

# Atelier Zend Framework : Création d'un processus de listage du contenu d'une table

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

Date de publication : 15/09/2007

Dernière mise à jour :

Nous allons voir comment créer un schéma sous Zend Framework permettant de lister le contenu d'une table de base de données.

Le projet prend en compte la pagination, le classement et le tri par colonne, et doit pouvoir se généraliser quelle que soit la table interrogée.

Le processus utilisera MVC mais une grande partie du travail se fait sur la couche Modèle.

- I - Introduction
- II - Dans le vif du sujet
  - II-A - Un peu de théorie
  - II-B - Voyons le code
- III - Le fonctionnement interne
- IV - Conclusion

## I - Introduction

Je suis parti du principe que souvent, on a besoin de lister le contenu entier d'une table pour tout simplement afficher ses données. "Voici la liste de nos membres", "Liste des reservations en cours" ...

Evidemment, on pourrait dans chaque classe et à chaque endroit, écrire le même code. DRY (Don't Repeat Yourself) oblige : une bonne partie de ce code est factorisable, et c'est ce que nous allons faire.

Nous allons redéfinir `Zend_Db_Table_Abstract` afin d'avoir à disposition une méthode comme suit : `liste(ORDRE , TRI, PAGE, NOM-COLONNES)`.

Chaque table pourra donc être listée, et classée sur une colonne (ORDRE), triée ascendante descendante (TRI). De plus on pourra demander à accéder à une certaine page du resultat (PAGE), et comme on ne veut pas forcément que nos noms de colonne SQL apparaissent tels quels, on pourra effectuer des remplacements (NOM-COLONNES).

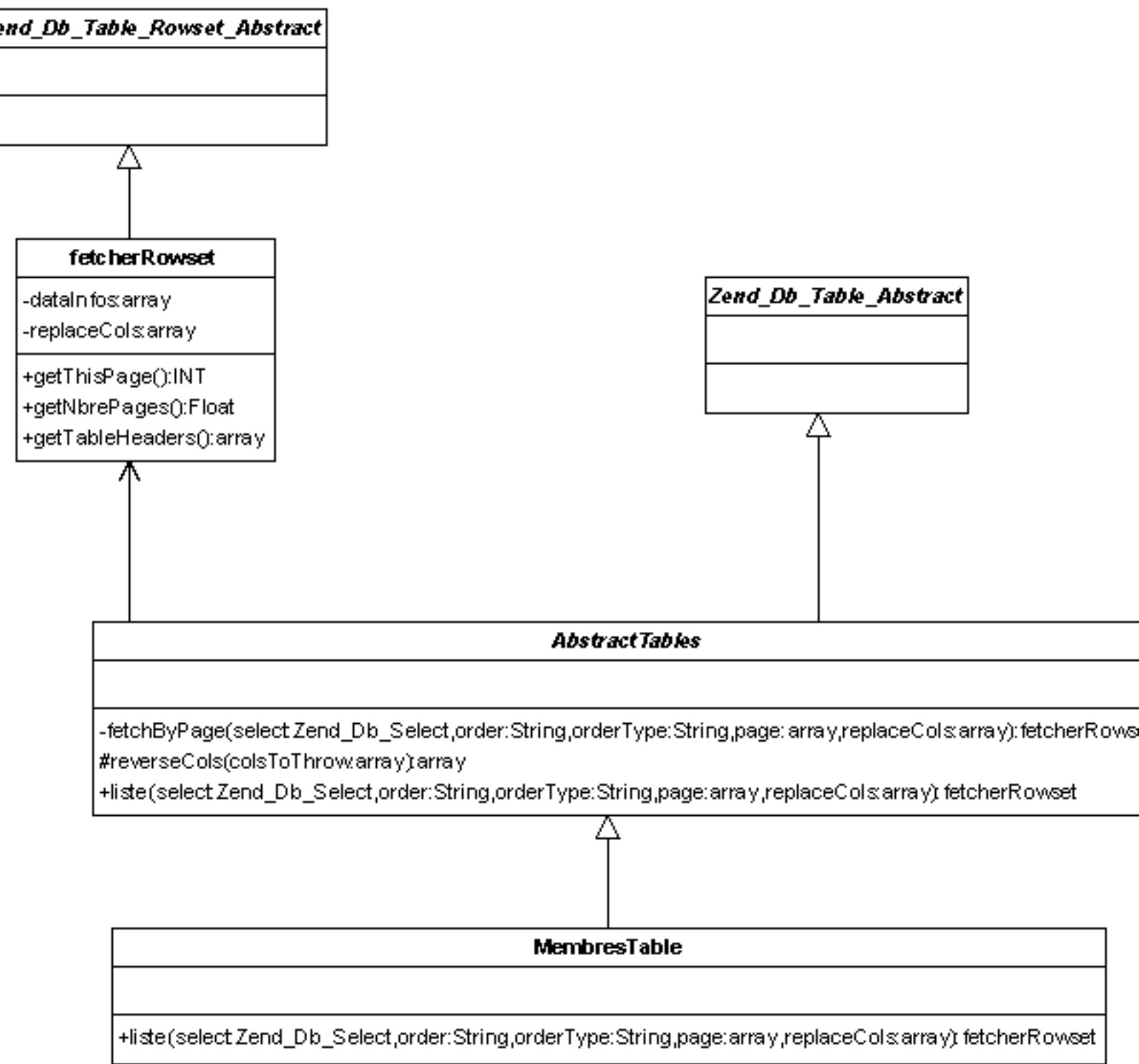
Les pré-requis sont donc : avoir déjà manipulé en profondeur le composant `Zend_Db`. Pour notre exemple, nous utiliserons `PDO_MYSQL`.

