



Beste de savoir

Doctrine 2 : À l'assaut de l'ORM phare de PHP

5 janvier 2019

Table des matières

I. Introduction	4
II. Les bases de Doctrine 2	6
1. Installation et configuration de Doctrine 2	8
1.1. Le fil conducteur du cours	8
1.2. Pourquoi utiliser Doctrine 2 ?	8
1.3. Installation et configuration	9
1.3.1. Installation	9
1.3.2. Configuration	13
2. Sauvegarder des entités grâce à Doctrine	15
2.1. Création des entités	15
2.2. Lier nos entités à Doctrine	16
2.2.1. Déclarer une entité	17
2.2.2. Configurer l'entité	18
2.3. Sauvegarder les entités : À la découverte du gestionnaire d'entités	22
3. Récupérer des entités avec Doctrine	24
3.1. Récupérer une entité avec la clé primaire	24
3.2. Récupérer une ou plusieurs entités selon des critères différents	25
3.2.1. Les <i>repositories</i>	25
3.2.2. Les méthodes du <i>repository</i>	26
3.3. Les index	30
4. Modifier et supprimer des entités	33
4.1. La notion d'entité gérée par Doctrine	33
4.2. Supprimer une entité	34
4.3. Mise à jour d'une entité	36
4.4. Fusionner une entité détachée	37
III. Les relations avec Doctrine 2	39
5. Relation OneToOne - 1..1	41
5.1. L'annotation OneToOne	41
5.2. Interaction avec une entité ayant une relation	44
5.2.1. Gestion manuelle des relations	44
5.2.2. Délégation à Doctrine	45
5.3. La relation OneToOne bidirectionnelle	47

5.4. L'annotation JoinColumn	48
6. Relation ManyToOne et OneToMany - 1..n	50
6.1. Relation ManyToOne	50
6.2. Relation OneToMany	53
6.3. Pratiqons : Création d'une entité sondage	57
6.3.1. Énoncé	57
6.3.2. Proposition de solution	58
7. Relation ManyToMany - n..m	64
7.1. Relation ManyToMany simple	64
7.2. Relation ManyToMany avec attributs	66
7.3. Création des participations	72
8. TP : Finir la modélisation du système de sondage	74
8.1. Pratiqons	74
8.1.1. Énoncé	74
8.1.2. Indices	74
8.2. Proposition de solution	74
8.2.1. Définition et implémentation du schéma	74
8.2.2. Tests	79
8.3. Idées d'amélioration	82
IV. Exploiter une base de données avec Doctrine 2	84
9. À la rencontre du QueryBuilder	86
9.1. Le QueryBuilder	86
9.1.1. Création d'un <i>QueryBuilder</i>	87
9.1.2. Exemples de requêtes avec le <i>QueryBuilder</i>	88
9.1.3. Personnaliser les réponses du <i>QueryBuilder</i>	92
9.1.4. Les expressions	93
9.2. Les repositories personnalisés	94
9.2.1. Configuration	94
9.2.2. Utilisation des <i>repositories</i> personnalisés	95
10. Optimiser l'utilisation de Doctrine	98
10.1. Tracer les requêtes Doctrine	98
10.2. Fetch mode (Extra-Lazy, Lazy, Eager) et lazy-loading	100
10.3. Les jointures	104
11. Configurer Doctrine pour la production	106
11.1. La gestion du cache	106
11.2. Les proxies	107
11.3. Exemple de configuration	108
12. Annexes	111
12.1. Support des transactions	111
12.1.1. Transactions implicites	111

12.1.2. Transactions explicites	112
12.2. Les références (ou comment économiser une requête)	113
12.3. Owning side - Inverse side : gestion des relations	114
12.3.1. Définition	114
12.3.2. Identification de l' <i>owning side</i>	115
12.3.3. Contraintes	116
12.4. Les événements Doctrine	117
12.4.1. Les types d'événements	117
12.4.2. Les méthodes de rappel (<i>callbacks</i>)	118

V. Conclusion **120**

Première partie

Introduction

I. Introduction

La programmation orientée objet a fini par prendre une part importante dans le monde de PHP. Bien qu'à priori simple, le besoin récurrent de faire communiquer un code orienté objet avec des sources de données extérieures telles que des base de données cache une grande complexité.

?

Comment faire correspondre facilement nos objets PHP aux informations dans la base de données ? Comment maintenir la cohérence entre le modèle objet de l'application et le schéma de la base de données ?

Ces problématiques, communes à tous les langages, ont été résolues grâce au mapping objet-relation (*Object-Relational Mapping* ou ORM). Cette technique permet d'établir un lien étroit entre notre modèle de données et la base de données relationnelles et nous donne ainsi le sentiment d'avoir une base de données orientée objet.

i

Le sigle ORM (**Object-Relational Mapper**) est aussi utilisé pour désigner les bibliothèques qui implémentent cette technique.

Dans le monde du PHP, plusieurs bibliothèques permettent de remplir ce besoin. nous pouvons citer [Propel](#) ou encore [Eloquent \(L'ORM de Laravel\)](#).

Et parmi toutes ces bibliothèques, nous allons découvrir [Doctrine 2](#) qui est très mature et largement supportée par presque tous les frameworks de l'écosystème de PHP (Symfony, Zend Framework, etc.). Nous aborderons entre autres :

- comment installer et configurer *Doctrine 2* ;
- comment modéliser un système de données orienté objet avec *Doctrine 2* ;
- comment exploiter une base de données avec *Doctrine 2*.

Il est important de souligner que l'utilisation d'un ORM nécessite une bonne connaissance de la programmation orientée objet en PHP et des notions en modélisation avec des méthodes d'analyse et de conception comme UML¹, Merise, etc.

Pour suivre ce cours, il faudra aussi savoir installer et utiliser le gestionnaire de dépendances [Composer](#) ainsi qu'un moteur de base de données relationnelle (MySQL, PostgreSQL, MSSQL, Oracle, etc.).

1. Unified Modeling Language - Langage de modélisation unifié

Deuxième partie
Les bases de Doctrine 2

II. Les bases de Doctrine 2

Dans cette première partie du cours, nous allons voir ensemble comment installer, configurer et utiliser les opérations de base de l'ORM Doctrine.

À la fin de celle-ci, vous devriez être en mesure de créer une base de données et d'interagir avec elle grâce à Doctrine.

1. Installation et configuration de Doctrine

2

Nous allons commencer par un chapitre assez court pour introduire *Doctrine 2*.

Avant de rentrer dans le vif du sujet, nous allons voir les avantages qu'un ORM et plus particulièrement *Doctrine*² peut nous apporter tout au long de nos développements.

Nous finirons ensuite par l'installation et la configuration de celui-ci pour pouvoir passer rapidement à la pratique dès le chapitre suivant.

1.1. Le fil conducteur du cours

Pour explorer une grande partie des fonctionnalités de *Doctrine*, nous allons modéliser une application simple de gestion de sondages.

L'approche de ce cours étant basée sur la pratique, nous essayerons dans chaque chapitre d'aborder une notion liée à la gestion d'un sondage avant de l'implémenter en utilisant *Doctrine*.

