# Zend Framework - Ajouter un champ calculé dans une table

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

Date de publication : 10 octobre 2007

Dernière mise à jour :

Il arrive parfois qu'il soit intéressant de véhiculer un champ dans un objet de mapping qui n'est pas conservé en base. C'est le cas entre autre des champs calculés.

# I - Introduction

Il arrive parfois qu'il soit intéressant de véhiculer un champ dans un objet de mapping qui n'est pas conservé en base. C'est le cas entre autre des champs calculés. Je veux par exemple un objet facture. Lorsque je manipule ma facture, un élément important est le total de la facture. Mais une facture en elle-même est composée de champs qui lui sont propres et de lignes de facturation. Chaque ligne véhicule une partie du total de la facture. Lorsque je manipule la facture, je n'ai pas nécessairement besoin de ses lignes de facturation.

Par exemple, lorsque je vérifie que le montant payé et bien le bon, il est inutile de remonter toutes les lignes : seul le total m'intéresse. Je peux alors décider de garder en base le total. Il me faudra alors veiller à ce que ce total soit tenu à jour en adéquation avec mes lignes de facturation. Je peux aussi décider de ne pas le garder en base, mais alors il me faudra le calculer et donc faire deux accès à la base pour obtenir ma facture sans ses lignes mais avec son total.

## II - Ajouter un champ à un objet de mapping

Une solution consiste à calculer ce champ lors de la lecture en base et de le garder dans un coin. Si je le mets directement dans mon objet de mapping, je vais me heurter à quelques difficultés. Par exemple si j'enregistre cet objet, l'ORM de Zend va au mieux supprimer le champ, au pire lever une exception et il aura raison. Il ne sait qu'en faire. Le but de cet article est de voir les points à lever pour arriver à cette solution.

## III - La table facture

La toute première étape consiste à créer un classe pour la table **facture** :

```
1. Class Model_Facture_Table extends Zend_Db_Table_Abstract {
2.     protected $_name = 'facture';
3.     protected $_rowClass = 'Model_Facture_Row';
4.     public function __construct($config = array())
5.     {
6.         parent::__construct($config);
7.         $this->_cols[] = 'fac_total';
8.     }
9. }
```

Et d'y ajouter une colonne. Ainsi, lorsque je sortirai ou entrerai une facture dans ma table, le champ **fac_total** ne sera pas inconnu.

Mais il va falloir aller un peu plus loin si on ne veut pas se retrouver avec des Exception de partout. La première étape passe par le calcul de ce champ. Donc lorsqu'on lit un enregistrement dans la base. ZF est ainsi fait que, quelle que soit la façon dont vous interrogez votre table, il passe toujours par la même méthode. Sauf évidement si vous écrivez vous-même une requête. La méthode qui définit la requête à effectuer sur la base pour lire un ou plusieurs enregistrements s'appelle **_fetch**. Il nous faut donc la modifier pour obtenir le résultat que nous cherchons. Ainsi, toute lecture prendra en compte notre modification. Pour bien comprendre ce que fait cette méthode, il suffit de se pencher sur le fonctionnement d'une **Zend_Db_Table**.

De façon générale, une **Zend_Db_Table** c'est :

```
1. SELECT * FROM tableName;
```

Les autres méthodes de recherche ne font qu'ajouter des clauses WHERE, ORDER *etc*. Le but de la méthode **_fetch** est de construire cette requête.

Moi, je voudrais à la place :

```
1. SELECT
2.     facture .*,
3.     SUM(lig_prix*lig_qte) AS fac_total
4. FROM facture
5. INNER JOIN lignes USING (fac_id)
6. GROUP BY fac_id;
```

## IV - Modifier la méthode _fetch

Tout d'abord, voyons comment est faite la méthode de ZF :

```
    /**
     * Support method for fetching rows.
     *
     * @param  string|array $where  OPTIONAL An SQL WHERE clause.
     * @param  string|array $order  OPTIONAL An SQL ORDER clause.
     * @param  int          $count  OPTIONAL An SQL LIMIT count.
     * @param  int          $offset OPTIONAL An SQL LIMIT offset.
     * @return array The row results, in FETCH_ASSOC mode.
     */
    protected function _fetch($where = null, $order = null, $count = null, $offset = null)
    {
        // selection tool
        $select = $this->_db->select();

        // the FROM clause
        $select->from($this->_name, $this->_cols, $this->_schema);

        // the WHERE clause
        $where = (array) $where;
        foreach ($where as $key => $val) {
            // is $key an int?
            if (is_int($key)) {
                // $val is the full condition
                $select->where($val);
            } else {
                // $key is the condition with placeholder,
                // and $val is quoted into the condition
                $select->where($key, $val);
            }
        }

        // the ORDER clause
        if (!is_array($order)) {
            $order = array($order);
        }
        foreach ($order as $val) {
            $select->order($val);
        }

        // the LIMIT clause
        $select->limit($count, $offset);
        // return the results
        $stmt = $this->_db->query($select);
        $data = $stmt->fetchAll(Zend_Db::FETCH_ASSOC);
        return $data;
    }
```

