Zend Framework - De la granularité des actions

par JYT (Les expériences Zend de Sekaijin) (Blog)

Date de publication : 10 octobre 2007

Dernière mise à jour :

Le modèle MVC propose de découper son application en actions regroupées dans des contrôleurs. Une question qu'on est amené naturellement à se poser dans ce contexte est : quel est le bon le découpage de l'application en divers contrôleurs ?

I - Introduction

Le modèle MVC propose de découper son application en actions regroupées dans des contrôleurs. Une question qu'on est amené naturellement à se poser dans ce contexte est : quel est le bon le découpage de l'application en divers contrôleurs ? Pour répondre à cette question, il existe des méthodes de conception qui permettent des découpages logiques de l'application en sous ensembles connexes. De ces approches, on trouve facilement les contrôleurs qui vont composer l'application. Mais quid des actions qui composent ces contrôleurs ? Quel niveau de granularité adopter pour décider des actions à implémenter ? Si l'on regarde globalement l'application en imaginant que, comme sous MVC, nous n'ayons qu'un seul script, nous aurions une espèce d'immense « Switch » pour déterminer quoi faire et dans quelles conditions. Heureusement, avec MVC le dispatcher est là pour assurer cet aiguillage. Au niveau de notre contrôleur, nous pouvons faire une seule action qui à son tour va devoir en fonction du contexte déterminer ce qu'il y a à faire. Nous n'aurons alors peut-être pas un gros « Switch », mais probablement un ou plusieurs « If ». On trouve dans les exemples beaucoup de contrôleurs dont le code ressemble à ceci :

```
if ($this->_request->isPost()) {
   $formvar = $f->filter($this->_request->getPost('varname'));
   if (empty($formvar)) {
      $this->view->message = 'Please provide a ...';
   } else {
      ...
      if ($result->isValid()) {
         ...
      } else {
         ...
      }
   }
} else {
...
```

Dans cet exemple, on va utiliser la présence ou pas de données de formulaire pour savoir où on en est au cours du processus de gestion d'un formulaire. Dans le cas où on a des données, on va alors tenter de faire le nécessaire ; dans le cas contraire, on va afficher le formulaire pour la saisie. À ce moment, on va devoir déterminer si on est en cours d'une édition ou d'un ajout *etc*.

Au final, suivant la complexité de l'objet à traiter, on va se retrouver avec un gros paquet de « if » imbriqués. Ne pourrait-on pas, tout comme on l'a fait au niveau général pour découper l'application en contrôleurs, découper la gestion de notre formulaire en une succession d'actions ? La réponse est oui évidemment. Mais dans ce cas, quand s'arrêter ? Car si je commence à découper le gros algorithme de gestion de mon formulaire en plus petits bouts, je peux très bien découper et découper encore jusqu'à obtenir une seule instruction par action, ce qui deviendrait plutôt compliqué. Là encore, il faut trouver un juste milieu.

II - Une action atomique

Avec le temps et l'expérience, j'en suis arrivé à la même conclusion que pour le découpage en contrôleurs. Le bon niveau est celui qui permet d'avoir un découpage logique et connexe. Reprenons l'exemple de la gestion d'un formulaire.

Au niveau général, on peut découper le processus de gestion de la façon suivante :

- Préparer le formulaire ;
- Afficher le formulaire ;
- Vérifier les données du formulaire ;
- Traiter les données.

Voici déjà un découpage que l'on peut faire et qui peut être déporté vers des actions différentes. La préparation consiste soit à récupérer l'enregistrement à éditer, soit à pré remplir un enregistrement en vu d'un ajout. L'affichage ne fait que gérer la vue, la vérification récupère les données du formulaire et vérifie qu'elles sont acceptables. Traiter les données sera par exemple l'enregistrement en base. Logiquement, on voit que les actions Préparer, Vérifier et Traiter n'affichent rien, elles se terminent donc par un « rediect » vers une autre action. Seule l'action Afficher utilise une vue. Peut-on aller plus loin, ou bien avons-nous atteint un niveau logique unitaire ? En clair, chaque action ainsi déterminée fait-elle une action unique ou doit elle encore déterminer par rapport à son contexte des traitements différents à effectuer ?

