

Zend Framework - Utiliser une base de données

par JYT ([Les expériences Zend de Sekaijin](#)) (Blog)

Date de publication : 26 novembre 2007

Dernière mise à jour :

Zend Framework contient déjà tout le nécessaire pour accéder à de nombreuses base de données. Je vais aborder ici son utilisation. La littérature sur la connexion à une base est suffisamment dense pour que je considère qu'elle soit déjà faite. Je vais donc me concentrer sur l'accès aux données et leurs manipulations.

- I - Approche conceptuelle
- II - La représentation Objet
- III - Approche par dérivation
- IV - Le lien inverse
- V - Et l'ajout de données ?
- VI - Est-ce tout ?
- VII - Restriction d'accès
- VIII - Conclusion

I - Approche conceptuelle

Une base de données est avant tout un gisement organisé de données structurées. Que ce soit une base de données relationnelle ou un annuaire, ou encore un fichier de données structurées, on se trouve devant deux notions fondamentales : une collection et des éléments de cette dernière. Dans une base de données, les collections sont appelées **Tables** et les éléments **Rows**. Une base de donnée relationnelle peut comporter bien d'autres objets mais, dans son fondement, elle est constituée de Tables qui contiennent des Rows. Cette séparation distincte de ces deux éléments est primordiale pour obtenir une conception claire de son application. En effet, on ne fait pas les mêmes opérations sur une table que sur un row. En même temps, on voit que ces deux types d'objet sont intimement liés.

II - La représentation Objet

Zend Framework nous fournit deux classes pour représenter les éléments. Il s'agit de **Zend_Db_Table_Abstract** et **Zend_Db_Table_Row_Abstract**. Comme leur nom l'indique, ces classes étant abstraites, ne sont pas utilisables directement. Pour aller au plus simple, ZF fournit aussi un classe **Zend_Db_Table** dérivant de **Zend_Db_Table_Abstract** qui, par simple instantiation, permet de manipuler une table. Cette classe fournit déjà pas mal de possibilités mais on peut constater qu'il n'existe pas de classe pour les *row* correspondants. Simplement parce que de façon très générale, une classe `Zend_Db_Table_Row` ne comporterait rien de particulièrement intéressant. En effet, les opérations que l'on peut faire sur un *row* dépendent complètement du type de données qu'il contient. Et *a priori*, il peut s'agir de n'importe quoi.

III - Approche par dérivation

Si on définit des classes abstraites, c'est pour qu'elles soient dérivées en classes concrètes. Et si les concepteurs de ZF s'en sont donné la peine, ce n'est sûrement pas pour la beauté de la chose. Que peut nous apporter la dérivation des classes **Zend_Db_Table_Abstract** et **Zend_Db_Table_Row_Abstract** ? La première chose que nous apporte la dérivation c'est d'associer une classe dérivée de **Zend_Db_Table_Abstract** à une table particulière de notre base. Par exemple *Client_Table* lié à la table *client* de la base.

Pour cela il suffit de très peu de chose :

```
Class Client_Table extends Zend_Db_Table_Abstract {  
    protected $_name = 'client';  
}
```

Après instanciation de cette classe, un appel aux méthodes de consultation de la table remontera des rows de la table client, ces derniers étant les objets qu'est sensé manipuler mon modèle. La conception de mon application m'a normalement amené à la manipulation de clients et donc un ensemble de méthodes pour parvenir à mes fins. Mais ainsi définie, ma *Client_Table* me retourne juste des données. Il serait intéressant qu'elle me donne des objets clients, et c'est justement ce que permet la dérivation de **Zend_Db_Table_Row_Abstract**.