Avoir déjà manipulé le modèle MVC de ZendFramework, nous y ferons référence, sans y entrer en profondeur (pas de nécessité).

## II - Dans le vif du sujet

### II-A - Un peu de théorie

ur de tables



### Diagramme de classes listeur de tables

Dans cet exemple, notre classe maitresse est *MembresTable*, car nous allons supposer que nous allons lister, et afficher des membres.

Cette classe n'hérite pas directement de *Zend\_Db\_Table\_Abstract*, il y a une couche supplémentaire : *AbstractTables*, dans laquelle va être définie la logique de récupération des données de la table, en vue de la lister.

Toutes les tables réelles hériteront donc de ***AbstractTables***, elles seront donc toutes listables :

#### 3 méthodes ont été rajoutées :

- 1 ***fetchByPage()*** comporte toute la logique de selection des données, elle est privée, donc inaccessible
- 2 ***liste()*** va nous servir à commander le listage de la table
- 3 ***reverseCols()*** va nous permettre de lister certaines colonnes, et pas d'autres

Soit vous redéfinissez ***liste()*** dans la classe de mapping de votre table (nous le faisons pour *MembresTable*), soit vous héritez d'une méthode ***liste()*** déjà toute prête :

Par défaut, ***liste()*** renvoie toutes les colonnes de la table, sauf les clés primaires. Elle demande un affichage de la page 1, à raison de 10 résultats par page.

Une fois tous ces paramètres ajustés, elle délègue tout le processus de récupération de données, à ***fetchByPage()***, qui comporte véritablement tout le code actif.

Seulement, je vous ai dit que j'automatise tout. Je ne peux me permettre de récupérer des données, dans un *Zend\_Db\_Table\_Rowset*, comme on a l'habitude de le faire.

Je dois recréer un conteneur, que j'ai appelé *fetcherRowset*, qui hérite de *Zend\_Db\_Table\_Rowset\_Abstract*, mais va rajouter certains paramètres. En effet, mon résultat, en plus de contenir les données propres, va devoir participer à la pagination. Il devra donc fournir à ma vue le nombre total de pages, la page actuelle (pour laquelle il contient ses données), et tous les noms des colonnes SQL rapportées.