Notre tâche va donc consister à créer un modèle de données permettant de gérer entre autres :

- des sondages ;
- les questions qui constituent les sondages et leurs réponses ;
- et les utilisateurs qui participent aux sondages.

Avec ce système simple, nous pourrons nous concentrer exclusivement sur les fonctionnalités que nous apporte *Doctrine* tout en les explorant en détail.

1.2. Pourquoi utiliser Doctrine 2 ?

L'utilisation de *Doctrine* apporte une couche d'abstraction qui nous permet de nous focaliser sur la gestion de la logique métier de notre application.

Outre cette couche d'abstraction, l'ORM *Doctrine* est suffisamment générique pour nous permettre d'utiliser à partir d'une même configuration aussi bien une base de données MySQL, PostgreSQL qu'une base de données Oracle ou encore MS SQL Server.

Pour la sécurité de notre application, *Doctrine* apporte nativement un ensemble de bonnes pratiques qui permettent entre autres d'éviter toutes les failles de type injection SQL avec peu d'efforts de notre part.

2. Tout au long de ce cours, le terme *Doctrine* sera utilisé pour désigner la deuxième version de l'ORM.

II. Les bases de Doctrine 2

Enfin, *Doctrine* est l'un des ORM les plus répandus dans l'écosystème de PHP. Il supporte beaucoup de fonctionnalités, sa prise en main est assez simple et [sa documentation](#) est très complète.

Le maîtriser pourra aussi faciliter l'apprentissage de framework PHP comme *Symfony* qui intègrent par défaut l'ORM *Doctrine*.

1.3. Installation et configuration

Pour commencer à utiliser *Doctrine*, nous devons mettre en place un minimum de configuration.

1.3.1. Installation

1.3.1.1. La base

La méthode d'installation la plus simple est d'utiliser le gestionnaire de dépendances [Composer](#). Vous pouvez l'installer depuis le [site officiel](#).

Toutes les dépendances seront installées dans un dossier vide nommé *doctrine2-tuto*. Dans ce dossier, lancer la commande :

```
1 composer require doctrine/orm:^2.5
```

Une fois la dépendance installée, il nous faut quelques lignes de configuration. Le code fourni durant ce cours sera compatible avec PHP 7. Il faudra donc avoir une version de PHP 7.X pour s'assurer du bon fonctionnement des extraits de code.

```
1 <?php
2 ##### bootstrap.php
3
4 require_once join(DIRECTORY_SEPARATOR, [__DIR__, 'vendor',
5     'autoload.php']);
6
7 use Doctrine\ORM\Tools\Setup;
8 use Doctrine\ORM\EntityManager;
9
10 $entitiesPath = [
11     join(DIRECTORY_SEPARATOR, [__DIR__, "src", "Entity"])
12 ];
13 $isDevMode = true;
14 $proxyDir = null;
15 $cache = null;
16 $useSimpleAnnotationReader = false;
```

II. Les bases de Doctrine 2

```
17
18 // Connexion à la base de données
19 $dbParams = [
20     'driver'    => 'pdo_mysql',
21     'host'     => 'localhost',
22     'charset'  => 'utf8',
23     'user'     => 'root',
24     'password' => '',
25     'dbname'   => 'poll',
26 ];
27
28 $config = Setup::createAnnotationMetadataConfiguration(
29     $entitiesPath,
30     $isDevMode,
31     $proxyDir,
32     $cache,
33     $useSimpleAnnotationReader
34 );
35 $entityManager = EntityManager::create($dbParams, $config);
36
37 return $entityManager;
```

Cette première étape obligatoire permet d'avoir les bases communes pour utiliser *Doctrine*.



La variable `$dbParams` désigne les paramètres de connexion que nous allons utiliser. Pour notre cas, nous aurons donc une base de données MySQL en local nommée `poll` avec comme utilisateur `root` sans mot de passe. À ce stade, seul le paramètre `dbParams`, nous intéresse. Nous aurons l'occasion d'aborder et d'explicitier l'utilité des autres paramètres par la suite.

Assurez-vous d'avoir un moteur de base de données installé et configuré correctement avant de passer à l'étape suivante. Si vous voulez tester ce cours rapidement sans beaucoup d'efforts sur ce point, vous pouvez utiliser le pilote SQLite.

Voici quelques paramètres de configuration utilisables selon votre moteur de base de données.

— Pilote MySQL :

```
1 <?php
2 [
3     'driver'    => 'pdo_mysql',
4     'host'     => 'localhost',
5     'charset'  => 'utf8',
6     'user'     => 'root',
7     'password' => '',
8     'dbname'   => 'poll',
9 ];
```

II. Les bases de Doctrine 2

— Pilote SQLite :

```
1 <?php
2 [
3     'driver'    => 'pdo_sqlite',
4     'path'     => 'data/poll.sqlite'
5 ];
```

Pensez à créer le dossier **data** si vous optez pour *SQLite*.

— Pilote PostgreSQL :

```
1 <?php
2 [
3     'driver'    => 'pdo_pgsql',
4     'host'     => 'localhost',
5     'charset'  => 'utf8',
6     'user'     => 'root',
7     'password' => '',
8     'dbname'   => 'poll',
9 ];
```

N'hésitez pas à consulter la documentation officielle du composant [Doctrine DBAL](#) pour plus de détails sur les paramètres de connexion.

Le composant *Doctrine DBAL*³ est une couche par-dessus [PDO](#)⁴ utilisée en interne par *Doctrine ORM* pour accéder à nos bases de données. Nous n'aurons donc pas à l'utiliser directement dans ce cours.

1.3.1.2. La ligne de commande

La bibliothèque *doctrine/orm* fournit un ensemble de commandes qui nous seront d'une grande aide pendant nos développements. Cependant quelques lignes de configuration sont nécessaires pour les activer. Nous allons donc créer un fichier **cli-config.php** pour finaliser la configuration.

```
1 <?php
2 ##### cli-config.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
```

3. Database Abstraction & Access Layer

4. PHP Data Objects

II. Les bases de Doctrine 2

```
6 use Doctrine\ORM\Tools\Console\ConsoleRunner;
7
8 return ConsoleRunner::createHelperSet($entityManager);
```



Le fichier doit obligatoirement s'appeler `cli-config.php` ou alors `config/cli-config.php` pour que les commandes de Doctrine puissent fonctionner.

Il est maintenant possible de vérifier l'installation en lançant la commande :

```
1 vendor/bin/doctrine
2
3 Doctrine Command Line Interface 2.5.5-DEV
4 Usage:
5   command [options] [arguments]
6
7 Options:
8   -h, --help           Display this help message
9   -q, --quiet          Do not output any message
10  -V, --version         Display this application version
11      --ansi           Force ANSI output
12      --no-ansi        Disable ANSI output
13  -n, --no-interaction Do not ask any interactive question
14  -v|vv|vvv, --verbose Increase the verbosity of messages: 1 for
15                        normal output, 2 for more verbose output and 3 for debug
16
17 Available commands:
18   help                Displays help for a command
19   list                Lists commands
20   dbal
21   dbal:import         Import SQL file(s) directly to
22                       Database.
23   dbal:run-sql       Executes arbitrary SQL directly
24                       from the command line.
25   orm
26   orm:clear-cache:metadata Clear all metadata cache of the
27                           various cache drivers.
28   orm:clear-cache:query  Clear all query cache of the
29                           various cache drivers.
30   orm:clear-cache:result Clear all result cache of the
31                           various cache drivers.
```

Un ensemble de commandes sont affichées dans la console. Nous aurons l'occasion de voir en détails l'utilité de chacune d'elles plus tard.