Je peux ainsi définir ma classe :

```
Class Client_Row extends Zend_Db_Table_Row_Abstract {  
}
```

J'ai donc une classe pour mes clients, et je vais pouvoir y attacher les méthodes que ma conception ma permet de définir. Reste à associer la classe **Client_Table** à la classe **Client_Row** cela se fait très simplement.

```
Class Client_Table extends Zend_Db_Table_Abstract {  
    protected $_name = 'client';  
    protected $_rowClass = 'Client_Row';  
}
```

Je bénéficie maintenant de toutes les méthodes de manipulation de ma table définies par ZF et à chaque fois que je récupère un *row* j'ai entre mes mains un *client* qui possède toutes ses méthodes de manipulation.

IV - Le lien inverse

On vient de voir que la classe **Client_Table** est en relation avec la classe **Client_Row**, c'est-à-dire que tout *row* sorti de la *table* est un **Client_Row**. Mais qu'en est-il dans l'autre sens ? La réponse est : la relation existe aussi. Intrinsèquement, la classe **Client_Row** connaît la table qui a donné naissance à ses instances. En clair, chaque **Client_Row** sorti d'une instance de **Client_Table** contient une référence à cette instance.

Il devient alors simple d'accéder à la *table* depuis le *row*. On peut se demander pourquoi. Simplement pour se faciliter la vie.

```
//trouver l'enregistrement client d'id 25
$aclient = $clientTable->find(25);
$aClient->maMethode();
$aClient->save();
```

Ce n'est qu'une facilité, on pourrait très bien en repasser par une instance de la table pour faire cette mise à jour. Mais le client ayant sa propre instance de la table, pourquoi ne pas en profiter. ZF a d'ailleurs prévu cela et fournit la méthode **save** et quelques autres.

On le voit, il n'est pas bien compliqué d'utiliser la *table* à partir du *row*.

V - Et l'ajout de données ?

Si je crée une instance de **Client_Row** directement je n'aurai pas associé la *table* à mon nouvel objet. J'aurai bien toutes les méthodes définies mais je ne pourrai pas mettre mon objet en base sans en passer par une instance de la table. Pour recréer ce client on peut soit passer par la méthode **setTable**, soit par le tableau de paramètres du constructeur. Personnellement, j'ai l'habitude de demander mes *Rows* à la *Table* : pourquoi ne pas demander les nouveaux rows à la table ?

```
Class Client_Table extends Zend_Db_Table_Abstract {
    protected $_name = 'client';
    protected $_rowClass = 'Client_Row';

    /**
     * Make new row associated to this table.
     *
     * @param StdClass|array $obj OPTIONAL object to cast
     * @return Fast_Db_Row
     */
    public function newRow($obj = null) {
        if ($obj)
        {
            if (is_object($obj))
                $obj = get_object_vars($obj);
            $row = $this->createRow($obj);
        } else {
            $row = $this->createRow();
        }
        return $row;
    }
}
```

Ainsi, j'ai le même fonctionnement que précédemment.

```
//Créer un nouvel enregistrement client.
$aclient = $clientTable->newRow();
$aclient->maMethode();
$aclient->save();
```

VI - Est-ce tout ?

Nous venons de voir que cette approche permet de faire un mapping d'une classe issue de ma conception sur les enregistrements d'une table de ma base. ZF offre au passage toute la manipulation des relations. Et toutes les fonctionnalités de manipulation des données sur de la table ajout, mise à jour, suppression. Mais cette approche permet aussi d'enrichir cette couverture déjà riche de méthodes spécifiques à notre modèle de données. Ainsi, la classe **Client_Table** peut être enrichie de toutes les méthodes qui me sont nécessaires et qui doivent me retourner un **Client_Row**. J'ai un réceptacle naturel pour toutes les requêtes qui retournent un ou des **Client_Row** mais aussi des données ayant un rapport direct avec la table **client**.

Par exemple, la méthode **getBoutiqueList** qui me retourne la liste des noms de boutique, trouve sa place dans **Boutique_Table**.