Ma vue n'aura alors quasiment plus rien à faire, et elle n'aura aucun calcul à effectuer. Mon contrôleur récupère un objet *fetcherRowset* qu'il va directement passer à la vue; voyez le diagramme de séquence :

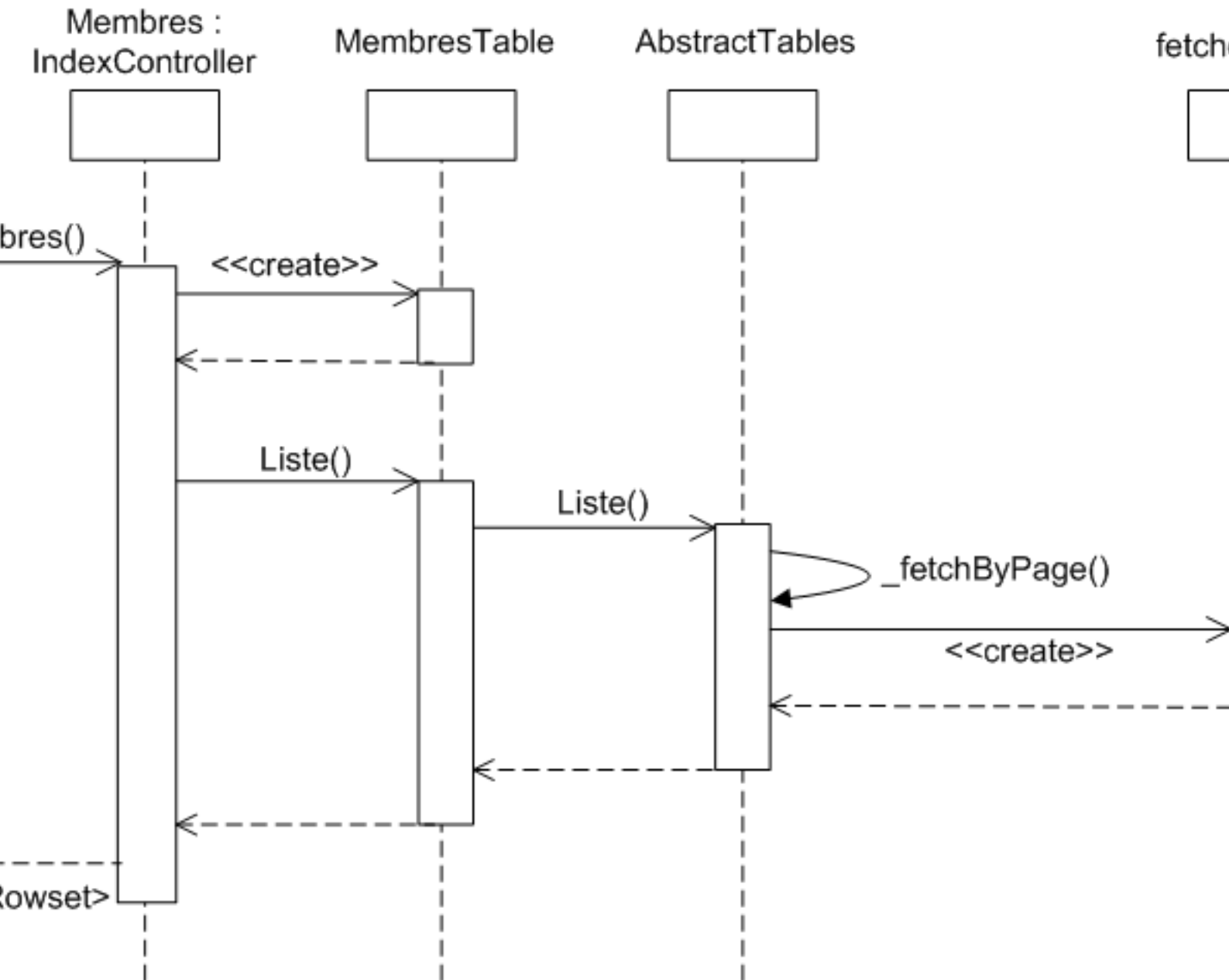


Diagramme de séquence - listeur de tables

Si on suit cette logique, tous les résultats de vos tables seront des *fetcherRowset* et non plus des *Zend\_Db\_Table\_Rowset*, mais la magie de l'héritage va faire que votre application continuera parfaitement de fonctionner, car mon conteneur de résultats (fetcheur) personnalisé ne gêne en rien le comportement de celui qu'il redéfinit.