Si on regarde de plus près, on voit que l'action Préparer fait soit une édition soit un pré remplissage. On peut donc la couper en deux. Les autres n'ont pas de choix à faire autre que ceux induits par les données. Ces choix ne relèvent pas de la logique d'enchaînement mais des données traitées. Ainsi, la Vérification va faire un « redirect » soit vers le Traitement soit vers l'Affichage selon la valididité des données.

On peut donc afiner ce découpage ainsi :

- Préparer formulaire pour un ajout ;
- Rechercher les données de l'enregistrement à éditer ;
- Afficher le formulaire ;
- Vérifier les données du formulaire ;
- Traiter les données.

À ce niveau, on a un découpage logique du traitement d'un formulaire. Il faut noter qu'on travaille sur un contrôleur qui doit traiter un formulaire. Donc il n'est pas question d'aborder le traitement du métier, ni la façon de coder la vue. On peut aussi remarquer qu'en agissant ainsi, on a déterminé un algorithme général de traitement des formulaires.

III - Tout cela sert-il à quelque chose ?

On peut légitimement se poser la question. Rien ne m'empêche de faire cinq méthodes privées appelées par une seule action pour arriver au résultat. J'aurais un découpage clair et je n'aurais qu'une action. L'action contiendra alors un aiguillage vers la fonction à effectuer, les enchaînements de fonctions se faisant simplement. Or le dispatcher de MVC sert justement à faire les aiguillages. En reportant les fonctions vers des actions, on va utiliser le dispatcher plutôt qu'écrire l'aiguillage. En imaginant qu'une autre partie de l'application n'ait besoin que d'une partie des fonctions, il suffira d'enchaîner vers cette partie de l'application pour obtenir le résultat. Si une nouvelle étape intermédiaire (une confirmation par exemple) venait à être insérée, il suffirait de changer les redirections.

Bref, passer par les actions va permettre :

- De ne pas écrire de code pour aiguiller les actions, le dispatcher le faisant très bien ;
- D'ajouter de la souplesse dans l'évolution de l'application.

Mais il y a un autre petit avantage. C'est que finalement, il est possible d'écrire les enchaînements sans savoir quels objets un traite. Je n'ai à aucun moment discouru sur l'objet que traitait mon formulaire. Or je peux déjà écrire le contrôleur et ses enchaînements.

IV - Transmission des données

Reste qu'à changer d'action, j'ai introduit une complexité nouvelle. Lorsque dans une fonction je passe les valeurs issues d'une fonction à une autre, je n'ai pas de problème ; lorsque je passe d'une action à une autre, je perds toutes mes variables. En effet, une redirection est un nouvel appel au contrôleur. Je vais devoir utiliser la session pour les conserver. De même, lorsque je n'ai qu'une action pour afficher un message, je le mets dans la vue. Lorsque j'en ai plusieurs, il me faut les garder dans la session.

