Utilisation d'un ORM : Les bases de Doctrine

Mis à jour le mardi 8 janvier 2013

Difficulté : **Moyenne**

Introduction du cours

Bienvenue dans ce tutoriel qui a pour vocation de vous apprendre à utiliser l'ORM Doctrine. Qu'est-ce qu'un ORM ? À quoi sert-il ? Comment l'utilise-t-on ? Tant de questions auxquelles ce tutoriel répond.

Notions prérequises

- Bases du PHP.
- Bases de la POO.
- Bases du SQL.

Un ORM : Doctrine

Définition d'un ORM

Un ORM est une classe (ou bien plus souvent un ensemble de classes) visant à ce que l'utilisateur puisse manipuler ses tables de données comme si c'étaient des objets.
Voici un petit code pour vous mettre l'eau à la bouche :

```php
<?php
$maNews = new News();

// On définit les caractéristiques de la news.
$maNews->titre = 'La première news du site';
$maNews->auteur = 'christophetd';
$maNews->contenu = 'Bienvenue sur mon site, j\'espère qu\'il vous plaira !';

// Puis, on sauvegarde le tout dans la base de données.
$maNews->save();
?>
```

Comme vous le voyez, on considère les champs d'une table (ici **news**) comme de simples attributs (ici **titre, auteur** et **contenu**).
Ensuite, c'est l'ORM qui se chargera de la communication avec la base de données, c'est lui qui fait le sale boulot.

Doctrine

Présentation

Doctrine est, comme vous devez maintenant vous douter, l'un des ORM les plus connus qui existent actuellement.
Il est utilisé dans des *frameworks* très connus (symfony, Zend Framework), et est aussi simple à prendre en main que puissant.
Dans ce tutoriel, seules les bases seront présentées.

Téléchargement

Pour commencer, rendez-vous sur le site http://www.doctrine-project.org, puis dans la section « *Download* ».
Prenez la version marquée comme stable, puis lancez le téléchargement en cliquant sur **Doctrine-X.X.X.tgz**.

Pour décompresser cette archive, il vous faudra un utilitaire comme **7-Zip**.
Une fois cela fait, vous devriez vous retrouver avec un fichier **Doctrine.php**, ainsi qu'un dossier **Doctrine**.
Déplacez les deux dans le dossier **lib** de votre dossier Web (**C:\wamp\www\Tests\doctrine\lib** dans mon cas), puis créez un fichier **index.php** à la racine.

Ce sera tout pour l'installation, maintenant nous allons commencer les choses sérieuses. 

Création des modèles et de la table

Création des modèles

Pour fonctionner, Doctrine a besoin que vous lui indiquiez la structure de votre ou de vos tables.
Pour cela, nous allons créer une classe qui hérite de la classe Doctrine_Record, portant le même nom que la table.
Notez que par convention, ce nom doit être au singulier.

```php
<?php
// Nous allons travailler sur un module de news, donc la classe s'appellera News.
class News extends Doctrine_Record
{
}
```

Ensuite, nous allons indiquer à Doctrine le nom de la table, ainsi que les différents champs, les types et spécificités que contiendra la table.
Pour ce faire, nous allons nous servir de deux méthodes de Doctrine_Record, héritées par notre classe :

- setTableName, qui prend en paramètre le nom que nous voulons donner à notre table ;
- hasColumn, qui prend trois (ou quatre) paramètres : le nom du champ, son type, sa valeur, et optionnellement, des paramètres supplémentaires (auto-incrémentation, valeur par défaut, clé primaire, etc.).

Voici ce que cela donne dans notre cas :

```php
<?php
class News extends Doctrine_Record
{
    public function setTableDefinition()
    {
        // On définit le nom de notre table : « news ».
        $this->setTableName('news');

    //Puis, tous les champs
        $this->hasColumn('id', 'integer', 8, array('primary' => true,
                         'autoincrement' => true));
        $this->hasColumn('titre', 'string', 100);
    $this->hasColumn('auteur', 'string', 100);
    $this->hasColumn('contenu', 'string', 4000);
    }
}
?>
```

C'est une chose de faite.
Enregistrez cette classe dans le fichier *modeles/News.class.php*, puis créez *modeles/NewsTable.class.php*.
Nous allons mettre à l'intérieur, une classe "*NewsTable*" héritant de Doctrine_Table que nous remplirons plus tard. Elle nous permettra de définir certaines méthodes dont nous nous servirons plus bas.

```php
<?php
// On inclut le modèle «News ».
require_once dirname(__FILE__).'/modeles/News.class.php';

class NewsTable extends Doctrine_Table
{

}
?>
```

