

Apprendre à utiliser Doctrine

Par Cybermanu



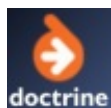
www.openclassrooms.com

*Licence Creative Commons 6 2.0
Dernière mise à jour le 15/02/2011*

Sommaire

Sommaire	2
Partager	2
Apprendre à utiliser Doctrine	4
Connaissances requises	4
Remerciements	4
Partie 1 : Débuter avec Doctrine	5
Conventions utilisées	5
Présentation et installation	5
Doctrine, c'est quoi ?	5
Installation	6
Téléchargement	6
Implémentation dans notre projet	7
Configuration de la base de données	9
Connexion	9
La classe Doctrine_Manager	9
Connexion à la base de données	9
Schéma de données (Yaml)	9
Le langage Yaml	9
Le schéma	10
Schéma de données (PHP)	11
Les classes de modèle	11
Les classes représentant les tables	12
Génération des tables et modèles	13
Manipulation des données stockées dans la base	15
Utilisation des objets et collections	15
Utilisation des objets	15
Utilisation des collections d'objets	17
Le langage DQL	18
Sélection simple de données	18
SELECT : Doctrine_Query en détails	19
UPDATE et DELETE	21
Utilité des classes de tables	22
TP : Création d'un CRUD	25
Préparation du terrain	25
Récapitulatif	25
Le contrôleur	25
Données de test	26
Affichage de données	26
Création de la requête	27
Affichage des données récupérées	27
Création d'objets	28
Le formulaire	28
Traitement des informations	28
Modification d'objets	29
Le formulaire	29
Traitement des informations	29
Refactorisation du code	30
Suppression de données	31
Traitement	31
Partie 2 : Utilisation avancée de Doctrine	33
Les Events Listeners	33
Introduction aux Events Listeners	33
Déclenchement d'évènements	33
Intercepter un évènement	33
Les méthodes utilisées	33
Doctrine_Event	34
Listener de connexion	35
Créer un listener	35
Attacher un listener	36
Listeners d'objets	37
Créer un listener	37
Attacher un listener	37
Les hooks	38
Intercepter les requêtes	38
Informations supplémentaires	39
Interrompre l'action en cours	39
Ne pas exécuter le listener suivant	40
Les templates	41
Introduction	41
Qu'est-ce qu'un template ?	41
Et Doctrine dans tout ça ?	41
Utilisation des templates	42
Découverte des behaviors	42

Timestampable	43
SoftDelete	45
Versionable	46
Sluggable	47
Structure d'un template	47
Comment ça marche ?	47
Méthodes déléguées	48
TP : Création d'un template	51
Créer son propre template	51
Introduction	51
Création de la classe	51
Détails supplémentaires	51
Ajout d'un listener	52
Correction : le template	52
Les options	52
Les colonnes	53
Les relations	54
Correction : le listener	55
Pré-insertion d'un article	56
Mise à jour d'un article	57
Sélection d'articles	58
Conclusion	59
Partie 3 : Annexes	63
Personnalisation	64
Avec Doctrine_Manager	64
Comment ça marche ?	64
Détail des principaux attributs	64



Apprendre à utiliser Doctrine

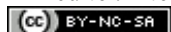


Par

Cybermanu

Mise à jour : 15/02/2011

Difficulté : Intermédiaire



Bienvenue à tous dans ce tutoriel. 😊

Je vais vous présenter ici un ORM pour PHP : Doctrine.

Si vous ne savez pas ce qu'est un ORM, ça tombe bien, je vous explique tout cela juste après. 😊

Connaissances requises

Pour suivre ce tuto, il est nécessaire :

- de bien connaître PHP et la POO ;
- de savoir comment fonctionnent les relations dans une base de données (au moins basiquement) ;
- d'avoir un serveur avec PHP 5.2.3 minimum ;
- et... d'avoir du temps libre !

En effet, Doctrine est vraiment TRÈS complet, et on ne pourra de toute façon pas tout voir dans ce tutoriel.

Si vous ne connaissez pas la POO, arrêtez-vous là ! 😊 Je vous conseille d'aller faire un tour sur [le tuto de vyk12](#). Revenez juste après, vous verrez, ça vaut le coup. 😊

Je vous recommande également de savoir, en gros, ce qu'est l'architecture MVC. Si vous ne la connaissez pas, ce n'est pas dramatique, vous comprendrez comment on organise notre projet au fur et à mesure du tutoriel.

Remerciements

Avant de commencer, je souhaite remercier les personnes qui ont contribué à ce tuto, m'ont donné des idées, des conseils, notamment [Nami Doc](#). 😊



La version que j'utiliserai et expliquerai dans ce tutoriel est la version 1.2. Il est possible que je mette à jour le tutoriel lorsque la version 2 sortira en version stable.

Partie 1 : Débuter avec Doctrine

Dans cette première partie, on va découvrir ce qu'est Doctrine, et commencer à l'utiliser tout en douceur. 😊

Conventions utilisées

Dans les exemples que je donnerai tout au long du tutoriel, j'emploierai souvent certaines variables « spéciales » :

- **\$manager** : il représente une instance de *Doctrine_Manager* et peut être récupéré avec *Doctrine_Manager::getInstance()* ;
- **\$connexion**, **\$connection** ou **\$conn** : il s'agit d'une instance de *Doctrine_Connection*, qui peut être obtenue avec *Doctrine_Manager::connection()* ;
- **\$record** : c'est un objet héritant de *Doctrine_Record*, faisant partie du modèle ;
- **\$table** : il s'agit d'une instance d'une classe fille de *Doctrine_Table*, représentant l'une des tables de la base de données. Elle peut être récupérée avec *Doctrine_Core::getTable()* ou *\$record->getTable()*.

Rassurez-vous : vous ne connaissez pas du tout ces classes pour l'instant et c'est normal, nous les étudierons dans le tutoriel.



📄 Présentation et installation

Tout d'abord, nous allons voir ce qu'est Doctrine, puis nous l'installerons, pour pouvoir commencer à coder un peu, quand même. 😊

Attention, je vous préviens, je ne serai pas aussi direct que dans [le tuto de christophetd](#). Je passerai plus de temps à vous faire découvrir le fonctionnement interne de Doctrine.

Si vous voulez commencer à coder tout de suite, allez voir le sien, il est relativement bien fait pour commencer rapidement.

Si vous souhaitez en savoir un peu plus, et apprendre à bien utiliser Doctrine et pouvoir l'étendre et le personnaliser ensuite (*behaviors*, et tout ça ! 😊), restez ici. 😊

Doctrine, c'est quoi ?

Doctrine est un ORM, pour *object-relational mapping*, soit mapping objet-relationnel en français.



Attends, euh... c'est quoi un ORM ???

Un ORM, c'est en quelque sorte une interface entre votre base de données et vous. Il s'agit d'un ensemble de classes (ou « *bibliothèques* ») qui fonctionne de manière indépendante. Le but est de nous permettre d'accéder au contenu de la base de données avec un langage orienté objet (PHP en l'occurrence).



Pour ceux qui ont déjà entendu parler de [Symfony](#), sachez que Doctrine est dorénavant l'ORM intégré par défaut.

Pour comprendre concrètement comment Doctrine fonctionne, voyons un exemple. Imaginons que l'on possède une table contenant des articles, avec un titre, un contenu et une certaine catégorie. On pourra y accéder avec un objet Article en PHP :

Code : PHP

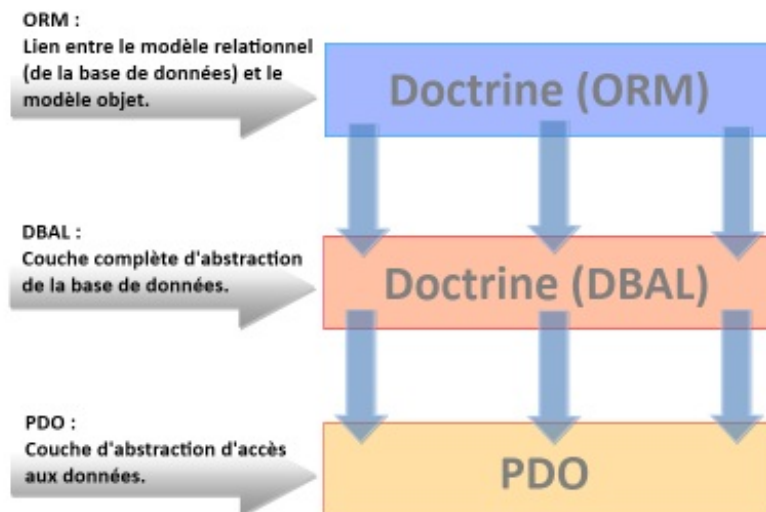
```
<?php
$article = new Article();
$article->title      = 'Tutoriel Doctrine';
$article->content     = 'blablabla...';
$article->categorie  = $monObjetCategorie;
$article->save();
```

Et voilà !

En quelques lignes, nous avons créé un objet `article`, l'avons rempli avec des données, et l'avons sauvegardé. Et c'est tout. Doctrine se charge tout seul d'insérer ces données dans les champs de notre table, on ne touche pas au code SQL.

Et c'est l'un des gros avantages d'un ORM : si vous déplacez votre projet sur un autre serveur, avec un autre SGBD, vous n'aurez pas à vous soucier des différences qu'il peut y avoir entre les langages. En effet, Doctrine repose sur **PDO** et supporte les mêmes drivers. Vous n'aurez pas non plus à écrire les classes (par exemple `Categorie` ou `Article`) vous-même, elles sont générées automatiquement. Ce qui ne veut pas dire qu'on ne pourra pas les personnaliser, bien au contraire.

Voici un schéma pour comprendre rapidement comment Doctrine s'articule :



Doctrine est constitué de deux couches distinctes :

- Je vous l'ai dit, l'une repose sur PDO, à laquelle elle ajoute des fonctionnalités, c'est ce qui s'appelle la DBAL : *Database Abstraction Layer*. Elle pourrait s'utiliser directement, sans la couche ORM. Son but, comme son nom l'indique, est de fournir une couche d'abstraction, c'est-à-dire que grâce à elle, on ne se soucie plus de comment fonctionne notre base de données. À ce niveau-là, on n'utilise pas encore de langage-objet.
- La deuxième couche (ORM, donc) permet de faire le « lien » entre nos objets PHP et la DBAL, c'est ce qui nous fournit une interface orientée objet pour manipuler la base de données. Elle interagit avec la *DBAL* et nous retourne des résultats sous forme d'objets.

Ne faites pas de confusion avec PDO : j'ai parlé d'*abstraction* dans le schéma, mais PDO ne propose pas d'abstraction de la base de données. Elle fournit simplement des fonctions communes, peu importe le SGBD, mais il faut quand même se plier au langage « local ».

Bon assez bavardé, je suppose que vous êtes tous impatients de l'installer ! Comment ça, non ? 🤪

Installation Téléchargement

Ici, je vais seulement vous montrer la manière la plus simple et accessible à tout le monde pour installer Doctrine.

Commencez par télécharger le **package .tgz** ici : <http://www.doctrine-project.org/projects/orm/download>. La dernière version stable à l'heure où j'écris ces lignes est la **1.2.2**. Pour extraire l'archive, utilisez un logiciel comme 7zip ou Winrar.



Si vous souhaitez l'installer via SVN ou PEAR, je vous renvoie à la [documentation en ligne \(in english\)](#). Mais nous ne nous attarderons pas là-dessus. Notez tout de même qu'il existe un [tuto sur SVN](#) sur le Site du Zéro.

À la racine de votre projet, créez un répertoire `~/lib/`. Nous placerons ici toutes nos classes. Nos classes de modèles seront placées dans `~/lib/models/`.

Placez le contenu du dossier [<Archive>/Doctrine-1.2.2/lib/](#) que vous venez de télécharger, dans `~/lib/vendor/doctrine/` (que vous créez pour l'occasion 😊).

Profitions-en pour créer aussi un répertoire `~/web/` à la racine. Tous les fichiers destinés à être vus par le visiteur seront placés ici.

J'insiste sur le fait que la racine Web est `~/web/`. C'est une bonne habitude à prendre de ne pas laisser vos sources librement accessibles. Ne mettez dans ce dossier Web que le strict nécessaire, notamment vos images, feuilles de style, javascript, etc.

Implémentation dans notre projet

Doctrine peut inclure les fichiers contenant nos classes automatiquement, nous n'aurons pas à faire d'`include()` ou `require()` manuellement.

Pour cela, il va falloir ajouter quelques lignes de configuration à votre projet. Créez un fichier `~/config/global.php`.

Tout d'abord, définissons quelques constantes et incluons la classe de base de Doctrine :

Code : PHP - `~/config/global.php`

```
<?php
// Adaptez bien sûr le DSN à votre cas
define('CFG_DB_DSN', 'mysql://root@localhost/db_doctrine_test');

define('LIB_DIR',    dirname(__FILE__) . '/../lib/');
define('CFG_DIR',    dirname(__FILE__) . '/');
define('WEB_DIR',    dirname(__FILE__) . '/../web/');
define('HTML_DIR',   dirname(__FILE__) . '/../html/');

require_once(LIB_DIR . 'vendor/doctrine/Doctrine.php');
spl_autoload_register(array('Doctrine_Core', 'autoload'));
```

Concernant la base de données, il est préférable (et je vous l'ordonne ! 😊) de dédier complètement une base de données à Doctrine. Cela permettra de laisser Doctrine s'occuper de toute la configuration. Si vous ne lui dédiez pas de base de données, il pourrait y avoir des interactions non désirées avec vos autres projets.

Nous enregistrons aussi l'autoloader de Doctrine avec `spl_autoload_register()`. C'est celui-ci qui va s'occuper d'inclure les fichiers contenant les classes pour nous.



Waaaa, c'est génial ! Comment ça marche ?

En fait, c'est assez simple. Pour commencer, l'autoloader (`Doctrine_Core::autoload()`) sera appelé par PHP à chaque fois qu'une classe non définie sera utilisée, avec en paramètre le nom de cette classe. C'est à cette fonction de se débrouiller ensuite pour faire les `include()` ou `require()` nécessaires.

Pour trouver où sont rangés ces fichiers, une convention de Doctrine est que le nom des classes indique aussi où elles sont situées : il suffit de remplacer les `_` par des `/` et on obtient le chemin du fichier (relatif par rapport au dossier contenant Doctrine). Par exemple, la classe `Doctrine_Record_Abstract` est située dans `~/lib/vendor/doctrine/Doctrine/Record/Abstract.php`. À noter que Doctrine ne nomme pas les fichiers contenant des classes avec l'extension `.class.php`.

Cependant, il y a une exception pour les fichiers contenant nos modèles, et Doctrine les inclura toujours (`~/lib/models/`).



Si vous êtes curieux et avez ouvert le fichier `Doctrine.php`, vous avez dû vous apercevoir que la classe `Doctrine` est vide ! En fait, elle hérite de `Doctrine_Core`. Je vous recommande d'utiliser `Doctrine_Core` par la suite, `Doctrine.php` étant là uniquement pour la rétro-compatibilité.

Avant de conclure, je vais faire un petit point sur l'organisation du projet que nous allons utiliser pour suivre ce tutoriel. Je vous ai brièvement parlé de MVC en introduction. Cependant, pour ce tutoriel, il ne sera pas toujours nécessaire de le

respecter, et pour ne pas compliquer la compréhension de ceux qui ne connaissent pas cette architecture, il m'arrivera parfois de ne pas suivre le pattern MVC à la lettre.

Je vous conseille la lecture de l'un des tutoriels présents sur ce site, par exemple [celui de Savageman](#), si voulez en savoir plus sur le sujet.

Vous êtes bien sûr libres d'intégrer directement à votre projet les exemples que je vous donnerai. 😊

Avant d'entrer dans le vif du sujet, créez le fichier `~/web/index.php` et placez-y les lignes suivantes pour vérifier que Doctrine est bien installé :

Code : PHP - `~/web/index.php`

```
<?php
require_once (dirname ( __FILE__ ) . '/../config/global.php' ) ;

echo Doctrine_Core::getPath () ;
```

Si le chemin vers Doctrine s'affiche, c'est bon. 😊 Sinon, reprenez depuis le début, mais il ne devrait pas y avoir de problème à ce stade-là.



Rappelez-vous bien que le fichier **global.php** devra être inclus dans tous vos fichiers !

Voilà voilà, le projet est maintenant opérationnel. Dans la partie suivante nous allons pouvoir nous connecter à la base de données. 😊

Configuration de la base de données

Nous allons maintenant voir comment nous connecter avec la base de données, puis configurer nos tables.

Connexion

La classe Doctrine_Manager

Toutes les connexions à la base de données sont gérées par un objet Doctrine_Manager.

Doctrine_Manager est un singleton, cela signifie qu'il ne peut y en avoir qu'une seule instance à la fois dans votre projet. Si vous êtes allés farfouiller dans `~/lib/vendor/doctrine/Doctrine/Manager.php` (ce que je vous encourage toujours à faire 😊), vous avez pu vous apercevoir que la méthode `__construct()` est privée : on ne pourra donc pas faire de `<?php new Doctrine_Manager(); ?>`.



