

Implémenter le Full Loading sur Zend Framework

par Baptiste Wicht ([home](#))

Date de publication : 22 Octobre 2007

Dernière mise à jour : 29 Avril 2009

Vous utilisez Zend Framework et plus précisément son module ORM (Mapping Objet Relationnel) et vous aimeriez implémenter du Full Loading ? C'est-à-dire charger directement les objets liés à d'autres. Alors, cet article est fait pour vous, il va vous montrer comment implémenter pas à pas le mode Full Loading sur le framework.

Cet article fonctionne avec Zend Framework 1.7.

1 - Introduction.....	3
2 - Le contexte.....	4
3 - Implémentation du Full Loading.....	5
4 - Appliquer le Full Loading à createRow().....	8
5 - Problèmes rencontrés.....	9
5.1 - Impossible de sauvegarder les enregistrements full loadé.....	9
5.2 - Problème avec les relations réciproques entre 2 objets d'une même table.....	11
6 - Conclusion.....	13
6.1 - Code complet.....	13

1 - Introduction

Pour commencer, nous allons voir ce qu'est le Full Loading. Si vous savez déjà ce qu'est le Full Loading, vous pouvez passer directement à la section suivante.

Tout d'abord, comment fonctionne le Zend Framework par défaut avec les tables liées ? Vous pouvez les définir dans un tableau à deux dimensions nommé **\$_referenceMap**. Dans ce tableau vous donnerez les paramètres de la liaison, c'est-à-dire la colonne qui est liée à une autre table, le modèle à utiliser et la colonne concernée dans le modèle de destination. Mais ensuite, il ne fait rien directement avec les données, il vous faut encore invoquer des méthodes vous-mêmes. Ce sont les méthodes **findParent**, **findDependentRowSet** et **find<table>Via<IntersectionTableClass>**. Comme vous le voyez, ce n'est donc pas tout simple et ça nécessite pas mal de code.

Cette façon de charger les relations s'appelle le Lazy Loading ou initialisation paresseuse. On ne charge que les éléments de la table et ensuite le chargement des données dépendantes est à la charge du programmeur. Le grand avantage de cette méthode est d'être moins gourmande. On est en effet sûr de charger seulement les données dont on a besoin.

Mais alors, à quoi sert-il de changer si ça va bien ? Si vous avez beaucoup de relations entre les tables, que vous ne voulez pas passer votre temps à écrire les méthodes pour charger les données dépendantes et que les performances ne sont pas le point le plus important de votre application, il peut être très intéressant d'implémenter le Full Loading ou Agressive Loading. Le Full Loading consiste à charger de manière automatique toutes les données en relation avec ce que le programmeur demande. Vous comprenez que le nombre et la complexité des requêtes effectuées augmente sensiblement. On augmente donc la charge du serveur de base de données. Mais vous verrez que l'avantage est indéniable à l'utilisation.

Voilà ce à quoi nous voulons arriver à la fin de cet article :

On a une base de données qui contient des membres, chaque membre est d'un certain type et un type a un nom. On aimerait récupérer le nom du type d'un membre de cette manière :

```
echo $membre->type->name;
```

Admettez que c'est très simple. On va donc voir comment implémenter ceci maintenant.

2 - Le contexte

Le contexte est tout simple pour mieux comprendre. Nous avons une petite base de données permettant de gérer des membres. Ces membres ont un nom, un prénom et un type. Les différents types sont stockés dans une table à part. On a donc 2 tables :

Création des tables

```
CREATE TABLE t_types (
    id INT(11) NOT NULL AUTO_INCREMENT,
    nom VARCHAR(75) NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE t_membres (
    id INT(11) NOT NULL AUTO_INCREMENT,
    nom VARCHAR(50) NOT NULL,
    prenom VARCHAR(50) NOT NULL,
    type_id INT(11) NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (type_id) REFERENCES t_types (id) ON DELETE CASCADE ON UPDATE CASCADE
);
```

La FOREIGN KEY n'est pas indispensable dans notre exemple, mais autant faire les choses de manière correcte dès le départ. On est sûr ainsi de conserver l'intégrité référentielle des données.