Création de la table

Avant de pouvoir commencer à manipuler réellement Doctrine et notre base de données, il nous va falloir établir une connexion à celle-ci.
Pour commencer, nous allons inclure le fichier **News.class.php** et **Doctrine.php**, la base de Doctrine, puis « enregistrer » sa fonction d'inclusion automatique de classes auprès de PHP :

```php
<?php
require_once 'lib/Doctrine.php';
spl_autoload_register(array('Doctrine_Core', 'autoload'));
require_once 'modeles/News.class.php';
```

La fonction spl_autoload_register() dit à PHP : « *Si je fais appel à une classe qui n'existe pas, avant de générer une erreur, appelle la méthode **autoload** de la classe **Doctrine** pour qu'elle tente de l'inclure.* »

Bien, maintenant, nous allons appeler la méthode connection de Doctrine, qui prend en paramètre une chaîne formatée selon la syntaxe :

mysql://utilisateur:motdepasse@serveur/base_de_donnees

```php
<?php
// Si l'utilisateur ne possède pas de mot de passe, il faut faire directement « utilisateur@serveur ».
$dsn = 'mysql://root@localhost/developpement';
$connexion = Doctrine_Manager::connection($dsn);
```

Pour finir, nous allons pouvoir générer notre table, comme ceci :

```php
<?php
try {
    $table = Doctrine_Core::getTable('News'); // On récupère l'objet de la table.
    $connexion->export->createTable($table->getTableName(),
                                    $table->getColumns()); // Puis, on la crée.
        echo 'La table a bien été créée';
} catch(Doctrine_Connection_Exception $e) { // Si une exception est lancée.
    echo $e->getMessage(); // On l'affiche.
}
```

Si vous voyez le message de confirmation s'afficher, c'est que tout s'est bien passé, et que la table « News » a bien été créée (vous pouvez aller vérifier via phpmyadmin). 
Sinon, c'est qu'il y a une erreur quelque part, essayez de comprendre le message, et si vous ne trouvez toujours pas la cause de l'erreur, postez sur le forum. 

Effectuer des requêtes avec Doctrine

Bien, maintenant que tout est prêt, on va pouvoir commencer à voir comment s'exécute une requête.
Tout d'abord, videz votre *index.php*, et ne gardez que ce code :

```php
<?php
require_once 'lib/Doctrine.php';
spl_autoload_register(array('Doctrine', 'autoload'));
require_once 'modeles/News.class.php';

// Adaptez cela selon vos besoins bien entendu.
$dsn = 'mysql://root@localhost/developpement';
$connexion = Doctrine_Manager::connection($dsn);
```

Pour commencer, on va créer une instance de notre classe News :

```php
<?php
$news = new News();
```

Vous vous souvenez de ce que je vous ai dit tout à l'heure ? On va pouvoir traiter cet objet comme si c'était notre table ! 
Essayons donc…

```php
<?php
$news->titre = 'Doctrine';
$news->auteur = 'Georges Clooney';
$news->contenu = 'Doctrine, what else ?';
```

Tu m'as bien eu, ça ne fait rien ce code ! 

Eh oui, c'est normal, on n'a pas dit à Doctrine d'appliquer les modifications dans la base de données. Pour cela, il faut utiliser la méthode save.

```php
<?php
$news->save();
```

Et là, vos yeux ébahis, rougis par les larmes que vous versez tant vous avez attendu ce jour (c'est bon, j'en fais assez là ?), vous voyez s'insérer comme par magie dans votre table une ligne correspondant aux valeurs que vous avez indiquées.

Récupérer les données de la table

J'imagine que vous aimeriez bien voir comment l'on récupère les données d'une table.

Pour que l'on ait quelques enregistrements sur lesquels travailler, exécutez ce code depuis PhpMyAdmin (onglet *SQL*) :

```sql
INSERT INTO `news` (`id`, `titre`, `auteur`, `contenu`) VALUES (NULL, 'La premiere news du site', 'christophetd', 'Bonjour,
Bienvenue sur mon site, j''espere que vous l''aimerez ...
@+'), (NULL, 'Fermeture temporaire du site', 'vyk12', 'Le site fermera ce mercredi 16 decembre de 10h00 a 20h00, le temps d''une grosse maintenance du code (passage a Doctrine =D).');
```

Pour récupérer tous les enregistrements contenus dans cette table, nous allons manipuler l'objet Doctrine_Query que nous renvoie la méthode `Doctrine_Query::create()` :

```php
<?php
$requete = Doctrine_Query::create();
// Spécifications de la requête
```

Pour commencer, nous allons utiliser la méthode from, puis exécuter la requête à l'aide de execute() :

```php
<?php
$requete = Doctrine_Query::create() // On crée une requête.
            ->from('news') // On veut les enregistrements de la table news.
            ->execute(); // On exécute la requête.
?>
```

C'est bien beau tout ça, mais si je veux parcourir les résultats, comment je peux faire ?