Mais alors, pourquoi tu nous parles de cette classe si elle ne sert à rien ? 🤔

Je m'attarde un peu là-dessus, parce que c'est le principe du singleton : on ne peut pas instancier des objets comme on veut. On va donc utiliser une méthode statique : `Doctrine_Manager::getInstance()`. C'est elle qui va nous retourner un nouvel objet ; et l'intérêt est qu'elle nous retournera toujours le même, au lieu d'en créer un à chaque fois. Nous n'avons en effet pas besoin d'utiliser plusieurs objets Doctrine_Manager.



Doctrine_Manager nous permet de configurer de nombreuses options, mais nous reviendrons dessus au fur et à mesure du tutoriel. Il y a aussi une partie en annexe qui récapitule tout ça.

Connexion à la base de données

Doctrine_Manager possède une méthode statique `connection()`. Cette méthode prend en paramètre un DSN, du même format que PDO (par exemple, `mysql://root:@localhost/db_doctrine_test`) et nous retourne un objet connexion. Un DSN doit être de la forme **SGBD://utilisateur:motdepasse@serveur/nomdelabasededonnées**.

Modifions le fichier `~/config/global.php` :

Code : PHP - `~/config/global.php`

```
<?php
// Adaptez bien sûr le DSN à votre cas.
define('CFG_DB_DSN', 'mysql://root@localhost/db_doctrine_test');

define('LIB_DIR',    dirname(__FILE__) . '/../lib/');
define('CFG_DIR',    dirname(__FILE__) . '/');
define('WEB_DIR',    dirname(__FILE__) . '/../web/');
define('HTML_DIR',   dirname(__FILE__) . '/../html/');

require_once(LIB_DIR . 'vendor/doctrine/Doctrine.php');
spl_autoload_register(array('Doctrine_Core', 'autoload'));

$conn = Doctrine_Manager::connection(CFG_DB_DSN);
```



Note : Doctrine_Manager peut prendre en charge plusieurs connexions à différentes bases de données. Il est conseillé pour cela d'affecter explicitement un label aux connexions créées :

`Doctrine_Manager::connection('dsn', 'label');` Il est possible ensuite de les récupérer avec `Doctrine_Manager::getConnection('label');`.

Schéma de données (Yaml)

Petite surprise : on va apprendre un nouveau langage. 😊 Mais pas de panique, c'est un langage très simple, utilisé pour définir le schéma de données de Doctrine.

Le langage Yaml

Ce langage est un langage de description, un peu comme le XML, mais moins lourd. Pour info, YAML est un acronyme récursif, tout comme PHP, et signifie *YAML Ain't Markup Language*.

Comme vous allez le voir, la syntaxe du Yaml repose presque uniquement sur l'indentation du code. Attention, quelque chose de très important : l'indentation doit se faire **UNIQUEMENT AVEC DES ESPACES**, jamais avec des tabulations.

Le langage repose sur ces règles principales :

- Les nombres (entiers, flottants, etc.) sont écrits naturellement.
- La valeur Null peut être indiquée avec `null` ou `~`.
- Les booléens sont exprimés par `true` ou `false`.
- Les dates sont écrites au format **ISO-8601**, c'est-à-dire, par exemple, `2010-06-03 23:30:53`.
- Les chaînes de caractères peuvent être écrites sans délimiteurs, délimitées par des simple quotes (`'`), ou par des doubles quotes (`"`). Lorsque la chaîne s'étend sur plusieurs lignes, il faut utiliser le symbole pipe `|` pour l'indiquer.
- Les paires clé/valeur sont séparées par `:`.
- Plutôt que l'indentation, on peut utiliser les symboles `[]` et `{}` pour indiquer explicitement un tableau, par exemple, `tableau: { cle: valeur, cle2: valeur2 }`. Les accolades indiquent des paires clé/valeur, les crochets indiquent simplement des valeurs. Par exemple : `tableau: [valeur1, valeur2, valeur3]`.
- Les commentaires sont indiqués par `#`. Toute ligne contenant un `#` devient commentaire sur le reste de la ligne.

Ces règles sont très intuitives, voyez par vous-mêmes.

Le schéma

Voyons tout de suite le schéma que j'utiliserai dans la suite de ce tutoriel. Nous placerons tout ceci dans le fichier `~/config/schema.yml` :

Code : Autre - `~/config/schema.yml`

```
# Cela représentera nos articles :
Article:
  columns:
    title:
      type: string(255)
      notnull: true
    content:
      type: string
      notnull: true
    user_id:
      type: integer
  relations:
    User:
      class: User
      local: user_id
      foreign: id
      alias: User # Inutile, puisque l'alias par défaut est le nom de la classe.
      foreignAlias: Articles

# Et là les utilisateurs :
User:
  tableName: member
  columns:
    login:
      type: string(255)
      unique: true
      notnull: true
    password:
      type: string(255)
      notnull: true
```

Détaillons maintenant le schéma ci-dessus. Les clés de premier niveau représentent nos classes (Article et User). Chaque classe sera représentée par une table dans la base de données. Doctrine déduit automatiquement le nom de la table, si on ne spécifie pas l'option `tableName`.

À l'intérieur, on retrouve les clés `columns` (qui contient la définition des colonnes de la table) et `relations` (qui définit les relations entre les tables).

Les colonnes

Chaque colonne accepte plusieurs options, dont :

- `type` : un des types suivants : `boolean`, `integer`, `float`, `decimal`, `string`, `blob`, `clob`, `timestamp`, `time`, `date`, `enum`, `gzip`, `array`, `object`. Doctrine adapte ensuite le type utilisé dans la base de données selon ce que le SGBD supporte ;
- `notnull` : indique si le champ peut être vide ou s'il doit toujours contenir une valeur (`true` ou `false`);
- `unique` : indique si le champ doit être unique ou non (`true` ou `false`).



Vous avez dû remarquer que l'on peut indiquer comme type `array` ou `object` ; or, ces types n'existent pas dans un SGBD classique. En fait, Doctrine va s'occuper de sérialiser/désérialiser automatiquement la valeur de ces champs. Ils sont utiles lorsque l'on veut éviter de créer une table supplémentaire avec des relations, parfois un peu lourdes à gérer pour pas grand-chose.

Les relations

Il y a plusieurs manières de définir les relations. Ici, nous sommes partis de l'objet Article et avons indiqué qu'ils ont un auteur. Nous aurions aussi pu indiquer cette relation à partir de l'objet User et indiquer que chacun peut posséder plusieurs articles.

Nous commençons par donner un nom à notre relation ('User'). La classe étrangère est indiquée par `class`. L'identifiant local est spécifié par `local`, et l'identifiant étranger par `foreign`. Doctrine nous donne la possibilité de donner des alias avec `alias` et `foreignAlias`. Ceci nous permettra d'utiliser ensuite : `<?php $article->AliasPourUser->login`.

Schéma de données (PHP)

Le langage Yaml n'est en fait qu'une manière alternative de définir notre modèle. Doctrine le traduit ensuite en PHP ; mais nous pouvons bien entendu écrire directement tout cela en PHP.

Les classes de modèle

Voyons l'équivalent du schéma précédent :

Code : PHP - `~/lib/models/generated/BaseArticle.php`

```
<?php
class BaseArticle extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 255, array('notnull'
=> true));
        $this->hasColumn('content', 'string', null, array('notnull'
=> true));
        $this->hasColumn('user_id', 'integer', null, array());
    }

    public function setUp()
    {
        $this->hasOne('User as User', array('local' => 'user_id',
'foreign' => 'id'));
        // Même remarque que pour l'alias ici : il s'agit par défaut
```

```

    du nom de la classe, donc il est inutile
    // de préciser 'as User'.
    }
}

```

Code : PHP - ~/lib/models/generated/BaseUser.php

```

<?php
class BaseUser extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->setTableName('member');
        $this->hasColumn('login', 'string', 255, array('unique' =>
true, 'notnull' => true));
        $this->hasColumn('password', 'string', 255, array('notnull'
=> true));
    }

    public function setUp()
    {
        $this->hasMany('Article as Articles', array('local' => 'id',
'foreign' => 'user_id'));
    }
}

```

La méthode `hasColumn()` permet de définir les colonnes ; elle prend en argument le nom, le type, la longueur, et les options supplémentaires. Les méthodes `hasOne()` / `hasMany()` permettent de définir les relations ; avec en argument le nom du modèle et éventuellement un alias, et les options.

L'héritage des classes de base

Lisez bien ceci, c'est important.

Vous avez vu que nous avons appelé nos classes `BaseArticle` et `BaseUser` et que nous les avons placées dans le dossier `generated/`. Ceci est peu important si vous écrivez le PHP à la main, et vous pouvez directement nommer vos classes `Article` et `User`. Je le fais pour que dans la suite du tuto, la configuration du projet soit la même, peu importe la méthode que vous avez choisie.



Je ne comprends toujours pas l'utilité de ces classes « BaseMachin » moi...

Lorsque nous demanderons à Doctrine de les générer automatiquement à partir du Yaml, Doctrine aura besoin de recréer ces fichiers. Pour ne pas effacer ceux qui nous servent (et que l'on modifiera ensuite), Doctrine génère une classe « intermédiaire », que nous NE MODIFIERONS PAS. Nous modifierons seulement les classes `Article` et `User` (qui héritent donc de `BaseArticle` et `BaseUser` 🐼).

Si vous ne comprenez pas tout de suite, ne vous inquiétez pas, je récapitulerai juste après.

Les classes représentant les tables

Si vous décidez d'utiliser le PHP, il vous faudra aussi créer une classe pour chaque table. Elles nous serviront plus tard pour effectuer certaines tâches.

Code : PHP - ~/lib/models/ArticleTable.php et ~/lib/models/UserTable.php

```

<?php
// Fichier ~/lib/models/ArticleTable.php
class ArticleTable extends Doctrine_Table
{ }

```

```
// Fichier ~/lib/models/UserTable.php
class UserTable extends Doctrine_Table
{ }
```

Vous noterez que nos modèles héritent de `Doctrine_Record` et que nos tables héritent de `Doctrine_Table`. Ceci est **OBLIGATOIRE** ! Les méthodes `setTableName()`, `hasColumn()`, `hasMany()` etc... sont héritées de `Doctrine_Record`. Allez donc jeter un coup d'oeil au fichier `~/lib/vendor/doctrine/Doctrine/Record.php`.



Génération des tables et modèles

Voyons maintenant comment générer automatiquement notre base de données à partir du schéma que nous venons de créer.

Modifions le fichier `~/config/global.php` pour correspondre à nos besoins :

Code : PHP - `~/config/global.php`

```
<?php
// Adaptez bien sûr le DSN à votre cas.
define('CFG_DB_DSN', 'mysql://root@localhost/db_doctrine_test');

define('LIB_DIR',    dirname(__FILE__) . '/../lib/');
define('CFG_DIR',    dirname(__FILE__) . '/');
define('WEB_DIR',    dirname(__FILE__) . '/../web/');
define('HTML_DIR',   dirname(__FILE__) . '/../html/');

require_once(LIB_DIR . 'vendor/doctrine/Doctrine.php');
spl_autoload_register(array('Doctrine_Core', 'autoload'));
spl_autoload_register(array('Doctrine_Core', 'modelsAutoload'));

$manager = Doctrine_Manager::getInstance();
$conn     = Doctrine_Manager::connection(CFG_DB_DSN);

$manager->setAttribute(Doctrine_Core::ATTR_VALIDATE,
Doctrine_Core::VALIDATE_ALL);
$manager->setAttribute(Doctrine_Core::ATTR_AUTO_ACCESSOR_OVERRIDE,
true);
$manager->setAttribute(Doctrine_Core::ATTR_AUTOLOAD_TABLE_CLASSES,
true);
$manager->setAttribute(Doctrine_Core::ATTR_MODEL_LOADING,
Doctrine_Core::MODEL_LOADING_CONSERVATIVE);

Doctrine_Core::loadModels(LIB_DIR . 'models/');
```

IMPORTANT



Nous ajoutons un autre autoloader (`Doctrine_Core::modelsAutoload`). Celui-ci se chargera d'inclure les classes de modèles. En effet, l'autoloader basique ne trouve les fichiers que par leur nom (souvenez-vous, je l'ai expliqué en première partie du tuto). Or, les fichiers contenant les modèles ne respectent pas cette règle : souvenez-vous lorsque nous avons défini le schéma. Je vous avais parlé des classes **BaseXXX** : elles sont placées par Doctrine dans un dossier `generated/`.

Nous utilisons la méthode `setAttribute()` de `Doctrine_Manager` pour personnaliser plusieurs attributs. Je vous invite à vous reporter à la partie Annexes pour une explication détaillée de ces attributs. Sachez simplement que nous définissons le nécessaire pour une utilisation classique, et pour le chargement automatique de tous les fichiers.

La dernière instruction `<?php Doctrine_Core::loadModels(LIB_DIR . 'models/')` demande d'inclure les modèles se situant dans le dossier indiqué.

Pour le code des modèles, je vous laisse vous reporter aux chapitres précédents.

Voici les commandes à exécuter pour que Doctrine génère les fichiers de modèle et construise notre base de données :

ATTENTION :

Si vous n'avez pas dédié une base de données entière à Doctrine comme je vous l'ai conseillé lors de l'installation, n'exécutez pas `Doctrine_Core::dropDatabases()` ! Cela efface la base de données avant de la reconstruire. Attention donc aux autres projets partageant la même base de données ! Si vous partagez la base de données avec d'autres applications, vous pouvez simplement supprimer les tables utilisées par Doctrine à la main, ce qui peut vite devenir fastidieux, je vous l'accorde... 🤔

Code : PHP - ~/web/builder.php

```
<?php
require(dirname(__FILE__).'../config/global.php');

// Si elle existe, supprimez la base existante.
// Attention, cela vide totalement la base de données !
Doctrine_Core::dropDatabases();

// Création de la base (uniquement si elle n'EXISTE PAS)
Doctrine_Core::createDatabases();

// Création des fichiers de modèle à partir du schema.yml
// Si vous n'utilisez pas le Yaml, n'exécutez pas cette ligne !
Doctrine_Core::generateModelsFromYaml(CFG_DIR.'schema.yml',
LIB_DIR.'models', array('generateTableClasses' => true));

// Création des tables
Doctrine_Core::createTablesFromModels(LIB_DIR.'models');
```

Lorsque vous appellerez le fichier builder.php, votre base de données et les fichiers des classes **BaseXXX** seront reconstruits (les autres seront créés s'ils n'existent pas ; s'ils existent déjà, ils ne seront pas modifiés).

Tout ce fonctionnement *automatique* peut vous paraître un peu obscur. Exécutez donc le fichier ~/web/builder.php. Je vous laisse maintenant aller jeter un coup d'œil dans votre base de données et dans le répertoire ~/lib/models/. Un dossier **generated/** y a été créé, ainsi que les classes **BaseXXX**. Notez que les classes « normales » (c'est-à-dire **Article**, **User**, **ArticleTable** et **UserTable**) ont été créées parce qu'elles n'existaient pas. Par la suite, si vous régénerez à nouveau tout cela, elles ne seront plus modifiées. C'est donc dans celles-ci que nous travaillerons.

Pour finir, notez que j'ai indiqué ici (en Yaml et en PHP) les options les plus utilisées, mais il en existe d'autres et je vous les montrerai au cours du tutoriel.

Rappelez-moi comment vous faisiez jusqu'ici ? Vous construisiez vos bases de données à la main ? Eh bien maintenant vous pouvez oublier tout ça ! 😄

(Oui enfin, ne prenez pas non plus à la lettre tout ce que je peux raconter... 🤔)

Manipulation des données stockées dans la base

On y arrive enfin !

J'ai pris le temps de bien vous montrer comment fonctionne Doctrine, car cela me paraît important pour la suite. Ce n'est pas primordial pour l'instant, mais comme je l'ai dit, nous explorerons plus en détail le fonctionnement de Doctrine plus loin dans le tuto. Je commence donc à préparer le terrain. 😊

On va maintenant pouvoir créer des objets, les modifier, les sauvegarder... Cool, n'est-ce pas ? 😊

Voyons cela tout de suite.



Ne cherchez pas forcément à tester le code qui suit tout de suite. Le prochain chapitre est consacré à un TP qui vous permettra de mettre tout ça en pratique.

Utilisation des objets et collections

Vous vous rappelez l'exemple que je vous avais donné au début ?
Et bien, on va utiliser exactement cela.

Utilisation des objets

Créer un objet

Commençons par créer un nouvel objet :

Code : PHP

```
<?php  
$article = new Article();
```

Doctrine nous fournit plusieurs manières d'accéder aux propriétés de nos objets :

Code : PHP

```
<?php  
$article->title = 'Mon titre';  
$article['title'] = 'Mon titre';  
$article->set('title', 'Mon titre');  
  
$article->content = "Contenu de l'article...";  
$article['content'] = "Contenu de l'article...";  
// etc.  
  
// Et on les récupère de la même manière :  
echo $article->title;
```

Rien que ça !

Personnellement, j'aime bien utiliser la notation `$obj->title`, mais vous faites bien sûr comme vous voulez. 😊



Si vous utilisez Doctrine par l'intermédiaire de **symfony**, notez que ce dernier ajoute la possibilité d'utiliser des *getters* et *setters* (des méthodes commençant par *get* ou *set* permettant de récupérer/modifier les attributs) ; par exemple, `<?php $obj->setTitle('mon titre')` ou `<?php echo $obj->getContent()`.