## II-B - Voyons le code

La code final ressemble donc à ce :

Le modèle : la classe Membre

```
<?php
class MembreTable extends AbstractTables
```

## Le modèle : la classe Membres

```
{
    protected $_name = 'membres';
    protected $_primary = 'id';

    public function liste($order = null, $orderType = null, array $page = array(1,10), array
    $replaceCols = array())
    {
        $select = $this->_db->select()
            ->from($this->_name,array('login','email','nbreconnect'));
        return parent::liste($select, $order, $orderType, $page, $replaceCols);
    }
}
```

Je redéfinit **liste()**, hérité de *AbstractTables*. Le premier paramètre doit être un objet *Zend\_Db\_Select*, obligatoirement.

Je définis donc mon objet pour requêter cette table (*\$this->\_name*), et je lui dis que je veux sélectionner les colonnes login, email, et nbreconnect.

Passons au contrôleur qui demande ceci :

## Le contrôleur

```
<?php
class Membres_IndexController extends Zend_Controller_Action
{
    function listeAction()
    {
        $table = new MembreTable();
        $result = $table->liste($this->_getParam('order'),
            $this->_getParam('orderType')
            ,array($this->_hasParam('page') ? $this->_getParam('page') : 1,10)
        );
        $this->view->data = $result;
        $this->view->title = sprintf('Notre site compte aujourd\'hui %d membres',$result->count());

        $this->view->baseUrl = $this->getRequest()->getBaseUrl();
        $this->view->thisUrl = $this->view->baseUrl . $this->getRequest()->getPathInfo();

        $this->render('reader');
    }
}
```

Le premier paramètre, *\$this->\_getParam('order')*, représente la colonne de classement, reçue en GET, de la vue.

S'en suit le tri, *\$this->\_getParam('orderType')*, ascendant ou descendant.

Ensuite le numéro de la page. La page est notée comme un array, la première valeur désigne la page que l'on demande d'afficher. La deuxième valeur du tableau désigne le nombre de résultats par page.

Par défaut, il s'agit de 1,10 : la page 1 pour un ensemble de résultats triés à raison de 10 par page.

*\$result* est notre *fetcherRowset*, on le passe à une variable de vue, nommée data. On lui fournit ensuite l'URL de la page appelée, pour qu'elle puisse générer les liens qui vont permettre de la rappeler, mais en changeant par exemple le tri, ou la page que l'on désire.

On rend ensuite le script qui va en fait mettre en forme le résultat, en utilisant l'objet *fetcherRowset*, voyons voyons :

## La vue

```
<p><?php echo $this->title ?></p>
<table border="1">
  <tr>
    <?php foreach($this->data->getTableHeaders() AS $titre) :
      printf("<th> <a href=\"%1\$s?page=%2\$d&order=%3\$s&ordertype=asc\">^</a>%3\$s
        <a href=\"%1\$s?page=%2\$d&order=%3\$s&ordertype=desc\">v</a></th>",
          $this->thisUrl,
          $this->data->getThisPage(),
          $this->escape($titre)
        );
    endforeach; ?>
  </tr>

  <?php foreach($this->data AS $id=>$row) :
    $row = $row->toArray();?>
  <tr>
    <?php
      foreach($row AS $data) : ?>
        <td> <?php echo $this->escape($data); ?> </td>
      endforeach; ?>
    </tr>
  <?php endforeach;?>
</table>
<br>
<form method="get" action="<?php echo $this->thisUrl ?>" >
<?php $listePages = range(0,$this->data->getNbPages());unset($listePages[0]);
  echo $this->formSelect('page',$this->data->getThisPage(), null, $listePages);
  echo $this->formSubmit(null,'go'); ?>
</form>
</center>

</div>
```

***getTableHeaders()*** provient de notre objet *fetcherRowset*, c'est un tableau qui contient, dans leur ordre d'apparition, les noms des colonnes sélectionnées dans la requête de listage, pour les afficher.