En fait, l'objet renvoyé implémente une interface qui fait qu'on peut le parcourir comme si c'était un *array* !
Nous allons donc faire cela à l'aide d'un *foreach*, puis accéder à la colonne que nous voulons :

```php
<?php
// On parcourt les résultats de la requête.
foreach($requete as $news) {
        // On affiche le titre de chaque news.
    echo $news->titre.'<br />';
}
```

Pour accéder à une colonne, nous pouvons aussi bien faire **$requete->colonne** que **$requete['colonne']**. 

Maintenant que vous savez faire une requête basique, vous pouvez y apporter des spécifications.
Pour faire simple, vous pouvez généralement (quasiment souvent) définir le nom d'une clause via la méthode qui porte son nom.
Exemples :

- **where(condition)** pour la clause `WHERE condition` ;
- **orderBy(champ)** pour la clause `ORDER BY champ` ;
- **groupBy(champ)** pour la clause `GROUP BY champ` ;
- **leftJoin** pour la clause `LEFT JOIN` . Raté.  C'est une exception que nous verrons dans une autre partie. 

Pour être sûr que vous avez bien compris le principe, voici une requête qui en regroupe cinq ou six :

```php
<?php
$requete = Doctrine_Query::create()
            ->select('titre, auteur')
            ->from('news n')
            ->orderBy('titre')
            ->groupBy('auteur')
            ->limit(2);
// N'oubliez pas de rajouter le execute() si vous voulez utiliser la requête par la suite.
```

L'ensemble des méthodes de Doctrine utilisées pour construire des requêtes est appelé le DQL.

Créer et utiliser des fonctions de récupération à partir du modèle

Vous vous souvenez de la classe **NewsTable** que nous avions créée ? Elle hérite de la classe **Doctrine_Table**.
Il se trouve que cette dernière hérite elle-même de deux méthodes auxquelles nous allons nous intéresser.

findAll()

Étant donné que cette méthode appartient à la classe **Doctrine_Record**, nous allons devoir utiliser la méthode de Doctrine_CoregetTable, qui prend en paramètre le nom de notre table, et qui renvoie un objet de type News.
Elle permet de récupérer tous les enregistrements d'une table.
Par exemple, pour récupérer toutes nos *news*, nous pourrions faire :

```php
<?php
$requete = Doctrine_Core::getTable('News')->findAll();

foreach($requete as $news)
{
    echo $news->titre.', par <strong>'.$news->auteur.'</strong><br />';
}
```

Vous avouerez que c'est plus pratique de faire comme ça que de devoir passer par un `Doctrine_Query::create()...` 

find

Cette méthode prend en paramètre l'identifiant de l'enregistrement à récupérer.

```php
<?php
// On veut la news n° 1.
$news = Doctrine_Core::getTable('News')->find(1);

// Notre objet ne contient forcément qu'un seul enregistrement, on peut l'afficher sans avoir à faire de foreach.
echo $news->titre.', par <strong>'.$news->auteur.'</strong>';
```

Grâce à cela, on peut facilement éditer une *news* :

```php
<?php
// On veut éditer la news numéro 1.
$requete = Doctrine_Core::getTable('News')->find(1);

// On modifie ses attributs.
$requete->titre = 'Mon nouveau titre';

// Puis on sauvegarde.
$requete->save();
```

Utiliser des méthodes de récupération à partir du modèle

Vous ne trouvez pas ça lourd de devoir créer une requête longue et fastidieuse à chaque fois ? Que diriez-vous s'il était possible de créer une méthode recupererDernieresNews($nombreDeNews) qui nous permettrait de récupérer les $nombreDeNews dernières *news* ?
Il se trouve que c'est tout à fait faisable. 

Tout d'abord, rendons-nous dans notre classe NewsTable qui est pour l'instant vierge. Nous allons y créer une méthode recupererNews qui prend en paramètre un nombre entier (le nombre de *news* à récupérer). Les *news* doivent être ordonnées par ordre d'identifiant décroissant.
Essayez de faire ça pour vous-même, c'est un bon entraînement. 