V - Un prototype de contrôleur

Je vais ici donner un prototype d'un tel contrôleur. Je reviendrai dessus dans un article ultérieur car on va voir que la gestion des messages et de la session peut être rendu générique pour toute l'application, ce qui fera l'objet d'un article prochain.

```php
<?php
class FormController extends Zend_Controller_Action
{
    /**
     * Affiche la liste des éléments
     *
     * @return null
     */
    public function showListAction(){
        $messenger = new Zend_Session_Namespace('messenger');
        $this->view->list = $this->model->getObjectList();
        $this->view->messages = $messenger;
        unset($messenger); //on supprime les messages ils sont affiché on en a plus besoin en session
    }

    /**
     * Prépare un enregistrement pour l'afficher dans le formulaire
     * redirige vers showForm
     * @see showForm
     */
    public function addAction() {
        $context = new Zend_Session_Namespace('context');
        $messenger = new Zend_Session_Namespace('messenger');

        $context->returnPath = '/FormController/showList';
        $context->saveMethod = 'add';
        // Demander au model un nouvel enregistrement avec les valeurs par défaut
        $context->formData = $this->model->newObject();
        $this->_redirect('/FormController/showForm');
    }

    /**
     * Recherche l'enregistrement pour l'afficher dans le formulaire
     * redirige vers showForm
     * @see showForm
     */
    public function editAction() {
        $context = new Zend_Session_Namespace('context');
        $messenger = new Zend_Session_Namespace('messenger');

        $context->returnPath = '/FormController/showList';
        $context->saveMethod = 'update';

        $id = $this->_request->get('id');
        $context->formData = $this->model->getItemById($id);
        if (!$context->formData) {
            // revenir au point de retour
            $messenger = 'no object for: '.$id;
            $redirect = $context->returnPath;
        } else {
            $redirect = '/FormController/showForm';
        }
        $this->_redirect($redirect);
    }

    /**
     * Affiche le formulaire d'édition d'un élément.
     *
```

```
59.      * @return null
60.      */
61.     public function showFormAction(){
62.        $context = new Zend_Session_Namespace('context');
63.        $messenger = new Zend_Session_Namespace('messenger');
64.
65.        $this->view->cancelAction = $context->returnPath;
66.        $this->view->saveAction = '/FormController/checkForm/';
67.
68.        $this->view->form = clone($context->formData);
69.        $this->view->messages = $messenger;
70.        unset($messenger); //on supprime les messages ils sont affiché on en a plus besoin en
 session
71.     }
72.
73.     /**
74.      * Récupère les données du formulaire les filtres et les vérifie
75.      * redirige vers save si le formulaire est valide showFrom sinon
76.      * @see save
77.      */
78.     public function checkFormAction() {
79.        $context = new Zend_Session_Namespace('context');
80.        if ($context->formData = $this->_request->get('form'))
81.        // vérification
82.        $ok = true;
83.        if ($ok) {
84.           $redirect = '/FormController/save';
85.        } else {
86.           $messenger = new Zend_Session_Namespace('messenger');
87.           $messenger = 'invalid datas';
88.           $redirect = '/FormController/showForm';
89.        }
90.        $this->_redirect($redirect);
91.     }
92.
93.     /**
94.      * enregistre l'élément dans la collection
95.      * redirige vers l'action qui précédé l'action add ou edit en cas de succès
96.      * vers showForm en cas d'échec
97.      * @see add
98.      * @see edit
99.      * @see showForm
100.      */
101.     public function saveAction($perform = true) {
102.        $context = new Zend_Session_Namespace('context');
103.        $messenger = new Zend_Session_Namespace('messenger');
104.
105.        $data = $context->formData;
106.        $method = $context->saveMethod;
107.
108.        $ok = $this->model->saveObject($data, $method);
109.        if ($ok) {
110.           $messenger = 'data saved';
111.           $redirect = $context->returnPath;
112.        } else {
113.           $messenger = 'error data could not be saved';
114.           $redirect = '/FormController/showForm';
115.        }
116.        $this->_redirect($redirect);
117.     }
118. }
```

On voit ici l'un des avantages de procéder ainsi : le contrôleur peut être rendu générique. On pourrait en faire une classe abstraite et la dériver en autant de formulaires à traiter. Mais on peut aussi l'utiliser sans traitement pour présenter un prototype de l'application.

Enfin, ce contrôleur tel qu'il est écrit pose un petit problème. En effet, on met des choses dans la session dans un namespace nommé **contexte** mais, si on utilise plusieurs instances de ce contrôleur, on va avoir des conflits ; de même avec le **messager**.

Je pratique cette approche en PHP maintenant depuis plusieurs années et elle a montré de gros avantages. Par exemple, dans une application les utilisateurs ont plusieurs profils. Et en calquant dans une première version les développements sur ce principe, nous avions l'affichage d'une liste d'utilisateurs, puis tout le processus d'ajout-modification. Puis pour chaque fiche utilisateur, un lien vers la liste de ses profils qui eux-mêmes avaient ces enchaînements. Une évolution qui n'a quasiment rien coûté fut de reporter la liste des profils de l'utilisateur dans sa fiche. Le contenu du **showList** de **ProfileController** a été reporté dans **showForm** de **UserController**. Il a suffit de changer la valeur de **returnPath** dans **>ProfileController** pour que tout soit opérationnel.

Au fil du temps, j'ai automatisé pas mal de choses liées à cette approche. Je vous les détaillerai dans les prochains articles.

A+JYT