1.3.2. Configuration

1.3.2.1. Configuration des entités

Durant tout le long de cours, nous parlerons souvent d'entités.



Mais qu'est qu'une entité ?

Une entité désigne tout simplement une classe permettant de faire le lien entre notre application et notre base de données. Elle représente donc une classe avec un ensemble d'attributs qui seront liés à des colonnes d'une table de notre base de données grâce à *Doctrine*.

Pour le bon fonctionnement de *Doctrine*, nous devons spécifier le ou les dossiers contenant ses entités. Pour notre cas, nous placerons les entités dans le dossier *src/Entity*.

```
1 <?php
2 ##### bootstrap.php
3
4 require_once join(DIRECTORY_SEPARATOR, [__DIR__, 'vendor',
5     'autoload.php']);
6
7 use Doctrine\ORM\Tools\Setup;
8 use Doctrine\ORM\EntityManager;
9
10 $entitiesPath = [
11     join(DIRECTORY_SEPARATOR, [__DIR__, "src", "Entity"])
12 ];
13 // ...
```

1.3.2.2. Configuration de Composer

Pour finir, nous allons aussi configurer *Composer* pour gérer le système d'*autoloading* de notre application. Cette étape n'est certes pas obligatoire pour utiliser *Doctrine* mais pour nous faciliter le travail, nous allons créer une règle d'*autoloading* pour charger via *Composer* toutes nos classes.

Modifions le fichier `composer.json` pour charger le préfixe `Tuto` depuis le dossier *src*.

```
1 {
2     "require": {
3         "doctrine/orm": "^2.5"
4     },
5     "autoload": {
```

II. Les bases de Doctrine 2

```
6         "psr-4": {
7             "Tuto\\": "src/"
8         }
9     }
10 }
```

Nous pouvons maintenant recharger la configuration de l'*autoloader* de *Composer* avec la commande :

```
1 composer dump-autoload
```

L'arborescence de notre projet est maintenant :

```
1 |— bootstrap.php
2 |— cli-config.php
3 |— composer.json
4 |— composer.lock
5 |— src
6 |— vendor
```

Maintenant que *Doctrine* est correctement installé et configuré, nous allons pouvoir l'exploiter dans notre projet pour mettre en place un modèle de données 100 % orienté objet.

Les notions propres à *Doctrine* seront introduites au fur et à mesure que nous modéliserons notre application.

Gardez en tête que toutes les fonctionnalités de l'ORM *Doctrine* ne pourront être abordées dans ce cours. Mais l'objectif est de devenir suffisamment autonome à la fin pour trouver les informations utiles dans la documentation officielle.

2. Sauvegarder des entités grâce à Doctrine

L'installation étant finie, nous pouvons maintenant utiliser *Doctrine* à proprement parler. Nous aborderons une opération de base dans *Doctrine* à savoir : la sauvegarde des données.

Toutes les configurations introduites dans ce chapitre seront primordiales pour la suite du cours. Il faudra donc porter une attention particulière sur les notions qui seront abordées.

2.1. Création des entités

Nous allons rentrer dans le vif du sujet en modélisant les utilisateurs qui participeront et créeront éventuellement les sondages.

Nous allons considérer qu'un utilisateur a un identifiant, un nom, un prénom et un rôle (administrateur, modérateur, etc.).

Cette classe simple peut représenter notre objet utilisateur.

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 class User
7 {
8     protected $id;
9
10    protected $firstname;
11
12    protected $lastname;
13
14    protected $role;
15
16    public function getId()
17    {
18        return $this->id;
19    }
20
21    public function setId($id)
22    {
23        $this->id = $id;
24    }
```

```
25
26     public function getFirstname()
27     {
28         return $this->firstname;
29     }
30
31     public function setFirstname($firstname)
32     {
33         $this->firstname = $firstname;
34     }
35
36     public function getLastname()
37     {
38         return $this->lastname;
39     }
40
41     public function setLastname($lastname)
42     {
43         $this->lastname = $lastname;
44     }
45
46     public function getRole()
47     {
48         return $this->role;
49     }
50
51     public function setRole($role)
52     {
53         $this->role = $role;
54     }
55 }
```

À l'état actuel, nous n'avons qu'une classe PHP comme tant d'autres. Pour l'utiliser avec *Doctrine*, nous devons le décorer.

2.2. Lier nos entités à Doctrine

Une bonne partie du fonctionnement de *Doctrine* est basée sur la configuration des entités. Pour ce faire nous disposons de quatre (4) moyens différents :

- les fichiers [YAML](#) [↗](#) ;
- les fichiers [XML](#) [↗](#) ;
- du code [PHP](#) [↗](#) ;
- et pour finir [les annotations PHP](#) [↗](#) .



Les annotations PHP sont des blocs de commentaires qui permettent de rajouter un ensemble de métadonnées à du code. Elles peuvent être présentes sur les attributs des classes, les noms des méthodes et les noms des classes. Et elles sont utilisées par beaucoup de bibliothèques PHP comme PHPUnit (`@dataProvider`, `@expectedException`, etc.) ou encore le framework Symfony [↗](#) (`@Route`, `@Security`, etc.).

Nous utiliserons les annotations tout le long du cours et c'est d'ailleurs la technique la plus répandue dans le monde PHP. Mais il faut garder en tête qu'il est relativement simple de passer d'un type de configuration à un autre.

Les annotations ont un grand atout car elles seront directement sur les entités que nous voulons décrire. Il sera donc facile de faire le pont entre nos entités et leurs correspondants dans notre base de données. Ainsi, toutes les modifications du modèle de données seront concentrées à un seul endroit de notre code ce qui facilitera sa prise en main.

2.2.1. Déclarer une entité

Toutes les annotations de *Doctrine* se trouvent dans l'espace de nom `Doctrine\ORM\Mapping`. Pour une meilleure lisibilité et un code plus concis, nous utiliserons l'alias `ORM`. Pour déclarer une entité, il suffit d'utiliser l'annotation `Entity`. Avec cette annotation, *Doctrine* sait qu'il doit prendre en compte cette classe.

Notre code devient maintenant :

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 */
11 class User
12 {
13     protected $id;
14
15     protected $firstname;
16
17     protected $lastname;
18
19     protected $role;
20
21     // getters et setters
22 }
```



Toutes les entités doivent avoir l'annotation `Entity`. C'est le lien **obligatoire** entre notre code PHP et *Doctrine*.

2.2.2. Configurer l'entité

Maintenant que l'entité est *déclarée*, nous pouvons maintenant la configurer à notre guise. Considérons donc que les identifiants des utilisateurs sont des entiers auto-incrémentés et que le nom, prénom et le rôle sont des chaînes de caractères.

Toutes ses attributs sont déclarés en utilisant l'annotation `Column`. Cette annotation comporte plusieurs attributs dont `type` qui permet de définir le type de l'attribut.