Définissons maintenant l'auteur de l'article :

Code : PHP

```
<?php
// Création d'un nouvel utilisateur :
$author = new User();
$author->login = 'Gérard';
$author->password = 'pass';

// Affectation de cet utilisateur à notre article :
$article->user_id = $author->id;
// Ou bien directement :
$article->User = $author;
```

Comme vous le voyez, on peut directement affecter l'auteur, sans indiquer explicitement son id, même si c'est cette valeur qui est stockée dans la base de données. Doctrine s'occupe de tout cela automatiquement pour nous simplifier la tâche. 😊

Pour terminer, on sauvegarde tous les changements effectués sur notre article :

Code : PHP

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

N'oubliez pas de sauvegarder ; sans ça, la base de données ne sera pas modifiée ! 😊

Récupérer un objet

C'est là que les classes représentant les tables vont nous être utiles. Doctrine nous fournit plusieurs « finders magiques » :

Code : PHP

```
<?php
$table = Doctrine_Core::getTable('Article');

// Récupérer un article avec son id, par exemple $id = 1
$article = $table->find($id);

// Récupérer un article avec une autre propriété
$article = $table->findOneByTitle('Mon titre');

// Avec deux propriétés
$article = $table->findOneByTitleAndContent('Mon titre', "Contenu de l'article...");

// Avec une relation
$article = $table->findOneByUser($user);
```



Comment ça marche ? Je n'ai jamais créé ces méthodes moi ! C'est Doctrine_Table qui les définit ?

Haha, mystère... 😊

Magie, magie...

Pour répondre à la question, et bien... oui et non.

En fait, Doctrine utilise très largement une méthode magique de PHP : `__call()`. PHP appelle cette méthode lorsque nous en utilisons une qui n'est pas définie. Ensuite, c'est à l'intérieur de cette méthode que tout le travail est fait.

Par exemple, en appelant `findOneByTitleAndContent("Mon titre", "Contenu de l'article...")`, Doctrine va se dire « *Tiens, il veut que je lui donne un article dont le titre est "Mon titre" et le contenu est "Contenu de l'article"...* », et exécuter une requête en conséquence. Bon, en réalité, c'est un peu moins magique que ça, et je vous invite à aller voir le détail de cette fonction dans le code de Doctrine (essayez de la retrouver 😊).

Pendant qu'on y est, parlons des attributs de nos objets (*title*, *content* et *user_id*). En fait, ils ne sont pas définis « en dur » en tant qu'attributs. Pour faire simple, c'est comme s'ils étaient stockés dans un tableau.

Doctrine_Record implémente la méthode magique `__set()`, qui est appelée automatiquement lorsque l'on essaye d'utiliser un attribut qui n'existe pas. Ensuite, c'est comme pour `__call` : c'est ici que l'attribut est récupéré.

Et pour ce qui est de `$article['title'] = '...'`, et bien *Doctrine_Record* implémente l'interface [ArrayAccess](#) (dont le nom est assez explicite 😊).

Je vous encourage par la même occasion à vous documenter au sujet de cette interface. Elle n'est pas extrêmement compliquée à comprendre, et beaucoup de composants de Doctrine l'utilisent.

J'espère que maintenant tout ce fonctionnement « magique » vous paraîtra un peu moins obscur. 😊

Utilisation des collections d'objets

À l'instant, nous avons utilisé des méthodes nommées `findOneBy...`, qui nous retournent un seul objet. Mais si plusieurs objets peuvent correspondre aux critères, alors il est préférable d'utiliser `findBy...` qui retourne un objet *Doctrine_Collection* contenant une liste d'objets. Ces collections peuvent s'utiliser comme des tableaux.

À noter qu'il existe une méthode `findAll()` qui retourne TOUS les enregistrements de la table. Attention en l'utilisant sur des tables qui contiennent beaucoup d'enregistrements...

Voyons un exemple :

Code : PHP

```
<?php
$table = Doctrine_Core::getTable('Article');

$articles = $table->findByTitle('Titre');
foreach($articles as $article)
{
    echo $article->title;
}
```

Il est bien entendu aussi possible de créer/modifier des collections :

Code : PHP

```
<?php
$articles = new Doctrine_Collection('Article');
// Ou bien
$articles = $table->findAll();

$articles[0] = new Article();
$articles[1] = new Article();

$articles[0]->title    = "Titre de l'article n°1";
$articles[0]->content = 'Contenu 1';
$articles[1]->title    = "Titre de l'article n°2";
// etc.
```



Je vous ai dit que beaucoup de composants de Doctrine implémentaient l'interface *ArrayAccess*. Et bien comme vous le voyez dans le code ci-dessus, c'est le cas de *Doctrine_Collection*. Notez aussi que pour pouvoir être parcouru avec *foreach*, il faut aussi implémenter une interface « iterator » (*IteratorAggregate* en l'occurrence).

Pour sauvegarder tous nos objets, il existe un raccourci : nous ne serons pas obligés de sauvegarder nos objets un par un (ouf ! 🤗).

Doctrine_Collection possède tout simplement une méthode `save()` qui appelle cette même méthode sur tous ses objets.

Code : PHP

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

Le langage DQL

Nous allons ici explorer un peu plus en profondeur Doctrine et découvrir son langage interne : **le DQL**.

DQL signifie *Doctrine Query Language*. L'intérêt est encore une fois de ne pas se soucier du type de base de données utilisé. En effet, même s'ils utilisent le langage SQL, beaucoup de SGBD (tous ? 🤗) implémentent leurs propres fonctions et spécificités dans le langage. Doctrine se chargera de traduire le DQL dans le langage approprié.

Sélection simple de données

Rassurez-vous, vous n'allez pas devoir apprendre (encore !) un nouveau langage ultra complexe...
Le DQL utilise en fait les mêmes instructions que le SQL.

Pour ce premier chapitre, nous allons donc utiliser le mot clé... **SELECT** ! 🤗

Tout d'abord, petite explication.

Lorsque vous sélectionnez un objet de cette manière :

Code : PHP

```
<?php
$article = Doctrine_Core::getTable('Article')->find(1);
```

Doctrine va générer des instructions en SQL. Mais nous avons la possibilité d'écrire nous-mêmes les requêtes à la main.

Pour cela, nous allons utiliser des objets *Doctrine_Query*.

Code : PHP

```
<?php
$q = Doctrine_Query::create()
    ->from('Article a')
    ->leftJoin('a.User u')
    ->where('a.id = ?', 1);

$article = $q->fetchOne();
```

Cet exemple produit le même effet que précédemment, le SQL généré étant le même ; mais à la différence près que nous avons construit la requête à la main.

Rassurez-vous, nous allons détailler tout cela juste après. 🤗



Euh... C'est quoi l'intérêt de faire ça alors qu'on peut le récupérer en une seule ligne, notre article ?!

Oui, bon, j'avoue, sur cet exemple, l'intérêt n'est pas flagrant. 😊 Mais dès que vous allez vouloir faire des sélections plus complexes qu'avec les *finders* (souvenez-vous, les méthodes `$table->findByXXX()`), cela va devenir indispensable.

Imaginons que nos articles possèdent un champ `publishable` dont le contenu indique si l'article est publiable ou non. Pour récupérer un article, il nous faut alors prendre ceci en compte et filtrer les résultats :

Code : PHP

```
<?php
$q = Doctrine_Query::create()
    ->from('Article a')
    ->leftJoin('a.User u')
    ->where('a.id = ?', 1)
    ->andWhere('a.publishable = ?', true);

$article = $q->fetchOne();
```

SELECT : Doctrine_Query en détails

Passons maintenant en revue les différentes méthodes disponibles pour construire notre `Doctrine_Query`.

SELECT

`Doctrine_Query::select()` permet d'indiquer les champs à sélectionner. Notez que pour l'instant, nous récupérons tous nos enregistrements sous forme d'objets ; il est donc inutile de préciser quels champs récupérer, car Doctrine sélectionne automatiquement tous les champs. Nous verrons plus tard comment personnaliser tout ça. 😊

FROM

`Doctrine_Query::from()` permet d'indiquer la (ou les) table(s) sur lesquelles effectuer la sélection. Pour ajouter d'autres champs à une requête existante, utilisez `Doctrine_Query::addFrom()`.

WHERE

`Doctrine_Query::where()` permet de filtrer les enregistrements. Utilisez la syntaxe « classique » de la clause WHERE du SQL.

Notez que cette méthode efface toutes les conditions précédemment définies. Pour ajouter une condition à celles existantes, il faut utiliser `Doctrine_Query::addWhere()` (identique à `Doctrine_Query::andWhere()`) qui ajoute une condition avec AND, ou `Doctrine_Query::orWhere()` qui ajoute une condition avec OR.

Cette fonction accepte en deuxième argument la liste des paramètres :

Code : PHP

```
<?php
$q->andWhere('a.id = ? OR a.id = ?', array(1, 2));
$q->orWhere('a.title = ?', array('Mon titre'));
// Ou :
$q->orWhere('a.title = ?', 'Mon titre');
// Ou encore :
$q->orWhere('a.title = :title', array(':title' => 'Mon titre'));
```

JOIN

Doctrine supporte deux types de jointures : INNER JOIN et LEFT JOIN. RIGHT JOIN n'est pas supporté, mais il suffit de construire ses requêtes autrement pour y pallier. Les méthodes utilisées sont `Doctrine_Query::innerJoin()` et

`Doctrine_Query::leftJoin()`.

Par défaut, Doctrine ajoute automatiquement la condition de jointure (mot-clé `ON`) en fonction des relations définies entre les tables. Pour utiliser une condition personnalisée, il suffit d'inclure le mot clé `ON` dans le DQL passé à l'une de ces méthodes :

Code : PHP

```
<?php
$q = Doctrine_Query::create()
    ->from('Article a')
    ->leftJoin('a.User u ON u.id = ?', 3);
```

- Le mot-clé `WITH` (ajout d'une condition sur la jointure) s'utilise de la même manière.
- Les paramètres sont passés de la même façon que pour le `WHERE` (voir ci-dessus).

ORDER BY

La clause `ORDER BY` se définit avec `Doctrine_Query::orderBy()` (ou `Doctrine_Query::addOrderBy()`).

LIMIT et OFFSET

Utilisez `Doctrine_Query::limit()` et `Doctrine_Query::offset()`.



Pour ceux qui formulent leurs limites en faisant `LIMIT 14, 50` (avec MySQL par exemple), sachez que cela ne fait pas partie des spécifications officielles de SQL. L'autre formulation possible est `LIMIT 50 OFFSET 14`. Avec Doctrine, vous devrez donc utiliser la deuxième manière et utiliser `limit()` et `offset()`.

GROUP BY

Utilisez `Doctrine_Query::groupBy()` ou `Doctrine_Query::addGroupBy()`.

HAVING

Utilisez `Doctrine_Query::having()` ou `Doctrine_Query::addHaving()`.

Récupération des résultats

Pour finir, la requête doit être exécutée pour pouvoir récupérer les résultats.

Doctrine possède plusieurs méthodes pour cela :

- **execute()** : cette méthode retourne un objet `Doctrine_Collection` contenant un enregistrement ou plus. Notez que vous pouvez passer en argument un tableau contenant les paramètres n'ayant pas encore été renseignés. On peut également passer en deuxième argument le mode d'hydratation des résultats, mais nous verrons cela plus tard. 😊
- **fetchOne()** : cette méthode retourne un objet `Doctrine_Record`. Cette méthode accepte les mêmes arguments que `execute()`.

Il existe d'autres méthodes qui retournent des résultats, telle que `fetchArray()`. Mais nous verrons cela plus tard (oui, encore 😊).

Note sur le passage des paramètres

Vous avez vu que les méthodes acceptant des *variables* peuvent recevoir en paramètre ces valeurs à remplir. Lorsque vous utilisez des valeurs dynamiques comme cela, vous devez toujours les passer en paramètre. De cette manière, les caractères dangereux seront échappés automatiquement, évitant des injections SQL.

Ces paramètres s'utilisent de la même manière qu'avec PDO. Deux choix s'offrent à vous pour indiquer leur emplacement :

- Utiliser des `?`.
Dans ce cas, les paramètres doivent être passés dans le même ordre que les `?`.
Par exemple : `<?php $q->where('a.title = ?', $title)`.

- Utiliser des paramètres nommés, comme **:name**.
 Dans ce cas, l'argument supplémentaire doit être un tableau associatif de paires **nom => valeur**.
 Par exemple : `<?php $q->where('a.title = :title', array(':title' => $title))`.

Pour finir, comme je vous l'ai dit, les méthodes qui exécutent la requête (telles que `execute()` et `fetchOne()`) acceptent elles aussi comme premier argument un tableau de paramètres à passer.

Pourquoi ? Et bien, il se peut qu'au moment de construire votre requête, vous ne connaissiez pas encore la valeur des variables que vous utiliserez à l'intérieur. Vous avez donc la possibilité de les passer tout à la fin, juste au moment de l'exécution.

Par exemple, les deux codes ci-dessous sont identiques :

Code : PHP - Passage des paramètres lors de la construction

```
<?php
$q = Doctrine_Query::create()
->from('Article a')
->where('a.title = :title AND a.publishable = :publishable',
array(
    ':title' => $title,
    ':publishable' => $publishable
))
->andWhere('a.content LIKE :keyword', '%'.$keyword.'%');

$resultats = $q->execute();
```

Code : PHP - Passage des paramètres lors de l'exécution

```
<?php
$q = Doctrine_Query::create()
->from('Article a')
->where('a.title = :title OR a.publishable = :publishable')
->andWhere('a.content LIKE :keyword');

$resultats = $q->execute(array(
    ':title' => $title,
    ':publishable' => $publishable,
    ':keyword' => '%'.$keyword.'%',
));
```

Ceci peut vous être utile lorsque vous utilisez des requêtes comme *modèles*. Notez que dans ce cas, il est conseillé d'utiliser les paramètres nommés (tels que **:title**), à moins que vous ne connaissiez par coeur l'ordre dans lequel vous devez passer les paramètres. 😊

UPDATE et DELETE

Les mises à jour et suppressions de données se font tout aussi simplement avec un objet `Doctrine_Query`.

DELETE

Ici, au lieu d'utiliser `Doctrine_Query::select()`, nous allons utiliser... `Doctrine_Query::delete()` ! Surprenant, hein ? 😊

Nous pouvons appliquer les mêmes filtres que pour un SELECT ; voici un exemple :

Code : PHP

```
<?php
$q = Doctrine_Query::create()
->delete('Article a')
->where('a.title LIKE ?', 'Mauvais titre')
->orWhere('a.published = ?', false)
```

```
->limit(20);
$number = $q->execute();
```

UPDATE

De la même manière, nous pouvons mettre à jour des champs. Il faut en revanche spécifier en plus les champs à mettre à jour avec `Doctrine_Query::set()` :

Code : PHP

```
<?php
$q = Doctrine_Query::create()
->update('Article a')
->where('a.title = ?', 'mon titre')
->orWhere('a.id = ?', 1)

// Soit :
->set('content', 'un super texte');
// ou soit :
->set('content', '?', $content);
// ou encore
->set(array('content' => $content))

$number = $q->execute();
```



Le nombre de lignes affectées (mises à jour ou supprimées) est retourné par la méthode `execute()` dans le cas d'un update d'un delete.

Utilité des classes de tables

Au chapitre précédent, je vous avais dit de créer, en plus des classes classiques de modèles, des classes de tables (par exemple `ArticleTable.class.php`). Mais peut-être n'en voyez-vous pas l'utilité.

En fait, nous allons nous en servir pour effectuer des traitements lorsqu'il y en aura besoin. Tout le code (ou du moins le plus possible 🤖) concernant des actions sur la table et ses données devra être placé dans cette classe.

Prenons un exemple. Imaginons que nos articles contiennent un champ `is_publishable` qui indique si l'article est publiable ou non. Par défaut, la méthode `find()` cherche un article avec son id. Mais nous voulons en plus une « sécurité », c'est-à-dire ne pas pouvoir récupérer les articles non publiables, par exemple.

Pour cela, nous allons redéfinir cette méthode :

Code : PHP

```
<?php
class ArticleTable extends Doctrine_Table
{
    public function find($id)
    {
        $q = $this->createQuery('a')
        ->where('a.id = ?', $id)
        ->andWhere('a.is_publishable = ?', true);

        return $q->fetchOne();
    }
}
```

Maintenant, lorsque nous utiliserons `<?php Doctrine_Core::getTable('Article')->find(3)`, ce sera cette méthode qui sera appelée (merci l'héritage 🤖).

Remarquez que nous avons utilisé `<?php $this->createQuery('a')`. Ceci est en fait un raccourci pour :

Code : PHP

```
<?php
Doctrine_Query::create()
    ->from('Article a')
```

Mais nous avons encore un problème : si on utilise une autre méthode que *find()* pour récupérer un article, le filtrage ne sera pas effectué.