Solution :

```php
<?php
class NewsTable extends Doctrine_Table
{
    public function recupererNews($nombreNews) {
        $q = Doctrine_Query::create()
                ->from('news')
                ->orderBy('id DESC')
                ->limit((int) $nombreNews);
        return $q;

    }
}
```

Maintenant, dans notre index.php, nous pouvons faire :

```php
<?php
$news = Doctrine_Core::getTable('News');
foreach($news->recupererNews(2)->execute() as $news) {
    echo $news->titre.'<br />';
}
```

Détaillons un peu la troisième ligne : `<?php $news->recupererNews(2)->execute() ?>` .
Tout d'abord, on appelle la méthode recupererNews de l'objet $news. Cette méthode renvoyant un objet Doctrine_Query, nous pouvons ensuite exécuter sa méthode execute.

Mais à quoi ça sert de se compliquer la vie à créer une méthode de notre classe NewsTable ? Pourquoi ne pas faire directement la requête dans notre index.php ?

Ça sert surtout si vous utilisez le modèle MVC.
En effet, le contrôleur, qui inclut les modèles, appelle des fonctions et plein d'autres choses indispensables au fonctionnement d'une application, **ne doit pas contenir de requêtes** : il doit seulement faire appel à des fonctions du modèle qui s'en chargeront.

Vous voyez où je veux en venir ? Si vous avez pour projet d'utiliser le modèle MVC, il vous faudra obligatoirement user de cette fonctionnalité.

Lier deux tables

Maintenant que vous avez les bases, nous pouvons nous attaquer à quelque chose un chouïa plus compliqué.
Tout d'abord, nous allons utiliser une nouvelle table **commentaires** qui comporte trois champs : **id, id_news** et **contenu**.

Configuration des modèles

Créez un modèle « *Commentaire* » dans **modeles/** :

```php
<?php
// Souvenez-vous, le nom du modèle reste au singulier.
class Commentaire extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->setTableName('commentaires');

        $this->hasColumn('id', 'integer', 8, array('primary' => true,
                                'autoincrement' => true));
    $this->hasColumn('id_news', 'integer', 8);
        $this->hasColumn('contenu', 'string', 4000);
    }
}
```

Jusque-là rien de nouveau.
Maintenant, voici ce que nous voulons faire : lier les tables **news** et **commentaires** afin de pouvoir effectuer une jointure entre les deux.
Pour ce faire, nous aurons besoin d'ajouter une méthode setUp dans nos deux modèles qui contiendront ce code :

```php
<?php
    public function setUp() {
        $this->[type de la relation](
            'NomDeLaTableARelier [as Alias]',
            array(
                'local' => 'clé locale',
                'foreign' => 'clé étrangère'
            )
        );
    }
```

Détaillons ce code.

- **Type de la relation** : une relation peut être de deux types.
  - *hasOne* : cette relation est valable quand l'objet (= la table) ne peut avoir qu'une seule relation avec l'objet avec lequel il est lié.
    Exemple : un commentaire ne peut être lié **qu'à une seule***news*.
  - *hasMany* : cette relation est valable quand l'objet peut avoir plusieurs relations avec l'objet auquel il est lié.
    Exemple : une *news* peut être liée **à plusieurs** commentaires.
- **Alias** : c'est en quelque sorte un deuxième nom que vous pourrez par la suite utiliser (voir rappel plus bas). **Le « *as Alias* » est facultatif**.
- **Clé locale :** c'est le champ appartenant à l'objet qui le relie à celui auquel il est lié. Pour la table **news**, ce sera **id**. Pour la table **commentaires**, ce sera **id_news**.
- **Clé étrangère** : c'est le champ n'appartenant pas à l'objet qui le relie à celui auquel il est lié. Pour la table **news**, ce sera **id** (de l'objet commentaires). Pour la table **commentaires**, ce sera **id_news** (de la table news).

Rappel

Si vous vous êtes endormis pendant quelques secondes en lisant votre cours de SQL, il se peut que vous ayez sauté la notion de création d'alias.
Un alias se crée à la suite du nom de la table, dans la clause `FROM` . C'est un nom généralement plus court que celui de la table (les alias ont été principalement créés pour ça) :

```sql
SELECT titre FROM news n
```

On peut ensuite mettre le préfixe **<aliasTable>.champ** dans le `SELECT` pour indiquer que le champ **champ** appartient à la table **table** :

```sql
SELECT n.titre FROM news n
```

Maintenant, voici ce que devrait être la méthode setUp pour la table news :

```php
<?php
public function setUp() {
        $this->hasMany(
            'Commentaire as commentaires',
            array(
                'local' => 'id',
                'foreign' => 'id_news'
            )
        );
    }
```

Exercice

Codez la fonction setUp de la table commentaires. 