Doctrine supporte beaucoup de [type de données](#) et on peut citer entre autres :

- **string** pour faire correspondre un *VARCHAR* en SQL en chaîne de caractères PHP ;
- **text** pour faire correspondre un *LONGTEXT* en SQL en chaîne de caractères PHP ;
- **integer** pour faire correspondre un *INT* en SQL en entier PHP ;
- **boolean** pour faire correspondre un *TINYINT* en SQL en booléen PHP ;
- **datetime** pour faire correspondre un *DATETIME* en SQL en date PHP.

La configuration de l'entité devient :

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 */
11 class User
12 {
13     /**
14     * @ORM\Column(type="integer")
15     */
16     protected $id;
17
18     /**
19     * @ORM\Column(type="string")
20     */
21     protected $firstname;
22
23     /**
24     * @ORM\Column(type="string")
25     */
```

II. Les bases de Doctrine 2

```
26     protected $lastname;
27
28     /**
29     * @ORM\Column(type="string")
30     */
31     protected $role;
32
33     // ...
34 }
```

Maintenant, nous devons déclarer l'identifiant comme étant la clé primaire auto-incrémentée. Il existe une annotation Doctrine nommée `Id` qui permet de déclarer un attribut comme étant la clé primaire et l'annotation `GeneratedValue` permet quant à elle de gérer l'auto-incrémentation de celui-ci.

La configuration finale devient donc :

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9 * @ORM\Entity
10 */
11 class User
12 {
13     /**
14     * @ORM\Id
15     * @ORM\GeneratedValue
16     * @ORM\Column(type="integer")
17     */
18     protected $id;
19
20     /**
21     * @ORM\Column(type="string")
22     */
23     protected $firstname;
24
25     /**
26     * @ORM\Column(type="string")
27     */
28     protected $lastname;
29
30     /**
31     * @ORM\Column(type="string")
```

II. Les bases de Doctrine 2

```
32     */
33     protected $role;
34
35     // ...
36 }
```

Nous avons, à présent, une entité configurée comme il faut mais pour l'instant notre base de données n'a pas changé. Et c'est là que les commandes *Doctrine* entrent en jeu. Nous pouvons commencer par valider nos annotations en lançant la commande :

```
1 vendor/bin/doctrine orm:validate-schema
2 [Mapping] OK - The mapping files are correct.
3 [Database] FAIL - The database schema is not in sync with the
   current mapping file.
```

Doctrine nous dit que les annotations sont correctes mais la base de données n'est pas synchronisé avec notre configuration.

Une autre commande nous permet de mettre à jour notre base de données ou de récupérer les requêtes SQL que *Doctrine* va exécuter pour faire la mise à jour.

Voyons la requête SQL que *Doctrine* va exécuter avec cette configuration :

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql
2
3 CREATE TABLE User (id INT AUTO_INCREMENT NOT NULL, firstname
   VARCHAR(255) NOT NULL, lastname VARCHAR(255)
4 NOT NULL, role VARCHAR(255) NOT NULL, PRIMARY KEY(id)) DEFAULT
   CHARACTER SET utf8 COLLATE utf8_unicode_ci
5 ENGINE = InnoDB;
```

Pour l'instant, nous ne faisons qu'afficher la requête qui sera exécutée. Cela permet de faire une dernière vérification de l'ensemble des modifications que nous voulons appliquées.

Le nom de la table que *Doctrine* veut créer est `User`. Ce nom est exactement le même que celui de l'entité.

Mais par convention, les noms des tables SQL sont souvent en minuscules et au pluriel. Heureusement, avec l'annotation `Table`, nous pouvons définir le nom de chaque table. La configuration de l'entité devient maintenant :

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
```

II. Les bases de Doctrine 2

```
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="users")
11 */
12 class User
13 {
14     // ...
15 }
```

La configuration étant satisfaisante, nous pouvons créer la table des utilisateurs. Notez la présence de l'option `--force` qui permet d'exécuter la requête SQL.

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql --force
2
3 CREATE TABLE users (id INT AUTO_INCREMENT NOT NULL, firstname
4   VARCHAR(255) NOT NULL, lastname VARCHAR(255)
5   NOT NULL, role VARCHAR(255) NOT NULL, PRIMARY KEY(id)) DEFAULT
6   CHARACTER SET utf8 COLLATE utf8_unicode_ci
7   ENGINE = InnoDB;
8
9 -- Updating database schema...
10 -- Database schema updated successfully! "1" query was executed
```

Nous revalidons l'état de notre configuration.

```
1 vendor/bin/doctrine orm:validate-schema
2 [Mapping] OK - The mapping files are correct.
3 [Database] OK - The database schema is in sync with the mapping
4   files.
```

#	Nom	Type	Interclassement	Attributs	Null	Défaut	Extra
<input type="checkbox"/>	1 id 	int(11)			Non	Aucune	AUTO_INCREMENT
<input type="checkbox"/>	2 firstname	varchar(255)	utf8_unicode_ci		Non	Aucune	
<input type="checkbox"/>	3 lastname	varchar(255)	utf8_unicode_ci		Non	Aucune	
<input type="checkbox"/>	4 role	varchar(255)	utf8_unicode_ci		Non	Aucune	

FIGURE 2.1. – Création de la table users

II. Les bases de Doctrine 2

Notre base de données est maintenant synchronisée!

Par défaut, les noms des colonnes sont créés en se basant sur les noms des attributs des entités. Mais nous pouvons les décoller en utilisant l'attribut `name` dans les annotations `Column`.

Ainsi `@ORM\Column(type="string", name="my_column")` créerait une colonne nommée `my_column` peut importe le nom de l'attribut sur lequel il est situé.

i

Il est possible de supprimer la contrainte `NOT NULL` sur un attribut de notre entité en spécifiant le paramètre `nullable` de l'annotation `Column` à `true`. Ainsi, cet attribut pourra être nul (valeur à `null`).

Bref, nous avons là une configuration fonctionnelle mais qui repose sur un grand nombre de paramètres par défaut de *Doctrine*. Mais il est toujours possible de personnaliser à souhait notre configuration.

2.3. Sauvegarder les entités : À la découverte du gestionnaire d'entités

Maintenant que notre entité est bien configurée et que la base de données est à jour, nous pouvons utiliser l'ORM pour la manipuler.

Nous avons à notre disposition un objet que vous avez sûrement remarqué durant l'étape d'installation et de configuration de *Doctrine* : le gestionnaire d'entités (entity manager).

Comme son nom l'indique, cet objet est le chef d'orchestre de l'ORM *Doctrine*. Toutes nos opérations sur les entités se feront directement ou indirectement par son intermédiaire.

Nous l'appellerons durant le reste du cours avec son nom anglais : *entity manager*.

Pour créer notre premier utilisateur, nous allons créer un fichier `create-user.php`. Avec *Doctrine*, la création d'une entité se réduit à deux actions :

- l'instanciation de l'entité;
- la persistance de celle-ci grâce à l'*entity manager*.

Voyez par vous-même.