On va maintenant créer nos 2 modèles de données :

Classe Membres

```
<?php
class Membres extends Zend_Db_Table {
    protected $_name = 't_membres';
}
?>
```

Classe Types

```
<?php
class Types extends Zend_Db_Table {
    protected $_name = 't_types';
}
?>
```

On va ensuite insérer une petite série d'enregistrements pour pouvoir travailler avec :

Insertion des données de test

```
INSERT INTO t_types (nom) VALUES ('Visiteur');
INSERT INTO t_types (nom) VALUES ('VIP');
INSERT INTO t_types (nom) VALUES ('Administrateur');

INSERT INTO t_membres (nom, prenom, type_id) VALUES ('Zoth', 'Hans-Ruedi', 1);
INSERT INTO t_membres (nom, prenom, type_id) VALUES ('Henroz', 'Georges', 1);
INSERT INTO t_membres (nom, prenom, type_id) VALUES ('Goeth', 'Steven', 1);
INSERT INTO t_membres (nom, prenom, type_id) VALUES ('Einstein', 'Albert', 2);
INSERT INTO t_membres (nom, prenom, type_id) VALUES ('Curie', 'Marie', 2);
INSERT INTO t_membres (nom, prenom, type_id) VALUES ('Wicht', 'Baptiste', 3);
```

Et voilà, nous sommes fin prêts à commencer notre tutoriel. Je supposerai que vous avez déjà une application PHP basée sur Zend Framework pour effectuer ces tests, je vais donc pas donner de détails sur la configuration de base à faire. Si vous n'avez jamais utilisé Zend-Framework, je vous invite à lire ce tutoriel : <http://g-rossolini.developpez.com/tutoriels/php/zend-framework/debuter/>

3 - Implémentation du Full Loading

Pour commencer, la première chose à faire est de définir les colonnes qui ont un lien avec une autre table et donc un autre modèle. Pour cela, on a 2 choix, soit on utilise le tableau **\$_referenceMap** pour récupérer les données dont on a besoin, soit on crée notre propre configuration. C'est la deuxième solution qu'on va utiliser pour ce tutoriel, on a besoin de moins de données que ce qui est présent dans le référence mapping de base. On va donc utiliser un tableau liant des colonnes et des modèles. Voilà, ce qu'on aura pour le modèle **Membres** :

```
protected $_relations = array (
    "type_id" => "Types",
);
```

On a donc un tableau associatif dont la clé est le nom de la colonne de la table et la valeur est la classe du modèle qu'on devra utiliser.

Si vous tenez vraiment à garder le tableau de base, il vous faut écrire une méthode qui va transformer le **\$_referenceMap** en un tableau utilisable par notre système.

Ensuite, pour que le Full Loading soit implémenté pour toutes les méthodes de récupération sur le modèle (**fetchAll()**, **fetchRow()**, **find()**, **□**), on va implémenter la méthode **_fetch** qui est la méthode de base utilisée ensuite par les autres méthodes pour la récupération des données. Pour ne pas avoir besoin d'implémenter cette méthode partout, on va créer une nouvelle classe abstraite **FullModel** dans laquelle on va redéfinir la méthode **_fetch**. Tous les modèles devront donc étendre **FullModel**.

Voilà comment se présente la méthode **_fetch** :

```
protected function _fetch(Zend_Db_Table_Select $select)
```

La première chose qu'on va faire est d'appeler la méthode parente pour pouvoir ensuite traiter les données :

```
$rows = parent::_fetch($select);
```

On lui repasse exactement les paramètres de la méthode, donc aucun changement de ce côté-là. Ensuite, on va initialiser les modèles dont on a besoin :

```
$models = array();

foreach ($this->relations as $columnName => $modelClass) {
    Zend_Loader::loadClass($modelClass); //Chargement du modèle
    $models[$columnName] = new $modelClass(); //Instanciation du modèle
}
```

Une fois ceci fait, on va pouvoir parcourir tous les enregistrements retournés par la méthode **_fetch** parente. Pour chacun des enregistrements, on va parcourir les modèles et si on trouve une colonne dans un enregistrement qu'on trouve aussi dans le modèle, on va remplacer la valeur de cette colonne par l'objet correspondant :

```
foreach($rows as $row) {
    foreach ($models as $column => $model) {
        if(array_key_exists($column, $row)) {
            $row[$column] = $model->find($row[$column])->current();
        }
    }
}
```

Et finalement, on va retourner les données :

```
return $rows;
```

Voilà donc ce que nous donne cette méthode pour le moment :