Afin de simplifier l'ajout de cette condition à toutes nos requêtes, nous allons créer une méthode *getPublishableArticleQuery()* :

Code : PHP

```
<?php
class ArticleTable extends Doctrine_Table
{
    public function getPublishableArticleQuery(Doctrine_Query $q =
null)
    {
        if(null === $q)
        {
            $q = $this->createQuery();
        }

        // getRootAlias() renvoie l'alias principal de la requête
        // (jusqu'à maintenant j'ai utilisé 'a' pour cette table,
        // mais j'aurais pu utiliser n'importe quoi).
        $q->andWhere($q->getRootAlias().'.is_publishable = ?', true);

        return $q;
    }

    public function find($id)
    {
        $q = $this->createQuery('a')
            ->where('a.id = ?', $id);

        return $this
            ->getPublishableArticleQuery($q)
            ->fetchOne();
    }
}
```

Explications :

La méthode `ArticleTable::getPublishableArticleQuery()` accepte en argument un objet `Doctrine_Query`, et lui ajoute une condition `WHERE`, puis la retourne.



Nous voyons ici toute la puissance de `Doctrine_Query` : il est possible de modifier la requête à plusieurs endroits dans notre programme, de lui ajouter des conditions, etc. sans se soucier de ce qu'elle fait. Notre méthode *getPublishableArticleQuery* ajoute la condition, sans savoir comment la requête est construite ni les éléments qu'elle sélectionne déjà.

Il reste un petit inconvénient : dans toutes les requêtes de sélection telles que *find()*, nous devons penser à bien passer par cette méthode afin d'ajouter le filtre.

Nous verrons plus tard une méthode beaucoup plus propre et pratique pour pallier cela, afin que ce soit fait automatiquement (oui oui, Doctrine réserve encore beaucoup de surprises... 😊).

Je vous ai expliqué cela ici pour vous montrer la force de l'héritage : nous pouvons redéfinir tous les comportements que nous voulons (tout en restant raisonnable, et en ne « détruisant » pas tout... 😊).

Pour conclure cette partie, notez que l'objet *Doctrine_Connection* (à ne pas confondre avec *Doctrine_Collection* 😊) possède une méthode *flush()*, qui permet de sauvegarder tout ce qui a été modifié avec la connexion courante. C'est utile lorsque vous

modifiez un grand nombre d'objets et collections différents : cela vous évite de sauvegarder les objets/collections un par un.

TP : Création d'un CRUD

Nous allons maintenant passer un peu plus à la pratique et créer un CRUD. Pour ceux qui ne connaissent pas, un CRUD est un système de manipulation des données de la base : ça signifie *Create, Read, Update, Delete* (*Créer, Lire, Mettre à jour, Supprimer*).

Autrement dit, on va faire une interface pour gérer nos données. Cela peut entre autres servir à créer une interface d'administration pour votre site.



Attention, ce TP est là uniquement pour vous faire pratiquer un peu sur des choses simples, pour commencer à retenir un peu le fonctionnement de Doctrine et les méthodes utiles. Je ne traiterai pas, notamment, de la sécurité des scripts. Ceci est à intégrer à un projet, c'est pourquoi je vous donne un contrôleur frontal si basique.

Préparation du terrain Récapitulatif

Avant de commencer, récapitulons la structure de notre projet. Jusqu'à maintenant je vous ai simplement indiqué où placer quelques fichiers.

Nous utiliserons cette structure :

```
~/config/  
----global.php  
----schema.yml  
~/html/  
~/lib/  
----models/  
-----Article.php  
-----ArticleTable.php  
-----User.php  
-----UserTable.php  
-----generated/  
-----BaseArticle.php  
-----BaseUser.php  
~/web/  
----index.php
```

Si vous avez suivi le tutoriel en entier, vous ne devriez rien avoir à faire. Dans le cas contraire, je vous laisse vous reporter aux chapitres précédents pour connaître le contenu de ces fichiers.

Comme je vous l'ai dit, nous allons grandement simplifier le modèle MVC : nous aurons un contrôleur frontal (`~/web/index.php`) qui se chargera aussi d'inclure la page HTML.



Le but ici n'est pas d'expliquer le pattern MVC. Libre à vous d'utiliser votre propre système de templates/framework, d'adapter tout cela à un projet existant, etc. Je vous encourage d'ailleurs à le faire.

Le contrôleur

Voici la structure de notre contrôleur (tout pas beau tout moche, je sais 😊) :

Code : PHP - `~/web/index.php`

```
<?php  
require(dirname(__FILE__).'../config/global.php');  
  
// On récupère l'action à effectuer (Create, Read, Update ou  
Delete).  
// Je compte sur vous pour utiliser un système plus perfectionné  
que ça. ^^
```

```

$action = 'read';
if(isset($_GET['action']) && in_array($_GET['action'],
array('create', 'read', 'update', 'delete')))
{
    $action = $_GET['action'];
}

/* Nous ferons ici les traitements concernant la page. */
switch($action)
{
    case 'read':
        break;
    case 'create':
        break;
    case 'update':
        break;
    case 'delete':
        break;
}

/* Nous appellerons ici la page HTML appropriée. */
include(HTML_DIR.$action.'.php');

```

Données de test

Avant de commencer, il va bien falloir avoir quelques données pour tester. Insérez donc ceci :

Code : SQL - Données de base

```

INSERT INTO member
(login, password)
VALUES
('Jean', 'mypass'),
('Bernard', 'mysuperpass'),
('Superman', 'batman');

INSERT INTO article
(title, content)
VALUES
("Mon premier article", "Voici le contenu de l'article, comme c'est
simplement pour tester, je mets du texte tout bidon. D'ailleurs, je
ne sais pas si vous avez remarqué, mais je n'ai pas beaucoup
d'imagination !"),
("Mon second article", "Voici le contenu du deuxième article, comme
c'est simplement pour tester, je mets du texte tout bidon.
D'ailleurs, je ne sais pas si vous avez remarqué, mais je n'ai pas
beaucoup d'imagination !"),
("Mon troisième et dernier article", "Voici le contenu du troisième
article, comme c'est simplement pour tester, je mets du texte tout
bidon. D'ailleurs, je ne sais pas si vous avez remarqué, mais je
n'ai pas beaucoup d'imagination !");

```

Bien sûr, si vous avez des données plus intéressantes, utilisez-les. 😊

À partir d'ici, je vais vous donner le code au fur et à mesure que l'on avance. Je ne vous prends pas pour des gamins, si vous n'avez pas envie de réfléchir et voulez copier/coller directement, libre à vous. 😊 Mais dans ce cas, ce n'est presque pas la peine de lire jusqu'en bas, parce que je ne fais que reprendre ce que l'on a vu jusqu'ici. À vous de voir ce que vous préférez. 😊

Affichage de données

Ici, nous voulons afficher une liste de tous nos articles dans un tableau. Nous allons devoir récupérer les données dans la base de données, puis les afficher. Ce n'est vraiment pas compliqué, à vous de jouer.

Création de la requête

Ajoutez ceci dans le contrôleur :

Code : PHP - ~/web/index.php

```
<?php
/* ... */
case 'read':
    $articles = Doctrine_Core::getTable('Article')->findAll();
    break;
/* ... */
```

Comme nous l'avons vu, *findAll()* nous retourne tous les articles de la table.

Affichage des données récupérées

Créons maintenant la vue pour l'action *read* :

Code : PHP - ~/html/read.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
lang="en-US">
<head>
<title>Tutoriel Doctrine</title>
<meta http-equiv="content-type" content="text/html; charset=iso-
8859-1" />
</head>
<body>
<h1>Liste des articles</h1>
<a href="?action=create">Ajouter un nouvel article</a>
<table>
<tr>
<th>Titre</th>
<th>Contenu</th>
<th>Auteur</th>
<th><em>Action</em></th>
</tr>
<?php foreach($articles as $article): ?>
<tr>
<td><?php echo $article['title'] ?></td>
<td><?php echo $article['content'] ?></td>
<td><?php echo $article['User']['login'] ?></td>
<td>
<a href="?action=update&id=<?php echo $article['id'] ?
>">Modifier</a>
<a href="?action=delete&id=<?php echo $article['id'] ?
>">Supprimer</a>
</td>
</tr>
<?php endforeach; ?>
</table>
</body>
</html>
```

Tout ceci est assez simple, je ne pense pas qu'il y ait de problème particulier. Nous utilisons simplement *findAll()* pour récupérer tous les articles. Comme on récupère un objet Doctrine Collection (qui peut s'utiliser comme un tableau, rappelez-vous),

nous utilisons `foreach` pour les parcourir.



La notation « tableau » est recommandée lorsque nous accédons à l'objet dans du code HTML. Ceci, simplement parce que si la personne qui s'occupe d'écrire le code HTML n'est pas la même que celle qui écrit le code PHP, elle n'est pas censée savoir comment PHP fonctionne. Cette notation lui permet de ne pas se soucier de la nature de la variable `$article`. Cela dit, si vous vous occupez de tout, vous faites bien comme vous voulez, vous avez quand même tout à fait le droit d'utiliser la notation objet si ça vous chante. 😊

Création d'objets

Cette fois, il va nous falloir afficher un formulaire nous permettant de renseigner les informations sur un nouvel article. Attention, pour l'utilisateur, je suppose que vous ne voulez pas laisser le choix à celui qui écrit l'article de « choisir qui écrit l'article » (logique, non ? 😊).

Le formulaire

Nous allons voir ici comment créer de nouveaux objets et les sauvegarder dans la base.

Nous allons avoir besoin d'un formulaire afin de rentrer les informations :

Code : PHP - `~/html/create.php`

```
<body>
<h1>Création d'un article</h1>
<form action="index.php?action=create" method="post">
  <label for="article_title">Titre : </label><br />
  <input type="text" name="title" id="article_title" /><br />
  <label for="article_text">Texte : </label><br />
  <textarea name="content" id="article_text"></textarea><br />
  <input type="submit" value="Sauvegarder" />
</form>
<a href="?action=read">Retour à la liste</a>
</body>
```

Nous ne mettons pas de champ 'Auteur'. Personnalisez tout cela à votre sauce, par exemple en récupérant l'id de l'auteur depuis une variable de session.

Traitement des informations

Code : PHP - `~/web/index.php`

```
<?php
/* ... */
case 'create':
  if (!empty($_POST))
  {
    $article = new Article();
    $article->title = $_POST['title'];
    $article->content = $_POST['content'];

    // On indique l'auteur. Adaptez cela à votre projet.
    $article->User = $user;

    $article->save();
  }
  break;
/* ... */
```

Nous récupérons ici les données postées par le formulaire, et les affectons à un nouvel article, puis nous le sauvegardons.

Modification d'objets

La modification d'objets sera semblable à la création. La principale différence réside dans le fait que nous allons préremplir le formulaire avec les informations déjà existantes. Il nous faut donc récupérer l'article dans tous les cas. Et si on poste des informations le concernant, il faut le mettre à jour et sauvegarder en conséquence.

Le formulaire

Nous allons reprendre le formulaire de création, en ajoutant simplement les valeurs par défaut :

Code : PHP - ~/html/update.php

```
<body>
<h1>Modification d'un article</h1>
<form action="index.php?action=update" method="post">
  <label for="article_title">Titre : </label><br />
  <input type="text" name="title" id="article_title" value="<?php
echo $article['title'] ?>" /><br />
  <label for="article_text">Texte : </label><br />
  <textarea name="content" id="article_text"><?php echo
$article['content'] ?></textarea><br />
  <input type="submit" value="Sauvegarder" />
  <!-- Pour se souvenir de quel article il s'agit -->
  <input type="hidden" name="id" value="<?php echo $article['id'] ?
>" />
</form>
<a href="?action=read">Retour à la liste</a>
</body>
```

Traitement des informations

Code : PHP - ~/web/index.php

```
<?php
/* ... */
case 'update':
  $id = isset($_POST['id']) ? $_POST['id'] : $_GET['id'];
  // Il faut évidemment s'assurer que l'article existe
  if(!($article = Doctrine_Core::getTable('Article')->find($id))
  {
    // À vous de mettre un traitement personnalisé !

    // Par exemple, on considère qu'il s'agit d'un nouvel article.
    $article = new Article();
    // Ou bien :
    exit();
  }

  if(!empty($_POST))
  {
    $article->title = $_POST['title'];
    $article->content = $_POST['content'];

    // On indique l'auteur. Adaptez cela à votre projet, par exemple
    si vous stockez l'id dans la session.
    $article->User = $user;

    $article->save();
```

```

    }
    break;
    /* ... */

```

Petit exercice : trouvez-moi l'équivalent en construisant une requête (Doctrine_Query).

Correction :

Secret (cliquez pour afficher)

Code : PHP

```

<?php
if (!empty($_POST))
{
    $q = Doctrine_Query::create()
    $q->update('Article a')
    ->set('a.title', '?', $_POST['title'])
    ->set('a.content', '?', $_POST['content'])
    ->where('a.id = ?', $id)
    ->execute();
}

```

Refactorisation du code

Ne voyez-vous pas quelque chose d'évident ?

Les pages *create* et *update* sont pratiquement identiques, il est donc intéressant de les fusionner en une seule.

Le formulaire

La page contenant le formulaire reste inchangée. Si l'article est « vide », alors les champs seront tout simplement préremplis avec des valeurs vides :

Code : PHP - ~/html/update.php

```

<body>
<?php if($article->exists()): ?>
    <h1>Modification d'un article</h1>
<?php else: ?>
    <h1>Création d'un article</h1>
<?php endif; ?>

<form action="index.php?action=<?php echo $action ?>"
method="post">
    <label for="article_title">Titre : </label><br />
    <input type="text" name="title" id="article_title" value="<?php
echo $article['title'] ?>" /><br />
    <br />
    <label for="article_text">Texte : </label><br />
    <textarea name="content" id="article_text"><?php echo
$article['content'] ?></textarea><br />
    <br />
    <input type="submit" value="Sauvegarder" />
    <?php if($article->exists()): ?>
    <input type="hidden" name="id" value="<?php echo $article['id'] ?
"> />
    <?php endif; ?>
</form>
<a href="?action=read">Retour à la liste</a>
</body>

```



Remarquez la méthode `exists()` de l'objet `Doctrine_Record`. Elle indique si l'objet existe dans la base de données, c'est-à-dire s'il est nouveau ou non.

Le contrôleur

Le traitement devra créer l'objet au cas où il n'existe pas :

Code : PHP - ~/web/index.php

```
<?php
/* ... */
case 'create':
case 'update':
    if(!isset($_GET['id']) && !isset($_POST['id']))
    {
        $article = new Article();
    }
    else
    {
        $id = isset($_POST['id']) ? $_POST['id'] : $_GET['id'];
        if(!($article = Doctrine_Core::getTable('Article')->find($id))
        {
            $article = new Article();
        }
    }

    if(!empty($_POST))
    {
        $article->title = $_POST['title'];
        $article->content = $_POST['content'];

        // On indique l'auteur. Adaptez cela à votre projet, par exemple
        // si vous stockez l'id dans la session.
        $article->User = $user;

        $article->save();
    }
    break;
/* ... */
```

Dans le cas où la page est appelée avec un id, on l'utilise pour récupérer l'article existant correspondant. Sinon, on en crée un nouveau.

Ensuite, peu importe que l'article soit un nouveau ou un existant, le traitement est le même pour modifier les valeurs et sauvegarder.

Par ailleurs, le fichier `create.php` ne nous sert maintenant plus à rien. 😊

Suppression de données

La suppression de données est très simple. Pour la vue, je vous invite juste à afficher un message de confirmation, par exemple, dans le fichier `~/html/delete.php`, vous devriez y arriver sans problème. 😊

Traitement

Code : PHP - ~/web/index.php

```
<?php
/* ... */
```

```
case 'delete':  
    $id = isset($_POST['id']) ? $_POST['id'] : $_GET['id'];  
    // On s'assure que l'article existe  
    if($article = Doctrine_Core::getTable('Article')->find($id))  
    {  
        $article->delete();  
    }  
    break;  
/* ... */
```

Trouvez-moi maintenant l'équivalent avec une Doctrine_Query 😊

Secret (cliquez pour afficher)

Tout simplement :

Code : PHP