```
1 <?php
2 ##### create-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7
8 // Instanciation de l'utilisateur
```

II. Les bases de Doctrine 2

```
9
10 $admin = new User();
11 $admin->setFirstname("First");
12 $admin->setLastname("LAST");
13 $admin->setRole("admin");
14
15 // Gestion de la persistance
16 $entityManager->persist($admin);
17 $entityManager->flush();
18
19 // Vérification du résultats
20 echo "Identifiant de l'utilisateur créé : ", $admin->getId();
```

Si nous exécutons ce code, nous obtenons comme résultat :

```
1 Identifiant de l'utilisateur créé : 1
```



	id	firstname	lastname	role
	1	First	LAST	admin

FIGURE 2.2. – Création du premier utilisateur

En utilisant la méthode `persist` de l'*entity manager*, nous disons à *Doctrine* qu'il peut planifier la sauvegarde de cet objet. Mais à ce stade, aucune requête SQL n'est exécutée.

C'est à l'appel de la méthode `flush` que *Doctrine* effectue la sauvegarde réelle de l'entité. Nous verrons plus tard l'intérêt de l'utilisation de ces deux méthodes distinctes.

Un dernier point intéressant à relever aussi est l'affectation automatique d'un identifiant à notre utilisateur. En effet, grâce à *Doctrine*, l'identifiant SQL auto-incrémenté est récupéré dès qu'on fait appel à la méthode `flush`. Notre entité se retrouve donc tout le temps en parfaite synchronisation avec la base de données.

La configuration des entités est le moteur de toute application voulant profiter des fonctionnalités de *Doctrine*. N'hésitez donc pas rajouter des propriétés, modifier le nom des colonnes, etc. pour bien cerner les mécanismes derrière l'ORM.

La première opération de sauvegarde n'est que le début d'une longue série pour obtenir rapidement [un système de CRUD](#)⁵ (Créer, Lire, Mettre à jour et Supprimer) simple et fonctionnel avec *Doctrine*. Aucune ligne de code SQL n'a été et ne sera écrite par nos soins.

3. Récupérer des entités avec Doctrine

Maintenant que nous sommes en mesure de sauvegarder des informations dans notre base de données, il est légitime de se poser la question suivante :

?

Comment lire les données dans une base avec *Doctrine* ?

Là aussi, l'*entity manager* sera notre partenaire privilégié pour réaliser cette opération. Comme pour la sauvegarde des données, nous n'allons manipuler que des objets PHP.

Doctrine effectuera pour nous automatiquement la récupération des données, l'instanciation et l'hydratation de nos classes.

3.1. Récupérer une entité avec la clé primaire

Pour accéder aux informations grâce à une clé primaire, nous devons juste renseigner deux paramètres :

- la classe de l'entité que nous voulons récupérer ;
- et l'identifiant de celle-ci.

L'*entity manager* se charge alors de faire la requête SQL et d'instancier notre classe. Un exemple valant mieux que mille discours, nous allons récupérer l'utilisateur que nous venons de créer.

```
1 <?php
2 ##### get-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7
8 $user = $entityManager->find(User::class, 1);
9
10 echo sprintf(
11     "User (id: %s, firstname: %s, lastname: %s, role: %s)",
12     $user->getId(), $user->getFirstname(), $user->getLastname(),
13     $user->getRole()
14 );
```

II. Les bases de Doctrine 2

En exécutant ce code, nous obtenons comme réponse :



Si l'identifiant n'existe pas, la variable `$user` vaudra `null`.

Pour la suite, nous allons rajouter une méthode `__toString` à nos entités pour pouvoir les afficher plus facilement. Vous pouvez donc dès à présent rajouter dans l'entité *utilisateur*, ce code :

```
1 <?php
2 public function __toString()
3 {
4     $format =
5         "User (id: %s, firstname: %s, lastname: %s, role: %s)\n";
6     return sprintf($format, $this->id, $this->firstname,
7         $this->lastname, $this->role);
8 }
```

3.2. Récupérer une ou plusieurs entités selon des critères différents

L'intérêt de *Doctrine* ne se limite pas qu'à la récupération d'une entité en se basant sur la clé primaire. Grâce aux métadonnées que nous avons rajoutées à notre entité, nous sommes maintenant en mesure de faire plusieurs recherches sur les utilisateurs en utilisant **tous ses attributs**.

Nous pouvons nativement faire des recherches sur le nom, le prénom ou encore sur deux ou plusieurs champs en même temps et ce, sans écrire aucune ligne de code personnalisée.

3.2.1. Les repositories

Les *repositories* (entrepôts) sont des classes spécialisées qui nous permettent de récupérer nos entités. Chaque *repository* permet ainsi d'interagir avec un type d'entité et faire la liaison entre notre code et la base de données.

Pour, par exemple, lire les informations sur les utilisateurs, nous aurons un *repository* dédié à l'entité *utilisateur*. C'est une bonne pratique de travailler avec les *repositories* car elle facilite grandement les interactions avec nos entités.

En utilisant un *repository* pour les utilisateurs, notre exemple précédent devient :

```
1 <?php
2 ##### get-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7
8 $userRepo = $entityManager->getRepository(User::class);
9
10 $user = $userRepo->find(1);
11
12 echo $user;
```

L'*entity manager* reste l'élément central et c'est grâce à lui que nous récupérons notre *repository*. Avec la méthode `find`, nous ne sommes plus obligés de préciser la classe de l'entité que nous cherchons car le *repository* ne gère qu'une seule classe : ici les utilisateurs.

3.2.2. Les méthodes du *repository*

3.2.2.1. Les méthodes `find`, `findAll`, `findBy` et `findOneBy`

En accédant au *repository*, nous disposons de plusieurs méthodes *simples* permettant de lire des données.

Voici un petit tableau descriptif de chacune de ses méthodes *simples*.

Méthode	Description
<code>find</code>	Récupère une entité grâce à sa clé primaire
<code>findAll</code>	Récupère toutes les entités
<code>findBy</code>	Récupère une liste d'entités selon un ensemble de critères
<code>findOneBy</code>	Récupère une entité selon un ensemble de critères

Pour tester toutes ces méthodes, nous allons d'abord créer plusieurs utilisateurs.

```
1 <?php
2 ##### create-users.php
3
```

```

4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
    'bootstrap.php']);
5
6 use Tuto\Entity\User;
7
8 foreach (range(1, 10) as $index) {
9     $user = new User();
10    $user->setFirstname("First ".$index);
11    $user->setLastname("LAST ".$index);
12    $user->setRole("user");
13    $entityManager->persist($user);
14 }
15
16 $entityManager->flush();

```

id	firstname	lastname	role
1	First	LAST	admin
2	First 1	LAST 1	user
3	First 2	LAST 2	user
4	First 3	LAST 3	user
5	First 4	LAST 4	user
6	First 5	LAST 5	user
7	First 6	LAST 6	user
8	First 7	LAST 7	user
9	First 8	LAST 8	user
10	First 9	LAST 9	user
11	First 10	LAST 10	user

FIGURE 3.1. – Jeu de test pour la lecture

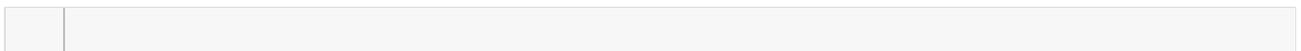


Nous ne sommes pas obligés d'appeler la méthode `flush` après chaque `persist`. Dans l'extrait de code, nous donnons d'abord à *Doctrine* toutes les entités à sauvegarder avant de faire le `flush`.

Voici maintenant un ensemble d'exemples de requêtes de lecture :