Cette méthode est un peu longue mais, au final, pas très complexe. On voit vite que notre SELECT * est à la ligne :

```
    $select->from($this->_name, $this->_cols, $this->_schema);
```

Et que c'est là qu'il faut intervenir. En effet, le reste n'est que l'ajout de clauses diverses.

Remplaçons donc :

```
    $cols = $this->_cols;
    unset($cols[array_search('fac_total',$cols)]);
```

```
$select->from($this->_name, $cols, $this->_schema)
    ->join('lignes','ligne.fac_id = facture.fac_id', array('fac_total' =>Zend_Db_Exp('SUM(lig_prix
 * lig_qte)')))
    ->group('fac_id');
```

Nous avons maintenant une table qui lit des factures avec leur total.

## V - L'écriture

Tant que nous ne faisons que lire dans la table avec cet objet, nous n'aurons pas de problème.

Mais si nous tentons un *update* ou un *insert*, nous allons avoir un problème. En effet, nous allons essayer de mettre à jour dans la base un champ qui n'y est pas. Il nous faut donc retirer ce champ de l'objet. Avant l'enregistrement :

```
        public function insert(array $data) {
            unset($data['fac_total']);
            parent::insert($data);
        }
```

Cela n'est en fait pas bien compliqué. La méthode **insert** va, en accord avec la liste des colonnes de la table, tenter d'ajouter l'enregistrement avec tous les champs présents dans **$this->_cols**. Si un champ de **$data** n'est pas dans la liste, il sera supprimé, mais **fac_total** y est puisque nous l'avons ajouté. Toutefois, ce champ n'est pas dans la table, le moteur SQL va donc rejeter la requête. Il suffit donc de supprimer ce champ des données.

La méthode **update** a un comportement équivalent mais légèrement différent car **update** va tenter de mettre à jour même les champs qui ne sont pas présents dans **$data**. Si je retire simplement le champ **fac_total**, la méthode **update** tentera de le mettre à **null** dans la base. Il faut donc retirer le champ des données mais aussi de la liste des colonnes, et le restituer ensuite car sinon notre objet **table** sera incohérent.

```
1.      public function update(array $data, $where)
2.      {
3.          unset($this->_cols[array_search('fac_total',$this->_cols)]);
4.          unset($data['fac_total']);
5.          $res = parent::update($data, $where);
6.          $this->_cols[] = 'fac_total';
7.          return $res;
8.      }
```

Il est ainsi possible d'ajouter de nombreux champs dans un objet **table** qui ne seront pas stockés en base. Il est assez simple de généraliser cette méthode. En se basant sur la description des relations dans les **Zend_Db_Table**, on peut imaginer une classe abstraite qui contiendrait un tableau des **autoJoinnedTable** et qui implémenterait ce principe.

## VI - Un exemple généralisé

Pour ma part, je l'utilise dans un tout autre contexte. J'utilise des tables hiérarchiques en POO ; il est simple et efficace d'utiliser une référence sur l'objet parent pour constituer une hiérarchie. Mais cette approche n'est pas performante dans une base de données. Une représentation intervallaire l'est bien mieux. Je vous conseille de lire les articles sur le sujet.

Dans le cas qui m'intéresse, j'ai donc des tables qui on un **id** numérique unique comme clef et des données. Le principe de la représentation intervallaire consiste à ajouter une borne droite et une borne gauche, j'ai aussi très souvent besoin de niveaux hiérarchiques relatifs. Par exemple tous les n#uds aux rangs N+1, N+2 et N+3 d'un n#ud donné. J'ajoute donc dans ma table un champ **level**.