Si on avait utilisé la clause *\$replaceCols*, on aurait pu changer ces noms, car souvent, les noms des colonnes SQL ne sont pas propres, et pas destinés à être affichés en clair.

***getThisPage()*** retourne tout simplement le numéro de la page que l'on a demandé de lister, ici c'est la 1.

L'affichage en lui-même des données, dans la boucle *foreach*, est hérité de *Zend\_Db\_Table\_Rowset*, je ne le commente pas, on utilise les interfaces *Iterator* et *Countable*, pour respectivement itérer sur le résultat, et compter le nombre total de résultats dans le conteneur.

***getNbPages()***, par contre est fabriqué par ma couche *fetcherRowset*, il retourne un entier : le nombre total de page. C'est important pour que la vue puisse créer le formulaire permettant à l'utilisateur d'accéder à une page, parmi toutes.

Le résultat visuel est donc un tableau, chaque nom de colonne est précédée d'un signe flèche montante, et suivie d'un signe flèche descendante, sur lesquelles des liens ont été fabriqués.

On aura deviné que ceci sert à demander à notre contrôleur un tri.

En bas de page, un formulaire permettant d'accéder à une page en particulier.

Notez qu'ici c'est MA vue, je la construis comme je veux, en fonction des données auxquelles j'ai accès.

Pour résumer, nous avons grâce à un méthode, le pouvoir de lister une table, en personnalisant le résultat. Pas mal hein ?

Mieux, si je ne redéfinit pas la méthode **liste()**, elle m'est quand même accessible (par défaut, on a dit, elle liste toutes les colonnes sauf les clés primaires, et renvoie la page 1, à raison de 10 résultats par page).

Voici à quoi sert la méthode protégée **reverseCols()** :

#### la méthode reverseCols()

```
<?php
// MembresTable.php

public function liste($order = null, $orderType = null, array $page = array(1,10), array
$replaceCols = array())
{
    $select = $this->_db->select()

->from($this->_name,$this->_reverseCols(array('login','email','nbreconnect')));
    return parent::liste($select, $order, $orderType, $page, $replaceCols);
}
}
```

Je demande presque la même chose que l'exemple d'au dessus. En fait presque, car je demande l'inverse ;-) Liste moi toutes les colonnes de la tables, SAUF login, email, et nbreconnect.

Et cette méthode est utile pour bien des besoins, il peut être intéressant de vouloir effectuer une selection inverse.

### III - Le fonctionnement interne

Non non, je ne pars pas avant de vous donner le code, le vrai. Je parle bien entendu du code de *AbstractTables* et *fetcherRowset*

Je vais même vous l'expliquer :