```
1 <?php
2 ##### get-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7
8 $userRepo = $entityManager->getRepository(User::class);
9
10 $user = $userRepo->find(1);
11 echo "User by primary key:\n";
12 echo $user;
13
14 $allUsers = $userRepo->findAll();
15 echo "All users:\n";
16 foreach ($allUsers as $user) {
17     echo $user;
18 }
19
20 $usersByRole = $userRepo->findBy(["role" => "admin"]);
21 echo "Users by role:\n";
22 foreach ($usersByRole as $user) {
23     echo $user;
24 }
25
26 $usersByRoleAndFirstname = $userRepo->findBy(["role" => "user",
27     "firstname" => "First 2"]);
28 echo "Users by role and firstname:\n";
29 foreach ($usersByRoleAndFirstname as $user) {
30     echo $user;
31 }
```

Le résultat est assez explicite :



Les méthodes les plus intéressantes ici sont le `findBy` et le `findOneBy`. Elles prennent toutes les deux un tableau associatif avec comme clé le nom des attributs de notre entité (id, firstname, lastname, role) et comme valeur l'information que nous voulons chercher.



Nous devons toujours mettre le nom des attributs des entités dans nos filtres et pas celui des colonnes dans la base de données. En utilisant l'ORM, nous devons faire fi de la configuration réelle de la base de données. *Doctrine* s'occupe en interne de la correspondance entre un attribut de notre classe et une colonne de la base de données.

II. Les bases de Doctrine 2

Si nous avons deux ou plusieurs clés dans le tableau comme `["role" => "user", "firstname" => "First 2"]`, *Doctrine* va chercher tous les utilisateurs ayant comme rôle `user` et comme prénom `First 2`.

?

Du coup, comment faire pour récupérer les utilisateurs ayant comme rôle `user` ou un prénom valant `First 2` ?

Il existe d'autres moyens pour récupérer des données avec des critères beaucoup plus avancés que ceux proposés nativement par le *repository*. Nous aurons l'occasion de tous les voir dans la suite de ce cours.

Mais avant cela, nous pouvons tester les filtres natifs comme *Order By*, *Limit*, *Offset* qui permettent d'affiner un tant soit peu les résultats de nos requêtes.

```
1 <?php
2 ##### get-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6
7 use Tuto\Entity\User;
8
9 $userRepo = $entityManager->getRepository(User::class);
10
11 $limit = 4;
12 $offset = 2;
13 $orderBy = ["firstname" => "DESC"];
14 $usersByRoleWithFilters = $userRepo->findBy(["role" => "user"],
15     $orderBy, $limit, $offset);
16
17 echo "Users by role with filters:\n";
18 foreach ($usersByRoleWithFilters as $user) {
19     echo $user;
20 }
```

Avec cette requête, nous récupérons une liste d'utilisateur en les triant par ordre décroissant du prénom (`["firstname" => "DESC"]`) et en limitant le nombre de résultats grâce à l'offset (2) et la limite (4).

Le résultat d'une telle requête est :

--

Le filtre *Order By* peut avoir comme valeur `ASC` pour un tri par ordre croissant et `DESC` pour un tri par ordre décroissant.

3.2.2.2. Les méthodes `findByXXX` et `findOneByXXX`

En plus des méthodes simples `findBy` et `findOneBy`, nous avons une panoplie de méthodes qui permettent de faire une recherche.

Ce sont des raccourcis qui rajoutent du sucre syntaxique aux méthodes simples mais restent néanmoins moins complets.

Pour chacune de nos entités, *Doctrine* gère grâce aux méthodes magiques plusieurs variantes de méthodes de sélection.

Si nous prenons le cas de notre utilisateur, l'attribut `role` permet d'avoir ainsi deux méthodes dans le *repository* : `findByRole` et `findOneByRole`. Ces deux méthodes représentent respectivement les méthodes `findBy(["role" => "XXX"])` et `findOneBy(["role" => "XXX"])`.

Donc l'exemple précédent peut devenir :

```
1 <?php
2 ##### get-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6
7 use Tuto\Entity\User;
8
9 $userRepo = $entityManager->getRepository(User::class);
10
11 // $usersByRole = $userRepo->findBy(["role" => "admin"]);
12 $usersByRole = $userRepo->findByRole("admin");
13 echo "Users by role:\n";
14 foreach ($usersByRole as $user) {
15     echo $user;
16 }
```



Ces méthodes magiques sont moins complètes car elles ne supportent pas les filtres natifs (*Order By*, *Limit*, etc.).

3.3. Les index

Nous avons actuellement une base de données peu volumineuse mais la quantité de données dans une application en production peut rapidement croître.

Ainsi les performances des requêtes de lecture sont à prendre en compte dès la conception du modèle de données. Avec *Doctrine*, nous pouvons mettre en place des index afin d'améliorer ceux-ci. Les index SQL sont des structures permettant aux bases de données de référencer une ou plusieurs colonnes et ainsi accélérer les recherches effectuées sur celles-ci.

II. Les bases de Doctrine 2

Pour déclarer des index grâce à *Doctrine*, nous pouvons utiliser l'annotation `Index`. Elle permet de déclarer un index en spécifiant son nom et la liste des colonnes indexées.



Les noms des colonnes **doivent être ceux dans la table SQL** et non pas le nom des attributs de la classe.

Si par exemple, nous voulions rajouter un index sur le nom et prénom des utilisateurs et un autre sur le rôle de ceux-ci, nous aurions comme configuration :

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10  * @ORM\Table(
11     name="users",
12     indexes={
13         @ORM\Index(name="search_firstname_lastname", columns={"firstname", "lastname"},
14         @ORM\Index(name="search_role", columns={"role"}))
15     }
16 )
17 */
18 class User
19 {
20     // ...
21 }
```

Bien sûr pour que les index soient créés, il faut mettre à jour la base de données avec la commande :

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql --force
2 CREATE INDEX search_firstname_lastname ON users (firstname,
3     lastname);
4 CREATE INDEX search_role ON users (role);
```

Doctrine se base sur nos entités pour nous offrir une API de lecture très simple et intuitive.

Nous avons ainsi pu créer et lire des données sans nous préoccuper de l'infrastructure derrière. Avec la configuration actuelle, on aurait pu avoir aussi bien une base SQLite ou PostgreSQL,

II. Les bases de Doctrine 2

Doctrine g rerait nativement toute la persistance des donn es gr ce   la couche d'abstraction qu'il apporte.

L' tape deux (2) du **CRUD** (Cr er, **Lire**, Mettre   jour et Supprimer) est maintenant atteinte.

4. Modifier et supprimer des entités

Continuons sur cette lancée et complétons notre système CRUD pour supporter les opérations d'édérations de nos entités.

Mais avant cela, nous allons faire un léger aparté indispensable sur la notion d'entité gérée par *Doctrine*.

4.1. La notion d'entité gérée par Doctrine

Reprenons l'exemple de création d'un utilisateur :