```
<?php  
$q = Doctrine_Query::create()  
$q->delete('Article a')  
    ->where('a.id = ?', $id)  
    ->execute();
```

Ce TP (extrêmement simple, non ? 😊) vous a permis de mettre en pratique simplement ce que nous avons vu jusque là. Bien sûr, comme je l'ai déjà dit, l'intérêt de Doctrine se révèle surtout dans les gros projets, et je ne vous encourage pas forcément à l'utiliser pour de petits projets simples.



Attention, comme je l'ai déjà dit dans l'introduction, ce TP n'est pas destiné à être un système complet. Je n'ai notamment pas mis l'accent sur la sécurité, mais il faut bien entendu vous assurer dans chaque action que les données correctes sont fournies.

Sur ce, et si vous voulez en savoir plus sur Doctrine, passons maintenant à des chapitres un peu plus complexes. Le suivant concerne les templates (ou *behaviors*).

Voilà, vous devez être maintenant capables d'utiliser Doctrine dans vos projets. Vous verrez, avec un peu de pratique, ça vous simplifiera grandement la tâche. 😊

Si vous êtes un peu curieux, je vous invite maintenant à entrer un peu plus en profondeur dans les entrailles de Doctrine (oulà, ça fait peur ! 😊). Plus vous en connaîtrez sur cet ORM, plus vous l'utiliserez d'une façon optimale.

Nous verrons beaucoup de trucs utiles et pratiques, qui simplifieront encore votre code. Je vous encourage vivement à ne pas vous arrêter ici. 😊

Partie 2 : Utilisation avancée de Doctrine

Maintenant que vous en connaissez un minimum sur Doctrine, on va pouvoir s'attaquer à des choses vraiment intéressantes.

Les Events Listeners

Dans cette partie, nous nous intéresserons aux *Events Listeners*. Comme leur nom l'indique, ces choses bizarres permettent... d'écouter des événements !

Non, je ne vous prends pas pour des billes, nous allons voir immédiatement de quoi il s'agit.

De manière générale et théorique, lorsqu'un événement est déclenché, les *listeners* associés en sont informés, et peuvent agir en conséquence. Ils peuvent non seulement faire leur petit travail de leur côté, mais aussi, et c'est très important, modifier l'exécution de l'évènement.

Petite précision avant de commencer : cette partie restera assez théorique, mais nous (c'est-à-dire vous 🤖) mettrons tout cela en pratique dans un prochain TP.

Introduction aux Events Listeners

Déclenchement d'évènements

La première chose à connaître, lorsque l'on parle d'évènements, est de savoir quand ils sont déclenchés.

Dans Doctrine, nous ne nous occuperons pas de les déclencher, cela est fait automatiquement. Par exemple, lors d'un enregistrement dans la base de données, ou lors d'une sélection de données, etc.



Pour ceux qui utilisent Symfony ou qui ont l'habitude de déclencher des événements manuellement, gardez bien à l'esprit qu'avec Doctrine, nous ne parlons que d'*Event Listener*, ce qui signifie que nous ne faisons que les écouter. Le framework Symfony (entre autres) intègre une gestion complète des événements, ce qui vous permet d'en déclencher vous-même. Ce n'est pas le cas ici !

Les différents composants de Doctrine déclenchent à plusieurs moments ces événements. Nous allons les étudier juste après.

Intercepter un événement

C'est le rôle des *listeners* et des *hooks*.

- Un *hook* est une méthode vide intégrée à l'objet écouté.
- Un *listener* est un objet à part, qui est *lié* à l'objet écouté. Il implémente lui-même les *hooks*, ce qui permet une meilleure séparation et organisation des différentes facettes de l'application.

Vous comprendrez mieux tout cela par la suite, ne vous inquiétez pas. 😊

Les classes susceptibles de déclencher des événements, sont les suivantes :

- Doctrine_Record ;
- Doctrine_Validator ;
- Doctrine_Connection ;
- Doctrine_Transaction ;
- Doctrine_Connection_Statement.

Les méthodes utilisées

Les méthodes (que ce soient des hooks, ou que ce soient celles des listeners) que nous allons utiliser seront expliquées par la

suite. Leur utilisation est intuitive, voyons un exemple :

Code : PHP

```
<?php
class Article extends Doctrine_Record
{
    public function preSave($event)
    {
        $this->user_id = $_SESSION['user_id'];
    }
}
```

Dans cet exemple, nous utilisons la méthode `preSave()` (un hook). Comme son nom l'indique, cette méthode est appelée juste avant l'enregistrement de l'article dans la base de données. Nous en profitons pour indiquer que c'est l'utilisateur courant qui a modifié cet article en dernier. Comme nous sommes à l'intérieur de l'objet en question, nous avons bien entendu accès à la variable `$this`, et nous pouvons le modifier directement, juste avant son enregistrement.

Les méthodes utilisées respectent toutes la même logique : elles sont préfixées par *pre* ou *post*, selon qu'elles soient appelées *avant* ou *après* l'évènement (ici, *save*).

Pour finir, elles reçoivent en paramètre un objet `$event`, instance de *Doctrine_Event*.



Mais comment Doctrine peut-il savoir que j'ai écrit la méthode *preSave()* si je ne le lui dis pas ?

En fait, tous les hooks sont déjà définis dans la classe *Doctrine_Record* : allez y jeter un coup d'œil. Ce sont des méthodes vides, et Doctrine les appelle à chaque évènement. Seulement, tant que vous ne redéfinissez pas cette méthode dans une classe fille (merci l'héritage 🐼), et bien il ne se passe rien !

C'est la même chose à chaque fois : l'astuce est simplement d'utiliser des méthodes vides !

Doctrine_Event

Les objets *Doctrine_Event* permettent d'agir sur certains objets. Par exemple, si l'évènement est l'exécution d'une requête, on peut récupérer celle-ci, et ainsi la modifier.

Voici quelques méthodes utilisées fréquemment :

getInvoker()

Retourne l'objet qui a « invoqué » l'évènement. Par exemple, un objet *Doctrine_Record*.

Il y a toujours un *Invoker*, et il s'agit d'une instance de l'une des classes citées plus haut.

Nous pouvons donc reformuler l'exemple précédent :

Code : PHP

```
<?php
class Article extends Doctrine_Record
{
    public function preSave($event)
    {
        $event->getInvoker()->user_id = $_SESSION['user_id'];
    }
}
```

getQuery()

Retourne l'objet *Doctrine_Query* associé à l'évènement, s'il existe.

Attention, il peut ne pas exister ! Dans l'exemple précédent, cela renverra la valeur *null*.

getParams()

Retourne un tableau d'éventuels paramètres. Par exemple, le paramètre *alias* contient l'alias utilisé dans la requête.

Voici les méthodes les plus courantes. Je vous invite à ouvrir le fichier [Event.php](#) pour plus de détails. Si vous avez bien suivi le début du tuto, vous devriez le trouver dans `~/lib/vendor/doctrine/Doctrine/`. 😊

Listener de connexion

Dans ce chapitre, nous nous intéressons aux listeners de connexion, c'est-à-dire les listeners que l'on peut attacher à la classe *Doctrine_Connection*.

Je vous l'ai dit dans le chapitre précédent, un listener est une classe qui implémente des méthodes spécifiques.

Créer un listener

Il existe trois manières de créer un listener.

Étendre une classe de base

Nous appellerons notre listener fictif *CustomListener*. *Doctrine_EventListener* offrant une base pour tout listener, le nôtre en héritera donc :

Code : PHP

```
<?php

class CustomListener extends Doctrine_EventListener
{ }
```

Ouvrez le fichier `~/lib/vendor/doctrine/Doctrine/EventListener.php`. Vous devriez maintenant mieux comprendre l'astuce des *méthodes vides* que je vous avais expliquée. La seule chose qu'il vous reste à faire maintenant, c'est de surcharger les méthodes que vous voulez implémenter. Chaque méthode porte un nom assez explicite, et je pense que vous devriez vous y retrouver. 😊

Étendre une autre classe

Si, pour une raison quelconque, votre listener ne peut pas hériter de *Doctrine_EventListener* (parce qu'il hérite déjà d'une autre classe, par exemple), il va falloir écrire en dur chaque méthode.

Explications

La seule condition pour créer un listener, c'est en fait qu'il **implémente l'interface *Doctrine_EventListener_Interface***. Comme vous avez pu le voir en ouvrant le fichier [EventListener.php](#), la seule chose que fait la classe *Doctrine_EventListener* est d'implémenter cette interface. Elle définit toutes les méthodes. Ainsi, la classe fille (*CustomListener*) implémente indirectement cette même interface, tout en n'étant pas obligée de redéfinir toutes les méthodes.

Donc, si vous n'utilisez pas toutes les méthodes dans votre listener, vous devrez quand même les définir !

Un exemple de code sera certainement plus parlant :

Code : PHP

```
<?php

class CustomListener2 extends UneAutreClasse implements
Doctrine_EventListener_Interface
{
    public function preTransactionCommit(Doctrine_Event $event)
```

```

    { }

    public function postTransactionCommit(Doctrine_Event $event)
    { }

    // Et ainsi de suite avec toutes les méthodes imposées par
    l'interface...
}

```

Bien sûr, c'est ici que vous personnalisez chaque méthode comme vous le voulez.

Le « faux » listener...

La dernière manière de créer un listener, est d'implémenter l'interface *Doctrine_Overloadable*. Cette interface n'impose qu'une chose : l'implémentation de la méthode magique `__call()`.

Cette méthode sera appelée lors de chaque évènement, à vous ensuite de traiter tout ça comme vous le voulez. 😊



Pour rappel, la méthode `__call()` reçoit deux paramètres : le nom de la méthode appelée, et un tableau des arguments.

Code : PHP

```

<?php
class CustomListener3 implements Doctrine_Overloadable
{
    public function __call($method, $arguments)
    {
        $event = $arguments[0];
    }
}

```

Attacher un listener

La dernière chose à savoir, c'est comment *lier* le listener à l'objet écouté.

Ceci se fait d'une manière très simple : *Doctrine_Connection* possède la méthode *addListener()* qui prend en paramètre... un listener !

Un exemple est plus simple que de longues explications :

Code : PHP

```

<?php

$listener1 = new CustomListener();
$listener2 = new CustomListener2();

$connexion->addListener($listener1);
$connexion->addListener($listener2);

```

Comme vous le voyez, il est possible d'ajouter plusieurs listeners. Lors d'un évènement, ils seront appelés dans le même ordre que celui dans lequel ils ont été affectés.



Notez que s'il n'y a qu'un seul listener, vous pouvez utiliser la méthode *setListener()*. Cela supprime également d'éventuels listeners existants.

Les listeners attachés à *Doctrine_Connection* écoutent également les classes proches, comme *Doctrine_Transaction* et *Doctrine_Connection_Statement*. Néanmoins, le listener est global à ces trois classes, et doit être attaché à *Doctrine_Connection*.

Je vous invite à vous référer à [la documentation officielle](#) (in english of course 😊) pour avoir le détail de chaque méthode. Dans les tableaux, la colonne de gauche indique le nom de la méthode du listener, celle du milieu indique quelle méthode est écoutée, et la troisième contient d'éventuels paramètres accessibles de l'objet *Doctrine_Event*.

Par exemple, lors de l'appel de `Doctrine_Connection::connect()`, avant toute action, la méthode *preConnect()* du/des listener(s) est appelée. Une fois que toutes les actions sont effectuées, c'est au tour de la méthode *postConnect()*.

Je vous rappelle aussi que toutes les méthodes reçoivent un objet *Doctrine_Event* en paramètre.

Je ne m'attarde pas plus sur les listeners de connexion, ça ne vous sera pas forcément utile pour une utilisation basique.

Listeners d'objets

Nous allons voir ici des listeners que vous trouverez certainement plus utiles. Ceux-ci sont appelés par les classes *Doctrine_Record* et *Doctrine_Validator*.

Créer un listener

De la même manière que pour les connexions, il y a trois façons de créer un listener :

- hériter de la classe *Doctrine_Record_Listener* ;
- implémenter l'interface *Doctrine_Record_Listener_Interface* ;
- implémenter l'interface *Doctrine_Overloadable*.

La manière de créer cette classe est identique à *Doctrine_Connection*, référez-vous donc au chapitre précédent. 😊

Voici le lien vers la doc pour plus de détails et la liste des méthodes : www.doctrine-project.org.

Attacher un listener

Notre listener peut être attaché au niveau global, au niveau d'une connexion ou au niveau d'une table.

Globalement

Il faut l'attacher à *Doctrine_Manager*. Toutes les tables de toutes les connexions en bénéficieront alors.

Code : PHP

```
<?php

class CustomRecordListener extends Doctrine_Record_Listener
{ /* ... */ }

Doctrine_Manager::getInstance()->addRecordListener(new
CustomRecordListener());
```

Par connexion

En attachant le listener à une instance de *Doctrine_Connexion*, toutes les tables de **cette** connexion sont affectées. Néanmoins, je pense que la plupart du temps, vous n'avez qu'une connexion à une seule base de données dans votre projet. Dans ce cas, cela équivaut à l'attacher globalement.

Code : PHP

```
<?php
$connexion->addRecordListener(new CustomRecordListener());
```

Par table

C'est certainement la fonctionnalité la plus intéressante, ou du moins celle dont vous vous servirez le plus. Il est possible d'attacher un listener à une table particulière.

Code : PHP

```
<?php

Doctrine_Core::getTable('Article')
    ->addRecordListener(new CustomRecordListener());

// Équivalent à :

class Article extends Doctrine_Record
{
    public function setUp()
    {
        $this->addListener(new CustomRecordListener());
    }
}
```

Ceci nous sera extrêmement utile, notamment, avec l'utilisation de templates...

Les hooks

Je vous en avais parlé en introduction, les hooks sont des listeners « simplifiés ». Ce sont de simples méthodes implémentées directement dans l'objet.

Code : PHP

```
<?php

class Article extends Doctrine_Record
{
    public function preSave($event)
    {
        // On fait ce que l'on veut juste avant l'enregistrement de
        l'objet.
    }
}
```

Comme vous le voyez, les hooks sont plus simples à utiliser que les listeners. Cependant, lorsque vous devez en utiliser beaucoup, préférez les listeners, cela permet une meilleure organisation du code.

Intercepter les requêtes

Il existe deux méthodes pour intercepter les requêtes : avec des hooks et dans les listeners.

Elles sont cependant identiques, et consistent en ces trois méthodes :

- **preDqlSelect()** est appelée lors d'une requête de type **SELECT** ;
- **preDqlUpdate()** est appelée lors d'une requête de type **UPDATE** ;
- **preDqlDelete()** est appelée lors d'une requête de type **DELETE**.

Leur fonctionnement est identique aux autres méthodes vues précédemment. Notez cependant que `Doctrine_Event::getQuery()` retourne dans ce cas la requête elle-même, ce qui offre la possibilité de la modifier.

Code : PHP

```
<?php

class CustomRecordListener extends Doctrine_Record_Listener
{
    public function preDqlSelect(Doctrine_Event $event)
    {
        $event->getQuery()->addWhere('ce que vous voulez...');
    }
}
```

Vous pouvez ainsi filtrer des données lors d'un **SELECT**, par exemple.

Attention !



Par défaut, ces trois méthodes ne sont pas appelées ! Pour les utiliser, il faut le spécifier explicitement, comme ceci :

```
<?php $manager->setAttribute(Doctrine_Core::ATTR_USE_DQL_CALLBACKS, true);
```

En effet, ceci ajoute une surcharge de travail, puisque Doctrine doit, pour chaque requête, appeler ces callbacks pour chaque table incluse dans la clause **FROM**.

Rendez-vous dans la partie Annexes pour plus de détails sur les attributs de *Doctrine_Manager*.

Informations supplémentaires

Interrompre l'action en cours

Il est possible d'annuler totalement l'évènement, grâce à la méthode `Doctrine_Event::skipOperation()`.

Code : PHP

```
<?php

class CustomRecordListener extends Doctrine_Record_Listener
{
    public function preSave(Doctrine_Event $event)
    {
        // Vous faites ce que vous voulez ici...