#### AbstractTables

```
<?php
abstract class AbstractTables extends Zend_Db_Table_Abstract
{
    /**
     * Méthode de récupération de résultat gérant la pagination
     *
     * @param Zend_Db_Select $select
     * @param string $order
     * @param string $orderType
     * @param array $page
     * @param array $replaceCols noms de colonne à remplacer pour affichage
     * @return FetcherRowset
     */
    final private function _fetchByPage(Zend_Db_Select $select,
        $order = null,
        $orderType = null,
        array $page = array(1,10),
        array $replaceCols = array())
    {
        $order = strtolower($order);
        $page = (array) $page;
        // $dataInfo est le tableau passé au jeu de résultat, qui va permettre de gérer la mise en
        page notamment
        $dataInfo = new ArrayObject(array(), ArrayObject::ARRAY_AS_PROPS);

        $dataInfo->pageEnCours = (int)$page[0];
        $dataInfo->resultParPage = (int)$page[1];

        $orderType = ( isset($orderType) && in_array(strtoupper($orderType), array('ASC', 'DESC')) ) ?
        $orderType : 'ASC';

        foreach($select->getPart(Zend_Db_Select::FROM) AS $table_name) {
            $tables[] = $table_name['tableName'];
        }
        if (!in_array($this->_name, $tables)) { // on est obligé de générer une requête concernant
        cette table
            throw new Zend_Db_Exception('Selection hors table');
        }

        // clause ORDER
        if (in_array($order, $this->_cols)) {
            $select->order($order.' '.$orderType);
        }
    }
    $selectQuery = clone($select);
    $select->reset(Zend_Db_Select::COLUMNS)
        >reset(Zend_Db_Select::FROM);
    $select->from($this->_name, 'COUNT(' . implode(', ', $this->_primary) . ')');
    //echo $select; // débogage
    try{
        $dataInfo->totalResults = $this->_db->query($select)->fetchColumn(); // nombre total de
        résultat
        $selectQuery->limit((int)$page[1], (( (int)$page[0] ) -1 ) * (int)$page[1] ); // ajout de
        la pagination
        $data = $this->_db->fetchAll($selectQuery); // on requête
    } catch (Exception $e) {
```

## AbstractTables

```

        throw new Zend_Db_Exception('Erreur de selection de données');
    }
    // Utilisation du Rowset spécial intégrant les dataInfo et les remplacements de noms de
    colonnes
    return new FetcherRowset(array(
        'db'          => $this->_db,
        'table'       => $this,
        'data'        => $data,
        'dataInfo'    => $dataInfo,
        'replaceCols' => $replaceCols,
    ));
}

/**
 * Fonction de listage de la table en vue de l'afficher
 * Cette fonction basique dont toutes les tables héritent
 * liste toutes les colonnes sauf les clés primaires,
 * si elle n'est pas redéfinie
 *
 * @param Zend_Db_Select $select
 * @param string $order
 * @param string $orderType
 * @param array $page
 * @param array $replaceCols
 * @return FetcherRowset
 */
public function liste(Zend_Db_Select $select = null,
    $order = null,
    $orderType = null,
    array $page = array(1,10),
    array $replaceCols = array())
{
    if (is_null($select)) {
        $select = $this->getAdapter()->select();
        // par défaut on sélectionne toutes les colonnes sauf les clés primaires
        $select->from($this->_name,array_diff($this->_cols,$this->_primary));
    }
    return $this->fetchByPage($select, $order, $orderType, $page, $replaceCols);
}

/**
 * Utilisée pour permettre de sélectionner toutes les colonnes
 * sauf celles passées en paramètre
 * Pratique pour les tables ayant beaucoup de colonnes et dont on ne
 * veut en éliminer qu'une pour la selection ...
 *
 * @param array $colsToThrow
 * @return array
 */
final protected function _reverseCols(array $colsToThrow = array())
{
    return array_diff($this->_cols,$colsToThrow);
}
}

```

Commençons par le bas, **reverseCols()** exécute une bête différence entre 2 tableaux. Le tableau `$this->_cols` représente les colonnes de la table actuelle.

Cette propriété est héritée de *Zend\_Db\_Table\_Abstract*, lors de l'héritage du constructeur

**liste()** comme nous l'avons déjà dit, existe; notez bien que tous les paramètres sont optionnels, ainsi vous pouvez redéfinir **liste()** en lui passant juste `$replaceCols`, par exemple.

\$replaceCols est un array, qui va servir au remplacement des noms de colonne SQL, utilisez le comme ceci :  
\$replaceCols = array('colonneSql'=>'nomDeRemplacement');, veillez à ce que la colonne originale existe bien dans le résultat.

La suite se corse légèrement, notre fameuse méthode privée **fetchByPage()**. Tout d'abord, nous initialisons notre \$dataInfo comme étant un ArrayObject, cette variable sera fournie au *fetcherRowset* qui lui même nous y donnera accès plus tard dans la vue.

