

Atelier Zend Framework : Des résultats pertinents avec les tables liées : l'ORM en mode FullLoading

par Julien Pauli ([Tutoriels, article et conférences PHP et développement web](#)) ([Blog](#))

Date de publication : 29/04/2007

Dernière mise à jour : 02/02/2008

Des résultats pertinents avec les tables liées : Implémentation d'un mode FullLoading simple dans la couche ORM de Zend Framework

Voici encore un petit atelier, qui a suivi ma question : Avec l'ORM sous ZF, lorsque je récupère un résultat d'une base de données, je récupère de simples valeurs de colonnes de base de données; par exemple, je veux afficher le message d'un forum qui a l'id numéro 1, je fais donc `$message = $tableMessage->find(1)->current()`; et j'obtiens ceci :

```
object(Zend_Db_Table_Row)#18 (3) {
  ["_data:protected"] => array(13) {
    ["id"] => string(1) "1"
    ["idmembre"] => "2"
    ["messagetxt"] => "coucou"
  }
  (...)
}
```

C'est bien, mais si je veux l'afficher directement, ça pose problème. Car, bien que je n'affiche pas à l'écran de mes visiteurs l'id du message, ça va de soit, je n'ai pas non plus envie d'afficher l'id du membre qui a écrit ce message, mais plutôt son nom.

Je suis alors obligé de refaire une requête, dans ma logique, pour dire " va me chercher le membre qui a l'id 2, et affiche moi son nom"; par exemple un `$message->findParentMembre()`;

Et bien en réécrivant un peu de code dans l'abstraction de l'ORM (en l'étendant), on peut s'arranger pour obtenir directement l'objet membre, et implicitement appeler un `findParentXYZ()` là où il le faut; dans le résultat message, ce qui nous donne :

```
object(Zend_Db_Table_Row)#18 (6) {
  ["_data:protected"] => array(13) {
    ["id"] => string(1) "1"
    ["idmembre"] => object(Zend_Db_Table_Row)#16 (6) {
      ["_data:protected"] => array(13) {
        ["id"] => string(1) "2"
        ["nom"] => string(4) "John"
      }
      (...)
    }
    ["messagetxt"] => "coucou"
  }
  (...)
}
```

Et je peux donc écrire `echo $message->messagetxt` , et `$message->idmembre->nom`.

J'aurai pu chercher plus loin et améliorer la logique en me permettant d'écrire plutôt que `$message->idmembre->nom`, un `$message->auteur->nom`, auteur étant le nom du rôle que j'aurai mappé dans ma `referenceMap`. Bon ça sera votre exercice tien :-)

Voici la logique :

On doit intercepter toute génération de résultats (`fetchAll`, `find()`, `fetchRow()`, mais aussi tout autre appel personnalisé, comme `fetchByNom()`), et fouiller dans ces résultats.

Si ils comportent une colonne ayant été définie dans le `referenceMap` de la table mappée (`idmembre` ici, mais il aurait pu y en avoir plusieurs) , alors on fait directement appel à la table dépendante, en lui suggérant un `find()` :

```
<?php
class MyAbstractTables extends Zend_Db_Table_Abstract {
```

```
protected function _fetch(Zend_Db_Table_Select $select){
    $data = parent::_fetch($select);
    if (is_array($data)){
        $referenceArray = $this->_searchRelation($data[0]);
        foreach($data AS &$result){
            foreach ($referenceArray AS $key=>$class){
                if (array_key_exists($key,$result)){
                    $classObj = new $class();
                    $result[$key] = $classObj->find($result[$key])->current();
                }
            }
        }
    }
    return $data;
}

protected function _searchRelation(array $searchResult){
    $out = array();
    foreach ($this->_referenceMap AS $referenceMap){
        if (!is_array($referenceMap['columns'])){
            throw new Zend_Db_Table_Exception('La referenceMap[columns] doit être présentée sous forme
de tableau');
        }

        $tmp =
array_intersect_key(array_combine($referenceMap['columns'],array($referenceMap['refTableClass'])), $searchResult);

        $out[key($tmp)] = current($tmp);
    }
    return $out;
}
}
?>
```

_searchRelation() va me chercher toutes les relations de la table, et me retourner un tableau de ce style :

```
array(1) {
    ["idmembre"] => string(20) "MembreTable"
}
```

A chaque colonne mappée, correspond la table de référence, qui va nous servir à instancier l'objet référence.

Ensuite _fetch est appelée quelque soit la requête envoyée à la table (find(), fetchAll() ...), ainsi on effectue la requête.

Si il y a un résultat, on prend le premier, et on analyse ses colonnes.

On remarque que la colonne "idmembre" est spécifiée dans le tableau retourné par _searchRelation(), on sait donc que la valeur qui est dans le résultat de "idmembre", dans notre exemple : 2, doit faire l'objet d'une requête find(), sur la classe MembreTable, qui est la classe d'ORM de ma table membre.

On réitère ce remplacement pour tous les autres résultats, avant de renvoyer finalement le résultat modifié.

Ce genre de système et de syntaxe est très emprunté à Rails, un framework très connu du langage Ruby.

On pourrait associer ça à la notion de mode de chargement gourmand (contrairement au mode paresseux, le "lazymode"), à savoir que lorsque je demande un résultat, si il y a des dépendances dedans, résouds les, et ceci de manière récursive : si dans les dépendances de mon résultat, il y a d'autres dépendances, résouds les, et ainsi de suite...

Même si j'ai allégé le code au maximum, ça reste très lourd au niveau système. Un objet pouvant avoir tout un tas de relations enfouies les unes dans les autres, un simple appel à un jeu de résultat peut envahir la base de données, car la logique SQL n'est pas manipulée, en d'autres termes, on ne fait pas de `join()`, mais bien d'autres requêtes.

Pour optimiser tout ceci (car ce n'est pas 'viable', mais c'est une idée ;-)) on peut par exemple instaurer un "level", à savoir, je ne descends pas chercher des relations au delà de X niveaux. De cette manière, si un message est écrit par un membre, qui appartient à un groupe; lorsque je requête un message, je résouds le membre, mais pas son groupe. Actuellement ça n'est pas le cas, avec la syntaxe que je propose ici, on résoud tout.

Même si logiquement, elles ne doivent pas apparaître : gare aux dépendances cycliques (un message est écrit par un membre, qui écrit des messages), je ne donne pas cher de votre serveur là :D, mais d'un autre côté, ça voudrait dire que votre logique de base de données n'est pas bonne.

Deuxième option, et c'est la plus intéressante : utiliser un cache. Zend Framework propose un package `Zend_Cache` qui sait très bien faire ceci : mémoriser des résultats de requêtes. Je vous laisse ceci en exercice, en factorisant encore un peu le code et en rajoutant quelques paramètres, on peut finalement arriver à une logique full-loading viable (c'est à dire pas gourmande), avec un cache de requêtes.