Méthode `_fetch`

```
protected function _fetch(Zend_Db_Table_Select $select){
    $rows = parent::_fetch($select);

    $models = array();

    foreach ($this->_relations as $columnName => $modelClass){
        Zend_Loader::loadClass($modelClass); //Chargement du modèle
        $models[$columnName] = new $modelClass(); //Instanciation du modèle
    }

    foreach($rows as &$row){
        foreach ($models as $column => $model){
            if($row[$column] != 0 && array_key_exists($column, $row)){
                $row[$column] = $model->find($row[$column])->current();
            }
        }
    }

    return $rows;
}
```

Vous verrez certainement qu'il y a certaines choses qui ne sont pas optimales avec ce code :

- On initialise le modèle même s'il a déjà été initialisé
- On initialise le modèle même si la table n'a aucune relation
- On initialise le modèle même s'il n'y a aucun enregistrement retourné

On va donc améliorer notre code. On va créer une méthode `initModels()` qui initialisera le modèle que s'il n'a pas été initialisé et on stockera les modèles en variable de classe. Ensuite, on va créer une méthode `hasRelations()` qui va tester s'il y a des relations pour ce modèle et on ne va initialiser le modèle que si on a des relations et des enregistrements. Commençons avec notre méthode `initModels()` :

Méthode `_fetch`

```
private $_models;
private $_loaded = FALSE;

private function initModels(){
    if(!$this->_loaded){
        $this->_models = array();

        foreach ($this->_relations as $columnName => $modelClass){
            Zend_Loader::loadClass($modelClass); //Chargement du modèle
            $this->_models[$columnName] = new $modelClass(); //Instanciation du modèle
        }

        $this->_loaded = TRUE;
    }
}
```

Rien de bien compliqué de ce côté-là, on vérifie avec une variable `$_loaded` si le modèle a déjà été initialisé et si ce n'est pas le cas, on le fait.

Pour la méthode `hasRelations()`, rien de compliqué non plus :

Méthode `hasRelations`

```
private function hasRelations(){
    return isset($this->_relations) && !empty($this->_relations);
}
```

Et enfin, on utilise nos nouvelles méthodes dans la méthode `_fetch`. Voici ce que donne notre classe pour le moment :

Classe FullModel

```
abstract class FullModel extends Zend_Db_Table {
    private $_models;
    private $_loaded = FALSE;

    protected function _fetch(Zend_Db_Table_Select $select){
        $rows = parent::_fetch($select);

        if($this->hasRelations() && is_array($rows) && !empty($rows)){
            $this->initModels();

            foreach($rows as &$row){
                foreach ($this->_models as $columnName => $model){
                    if(array_key_exists($columnName, $row)){
                        $row[$columnName] = $model->find($row[$columnName])->current();
                    }
                }
            }
        }

        return $rows;
    }

    private function hasRelations(){
        return isset($this->_relations) && !empty($this->_relations);
    }

    private function initModels(){
        if(!$this->_loaded){
            $this->_models = array();

            foreach ($this->_relations as $columnName => $modelClass){
                Zend_Loader::loadClass($modelClass); //Chargement du modèle
                $this->_models[$columnName] = new $modelClass(); //Instanciation du modèle
            }

            $this->_loaded = TRUE;
        }
    }
}
```

Votre classe est prête à l'emploi, vous pouvez d'ores et déjà tester ce code par exemple :

Test du Full Loading

```
Zend_Loader::loadClass("Members");
$members = new Members();

$member = $members->find(2)->current();

echo $member->type_id->name;
```

Ce code devrait vous retourner "Visiteur".

4 - Appliquer le Full Loading à createRow()

Il peut également être intéressant de *Full Loader* les enregistrements factices provenant de **createRow()**, cela permet ensuite de gérer ces enregistrements de la même manière que pour ceux qui viennent de la méthode **_fetch**. Le principe est le même que pour la méthode **_fetch**, on commence par récupérer le résultat de la méthode parente, on regarde s'il y a des relations et si c'est le cas on initialise les modèles. Ensuite, on remplace les valeurs des colonnes par la valeur du **createRow()** du modèle. Voilà ce que ça donne :

Méthode createRow avec Full Loading

```
public function createRow(array $data = array()) {
    $row = parent::createRow($data);

    if($this->hasRelations()) {
        $this->initModels();

        foreach ($this->_models as $column => $model) {
            $row->$column = $model->createRow();
        }
    }

    return $row;
}
```

Vous pourrez donc maintenant faire quelque chose dans ce genre :

test du Full Loading avec createRow

```
Zend_Loader::loadClass("Members");
$members = new Members();
$member = $members->createRow();

echo $member->type_id->name;
```

Ce qui ne vous retournera rien, mais ne vous lancera pas d'erreur.