```
1 <?php
2 ##### create-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6
7 use Tuto\Entity\User;
8
9 // Instanciation de l'utilisateur
10
11 $admin = new User();
12 $admin->setFirstname("First");
13 $admin->setLastname("LAST");
14 $admin->setRole("admin");
15
16 // Gestion de la persistance
17 $entityManager->persist($admin);
18 $entityManager->flush();
19
20 // Vérification du résultat
21 echo "Identifiant de l'utilisateur créé : ", $admin->getId();
```

Bien que l'entité *utilisateur* soit bien configurée avec les annotations de *Doctrine*, lorsque nous instancions un utilisateur, il n'y a rien qui lie l'instance elle-même à *Doctrine*.

Il faut bien faire la distinction entre la classe `User` et les instances de celle-ci. Les annotations sur la classe permettent de dire à *Doctrine* comment gérer des instances de ce type.

Par contre, tant que nous ne demandons pas à *Doctrine* de le faire, nos instances n'auront aucun lien avec notre ORM.

II. Les bases de Doctrine 2

Donc avant l'appel à la méthode `persist` de l'*entity manager*, l'instance n'était pas liée à *Doctrine* et n'était donc **pas gérée**.

Par contre, dès que nous récupérons une entité via l'*entity manager* ou le *repository* ou bien dès que la méthode `persist` est appelée, le ou les entités **sont gérées** par *Doctrine*.

Ainsi, *Doctrine* surveille les modifications faites sur ces entités et applique ces changements à l'appel de la méthode `flush`.

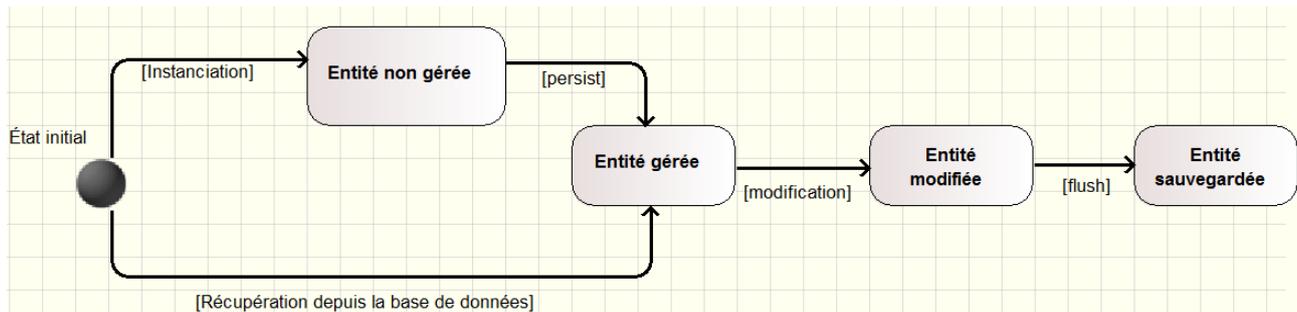


FIGURE 4.1. – Notion d'entité gérée par Doctrine

4.2. Supprimer une entité

Pour bien voir, l'utilité de la notion d'entité gérée par *Doctrine*, nous allons essayer de supprimer l'utilisateur avec comme identifiant deux (2).

L'*entity manager* (oui encore lui) propose une méthode `remove` qui prend en paramètre l'entité que nous voulons supprimer.

```
1 <?php
2 ##### delete-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7
8 $index = 1;
9
10 $user = new User();
11 $user->setId($index + 1);
12 $user->setFirstname("First ".$index);
13 $user->setLastname("LAST ".$index);
14 $user->setRole("user");
15
16 $entityManager->remove($user);
```

II. Les bases de Doctrine 2

On a beau mettre l'identifiant et toutes les informations de l'entité que nous voulons supprimer, en exécutant le code, nous obtenons une exception.

```
1 Fatal error: Uncaught Doctrine\ORM\ORMInvalidArgumentException:
2   Detached entity User (id: 2, firstname: First 1, lastname: LAST 1,
3     role: user)
   cannot be removed
```

L'entité est considérée comme *détachée*. *Doctrine* ne le gère pas et donc ne sait pas comment la supprimer. C'est une erreur assez courante et qui peut, sur de grand projet, facilement vous faire perdre beaucoup de temps.

Pour supprimer l'entité, nous allons donc commencer par le récupérer grâce au *repository*.

```
1 <?php
2 ##### delete-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5   'bootstrap.php']);
6
7 use Tuto\Entity\User;
8
9 $identifiant = 2;
10
11 $userRepo = $entityManager->getRepository(User::class);
12 // Récupération de l'utilisateur (donc automatiquement géré par
13   Doctrine)
14 $user = $userRepo->find($identifiant);
15
16 $entityManager->remove($user);
17 $entityManager->flush($user);
18
19 // Récupération pour vérifier la suppression effective de
20   l'utilisateur
21 $user = $userRepo->find($identifiant);
22
23 var_dump($user); // doit renvoyer NULL
```

id	firstname	lastname	role
1	First	LAST	admin
3	First 2	LAST 2	user
4	First 3	LAST 3	user

FIGURE 4.2. – Suppression de l'utilisateur

Comme pour la création d'une entité, nous disons d'abord à *Doctrine* que nous souhaitons supprimer une entité, et ensuite nous validons l'opération en appelant la méthode `flush`.

4.3. Mise à jour d'une entité

Puisque appeler la méthode `persist` permet de gérer une entité. Il est tentant de créer une nouvelle entité avec les informations à jour (l'identifiant restant le même) puis de sauvegarder celle-ci avec `persist`.

Mais comme vous vous en doutez, cela ne fera pas une mise à jour. Et pour en avoir le cœur net, nous allons effectuer cette manipulation.

```
1 <?php
2 ##### update-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6
7 use Tuto\Entity\User;
8
9 $index = 10;
10
11 $user = new User();
12 $user->setId($index + 1);
13 $user->setFirstname("First Modified ".$index);
14 $user->setLastname("LAST Modified ".$index);
15 $user->setRole("user");
16
17 $entityManager->persist($user);
18
19 $entityManager->flush();
```

11	First 10	LAST 10	user
12	First Modified 10	LAST Modified 10	user

FIGURE 4.3. – Mise à jour de l'utilisateur en echec

En consultant la base de données, nous pouvons voir d'une nouvelle ligne a été crée dans la table.



Dès que nous appelons la méthode `persist` sur une entité **pas encore gérée** par *Doctrine*, à l'appel de la méthode `flush`, une nouvelle entrée sera forcément créer dans la base de



données.

Donc comme pour la suppression d'une entité, nous allons d'abord commencer par la récupérer depuis le *repository*. Ensuite la mise à jour de cette entité se résume juste à modifier ses attributs puis à *flusher*.