        $event->skipOperation();
    }
}

class Article extends Doctrine_Record
{
    public function setUp()
    {
        $this->addListener(new CustomRecordListener());
    }
}
```

Attention, avec cet exemple, vos objets ne pourront pas être enregistrés avec la méthode `save()`... 😊

Ne pas exécuter le listener suivant

Lorsque vous ajoutez plusieurs listeners à une classe, il peut être utile, lors de l'appel de l'un d'entre eux, que le suivant (celui qui a été attaché juste après) ne soit pas appelé.

Ceci est permis par `Doctrine_Event::skipNextListener()`.

Code : PHP

```
<?php

class CustomRecordListener1 extends Doctrine_Record_Listener
{
    public function preSave(Doctrine_Event $event)
    {
        /* ... */

        $event->skipNextListener();
    }
}

class CustomRecordListener2 extends Doctrine_Record_Listener
{
    public function preSave(Doctrine_Event $event)
    { /* ... */ }
}

$record->addListener(new CustomRecordListener1());
$record->addListener(new CustomRecordListener2());

$record->save();
```

En temps normal, les méthodes `preSave()` de chaque listener sont appelées dans l'ordre (`CustomRecordListener1::preSave()` puis `CustomRecordListener2::preSave()`). Cependant ici, `CustomRecordListener1::preSave()` demande de « sauter » le prochain listener. `CustomRecordListener2::preSave()` ne sera donc jamais exécutée.

Ce petit chapitre est maintenant terminé et il vous a permis de découvrir les *Events Listeners* de Doctrine. Bien que ce chapitre soit très théorique, dans le prochain nous étudierons les templates. Ce qui permettra de mettre en pratique avec un TP tout ce que vous avez appris : vous ajouterez un listener à un template... 😊

Les templates

Nous allons maintenant nous attaquer à une partie importante de Doctrine : les *templates* (on parle aussi de *behavior*). Ces *behaviors* vont nous être utiles lorsque nous avons plusieurs modèles qui partagent des points communs. Ces points communs peuvent être :

- des colonnes de la base de données ;
- des algorithmes ;
- des relations ;
- n'importe quoi d'autre... 😊



Dans ce tutoriel, j'utiliserai indifféremment les termes *template* ou *behavior*. Littéralement, *template* signifie *modèle* alors que *behavior* se traduirait par *comportement*.

Introduction

Qu'est-ce qu'un template ?

De manière générale, en informatique, un *template* est un modèle. Comme vous le savez, un programmeur est fainéant (M@teo21 le répète pas mal dans ses tutoriels 😊). Cette fainéantise le pousse à détester une chose plus que tout : la **duplication de code**.

Cette réécriture de code identique (ou presque) se retrouve dans un grand nombre de concepts différents liés à l'informatique (ou pas, d'ailleurs 😊), et le principal inconvénient est une baisse de la maintenabilité de l'application.

Exemples

Peut-être utilisez-vous un système de templates dans votre application afin de séparer le code PHP de la mise en page (en HTML ou autre). Ceci offre un avantage considérable : l'utilisation, par exemple, d'une page HTML sans se soucier de comment sont récupérées les données. On se contente de les afficher. Cette page HTML (*template*) peut alors être réutilisée par plusieurs scripts différents, tout comme le script peut envoyer ses informations à plusieurs pages HTML différentes.

Un autre exemple, qui n'existe pas en PHP dû à son typage dynamique faible, mais dans d'autres langages comme le C++, est le template de fonctions. Cela permet d'outrepasser le fait qu'une fonction ne peut accepter qu'un type de paramètre défini à l'avance. Mais je ne m'étends pas trop là-dessus, puisque nous avons affaire au PHP.

Et Doctrine dans tout ça ?

Lien avec Doctrine

Après ces brèves explications concernant les templates, vous ne voyez peut-être pas le rapport avec Doctrine. Cherchez bien : où pourrions-nous avoir de la *duplication* de code dans notre projet ?

Vous ne voyez pas ? Dans notre schéma pardi ! 😊

Bien souvent, dans une application relativement complète, vos objets devront être datés. Ce peut être le cas d'articles, de commentaires liés à l'article, etc. Prenez par exemple Facebook (au hasard 😊), où toutes les actions sont datées : la mise en ligne de photos, publication de messages, ajout d'amis, participation à des événements...

Ajouté : 5 août

dimanche

il y a 7 minutes

Imaginez que les développeurs veuillent changer quelque chose dans la gestion de la datation des objets. Vous croyez qu'ils vont s'embêter à modifier toutes les classes une par une ? 😊 Les pauvres !

Bref, je vous ai assez fait mariner comme ça. Avec Doctrine, nous allons pouvoir définir des modèles de classes. Lorsque vous aurez plusieurs objets (articles, etc.) à dater, vous aurez juste à dire « *je veux le dater !* » et Doctrine s'en occupera automatiquement pour vous. Elle est pas belle la vie ? 😊

Quand dois-je utiliser les templates ?

Dès qu'il y a quelque chose de commun entre plusieurs classes, utilisez-les ! Votre application n'en sera que plus simple à maintenir en cas de changement.

Un *point commun* peut être des colonnes identiques (par exemple, une colonne *date*), un comportement, des relations, etc.



Je ne comprends pas. D'habitude, lorsque j'ai des points communs entre des classes, j'utilise l'héritage, ce n'est pas comme ça qu'il faut faire ?

Et bien, ça dépend. (Quoi, vous vous attendiez à une autre réponse ? 🤔)

Lorsque des classes ont entre elles une réelle relation parent-enfant, alors oui, l'héritage est là pour ça et il faut l'utiliser. Mais lorsque ce n'est pas le cas, mais qu'il y a des points communs, il paraît stupide de dupliquer du code. De plus, PHP lui-même, en ne supportant pas l'héritage multiple, nous limite.

La solution pour pallier cela : les *templates* ! 😊

Utilisation des templates

Doctrine propose plusieurs *behaviors* intégrés par défaut. Nous allons commencer par en étudier quelques-uns afin de voir comment ça marche.



J'utiliserai le plus souvent le terme *objet* pour définir une instance d'une classe du modèle (comme *Article*), et le terme *table* pour une instance d'une classe représentant une table (comme *ArticleTable*). Il s'agit d'un abus de langage, mais ça ne vous gênera pas pour comprendre. 😊

Découverte des behaviors

Il est très simple d'appliquer un *behavior* sur nos objets. Il faut l'indiquer dans le schéma de données.

Schéma en Yaml

Il suffit d'une ligne dans le schéma Yaml pour appliquer un behavior :

Code : Autre - ~/config/schema.yml

```
Article:
  actAs:
    Timestampable: ~
  columns:
    title:
      type: string(255)
      notnull: true
# ...
```

J'ai pris comme exemple le template nommé *Timestampable* (nous l'étudierons juste après). Comme vous le voyez, nous ajoutons une clé *actAs* qui indique la liste des behaviors et de leurs options. Ici, il n'y a pas d'option particulière (d'où le ~, qui est équivalent à *null*, pour rappel).

Notez que si aucun de vos templates n'a d'option, vous pouvez utiliser la notation simple de tableau :

Code : Autre

```
Article:
  actAs: [Timestampable, UnAutreTemplate]
# ...
```

Schéma en PHP

En PHP, il est recommandé de placer la déclaration dans la méthode `setUp()` :

Code : PHP - ~/lib/models/generated/BaseArticle.php

```
<?php
class BaseArticle extends Doctrine_Record
{
    /* ... */

    public function setUp()
    {
        /* ... */

        $this->actAs('Timestampable');
    }
}
```

Pour ceux qui génèrent les classes de base à partir du Yaml, vous aurez peut-être remarqué que le code généré n'est pas tout à fait identique :

Code : PHP

```
<?php
$timestampable0 = new Doctrine_Template_Timestampable();
$this->actAs($timestampable0);
```

C'est un objet qui est passé à la méthode `actAs()`. En effet, les templates sont des classes nommées de la forme `Doctrine_Template_NomDuTemplate`. Si vous passez juste son nom, Doctrine essaiera de l'instancier lui-même.

Nous allons maintenant passer en revue les templates intégrés.

Timestampable

C'est probablement le template le plus connu. Il donne le moyen de dater facilement les objets (date de *création* et date de *mise à jour*).

Tous vos objets déclarés comme *timestampables* se voient ajouter deux nouvelles colonnes : `created_at` et `updated_at`. Notez qu'il est possible de personnaliser leur nom, ou même de n'utiliser que l'une des deux. Jetez un œil :

Code : Autre

```
# ...
actAs:
  Timestampable:
    created:
      disabled: true
    updated:
      name: updated_date
      onInsert: false
      format: Y-m-d
# ...
```

Voici la correspondance en PHP :

Code : PHP

```
<?php
/* ... */
$this->actAs('Timestampable', array(
    'created' => array(
        'disabled' => true
    ),
    'updated' => array(
        'name'      => 'updated_date',
        'onInsert'  => false,
        'format'    => 'Y-m-d',
    )
));
/* ... */
```

Les options sont passées en deuxième argument de *actAs()*, qui lui-même les passe au constructeur du template.

Pour voir toutes les options disponibles, je vous invite à ouvrir le fichier

[~/lib/vendor/doctrine/Doctrine/Template/Timestampable.php](#). Les options par défaut sont définies dans l'attribut `$_options`.

Voyez ci-dessous pour quelques explications sur chacune.

(Notez que cela n'a pas beaucoup de sens de désactiver le champ *created_at* et de mettre l'option *onInsert* de *updated* à *false*, c'est juste un exemple. 🤔)

Un template possède un comportement très proche d'un objet. Il a, entre autres, la possibilité d'avoir des listeners. Je ne vous en parle pas plus pour l'instant, vous comprendrez leur fonctionnement lorsque nous en créerons un par la suite. 😊

Sachez simplement que cela donne la possibilité d'intervenir juste avant certains événements, par exemple juste avant la sauvegarde d'un objet dans la base de données. Et c'est précisément comme ça que le template *Timestampable* fonctionne : juste avant que vous sauvegardiez votre objet (méthode *save()* par exemple), Doctrine va mettre à jour ces champs avec la valeur de *timestamp* courante.

Vous n'avez donc jamais à vous soucier de ces champs. La seule chose qui compte pour vous, c'est de pouvoir les récupérer. Et cela se fait tout naturellement en y accédant comme on le ferait pour n'importe quel autre champ :

Code : PHP

```
<?php
echo $article->created_at;
```

Rien qu'avec ce premier exemple, assez banal en fait, vous voyez la subtilité et la puissance des *behaviors*. Doctrine fournit un comportement par défaut qui correspond à la plupart des besoins, mais en même temps vous laisse la possibilité de personnaliser chaque détail.

Options disponibles

Les options suivantes sont personnalisables. Elles sont identiques pour chacun des deux champs (*created* et *updated*).

- **name** : le nom de la colonne (par défaut, *created_at* et *updated_at*).
- **alias** : un alias pour chaque colonne. Par défaut il n'y en a pas.
- **type** : type du champ, par défaut *timestamp*.
- **format** : le format (nécessaire pour un champ de type *timestamp/date*). Par défaut *Y-m-d H:i:s*.
- **disabled** : si égal à *true*, désactive cette colonne. Vaut *false* par défaut.
- **expression** : donne la possibilité d'utiliser une *Doctrine_Expression* pour le champ. Désactivé par défaut.
- **options** : options à passer pour la définition de la colonne. Par défaut, indique seulement que la colonne ne peut être vide (**NOTNULL**).
- **onInsert** : utilisé uniquement sur le champ *updated*. Indique s'il doit être mis à jour aussi lors de la création de l'objet. Par défaut, vaut *true*.

SoftDelete

Ne vous êtes-vous jamais dit « *Ah mince ! Je voulais le garder !* » après avoir supprimé un article (ou n'importe quoi d'autre) ? SoftDelete est là pour vous !



Je ne vais pas remettre à chaque fois le code de l'implémentation Yaml ou PHP, ça fonctionne à chaque fois de la même manière. 😊 Seules les options sont variables.

Ce template vous permet de ne pas supprimer réellement vos objets. Il ajoute une colonne `deleted_at` dans votre modèle, qui indique la date à laquelle l'objet a été supprimé.

À chaque fois que vous ferez `<?php $objet->delete()`, votre objet ne sera pas supprimé. À la place, sa colonne `deleted_at` sera remplie avec la date courante.



Oui mais c'est pas très pratique tout ça, maintenant à chaque fois que je récupérerai un objet depuis ma base de données, il faudra que je vérifie s'il a été supprimé ou non. C'est nul ton truc !

Vous croyez donc que notre ORM préféré ne gère pas ça ? Erreur ! 😡

En effet, ce template ajoute un « filtrage » automatique de tout ce que vous sélectionnez dans la base. Et il ne vous renverra pas les éléments *virtuellement* supprimés.

De votre côté, vous ne vous en souciez donc vraiment plus.

Pour cela, il faut changer la valeur d'un attribut de `Doctrine_Manager` :

Code : PHP

```
<?php
Doctrine_Manager::getInstance()
    ->setAttribute(Doctrine_Core::ATTR_USE_DQL_CALLBACKS, true);
```

En effet, ce template utilise les *callbacks* DQL dans un listener. Je ne détaille pas plus pour l'instant, mais si vous avez lu le chapitre précédent, vous devriez comprendre de quoi il s'agit. 😊

Placez ceci dans `~/config/global.php` par exemple.

Notez que vous pouvez quand même récupérer ces enregistrements supprimés : il faut pour cela le spécifier explicitement dans une requête :

Code : PHP



```
<?php
$deletedArticles = Doctrine_Query::create()
    ->from('Article a')
    ->where('a.deleted_at IS NOT NULL') // Si deleted_at
    contient une valeur (une date), c'est que l'article est
    supprimé.
    ->execute();
```

Vous avez aussi la possibilité de supprimer définitivement vos objets :

Code : PHP

```
<?php
foreach($deletedArticles as $article)
{
    $article->hardDelete();
}
```

Comme vous le voyez, le template ajoute aussi une méthode *hardDelete()* aux objets (oui, un template, ça peut faire ça aussi 😊). Maintenant, si vous supprimez des enregistrements, c'est parce que vous l'aurez bien voulu ! 😊

Options disponibles

- **name** : nom de la colonne. Par défaut, c'est *deleted_at*.
- **type** : type de la colonne, *timestamp* par défaut. Vous pouvez aussi utiliser *boolean* pour simplement indiquer si l'article est supprimé ou non, sans la date.
- **hardDelete** : indique si les enregistrements doivent être supprimés définitivement. Par défaut, vaut *false*. Je ne vous conseille pas de le mettre à *true*, sinon le template n'est pas extrêmement utile...
- **options** : options à passer lors de la construction de la colonne.

Versionable

Ce template vous donne la possibilité de *versionner* vos objets. C'est-à-dire que vous conservez tout l'historique des modifications apportées à chaque objet, et que vous avez à tout moment la possibilité de revenir en arrière et d'annuler des modifications.

En fait, Doctrine va créer une deuxième table (dont vous ne vous servirez jamais directement), qui va stocker toutes les anciennes versions. Cette table est nommée avec le suffixe '_version'. Une colonne *version* est aussi ajoutée, qui stocke le numéro de version de l'objet.

Du point de vue utilisateur (c'est-à-dire vous 😊), vous ne vous apercevez de presque rien : vous voyez le numéro de version de votre objet s'incrémenter chaque fois que vous y apportez des modifications et le sauvegardez.

En interne, à chaque modification/suppression, l'ancienne version est déplacée dans la table dédiée, et la dernière version est enregistrée normalement dans la table avec un numéro de version incrémenté.

Pour finir, vous avez la possibilité de restaurer une ancienne version, grâce à la méthode *revert()* :

Code : PHP

```
<?php
// Nous restaurons l'objet tel qu'il était dans sa version N°2.
$objet->revert(2);
```



Attention cependant à ne pas abuser de ce behavior. Le fait d'ajouter une table supplémentaire pour chaque table *versionnable*, et de sauvegarder toutes les versions, peut finir par faire augmenter considérablement la taille de votre base de données.

Options disponibles

Je vous invite, comme toujours, à aller voir les options disponibles dans <~/lib/vendor/doctrine/Doctrine/Template/Versionable.php>.

Voici quelques explications :

- **generateFiles** : Doctrine génère une classe supplémentaire pour la table de version. Par défaut, ces classes ne sont pas créées « en dur », mais générées à la volée. Vous pouvez forcer à créer ces fichiers en passant cette option à *true*.
- **deleteVersions** : Indique tout simplement si oui ou non les anciennes versions doivent être supprimées lorsque la

version actuelle est supprimée, ou si elles doivent être conservées.

Sluggable

Si vous avez déjà eu envie de faire apparaître le titre de vos articles (par exemple) dans leur URL, vous vous êtes sûrement heurté au fait que certains caractères posent problème.

Doctrine apporte une solution à cela aussi, avec le template *Sluggable*. Celui-ci va ajouter une colonne *slug* qui contiendra le titre de vos articles, mais adapté à une URL (suppression des caractères spéciaux, etc.).

Pour indiquer sur quels champs doit être basé le *slug*, il faut renseigner l'option *fields*. Dans le cas de notre article, nous indiquerons par exemple la colonne *title*.

Vous avez aussi la possibilité de forcer cette colonne à être unique. De cette manière, vous pourrez retrouver un objet précis sans utiliser son ID, mais uniquement avec son *slug*. Pour cela, passez l'option *unique* à *true*. Si plusieurs objets ont le même slug, alors un chiffre sera ajouté pour les différencier.

Vous avez la possibilité de déterminer un slug unique avec des champs particuliers : indiquez-les avec *uniqueBy*.

Options disponibles

Voici un résumé des options :

- **unique** : si oui ou non l'unicité du champ doit être garantie. *True* par défaut.
- **fields** : champs à utiliser pour le *slug*.
- **uniqueBy** : champs par lesquels l'unicité est garantie.
- **uniqueIndex** : si oui ou non l'index est créé sur la colonne.
- **indexName** : nom à donner à l'index.
- **canUpdate** : si le slug peut être mis à jour à chaque modification de l'objet, ou s'il est défini une fois pour toutes.
- **builder** : fonction à utiliser pour construire le slug.
- **provider** : fonction à utiliser pour récupérer une *valeur* représentative de l'objet pour construire le slug. Est utilisé uniquement si aucun champ n'est indiqué dans l'option **fields**.

Structure d'un template

J'espère que vous voyez maintenant un peu mieux ce qu'est un template, et à quoi il sert. Voyons un peu à présent comment tout cela est structuré. Vous pourrez ainsi créer vos propres templates. Alors, lisez avec attention ce qui suit ! 🤖

Comment ça marche ?

Si vous avez lu le début du paragraphe sur l'utilisation des templates, vous devez vous souvenir qu'un template est une **classe** (par exemple, *Doctrine_Template_Timestampable*). Ouvrez donc le fichier [~/lib/vendor/doctrine/Doctrine/Template/Timestampable.php](#) si vous ne l'avez pas déjà fait, vous suivrez mieux ce qui va suivre.

Nous allons étudier cette classe en détail, elle nous servira d'exemple.

Doctrine_Template

Première chose que vous remarquez, elle hérite d'une classe de base : *Doctrine_Template*. Cette classe fournit les éléments essentiels au template :

- Des *mutateurs* pour accéder à l'*invoker*, c'est-à-dire le dernier enregistrement à avoir utilisé ce template : **setInvoker(\$invoker)** et **getInvoker()**.
- La possibilité d'utiliser un plug-in. Un plug-in est utile lorsque le template implique de faire appel à un fonctionnement relativement complexe. Par exemple, le template *Doctrine_Template_Searchable* (je ne vous en ai pas encore parlé) fait appel à *Doctrine_Search*. Le fonctionnement de ce plug-in est indépendant du template, et peut être utilisé de l'extérieur. Je vous laisse trouver les méthodes correspondantes... 🤖 (*Doctrine_Template::getPlugin()* etc.)
- Un accesseur vers la table de l'invoker : *getTable()*. On récupère simplement une instance de *Doctrine_Table*.

Héritage

Doctrine_Template possède aussi deux méthodes vides : *setUp()* et *setTableDefinition()*. Comme pour le modèle, ces méthodes servent à définir, entre autres, la table et les relations.

Les méthodes *hasColumn()*, *hasOne()*, *hasMany()*... sont à utiliser ici.

Et là vous allez me dire...



Eh ! Mais ça ressemble pas un peu à *Doctrine_Record* ça ?

Si !

Lien avec *Doctrine_Record*

Si vous êtes un tant soit peu curieux, vous aurez remarqué que *Doctrine_Template* et *Doctrine_Record* étendent tous les deux... *Doctrine_Record_Abstract* !



En effet, il faut bien comprendre qu'un objet est fusionné avec les templates qui lui sont associés (c'est une image hein 😊). Cela lui permet de partager tout ce que l'on vient de voir, comme des relations, etc. ainsi que des méthodes !

Méthodes déléguées

Un template offre la possibilité d'ajouter des méthodes aux objets et aux tables.

Ajout de méthodes aux objets

Il existe un moyen très simple d'ajouter des méthodes à nos objets : je vous ai dit qu'il fallait considérer un objet et ses templates comme étant fusionnés. Il suffit donc de définir une méthode dans un template, pour qu'elle soit accessible dans tous les objets l'implémentant !

Code : PHP