5 - Problèmes rencontrés

Vous allez vite vous rendre compte que cette façon de faire ne marche pas complètement. Pour ce qui est de la récupération des données, cela marche bien, mais ensuite, on va rencontrer quelques problèmes.

5.1 - Impossible de sauvegarder les enregistrements full loadé

Eh oui, à peine vous essayez que vous recevrez une jolie erreur de ce genre :

Erreur PHP

```
[08-Oct-2007 13:14:20] PHP Catchable fatal error:
Object of class Zend_Db_Table_Row could not be converted to string
in C:\wamp\www\cooper\library\Zend\Db\Table\Row\Abstract.php on line 371
```

Par exemple, si vous essayez ce code :

Sauvegarde d'un objet full loadé

```
Zend_Loader::loadClass("Members");
$members = new Members();

$member = $members->find(2)->current();

$member->type_id = 2;
$member->save();
```

J'ai mis un bout de temps pour comprendre cette erreur. En fait, la ligne en question, c'est ça :

```
$diffData = array_diff_assoc($this->_data, $this->_cleanData);
```

Il contrôle les données qui ont été changées par rapport au dernier état pour ne mettre à jour que les données modifiées dans la base de données. Le problème est qu'il faut qu'il ait des chaînes de caractères pour faire la comparaison. Il faut donc "unloader" nos données pour qu'elles soient de nouveaux stockables dans la base de données. Ce n'est pas très compliqué à faire, il faut refaire la même opération qu'au début, mais dans le sens contraire. On va donc créer une méthode **unload(\$row)** qui permettra de revenir à l'état initial des données :

Méthode unload

```
public function unload($row){
    if($this->hasRelations()){
        $this->initModels();

        foreach ($this->_models as $column => $model){
            if($row->$column != null && is_object($row->$column)){
                $row->$column = $row->$column->id;
            }
        }
    }
}
```

Donc on remplace tous les objets par leur **id**. Ce code est dépendant de la base puisqu'on va chercher directement l'**id**, on pourrait l'améliorer en utilisant ceci :

```
$row->$column = $row->$column->$this->_primary;
```

Et voilà, on peut maintenant tester. Mais ça ne marche toujours pas et on a toujours la même erreur, mais c'est normal, car il compare aussi les anciennes données (**\$_cleanData**). Il faut donc aussi unloader ces données, mais comment faire ? Sur ce coup-là, comme ces données ne sont pas accessibles, on va créer un nouveau type de **Row** (enregistrement) dans lequel on va ajouter une méthode nous permettant de rafraîchir ces données :

Classe FullRow

```
<?php
class FullRow extends Zend_Db_Table_Row_Abstract {
    public function refreshCleanData () {
        $this->_cleanData = $this->_data;
    }
}
?>
```

On a juste créé une méthode publique pour modifier la valeur de champs qui ne sont normalement pas accessibles. Maintenant, il faut encore indiquer au modèle qu'il doit utiliser cette classe plutôt que celle de base. Pour cela, rien de plus simple, il suffit d'appeler la méthode **setRowClass()**, on va le faire dans le constructeur de FullModel :

```
function __construct($config = array()) {
    parent::__construct($config);

    parent::setRowClass("FullRow");
}
```

Voilà, maintenant notre modèle va utiliser **FullRow** pour retourner des enregistrements. Ensuite, il nous faudra appeler notre nouvelle méthode dans la méthode **unload** :

Méthode unload

```
public function unload($row) {
    if ($this->hasRelations()) {
        $this->initModels();

        foreach ($this->_models as $column => $model) {
            if ($row->$column != null && is_object($row->$column)) {
                $row->$column = $row->$column->id;
            }
        }

        $row->refreshCleanData();
    }
}
```

Et on va maintenant utiliser cette méthode sur l'objet qu'on va modifier :