Je bénéficie au passage d'un nettoyage automatique des champs de mes objets. Il arrive parfois qu'on associe des valeurs à l'ensemble des données d'un formulaire pour se simplifier la vie. Si je donne tous les champs de formulaire à la méthode **newRow**, mon objet ne contiendra que les valeurs correspondant à un champ de la table, les autres étant supprimés au passage. Notez que si vous utilisez des tableaux de données à la place d'objets, la méthode **createRow** est là pour vous. De même pour obtenir le tableau des données d'un objet métier, vous avez la méthode **toArray**. Pour ma part, j'ai ajouté la méthode **toStdClass** pour obtenir un objet standard qui est parfois intéressant (pour la mise en session par exemple).

VII - Restriction d'accès

Les ACL permettent de restreindre l'accès aux fonctionnalités de votre application. Pour restreindre l'étendue des données, il est nécessaire d'en passer par des restrictions sur les requêtes.

Il est parfaitement possible d'intégrer cette restriction dans la table elle-même :

```

Class Client_Table extends Zend_Db_Table_Abstract {
    protected $_name = 'client';
    protected $_rowClass = 'Client_Row';
    /**
     * Restriction for query
     * @var string
     */
    protected $_restrict = array('cli_level > 0');
    public function __construct($config = array())
    {
        parent::__construct($config);
        $user = Zend_Auth::getInstance()->getIdentity();
        if ($user) {
            $this->_restrict[] = 'cli_group IN
                (SELECT grp_id FROM group WHERE usr_id = '.$user->usr_id.')';
        } else {
            //sans identité, on ne peut rien voir dans la base
            $this->_restrict = 'false';
        }
    }

    /**
     * Fetches one row in an object of type Zend_Db_Table_Row_Abstract,
     * or returns Boolean false if no row matches the specified criteria.
     *
     * @param string|array $where OPTIONAL An SQL WHERE clause.
     * @param string|array $order OPTIONAL An SQL ORDER clause.
     * @param boolean $restrict OPTIONAL use restrict SQL clause.
     * @return Fast_Db_Row The row results per the
     *         Zend_Db_Adapter fetch mode, or null if no row found.
     */
    public function fetchRow($where = null,
                            $order = null,
                            $restrict = true)
    {
        if ($restrict&&
            isset($this->_restrict)&&
            is_string($this->_restrict))
        {
            if (is_array($where))
            {
                $where[] = $this->_restrict;
            } else {
                $where = '('.$this->_restrict.') AND ( '.$where.' )';
            }
        } elseif ($restrict&&
            isset($this->_restrict)&&
            is_array($this->_restrict))
        {
            if (is_array($where))
            {
                $where = array_merge($where, $this->_restrict);
            } else {
                foreach ($this->_restrict as $constraint) {
                    $where = '('.$constraint.') AND ( '.$where.' )';
                }
            }
        }
        $res = parent::fetchRow($where, $order);
    }
}
    
```

Il est parfaitement possible d'intégrer cette restriction dans la table elle-même :

```
        return $res;
    }

    /**
     * Make new row associated to this table.
     *
     * @param StdClass|array $obj OPTIONAL object to cast
     * @return Fast_Db_Row
     */
    public function newRow($obj = null) {
        if ($obj)
        {
            if (is_object($obj))
                $obj = get_object_vars($obj);
            $row = $this->createRow($obj);
        } else {
            $row = $this->createRow();
        }
        return $row;
    }
}
```

Ainsi, lorsqu'on fait appel à une méthode d'extraction de données sur la table, la restriction s'applique automatiquement. Notez que j'ai prévu de pouvoir débraviller cette restriction avec le paramètre supplémentaire **\$restrict** : cela peut parfois s'avérer utile.

VIII - Conclusion

Cette approche conceptuelle permet facilement de coller la conception en classes sur son modèle de données. Elle permet un découpage logique de la partie métier de son application. Mais elle ne résout pas tout, il reste en effet des requêtes qu'on ne sait pas toujours où placer. Cependant, les possibilités d'extension de ces classes est grand et laisse libre court à votre imagination.

A+JYT