```

<?php

class Doctrine_Template_Timestampable extends Doctrine_Template
{
    /* ... */

    public function isOlderThan($date)
    {
        $date      = new DateTime($date);
        $created   = new DateTime($this->getInvoker()->{$this->
        >_options['created'] ['name']});

        return $date->getTimestamp() > $created->getTimestamp();
    }
  
```



```
}
```

Nous imaginons ici une méthode *isOlderThan()* qui indique si la date de création de l'objet est plus ancienne que la date passée en paramètre.

Maintenant, nous pouvons utiliser cette méthode comme ceci :

Code : PHP

```
<?php

if($article->isOlderThan("yesterday")) {
    echo "Cet article a été publié avant hier.";
}
else {
    echo "Cet article a été publié hier ou aujourd'hui.";
}
```

Difficile de faire plus simple, non ? 😊

Ajout de méthodes aux tables

Vous avez aussi la possibilité d'ajouter des méthodes aux classes représentant les tables (*ArticleTable*, etc.). Il y a simplement une petite subtilité pour les différencier.

Voiez plutôt :

Code : PHP - ~/lib/models/templates/Doctrine_Template_Publishable.php

```
<?php

class Doctrine_Template_Publishable extends Doctrine_Template
{
    /* ... */

    public function findArticlesOlderThanTableProxy($date)
    {
        $date = new DateTime($date);

        return $this->getTable()
            ->createQuery('a')
            ->where('a.'. $this->_options['created']['name'].' < ?',
                $date->getTimestamp())
            ->execute();
    }
}
```

Vous l'aurez deviné, lorsque vous voulez rendre accessible une méthode depuis la classe de table, il faut suffixer son nom de *TableProxy*.

Et vous pouvez l'utiliser comme ceci :

Code : PHP

```
<?php

$table = Doctrine_Core::getTable('Article');

// Ne pas mettre 'TableProxy' lors de l'utilisation de la méthode.
$articles = $table->findArticlesOlderThan("yesterday");
```

J'espère vous avoir mis l'eau à la bouche avec ce chapitre, parce que maintenant, tout ceci va être mis en pratique. 😊

Dans le prochain chapitre, vous créerez votre propre template. Un lien sera également fait avec le chapitre précédent, en lui attachant un listener.

TP : Création d'un template

Les deux chapitres précédents étaient assez théoriques. Ici, vous allez mettre en pratique tout ça ! 😊

Créer son propre template

Maintenant que vous voyez un peu comment les templates fonctionnent, et à quoi ils servent, vous allez certainement vouloir en créer de nouveaux adaptés à vos besoins.

Introduction

Nous allons détailler ici toutes les étapes de la création de notre template.

Je vais continuer avec l'exemple de notre article, que j'utilise depuis le début du tutoriel. Nous voulons maintenant que le nom de l'auteur original (celui qui a créé l'article), et le nom de la dernière personne à l'avoir modifié, soient enregistrés automatiquement. Nous allons aussi ajouter un champ qui indiquera si l'article est disponible ou non en lecture pour les visiteurs. Le fonctionnement ressemblera à *SoftDelete* et cela permettra d'assimiler un peu mieux le fonctionnement des événements.

Je vous propose de nommer notre template **Publishable**. 😊



Avant de commencer, vous pouvez supprimer la colonne *user_id* et la relation Article-User que nous avons définies dans le schéma. N'oubliez pas ensuite de tout reconstruire en appelant `~/web/builder.php`.

Création de la classe

Je vous ai dit dans le chapitre précédent que le fonctionnement des *templates* ressemblait à celui des objets *Doctrine_Record*. Et pour cause : *Doctrine_Template* hérite de la même classe de base, *Doctrine_Record_Abstract*. Ils utilisent donc les mêmes méthodes pour se configurer : *setUp()* et *setTableDefinition()*.

Nous définirons donc les colonnes à ajouter à l'intérieur de *setTableDefinition()*, tandis que *setUp()* servira à définir les relations.

Nous allons placer tous nos templates dans `~/lib/models/templates/`.

Code : PHP - `~/lib/models/templates/Doctrine_Template_Publishable.php`

```
<?php

class Doctrine_Template_Publishable extends Doctrine_Template
{ }
```



Notez que j'ai préfixé le nom du template de *Doctrine_Template_*, mais vous n'y êtes pas obligés. Si vous utilisez un préfixe particulier dans votre projet, vous pouvez tout à fait l'utiliser, ou même l'appeler *Publishable* tout court. Seul petit détail : si vous placez un préfixe (autre que *Doctrine_Template_*), il vous faudra utiliser le nom complet lors de la définition avec *actAs()*, sinon Doctrine ne pourra pas le trouver et le charger automatiquement. À moins de définir votre propre autoloader... mais ça n'entre pas dans le cadre de ce cours. Bref, vous faites comme vous voulez. 😊

Détails supplémentaires

Vous avez le code de base de la classe, maintenant, à vous de jouer ! 😊

Trois colonnes seront nécessaires pour ce template. Nommez-les comme vous le voulez. 😊 Deux d'entre elles contiendront des références à des utilisateurs (par exemple leur *ID*). La troisième sera simplement de type booléen.

Notez qu'il n'est pas obligatoire du tout d'utiliser l'attribut `$_options`. Néanmoins, je vous encourage à le faire, la personnalisation depuis le schéma étant impossible sinon. À vous de choisir quelles options implémenter (aidez-vous des

options « classiques » que vous trouverez dans les autres templates existants 😊).

Les colonnes supplémentaires sont à ajouter dans la méthode `setTableDefinition()`, et les relations à l'intérieur de `setUp()`.

Ajout d'un listener

Je vous propose d'ajouter un listener au template. Il servira notamment pour la mise à jour automatique de l'auteur de l'article.

Pour rappel, il doit hériter de `Doctrine_Record_Listener`.

Créez un répertoire `~/lib/models/templates/listeners/`. Nous placerons nos listeners ici.

Je vous propose d'appeler notre classe `Doctrine_Template_Listener_Publishable`.

Pour le construire, commencez par vous demander quelles méthodes seront utiles (c'est-à-dire quels événements intercepter).

Correction : le template

Vous avez bien travaillé ? Vous n'avez pas trouvé comment faire ?

Voici ci-dessous la correction détaillée.

Les options

Tout d'abord, voici un exemple d'options possibles. Bien sûr, ce n'est pas LA correction unique, vous mettez les options que vous voulez ! 😊

Code : PHP - `~/lib/models/templates/Doctrine_Template_Publishable.php`

```
<?php

class Doctrine_Template_Publishable extends Doctrine_Template
{
    protected $_options = array(
        'created' => array(
            // Nom de la colonne 'auteur original'.
            'name' => 'created_by',
            // Alias. On pourra faire $article->OriginalAuthor.
            'alias' => 'OriginalAuthor',
            // Classe représentant un auteur.
            'class' => 'User',
            // Champ de la classe User servant à la relation.
            'foreign' => 'id',
            // Type du champ, on stocke l'ID, donc integer.
            'type' => 'integer',
            // On donne la possibilité de ne pas utiliser cette
            colonne.
            'disabled' => false,
            // Options supplémentaires pour la colonne (nonnull,
            default, etc.).
            'options' => array(),
        ),
        // Mêmes options pour la colonne du dernier auteur.
        'updated' => array(
            'name' => 'updated_by',
            'alias' => 'Author',
            'class' => 'User',
            'foreign' => 'id',
            'type' => 'integer',
            'disabled' => false,
            'options' => array(),
        ),
        // Options pour la colonne 'publiable'.
        'publishable' => array(
            'name' => 'is_publishable',
            'alias' => null,
            'type' => 'boolean',
            'disabled' => false,
```

```

        // Par défaut, l'article sera publiable.
        'options' => array('notnull' => true, 'default' =>
true),
    ),
);

/* ... */
}

```

Comme vous le voyez, j'ai repris les options présentes dans la plupart des templates : on donne la possibilité de désactiver certaines colonnes, d'ajouter des alias pour accéder aux auteurs, etc. L'option *class* indique quelle classe sera utilisée pour la relation (c'est-à-dire celle qui représente un auteur, un membre, etc. : toute personne susceptible de rédiger l'article), et *foreign* indique le nom de la colonne sur laquelle la relation est définie (par défaut, *id*).

Les colonnes

Nous allons, comme nous l'avons dit, ajouter deux champs à notre table. Il n'y a rien d'extraordinaire, c'est exactement pareil que lors de la définition du schéma dans les classes de base.

Code : PHP - ~/lib/models/templates/Doctrine_Template_Publishable.php

```

<?php

class Doctrine_Template_Publishable extends Doctrine_Template
{
    /* ... */

    public function setTableDefinition()
    {
        if(!$this->_options['created']['disabled']) {
            $this->hasColumn(
                $this->_options['created']['name'],
                $this->_options['created']['type'],
                null,
                $this->_options['created']['options']
            );
        }

        if(!$this->_options['updated']['disabled']) {
            $this->hasColumn(
                $this->_options['updated']['name'],
                $this->_options['updated']['type'],
                null,
                $this->_options['updated']['options']
            );
        }

        if(!$this->_options['publishable']['disabled']) {
            $name = $this->_options['publishable']['name'];
            if($this->_options['publishable']['alias']) {
                $name .= ' as '.$this->
>_options['publishable']['alias'];
            }

            $this->hasColumn(
                $name,
                $this->_options['publishable']['type'],
                null,
                $this->_options['publishable']['options']
            );
        }
    }
}

```

```

        /* ... */
    }

```

Nous vérifions pour chaque élément qu'il n'est pas désactivé, et nous ajoutons la colonne au modèle en conséquence, en utilisant les options définies plus haut. Encore une fois, rien de spécial, la définition des colonnes se fait de la même manière que dans les classes de base.

Les relations

Nous avons choisi d'enregistrer l'ID de l'auteur original, et l'ID du dernier auteur à avoir modifié l'article. Nous allons donc définir deux relations dans la méthode `setUp()`.

Code : PHP - `~/lib/models/templates/Doctrine_Template_Publishable.php`

```

<?php

class Doctrine_Template_Publishable extends Doctrine_Template
{
    /* ... */

    public function setUp()
    {
        if (!$this->_options['created']['disabled']) {
            $name = $this->_options['created']['class'];
            if ($this->_options['created']['alias']) {
                $name .= ' as '.$this->_options['created']['alias'];
            }

            $this->hasOne($name, array(
                'local' => $this->_options['created']['name'],
                'foreign' => $this->_options['created']['foreign']
            ));
        }

        if (!$this->_options['updated']['disabled']) {
            $name = $this->_options['updated']['class'];
            if ($this->_options['updated']['alias']) {
                $name .= ' as '.$this->_options['updated']['alias'];
            }

            $this->hasOne($name, array(
                'local' => $this->_options['updated']['name'],
                'foreign' => $this->_options['updated']['foreign']
            ));
        }
    }
}

```

Une fois encore, la manière d'indiquer les relations ne diffère pas de lors de la définition du schéma. L'option *name* est utilisée en tant que clé locale, et *foreign* en tant que clé étrangère. *class* indique la classe à lier, et *alias* un éventuel alias (par défaut, *OriginalAuthor* et *Author*).

Voilà, à ce stade, nous avons défini toutes les colonnes et relations nécessaires à notre template. Nous pouvons par exemple accéder au login de la dernière personne ayant modifié un article :

Code : PHP

```

<?php

echo $article->Author->login;

```

```
$article->Author = $monObjetUser;
```

Correction : le listener

Sans le listener associé, notre template serait bien peu pratique. 😊

Réfléchissons à ce que nous voulons obtenir.

- Nous voulons que l'auteur soit ajouté lors du premier enregistrement de notre article. La méthode à utiliser est **preInsert()**.
- Nous voulons aussi qu'à chaque modification, le nom de l'auteur soit également mis à jour. Nous nous servirons de **preUpdate()**. **preDqlUpdate()** sera aussi utilisé, lorsque la mise à jour est effectuée depuis une requête DQL (construite à la main, reportez-vous au chapitre sur la manipulation des données pour plus de détails 😊).
- Pour filtrer automatiquement les articles non publiables afin qu'ils n'apparaissent pas lors d'une sélection de données, **preDqlSelect()** sera ici utile.

Voyons dès à présent le code de base de la classe :

Code : PHP - ~/lib/models/templates/listeners/Doctrine_Template_Listener_Publishable.php

```
<?php

class Doctrine_Template_Listener_Publishable extends
Doctrine_Record_Listener
{
    protected $_options = array();

    public function __construct(array $options)
    {
        $this->_options = $options;
    }

    public function preInsert(Doctrine_Event $event)
    { }

    public function preUpdate(Doctrine_Event $event)
    { }

    public function preDqlUpdate(Doctrine_Event $event)
    { }

    public function preDqlSelect(Doctrine_Event $event)
    { }

    protected function getCurrentUserId()
    {
        // À adapter à votre projet, bien sûr
        return $_SESSION['user_id'];
    }
}
```

Nous avons défini nos quatre méthodes. Nous récupérons aussi les options du template dans l'attribut `$_options` car nous en aurons besoin.

Chacune de ces méthodes reçoit en paramètre l'objet *Doctrine_Event* représentant l'évènement. Il nous sera possible de récupérer les objets concernés à partir de celui-ci.

Notez que nous savons déjà que l'on va avoir besoin de connaître l'utilisateur courant (celui qui rédige et sauvegarde l'article). Le rôle de la méthode *getCurrentUserId()* est de nous renvoyer... son ID. 😊 Bien entendu, la manière de le récupérer va différer selon votre projet. À vous d'adapter cela. 😊



S'il y en a parmi vous qui utilisent **Symfony**, vous pourrez le récupérer avec quelque chose de similaire à `<?php sfContext::getInstance()->getUser()->getId();`.

Pré-insertion d'un article

Nous allons ici travailler dans la méthode `preInsert()`, c'est-à-dire lors du premier enregistrement de l'article.

Code : PHP - ~/lib/models/templates/listeners/Doctrine_Template_Listener_Publishable.php

```
<?php

class Doctrine_Template_Listener_Publishable extends
Doctrine_Record_Listener
{
    /* ... */

    public function preInsert(Doctrine_Event $event)
    {
        // On récupère ce dont on aura besoin.
        $invoker      = $event->getInvoker();
        $invokerTable = $invoker->getTable();
        $modifiedFields = $invoker->getModified();

        // Si cette colonne n'est pas désactivée :
        if(!$this->_options['created']['disabled']) {
            $createdField = $invokerTable->getFieldName($this->
            _options['created']['name']);

            // On s'assure que le champ n'a pas été modifié
            manuellement.
            if(!isset($modifiedFields[$createdField])) {
                // On met le champ à jour.
                $invoker->$createdField = $this->getCurrentUserId();
            }
        }

        // Si cette colonne n'est pas désactivée :
        if(!$this->_options['updated']['disabled']) {
            $updatedField = $invokerTable->getFieldName($this->
            _options['updated']['name']);

            // On s'assure que le champ n'a pas été modifié
            manuellement.
            if(!isset($modifiedFields[$updatedField])) {
                // On met le champ à jour.
                $invoker->$updatedField = $this->getCurrentUserId();
            }
        }
    }