Utilisation de la méthode unload

```
$members = new Members();

$member = $members->find(2)->current();
$members->unload($member);

$member->type_id = 2;
$member->save();
```

Et cette fois, ça marche. Par contre, faites attention à faire l'unload avant les modifications, sinon, comme les données temporaires seront égales à celles modifiées, rien ne va être changé dans la base et vos changements seront perdus. Alors, bien sûr, c'est un peu lourd comme manière de faire, mais vous pouvez aussi créer une méthode dans votre modèle qui vous renvoie un enregistrement sans le Full Loading ou alors créer votre propre type de données et appeler directement la méthode **unload()** dans la méthode **save()**. Ou mieux encore, vous pouvez surcharger l'ORM pour qu'il puisse supporter complètement en interne le Full Loading, mais cela implique beaucoup de modifications.

Une deuxième chose à laquelle il faut faire attention est qu'il ne faut pas faire le **refreshCleanData()** sur un objet provenant de la méthode **createRow()**. Pourquoi ? Parce que le framework va vérifier s'il y a déjà des données dans **\$_cleanData** et si c'est le cas va considérer que l'enregistrement est déjà stocké dans la base. Dans ce cas, il ne s'agit plus d'un insert mais d'un update et il ne va pas trouver l'enregistrement dans la base. Mais il faut tout de même l'unloader, on va donc créer une méthode pour unloader des enregistrements non stockés. En fait, comme cette méthode sera pratiquement la même que celle pour les enregistrements normaux, on va factoriser le code des deux méthodes :

Refactorisation des méthodes d'unload

```

public function unload($row) {
    if($this->hasRelations()) {
        $this->_unload($row);

        $row->refreshCleanData();
    }
}

public function unloadCreatedRow($row) {
    if($this->hasRelations()) {
        $this->_unload($row);
    }
}

private function _unload($row) {
    $this->initModels();

    foreach ($this->_models as $column => $model) {
        if($row->$column != null && is_object($row->$column)) {
            $row->$column = $row->$column->id;
        }
    }
}
    
```

Cette fois, nous sommes bons. On peut sauvegarder sans problème nos données.

5.2 - Problème avec les relations réciproques entre 2 objets d'une même table

Qu'est-ce que j'entends par là. Je parle des relations entre des objets de même table. Par exemple, un type est un sous-type d'un autre type. La relation est donc sur la même base de données. Où est le problème ? En fait, il ne se pose que si vous avez la possibilité d'avoir un type qui n'est sous type de rien. Dans ce cas, comment va se comporter notre modèle ? Comme il ne trouve pas d'enregistrement dans la base correspondant à "rien", il va appeler la méthode **createRow()**, celle-ci va faire du Full Loading et va donc appeler à nouveau la méthode **createRow()** pour le champ qui définit le sous-type et ainsi de suite. On a donc une récursion infinie.

Comment parer à ce problème ? Il y a plusieurs solutions, soit on arrête de "Full Loader" la méthode **createRow()**, soit on décide de ne pas Full Loader une colonne si le modèle est de même type que celui dans lequel on est, soit on décide dans la base de données qu'il n'est pas possible qu'un type ne soit sous-type de rien, mais ce n'est pas toujours possible ou alors on peut aussi définir un niveau de récursion, c'est à dire à partir de quand on arrête de descendre dans le Full Loading.

On va implémenter la méthode 2 consistant à dire que quand la colonne fait référence à un objet de même type, on ne va pas loader l'objet correspondant. Il nous faut donc la classe du modèle. Ainsi, nous allons utiliser la méthode **get_class** sur l'objet courant (**\$this**) et on va ensuite pouvoir vérifier cela dans la méthode **createRow()** :

Nouvelle méthode createRow

```

public function createRow(array $data = array()) {
    $row = parent::createRow($data);

    if($this->hasRelations()) {
        $this->initModels();

        foreach ($this->_models as $column => $model) {
            if(!$model instanceof get_class($this)) {
                $row->$column = $model->createRow();
            }
        }
    }

    return $row;
}
    
```

Et voilà, ce problème est résolu. Par contre, ce problème se pose aussi dans le cas de dépendances cycliques, mais cela traduit une mauvaise conception de la base de données. Mais si vous voulez empêcher à tout prix une erreur même en cas de dépendances cycliques, vous serez obligé d'implémenter un niveau de récursion pour éviter que le programme aille trop loin.