Solution :

```php
<?php
public function setUp() {
        $this->hasOne(
            'News as news',
            array(
                'local' => 'id_news',
                'foreign' => 'id'
            )
        );
    }
```

Notez que nous ne sommes pas obligés de créer un alias dans ce cas-ci, car notre jointure ira de la table news vers la table commentaires.

Utilisation

Nous allons commencer par créer la table commentaires, ça pourrait nous être utile. 

```php
<?php
// Les inclusions et la connexion
try {
    $table = Doctrine_Core::getTable('Commentaire'); // On récupère l'objet de la table.
    $doctrine->export->createTable($table->getTableName(),
                                   $table->getColumns()); // Puis on la crée.
        echo 'La table a bien été créée';
} catch(Doctrine_Connection_Exception $e) { // Si une exception est lancée.
    echo $e->getMessage(); // On l'affiche.
}
```

Importez quelques commentaires dans votre table depuis phpMyAdmin (j'en profite pour enlever toutes les *news* et n'en laisser que deux) :

```sql
INSERT INTO `commentaires` (`id`, `id_news`, `contenu`) VALUES
(1, 1, 'Cool cette ouverture. :)'),
(2, 2, 'Quel dommage !'),
(3, 1, 'Vraiment bien ca :)');
DELETE FROM news;
INSERT INTO `news` (`id`, `titre`, `auteur`, `contenu`) VALUES
(1, 'Ouverture du site', 'christophetd', 'Le site ouvre. :)'),
(2, 'Fermeture du site', 'vyk12', 'Le site ferme. :(');
```

Ensuite, nous allons faire une requête pour récupérer le titre de chacune des *news* et les commentaires qui lui sont associés :

```html+php
<?php
$liste_news = Doctrine_Query::create() // Création de la requête
        ->select('n.titre, c.contenu') // On sélectionne le titre de la news et les commentaires associés.
        ->from('news n')
        ->leftJoin('n.commentaires c') // On joint les deux tables.
        ->execute(); // Et enfin, on exécute la requête.
```

Pourquoi met-on « *n.commentaires c* » dans le leftJoin ?

Décomposons :
n.commentairesc

- n. est l'alias de la table news que nous avons défini dans ->from('news **n**'). On aurait tout aussi bien pu mettre "*news*". Ce préfixe sert à indiquer la table **parente**, en l'occurrence la table news ;
- commentaires est l'alias que nous avions tout à l'heure donné à la table commentaires, dans le modèle de la classe News :

```html+php
<?php
$this->hasMany(
            'Commentaires as commentaires' /* <-- */,
                ...
              );
```

- c est l'alias que nous donnons dans la requête à la table commentaires. Nous nous en sommes servis dans la clause select : `->select('..., c.contenu') ?>` pour sélectionner le champ **contenu** de la table **commentaires**, alias **c**.

Maintenant, parcourons la requête pour afficher le titre des *news* :

```html+php
<?php
foreach($liste_news as $news) {
    echo $news->titre.'<br />';
}
```

Comment va-t-on parcourir les commentaires ? Il faut faire un echo de $news->contenu ?

Non. La liste de tous les commentaires est contenue dans $news->commentaires !
Nous n'avons donc qu'à parcourir cette liste, puis à afficher les commentaires un à un.

```php
<?php
foreach($liste_news as $news) {
    echo $news->titre.'<br />';
    echo '<p>Commentaires sur cette news :';
    echo '<ul>';
    foreach($news->commentaires as $commentaire) {
        echo '<li>'.$commentaire->contenu.'</li>';
    }
    echo '</ul></p><hr />';
}
```

Résultat :

Citation : Résultat

Ouverture du site

Commentaires sur cette *news* :

- Cool cette ouverture.
- Vraiment bien ça.

Fermeture du site

Commentaires sur cette *news* :

- Quel dommage...

Voilà, ce tutoriel se termine.
Il n'est cependant pas fini, il est possible (et même fort probable) que j'ajoute dans quelque temps une ou deux parties sur d'autres utilisations plus avancées de Doctrine.

Je vous laisse un lien vers la documentation officielle de Doctrine (en anglais *of course*).

Un grand merci à vyk12 pour ses corrections, suggestions et le suivi du tutoriel !