On vérifie ensuite \$orderListe avec une approche par liste blanche. Puis, grâce à la méthode **Zend\_Db\_Select::getPart()**, nous récupérons la partie "from" de la requête passée en paramètre, afin de vérifier si elle porte bien sur la table actuelle (\$this->\_name).

En effet, on ne peut utiliser **liste()** sur une table "Membre", pour récupérer la liste de la table "Message", ça n'a pas de sens.

Idem pour la clause "order", on vérifie que la colonne passée est bien une des colonnes de la table.

Ensuite, sans parler d'une éventuelle clause "limit", on compte le nombre de résultats de la requête : c'est le nombre total, nécessaire pour le calcul de la pagination. Pour ceci, il faut faire un "select count('clés-primaires)" sur la table. On utilise donc les propriétés de l'objet \$select pour effacer la clause "from" et la clause "columns" de notre selection, et les remplacer par "count('clés-primaires)". On ne fait pas de count(\*), car ceci renvoie les résultats ayant des valeurs NULL, même si ce n'est pas le cas ici, on généralise.

On effectue ensuite la requête, et via **fetchColumn()** (issu de PDO) on récupère le nombre de résultats total de la requête.

Etant donné qu'on a pris soin de cloner notre objet \$select, on reprend ensuite le clône auquel on rajoute enfin la clause "limit", pour finalement, envoyer la requête.

Enfin, nous construisons le jeu de résultat. Il est issu de *Zend\_Db\_Table\_Rowset\_Abstract*, il lui faut les clés "db", "table", et "data", pour se conformer à notre classe mère.

Notre *fetcherRowset* se permet de rajouter les "dataInfo" et "replaceCols", qui seront fournis à la vue, via le contrôleur.

Du côté du *fetcherRowset*, voici son code :

```
<?php
class FetcherRowset extends Zend_Db_Table_Rowset_Abstract{

    protected $_dataInfo;

    protected $_replaceCols = array();

    public function __construct(array $config = array()){
        if (!array_key_exists('dataInfo',$config) ||
            !array_key_exists('replaceCols',$config) ||
            !($config['dataInfo'] instanceof ArrayObject) ) {

            throw new Zend_Db_Table_Row_Exception('informations sur les données non fournies au
rowset ou erronées');
        }

        $this->_dataInfo    = $config['dataInfo'];
        $this->_replaceCols = $config['replaceCols'];
        parent::__construct($config);
    }
}
```

```
/**
 * retourne la page en cours de lecture
 *
 * @return int
 */
public function getThisPage(){
    return $this->_dataInfo->pageEnCours;
}

/**
 * Retourne le nombre total de pages de résultats
 *
 * @return int
 */
public function getNbPages(){
    return ceil($this->_dataInfo->totalResults/$this->_dataInfo->resultParPage);
}

/**
 * Retourne les titres de colonnes à afficher
 *
 * @return array
 */
public function getTableHeaders()
{
    if (count($this->_data) == 0) {
        throw new Zend_Db_Table_Exception('Pas de résultats à interpréter');
    }
    $data = array_keys(current($this->_data));
    $tableHeaders = array_combine($data, $data);
    return array_values(array_merge($tableHeaders,$this->_replaceCols));
}
}
```

On joue simplement avec les 2 nouveaux paramètres "dataInfo", et "replaceCols"; afin de fournir des méthodes supplémentaires

## IV - Conclusion

Voilà c'est ici que s'achève cet atelier Zend Framework. Je précise une fois de plus que ce n'est qu'une idée, et que je n'assure pas qu'elle soit sans bugs :-)

C'est pour montrer le genre de manipulations qu'offre le composant Zend\_Db, lorsqu'on a un besoin particulier. En théorie, cette fonctionnalité marche sur toute table; à vous après d'adapter ou de proposer d'autres idées, pourquoi pas :-)