6 - Conclusion

Et voilà, on arrive à la fin de notre implémentation. Cette méthode vous permet donc de charger toutes les instances des objets liées à celui que vous voulez. Néanmoins, cette méthode n'est pas encore optimale. Pour plusieurs raisons :

- On résout n'importe quel niveau de relation, ce qui peut se révéler très lourd avec une base de données ayant beaucoup de relations. Par exemple, est-ce qu'il vous est utile de connaître de quel type est le groupe de la personne qui a validé l'article sur lequel un commentaire a été posté. On pourrait limiter cela avec un niveau de récursion.
- On fait beaucoup de requêtes. En effet, on fait une requête pour chacune des relations de chacun des objets retournés par une méthode `_fetch`. Par exemple, si un type d'objets a 3 relations et qu'on fait une requête qui nous en retourne 100, on va faire 300 requêtes supplémentaires pour les compléter, ce qui n'est pas terrible. Il faudrait manipuler la logique SQL pour utiliser des `join()`. Mais cela voudrait dire qu'il faudrait qu'on recode la logique de récupération des données dans la base.
- On ne résout que les relations 1 à 1 et 1 à plusieurs et cette dernière que dans un seul sens. Par exemple, on peut savoir par quel membre a été écrit un commentaire, mais pas quels commentaires ont été écrits par tel membre. On ne résout pas non plus les relations plusieurs à plusieurs, c'est-à-dire qu'on peut récupérer quels auteurs ont écrit tel livre et quels livres ont été écrits par tel auteur. Il faudrait changer notre description des relations pour faire cela et changer notre logique de loading pour réussir à passer par des tables intermédiaires.

J'espère que malgré ces lacunes, ce mode Full Loading sur le Zend Framework et cet article vous auront été utiles.

Pour d'autres informations sur Zend Framework, je vous invite à consulter [notre rubrique](#) sur Developpez.com.

Un grand merci à [fabszn](#) ainsi qu'à [Yogui](#) pour leurs corrections.

6.1 - Code complet

```
FullModel
<?php
abstract class FullModel extends Zend_Db_Table {
    private $_models;
    private $_loaded = FALSE;

    function __construct($config = array()) {
        parent::__construct($config);

        parent::setRowClass("FullRow");
    }

    protected function _fetch(Zend_Db_Table_Select $select) {
        $rows = parent::_fetch($select);

        if($this->hasRelations() && is_array($rows) && !empty($rows)) {
            $this->initModels();

            foreach($rows as &$row) {
                foreach ($this->_models as $column => $model) {
                    if(array_key_exists($column, $row)) {
                        $row[$column] = $model->find($row[$column])->current();
                    }
                }
            }
        }

        return $rows;
    }

    public function createRow(array $data = array()) {
        $row = parent::createRow($data);
    }
}
```

FullModel

```

if($this->hasRelations()){
    $this->initModels();

    foreach ($this->_models as $column => $model){
        if(!($model instanceof get_class($this))){
            $row->$column = $model->createRow();
        }
    }
}

return $row;
}

private function hasRelations(){
    return isset($this->_relations) && !empty($this->_relations);
}

private function initModels(){
    if(!$this->_loaded){
        $this->_models = array();

        foreach ($this->_relations as $columnName => $modelClass){
            Zend_Loader::loadClass($modelClass); //Chargement du modèle
            $this->_models[$columnName] = new $modelClass(); //Instanciation du modèle
        }

        $this->_loaded = TRUE;
    }
}

public function unload($row){
    if($this->hasRelations()){
        $this->_unload($row);

        $row->refreshCleanData();
    }
}

public function unloadCreatedRow($row){
    if($this->hasRelations()){
        $this->_unload($row);
    }
}

private function _unload($row){
    $this->initModels();

    foreach ($this->_models as $column => $model){
        if($row->$column != null && is_object($row->$column)){
            $row->$column = $row->$column->id;
        }
    }
}
?>

```

FullRow

```

<?php
class FullRow extends Zend_Db_Table_Row_Abstract{
    public function refreshCleanData(){
        $this->_cleanData = $this->_data;
    }
}
?>

```

Définition d'un modèle

```

<?php

```

Définition d'un modèle

```
class Name extends FullModel {
    protected $_name = 'table_name';

    protected $_relations = array (
        "column" => "model",
    );
}
?>
```