Mais du côté PHP, il est plus simple d'utiliser une relation père-fils que la notion d'intervalle. Ma classe va donc masquer la représentation intervallaire à PHP. Elle va gérer elle-même les transactions nécessaires pour maintenir à jour les bornes des éléments de la table.

```
1. Zend_Loader::loadClass('Zend_Db_Table');
2.
3. Class Fast_Db_Hierarchical extends Zend _Db_Table {
4.
5.     /**
6.     * left field name in table
7.     *
8.     * @var string
9.     */
10.     protected $_left = NULL;
11.
12.     /**
13.     * right field name in table
14.     *
15.     * @var string
16.     */
17.     protected $_right = NULL;
18.
19.     /**
20.     * level field name in table
21.     *
22.     * @var string
23.     */
24.     protected $_level = NULL;
25.
26.     /**
27.     * virtual field name used has id of parent
28.     *
29.     * @var string
30.     */
31.     protected $_parent = NULL;
32.
33.     public function __construct($config = array())
34.     {
35.         parent::__construct($config);
36.         if (null == $this->_left)   throw new
 Fast_Exception_Db(Fast_Exception_Db::UNDEFINED_LEFT_KEY);
37.         if (null == $this->_right)  throw new
 Fast_Exception_Db(Fast_Exception_Db::UNDEFINED_RIGHT_KEY);
38.         if (null == $this->_level)  throw new
 Fast_Exception_Db(Fast_Exception_Db::UNDEFINED_LEVEL_KEY);
39.         if (null == $this->_parent) throw new
 Fast_Exception_Db(Fast_Exception_Db::UNDEFINED_PARENT);
40.         $this->_cols[] = $this->_parent;
41.     }
42.
```

```
43.    public function getById($id) {
44.       $rows = $this->find($id);
45.       if ($rows) {
46.          return $rows->current();
47.       }
48.       return false;
49.    }
50.
51.    public function deleteById($id) {
52.       if ($id == 1) return false; // on ne peut supprimer la racine
53.
54.       $this->_db->beginTransaction();
55.       $parent = $this->_db->select();
56.       $parent->from($this->_name, array('delete_left' => $this->_left, 'delete_right' =>
 $this->_right))
57.              ->where($this->_primary[1].' = :_deleteId');
58.       $statement = $this->_db->prepare($parent);
59.       $statement->execute(array('_deleteId' => $id));
60.       list($deleteLeft, $deleteRight) = array_values($statement->fetch());
61.       $res = false;
62.       if ($deleteLeft) {
63.          $row = $this->getById($id);
64.          $res = $row->delete();
65.
66.          if ($res) {
67.             $statement = $this->_db->prepare('UPDATE '.$this->_name.'
68.                                               SET '.$this->_left.' = '.$this->_left.' - 1
69.                                               WHERE '.$this->_left.' >= '.$deleteLeft.'
70.                                               AND '.$this->_right.' < '.$deleteRight.';');
71.             $statement->execute();
72.          }
73.
74.          if ($res) {
75.             $statement = $this->_db->prepare('UPDATE '.$this->_name.'
76.                                               SET '.$this->_left.' = '.$this->_left.' - 2
77.                                               WHERE '.$this->_left.' >= '.$deleteLeft.'
78.                                               AND '.$this->_right.' > '.$deleteRight.';');
79.             $statement->execute();
80.          }
81.
82.          if ($res) {
83.             $statement = $this->_db->prepare('UPDATE '.$this->_name.'
84.                                               SET '.$this->_right.' = '.$this->_right.' - 1
85.                                               WHERE '.$this->_right.' >= '.$deleteLeft.'
86.                                               AND '.$this->_right.' < '.$deleteRight.';');
87.             $statement->execute();
88.          }
89.
90.          if ($res) {
91.             $statement = $this->_db->prepare('UPDATE '.$this->_name.'
92.                                               SET '.$this->_right.' = '.$this->_right.' - 2
93.                                               WHERE '.$this->_right.' >= '.$deleteLeft.'
94.                                               AND '.$this->_right.' > '.$deleteRight.';');
95.             $statement->execute();
96.          }
97.
98.          if ($res) {
99.             $this->_db->commit();
100.          } else {
101.             $this->_db->rollback();
102.          }
103.       }
104.       return $res;
105.    }
106.
107.    public function UpdateById($data) {
108.       // on ne peut mettre à jour les donnée hiérarchique
109.       // ie on ne peut déplacer un noeud dans l'arbre.
```

```
110.        unset($data[$this->_parent]); // ne fait pas partie de la table
111.        unset($data[$this->_left]);   //ne peut être changé
112.        unset($data[$this->_right]);  //ne peut être changé
113.        unset($data[$this->_level]);  //ne peut être changé
114.        $res =  parent::UpdateById($data);
115.        return $res;
116.     }
117.
118.  public function insert(array $data) {