    /* ... */
}
```

L'objet `$event` est capable de nous retourner l'objet concerné avec la méthode `getInvoker()`. `Doctrine_Record` possède une méthode `getTable()` qui nous retourne une instance de cette table (donc, la classe `ArticleTable`).

Petit détail

La méthode `Doctrine_Record::getModified()` retourne la liste des champs de l'objet qui ont été modifiés.

Avant de mettre à jour les champs concernant l'auteur, nous vérifions que ceux-ci n'ont pas été modifiés manuellement (pour une quelconque raison). Si c'est le cas, alors on ne les modifie pas.

Enfin, si la colonne n'est pas désactivée (souvenez-vous que nous avons connaissance des options du template), on la met à jour en récupérant l'ID de l'utilisateur courant.

Mise à jour d'un article

Lors de la mise à jour d'un article, nous modifions seulement la valeur de la colonne *updated*.

Nous allons ici travailler sur deux méthodes : *preUpdate()* lors d'une mise à jour par l'objet directement, et *preDqlUpdate()* lors d'une mise à jour par l'intermédiaire d'une requête DQL.

Premier cas : avec Doctrine_Record

Commençons par *preUpdate()*. C'est identique à précédemment.

Code : PHP - ~/lib/models/templates/listeners/Doctrine_Template_Listener_Publishable.php

```
<?php

class Doctrine_Template_Listener_Publishable extends
Doctrine_Record_Listener
{
    /* ... */

    public function preUpdate(Doctrine_Event $event)
    {
        // On récupère ce dont on aura besoin.
        $invoker      = $event->getInvoker();
        $invokerTable  = $invoker->getTable();
        $modifiedFields = $invoker->getModified();

        // Si cette colonne n'est pas désactivée :
        if(!$this->_options['updated']['disabled']) {
            $updatedField = $invokerTable->getFieldName($this-
            >_options['updated']['name']);

            // On s'assure que le champ n'a pas été modifié
            manuellement.
            if(!isset($modifiedFields[$updatedField])) {
                // On met le champ à jour.
                $invoker->$updatedField = $this->getCurrentUserId();
            }
        }
    }

    /* ... */
}
```

Second cas : avec Doctrine_Query

Voyons maintenant comment gérer ça lors d'une requête DQL :

Code : PHP - ~/lib/models/templates/listeners/Doctrine_Template_Listener_Publishable.php

```
<?php

class Doctrine_Template_Listener_Publishable extends
Doctrine_Record_Listener
{
    /* ... */

    public function preDqlUpdate(Doctrine_Event $event)
```

```

    {
        // Si cette colonne n'est pas désactivée :
        if (!$this->_options['updated']['disabled']) {
            // On récupère ce dont on aura besoin.
            $invokerTable = $event->getInvoker()->getTable();
            $params       = $event->getParams();
            $updatedField = $invokerTable->getFieldName($this->
            >_options['updated']['name']);
            $q             = $event->getQuery();
            $field         = $params['alias'].'.'.$updatedField; //
            Contient par exemple 'a.updated_by'

            // On s'assure que le champ n'a pas été modifié
            manuellement.
            if (!$q->contains($field)) {
                // On met le champ à jour.
                $q->set($field, '?', $this->getCurrentUserId());
            }
        }
    }

    /* ... */
}

```

L'objet `$event` nous permet de récupérer la requête associée avec `getQuery()`. `$params` contient... des paramètres ! En l'occurrence, nous nous intéressons à `$params['alias']` qui contient l'alias utilisé dans la requête (que j'appelle souvent 'a'). Notez par la même occasion que l'alias principal de la requête peut aussi être récupéré avec `<?php $q->getRootAlias()`.

Pour nous assurer que le champ n'a pas été modifié à la main, et ne pas écraser les modifications, nous vérifions que la requête ne le contient pas avec `Doctrine_Query::contains()`. Si tout est bon, alors on peut l'ajouter à notre requête avec `set()`.

Voilà, à ce stade, à chaque fois que l'on créera/modifiera un article, le nom de l'auteur original et du dernier à l'avoir modifié seront automatiquement renseignés. Magique, non ? 🤖

Sélection d'articles

Reste un détail. Nous ne voulons pas qu'un article marqué comme supprimé apparaisse lorsque nous effectuons des sélections dans la base. Il nous faut pour ça modifier la requête à la volée.

Utilisons `preDqlSelect()` :

Code : PHP - `~/lib/models/templates/listeners/Doctrine_Template_Listener_Publishable.php`

```

<?php

class Doctrine_Template_Listener_Publishable extends
Doctrine_Record_Listener
{
    /* ... */

    public function preDqlSelect(Doctrine_Event $event)
    {
        // Si cette colonne n'est pas désactivée :
        if (!$this->_options['publishable']['disabled']) {
            // On récupère ce dont on aura besoin.
            $q       = $event->getQuery();
            $params   = $event->getParams();
            $field    = $params['alias'].'.'.$this->
            >_options['publishable']['name'];

            // On vérifie qu'il faille appliquer la restriction.
            if (!$q->contains($field)) {
                $q->andWhere($field.' = ?', true);
            }
        }
    }
}

```

```

    }
}

/* ... */
}

```

Premièrement, nous nous assurons que la colonne est activée, et nous récupérons les objets dont nous avons besoin. Ensuite, on s'assure que la requête ne contient pas déjà ce champ, et on ajoute une condition sur celui-ci.

Et bien, chers amis, j'ai une bonne nouvelle, notre listener est terminé ! 😊

Conclusion

Notre template ainsi que son listener sont maintenant opérationnels. 😊

Liaison des composants

Il reste juste une chose à régler : l'utiliser dans le template ! En effet, nous n'avons pas encore indiqué au template qu'il doit l'utiliser.

La méthode `addListener()` est là pour ça. Mais vous le saviez déjà, non ? Nous l'appellerons dans `setTableDefinition()` :

Code : PHP - ~/lib/models/templates/Doctrine_Template_Publishable.php

```

<?php

class Doctrine_Template_Publishable extends Doctrine_Template
{
    /* ... */

    public function setTableDefinition()
    {
        /* ... */

        $this->addListener(new
Doctrine_Template_Listener_Publishable($this->_options));
    }

    /* ... */
}

```

Résumé

Je vous donne ci-dessous le code complet du template et du listener. Oh que je suis gentil ! 😊

Doctrine_Template_Publishable :
Secret ([cliquez pour afficher](#))

Code : PHP - ~/lib/models/templates/Doctrine_Template_Publishable.php

```

<?php

class Doctrine_Template_Publishable extends Doctrine_Template
{
    protected $_options = array(
        'created' => array(
            'name' => 'created_by',
            'alias' => 'OriginalAuthor',

```

```

        'class'      => 'User',
        'foreign'    => 'id',
        'type'       => 'integer',
        'disabled'   => false,
        'options'    => array(),
    ),
    'updated' => array(
        'name'      => 'updated_by',
        'alias'      => 'Author',
        'class'      => 'User',
        'foreign'    => 'id',
        'type'       => 'integer',
        'disabled'   => false,
        'options'    => array(),
    ),
    'publishable' => array(
        'name'      => 'is_publishable',
        'alias'      => null,
        'type'       => 'boolean',
        'disabled'   => false,
        'options'    => array('notnull' => true, 'default' =>
true),
    ),
);

public function setTableDefinition()
{
    if (!$this->_options['created']['disabled']) {
        $this->hasColumn(
            $this->_options['created']['name'],
            $this->_options['created']['type'],
            null,
            $this->_options['created']['options']
        );
    }

    if (!$this->_options['updated']['disabled']) {
        $this->hasColumn(
            $this->_options['updated']['name'],
            $this->_options['updated']['type'],
            null,
            $this->_options['updated']['options']
        );
    }

    if (!$this->_options['publishable']['disabled']) {
        $name = $this->_options['publishable']['name'];
        if ($this->_options['publishable']['alias']) {
            $name .= ' as '.$this->
>_options['publishable']['alias'];
        }

        $this->hasColumn(
            $name,
            $this->_options['publishable']['type'],
            null,
            $this->_options['publishable']['options']
        );
    }

    $this->addListener(new
Doctrine_Template_Listener_Publishable($this->_options));
}

public function setUp()
{
    if (!$this->_options['created']['disabled']) {
        $name = $this->_options['created']['class'];
        if ($this->_options['created']['alias']) {
            $name .= ' as '.$this->

```

```

        >_options['created']['alias'];
    }

    $this->hasOne($name, array(
        'local' => $this->_options['created']['name'],
        'foreign' => $this->_options['created']['foreign']
    ));
}

if(!$this->_options['updated']['disabled']) {
    $name = $this->_options['updated']['class'];
    if($this->_options['updated']['alias']) {
        $name .= ' as '.$this->
    >_options['updated']['alias'];
    }

    $this->hasOne($name, array(
        'local' => $this->_options['updated']['name'],
        'foreign' => $this->_options['updated']['foreign']
    ));
}
}
}
}

```

Doctrine_Template_Listener_Publishable :
 Secret ([cliquez pour afficher](#))

Code : PHP - ~/lib/models/templates/listeners/Doctrine_Template_Listener_Publishable.php

```

<?php

class Doctrine_Template_Listener_Publishable extends
Doctrine_Record_Listener
{
    protected $_options = array();

    public function __construct(array $options)
    {
        $this->_options = $options;
    }

    public function preInsert(Doctrine_Event $event)
    {
        $invoker      = $event->getInvoker();
        $invokerTable = $invoker->getTable();
        $modifiedFields = $invoker->getModified();

        if(!$this->_options['created']['disabled']) {
            $createdField = $invokerTable->getFieldName($this->
            >_options['created']['name']);

            if(!isset($modifiedFields[$createdField])) {
                $invoker->$createdField = $this->
            >getCurrentUserId();
            }
        }

        if(!$this->_options['updated']['disabled']) {
            $updatedField = $invokerTable->getFieldName($this->
            >_options['updated']['name']);

            if(!isset($modifiedFields[$updatedField])) {
                $invoker->$updatedField = $this->
            >getCurrentUserId();
            }
        }
    }
}

```

```

    }

    public function preUpdate(Doctrine_Event $event)
    {
        $invoker      = $event->getInvoker();
        $invokerTable = $invoker->getTable();
        $modifiedFields = $invoker->getModified();

        if(!$this->_options['updated']['disabled']) {
            $updatedField = $invokerTable->getFieldName($this->
            >_options['updated']['name']);

            if(!isset($modifiedFields[$updatedField])) {
                $invoker->$updatedField = $this->
                >getCurrentUserId();
            }
        }
    }

    public function preDqlUpdate(Doctrine_Event $event)
    {
        if(!$this->_options['updated']['disabled']) {
            $invokerTable = $event->getInvoker()->getTable();
            $params        = $event->getParams();
            $updatedField = $invokerTable->getFieldName($this->
            >_options['updated']['name']);
            $q             = $event->getQuery();
            $field         = $params['alias'].'.'.$updatedField;

            if(!$q->contains($field)) {
                $q->set($field, '?', $this->getCurrentUserId());
            }
        }
    }

    public function preDqlSelect(Doctrine_Event $event)
    {
        if(!$this->_options['publishable']['disabled']) {
            $q      = $event->getQuery();
            $params = $event->getParams();
            $field  = $params['alias'].'.'.$this->
            >_options['publishable']['name'];

            if(!$q->contains($field)) {
                $q->andWhere($field.' = ?', true);
            }
        }
    }

    protected function getCurrentUserId()
    {
        // À adapter à votre projet, bien sûr.
        return $_SESSION['user_id'];
    }
}

```

Et maintenant ?

Ce TP est bien sûr un exemple parmi tant d'autres. Il y a notamment une notion que nous n'avons pas abordé : le partage de méthodes. Vous vous souvenez ?

Maintenant, à vous de jouer pour trouver des idées afin d'améliorer ce p'tit behavior. 😊

Vous avez pu apercevoir dans ce chapitre la grande modularité qu'offre Doctrine et l'intérêt des templates.

Attention cependant, ne les utilisez pas à tort et à travers ! Et souvenez-vous aussi que vous pouvez tout à fait ajouter un listener sur un objet en particulier, et non obligatoirement sur une classe entière !

Partie 3 : Annexes

Personnalisation

Doctrine nous permet de personnaliser pas mal de choses dans sa configuration.

Doctrine gère tout cela en *Cascade*, c'est-à-dire que l'on peut personnaliser certains aspects à plusieurs niveaux.

Le niveau le plus global est au niveau de Doctrine_Manager. Mais, bien entendu, il est possible de gérer tout cela plus finement.

Par exemple, nous pouvons récupérer les résultats de toutes les requêtes de différentes manières. On peut définir cela globalement ; par exemple, on choisit de récupérer des objets complets à chaque fois. Mais il reste tout à fait possible, au niveau de la requête elle-même par exemple, de dire « *Cette fois, je veux juste récupérer un tableau contenant certaines propriétés, et non l'objet en entier* ».

Notez que je rédigerai prochainement un chapitre spécifique sur l'*Hydratation des résultats*.

Avec Doctrine_Manager

Doctrine_Manager possède une méthode `setAttribute()` qui permet de changer certains... attributs !

Comment ça marche ?

Tout simplement, en appelant la méthode citée :

Code : PHP

```
<?php
$manager = Doctrine_Manager::getInstance();

$manager->setAttribute(Doctrine_Core::ATTR_VALIDATE,
Doctrine_Core::VALIDATE_ALL);
$manager->setAttribute(Doctrine_Core::ATTR_AUTO_ACCESSOR_OVERRIDE,
true);
```

Les attributs possibles sont définis dans Doctrine_Core et sont de la forme `ATTR_NOM_DE_LATTRIBUT`.

Leurs valeurs possibles sont des constantes de la forme `NOM_DE_LATTRIBUT_VALEUR`.

Détail des principaux attributs

ATTR_VALIDATE



Je mettrai prochainement en ligne un chapitre consacré à la validation des données. Nous parlerons plus en détail de cet attribut à ce moment-là.

Cet attribut indique la manière dont Doctrine va valider les données ; par exemple, si l'on doit forcer un champ déclaré de type *integer* à contenir bel et bien un nombre entier.

Les valeurs disponibles sont :

- `VALIDATE_ALL` (correspond à l'ensemble des validateurs suivants) ;
- `VALIDATE_LENGTHS` ;
- `VALIDATE_TYPES` ;
- `VALIDATE_CONSTRAINTS` ;
- `VALIDATE_NONE`.

Notez que l'on peut combiner plusieurs valeurs avec l'opérateur bit-à-bit « `|` ». Par exemple :

Code : PHP

```
<?php
$manager->setAttribute(Doctrine_Core::ATTR_VALIDATE,
Doctrine_Core::VALIDATE_LENGTHS | Doctrine_Core::VALIDATE_TYPES);
```

ATTR_AUTO_ACCESSOR_OVERRIDE

Autorise ou non la redéfinition des accesseurs, c'est-à-dire, la possibilité ou non de créer une méthode *getProperty()* pour accéder à la colonne *property*. L'intérêt est de pouvoir effectuer des traitements avant de renvoyer la valeur.

Valeurs possibles : *true* ou *false*.

ATTR_AUTOLOAD_TABLE_CLASSES

Indique si Doctrine doit ou non charger les classes de table (*ArticleTable* par exemple) automatiquement.

Valeurs possibles : *true* ou *false*. Je vous recommande de le laisser à *true* si vous les utilisez. Dans le cas contraire, cela améliorera (légèrement...) les performances.

ATTR_MODEL_LOADING

Indique la manière dont Doctrine va charger les modèles.

Les valeurs possibles sont :

- **MODEL_LOADING_AGGRESSIVE** : Doctrine inclut TOUS les modèles trouvés, même ceux que vous n'utilisez pas.
- **MODEL_LOADING_CONSERVATIVE** : stocke simplement le chemin de chaque fichier contenant un modèle, et l'inclut uniquement SI BESOIN.
- **MODEL_LOADING_PEAR** : ne précharge pas les modèles, vérifie à chaque appel dans le dossier indiqué à *loadModels()* (souvenez-vous 🤖) si le modèle existe.

ATTR_USE_DQL_CALLBACKS

Cet attribut indique si oui ou non les *hooks DQL* doivent être appelés. Reportez-vous au chapitre sur les Events Listeners pour plus de détails sur ces *hooks*.

Valeurs possibles : *true* ou *false*.

Attention, cet attribut est à *false* par défaut. Ceci signifie que vous devez explicitement le définir à *true*, quelque part dans votre application, pour utiliser les *hooks DQL*.

ATTR_QUOTE_IDENTIFIER

Cet attribut indique à Doctrine qu'il doit encapsuler tous les noms de tables, champs, etc. dans des délimiteurs. Pour info, **MySQL** utilise les *backticks* (```), **Oracle** utilise les guillemets doubles (`"`), **Access** utilise les crochets (`[et]`), etc.



Attention. N'utilisez jamais ces délimiteurs *en dur* dans votre code. Cela CASSE la portabilité de votre application, ce qui est justement tout l'inverse de ce que l'on veut avec Doctrine !

Valeurs possibles : *true* ou *false*. Je vous conseille de le mettre à *true* uniquement si vous avez des problèmes lors de l'exécution de requêtes.



Ce tutoriel n'est pas terminé !!!