119.        # select left and level of parent
120.        $parentId = $data[$this->_parent];
121.
122.        $this->_db->beginTransaction();
123.        $parent = $this->_db->select();
124.        if (null != $this->_level) {
125.           $fields = array('parent_left' => $this->_left, 'parent_level' => $this->_level);
126.        } else {
127.           $fields = array('parent_left' => $this->_left,);
128.        }
129.
130.        $parent->from($this->_name, $fields)
131.               ->where($this->_primary[1].' = :_parentId');
132.        $statement = $this->_db->prepare($parent);
133.        $statement->execute(array('_parentId' => $parentId));
134.        list($parentLeft, $parentLevel) = array_values($statement->fetch());
135.
136.        $res = false;
137.        if ($parentLeft) {
138.           #update tree
139.           $statement = $this->_db->prepare('UPDATE '.$this->_name.'
140.                                             SET '.$this->_left.' = '.$this->_left.' + 2
141.                                             WHERE '.$this->_left.' > '.$parentLeft.';');
142.           $res = $statement->execute();
143.           if ($res) {
144.              $statement = $this->_db->prepare('UPDATE '.$this->_name.'
145.                                                SET '.$this->_right.' = '.$this->_right.' + 2
146.                                                WHERE '.$this->_right.' > '.$parentLeft.';');
147.              $statement->execute();
148.           }
149.
150.           #insert node
151.           if ($res) {
152.              unset($data[$this->_parent]);
153.              $data[$this->_left] = $parentLeft + 1;
154.              $data[$this->_right] = $parentLeft + 2;
155.              if (null != $this->_level)
156.                 $data[$this->_level] = $parentLevel + 1;
157.              $res =  parent::insert($data);
158.       }
159.           if ($res) {
160.              $this->_db->commit();
161.           } else {
162.              $this->_db->rollback();
163.           }
164.        }
165.        return $res;
166.  }
167.
168.     /**
169.      * Support method for fetching rows.
170.      *
171.      * @param  string|array $where  OPTIONAL An SQL WHERE clause.
172.      * @param  string|array $order  OPTIONAL An SQL ORDER clause.
173.      * @param  int          $count  OPTIONAL An SQL LIMIT count.
174.      * @param  int          $offset OPTIONAL An SQL LIMIT offset.
175.      * @return array The row results, in FETCH_ASSOC mode.
176.      */
177.     protected function _fetch($where = null, $order = null, $count = null, $offset = null)
```

```
178.     {
179.         // selection tool
180.         $select = $this->_db->select();
181.
182.         //no _parent col on master table
183.         $cols = $this->_cols;
184.         unset($cols[array_search($this->_parent,$cols)]);
185.
186.         // the FROM clause
187.         $select->from($this->_name, $cols, $this->_schema);
188.         // add the parent col
189.         $select->join(array('parent' => $this->_name),
190.                       '(parent.'.$this->_left.' < workgroup.'.$this->_left.') AND
191.                        (parent.'.$this->_right.' > workgroup.'.$this->_right.') AND
192.                        (parent.'.$this->_level.' = workgroup.'.$this->_level.' -1)',
193.                       array('parent_id' => 'parent.'.$this->_primary[1].''));
194.
195.         // the WHERE clause
196.         $where = (array) $where;
197.         foreach ($where as $key => $val) {
198.             // is $key an int?
199.             if (is_int($key)) {
200.                 // $val is the full condition
201.                 $select->where($val);
202.             } else {
203.                 // $key is the condition with placeholder,
204.                 // and $val is quoted into the condition
205.                 $select->where($key, $val);
206.             }
207.         }
208.
209.         // the ORDER clause
210.         if (!is_array($order)) {
211.             $order = array($order);
212.         }
213.         foreach ($order as $val) {
214.             $select->order($val);
215.         }
216.
217.         // the LIMIT clause
218.         $select->limit($count, $offset);
219.         // return the results
220.         $stmt = $this->_db->query($select);
221.         $data = $stmt->fetchAll(Zend_Db::FETCH_ASSOC);
222.         return $data;
223.     }
224.
225.     public function update(array $data, $where)
226.     {
227.       unset($this->_cols[array_search($this->_parent,$this->_cols)]);
228.       unset($data[$this->_parent]);
229.       $res = parent::update($data, $where);
230.       $this->_cols[] = $this->_parent;
231.       return $res;
232.     }
233.
234.    protected function _parent($row, $fiels) {
235.       $parent = $this->_parents($row, $fiels)
236.              ->order($this->_right)
237.              ->limit(1);
238.       return $parent;
239.    }
240.    protected function _parents($row, $fiels) {
241.       $parent = $this->_db->select();
242.       $parent->from($this->_name, $fiels)
243.              ->where($this->_left.'  < '.$row->{$this->_left})
244.              ->where($this->_right.' > '.$row->{$this->_right});
245.       return $parent;
```

```
246.    }
247.    protected function _childs($row, $fiels) {
248.       $childs = $this->_db->select();
249.       $childs->from($this->_name, $fiels)
250.              ->where($this->_left.'  > '.$row->{$this->_left})
251.              ->where($this->_right.' < '.$row->{$this->_right});
252.       return $childs;
253.    }
254.
255.     /**
256.      * This is the find Zend_Db_Table Abstract method
257.      * But the where closes are prefixed by the table name
258.      *
259.      * Fetches rows by primary key.
260.      * The arguments specify the primary key values.
261.      * If the table has a multi-column primary key, you must
262.      * pass as many arguments as the count of column in the
263.      * primary key.
264.      *
265.      * To find multiple rows by primary key, the argument
266.      * should be an array.  If the table has a multi-column
267.      * primary key, all arguments must be arrays with the
268.      * same number of elements.
269.      *
270.      * The find() method always returns a Rowset object,
271.      * even if only one row was found.
272.      *
273.      * @param  mixed                        The value(s) of the primary key.
274.      * @return Zend_Db_Table_Rowset_Abstract Row(s) matching the criteria.
275.      * @throws Zend_Db_Table_Exception
276.      */
277.     public function find()
278.     {
279.         $args = func_get_args();
280.         $keyNames = array_values((array) $this->_primary);
281.
282.         if (empty($args)) {
283.             require_once 'Zend/Db/Table/Exception.php';
284.             throw new Zend_Db_Table_Exception(#No value(s) specified for the primary key#);
285.         }
286.
287.         if (count($args) != count($keyNames)) {
288.             require_once 'Zend/Db/Table/Exception.php';
289.             throw new Zend_Db_Table_Exception(#Missing value(s) for the primary key#);
290.         }
291.
292.         $whereList = array();
293.         $numberTerms = 0;
294.         foreach ($args as $keyPosition => $keyValues) {
295.             // Coerce the values to an array.
296.             // Don't simply typecast to array, because the values
297.             // might be Zend_Db_Expr objects.
298.             if (!is_array($keyValues)) {
299.                 $keyValues = array($keyValues);
300.             }
301.             if ($numberTerms == 0) {
302.                 $numberTerms = count($keyValues);
303.             } else if (count($keyValues) != $numberTerms) {
304.                 require_once 'Zend/Db/Table/Exception.php';
305.                 throw new Zend_Db_Table_Exception(#Missing value(s) for the primary key#);
306.             }
307.             for ($i = 0; $i < count($keyValues); ++$i) {
308.                 $whereList[$i][$keyPosition] = $keyValues[$i];
309.             }
310.         }
311.         $whereClause = null;
312.         if (count($whereList)) {
313.             $whereOrTerms = array();
```

```
314.             foreach ($whereList as $keyValueSets) {
315.                 $whereAndTerms = array();
316.                 foreach ($keyValueSets as $keyPosition => $keyValue) {
317.                     $whereAndTerms[] = $this->_db->quoteInto(
318.
 $this->_db->quoteIdentifier($this->_name).'.'.$this->_db->quoteIdentifier($keyNames[$keyPosition],
 true) . ' = ?',
319.                         $keyValue
320.                     );
321.                 }
322.                 $whereOrTerms[] = '(' . implode(' AND ', $whereAndTerms) . ')';
323.             }
324.             $whereClause = '(' . implode(' OR ', $whereOrTerms) . ')';
325.         }
326.
327.         return $this->fetchAll($whereClause);
328.     }
329.
330. }
```

Vous aurez noté la présence de la méthode **find** alors qu'elle est disponible dans la classe **Zend_Db_Table**. Cela vient du fait que je fais une auto-jointure : je joins la table sur elle-même. Du coup, tous les champs de la table sont en double dans la requête. Or la méthode **find** construit des closes **where** simples. Il est nécessaire dans ce cas de les préfixer du nom de la table, c'est ce que j'ai ajouté à la méthode **find**.

## VII - Conclusion

Cette façon de dériver la classe **Zend_Db_Table** permet d'imaginer toute sorte de mappings entre un objet et un ensemble de tables dans la base. Par exemple, un modèle dans lequel les adresses sont dans une table à part des clients alors que l'objet de mapping remonte toujours l'ensemble, ou la remontée systématique des valeurs des tables de références, *etc*.

A+JYT

**ZIP File : Hierarchical Table**