```

1  <?php
2  ##### update-user.php
3
4  $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
   'bootstrap.php']);
5
6  use Tuto\Entity\User;
7
8  $identifiant = 11;
9
10 $userRepo = $entityManager->getRepository(User::class);
11
12 // Récupération de l'utilisateur (donc automatiquement géré par
   Doctrine)
13 $user = $userRepo->find($identifiant);
14
15 $user->setFirstname("First Real Modification");
16 $user->setLastname("Last Real Modification");
17
18 $entityManager->flush();

```

11	First Real Modification	Last Real Modification	user
12	First Modified 10	LAST Modified 10	user

FIGURE 4.4. – Mise à jour de l'utilisateur avec comme identifiant : 11

Le comportement de *Doctrine* est très grandement influencé par l'état des entités. Il faut donc avant chaque mise à jour de celle-ci s'assurer que l'entité est bien gérée par *Doctrine*.

4.4. Fusionner une entité détachée

Nous avons la possibilité de gérer une entité qui n'est pas issue de la base de données sans passer la méthode `persist`. Prenons l'exemple classique où dans la session d'une application web, nous avons sérialisé un objet utilisateur.

Si nous dé-sérialisons ce même objet, il ne sera plus géré par *Doctrine*. Mais en utilisant la méthode `merge` de l'*entity manager*, l'entité est considérée comme si elle provenait de la base de données.

```
1 <?php
2 ##### update-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6
7 use Tuto\Entity\User;
8
9 $user = unserialize("informations de l'utilisateur serialisées");
10
11 // Reprise en charge de l'objet par l'entity manager
12 $entityManager->merge($user);
13
14 $user->setFirstname("First Real Modification");
15
16 $entityManager->flush();
```

Le prénom de l'utilisateur sera mis à jour avec cette méthode.

La méthode `merge` ne doit être utilisée que dans un cadre où vous avez **la certitude** que les informations contenues dans l'objet sont déjà en parfaite synchronisation avec ceux dans la base de données. Sinon vous risquez d'écraser des modifications faites sur l'entité entre sa sérialisation et sa dé-sérialisation.

Nous pouvons maintenant effectuer les quatre (4) opérations de base avec *Doctrine* : créer, lire, modifier et supprimer une entité.

Pour l'instant, nous n'avons effleuré que la surface de *Doctrine*. Prenez le temps de tester la configuration et l'ensemble des options déjà présentées pour bien vous familiariser avec l'ORM phare de PHP.

Nous aurons l'occasion de bien approfondir tous ces concepts et d'en aborder de nouveaux dans les chapitres suivants.

Voici un résumé de cette partie fourni par [artragis](#) sous licence [CC-BY](#).



FIGURE 4.5. – Résumé des bases de Doctrine

Vous pouvez consulter le [lien interactif](#) pour plus de détails.

Troisième partie
Les relations avec Doctrine 2

III. Les relations avec Doctrine 2

Maintenant que les opérations de base n'ont plus de secret pour nous, nous allons voir comment concevoir un modèle de données complet avec Doctrine.

?

Comment matérialiser les clés étrangères ? Comment Doctrine assure-t-il la cohérence de nos objets avec des relations ?

Nous allons répondre à ces questions en abordant dans cette partie tous les types de relations gérés par Doctrine.

5. Relation OneToOne - 1..1

MySQL est une base de données relationnelle (tout comme MariaDB, PostgreSQL, Oracle, etc.) et il n'est pas rare d'avoir des relations diverses lorsque nous modélisons une application.

Par exemple, un sondage est constitué d'un ensemble de questions, une question peut avoir plusieurs réponses possibles, un utilisateur peut avoir une adresse, etc.

Toutes ces relations peuvent être matérialisées en utilisant des clés étrangères en SQL.

Avec *Doctrine*, selon la nature de la relation, nous avons des moyens très **simples**, **efficaces** et **élégants** de la gérer. Nous allons donc étoffer notre modèle de données en implémentant des relations avec *Doctrine*.

5.1. L'annotation OneToOne

Reprenons notre exemple et considérons qu'un utilisateur peut avoir une seule adresse et que cette adresse ne peut être liée qu'à un seul utilisateur.

Dans la base de données, le schéma ressemblerait à :

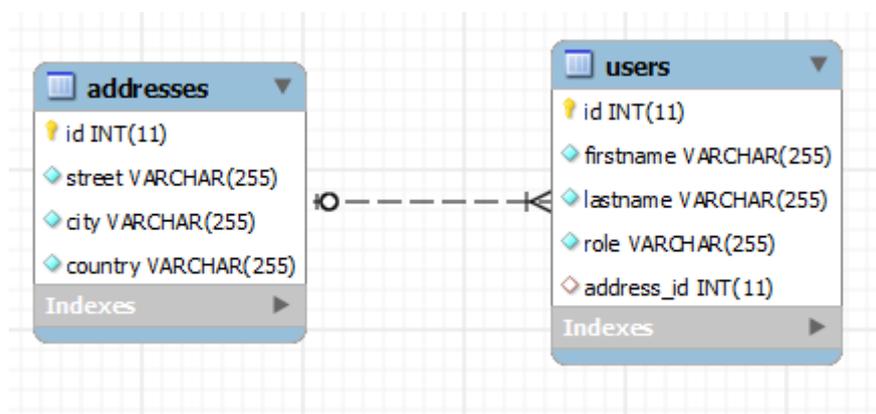


FIGURE 5.1. – Relation OneToOne : Un utilisateur avec une adresse

Avec *Doctrine*, nous pouvons obtenir ce résultat avec l'annotation `OneToOne`. Supposons que notre adresse est comme suit (vous pouvez aussi vous entraîner en créant vous-même une entité *adresse*) :

```
1 <?php
2 ##### src/Entity/Address.php
3
```

```

4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="addresses")
11 */
12 class Address
13 {
14     /**
15     * @ORM\Id
16     * @ORM\GeneratedValue
17     * @ORM\Column(type="integer")
18     */
19     protected $id;
20
21     /**
22     * @ORM\Column(type="string")
23     */
24     protected $street;
25
26     /**
27     * @ORM\Column(type="string")
28     */
29     protected $city;
30
31     /**
32     * @ORM\Column(type="string")
33     */
34     protected $country;
35
36     public function __toString()
37     {
38         $format =
39             "Address (id: %s, street: %s, city: %s, country: %s)";
40         return sprintf($format, $this->id, $this->street,
41             $this->city, $this->country);
42     }
43     // ... tous les getters et setters
44 }

```

Nous avons jusque-là rien de nouveau. Pour relier cette adresse à un utilisateur, nous devons modifier l'entité *utilisateur*. Pour des soucis de clarté tous les autres attributs seront masqués.

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="users")
11 */
12 class User
13 {
14     // ...
15
16     /**
17     * @ORM\OneToOne(targetEntity=Address::class)
18     */
19     protected $address;
20
21     // Ajout de l'adresse à la méthode __toString
22     public function __toString()
23     {
24         $format =
25             "User (id: %s, firstname: %s, lastname: %s, role: %s, address: %s)";
26         return sprintf($format, $this->id, $this->firstname,
27             $this->lastname, $this->role, $this->address);
28     }
29 }
```

Avec l'annotation `OneToOne`, nous disons à *Doctrine* que notre utilisateur peut être lié à une adresse (grâce à l'attribut `targetEntity`).



Dans `targetEntity`, il faut spécifier un espace de nom complet. Avec PHP 7, l'utilisation de la constante `class` nous facilite la tâche.

En lançant une mise à jour de la base de données, *Doctrine* génère la nouvelle table pour les adresses et crée la clé étrangère dans la table des utilisateurs.

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql --force
2 CREATE TABLE addresses (id INT AUTO_INCREMENT NOT NULL, street
3   VARCHAR(255) NOT NULL, city VARCHAR(255) NOT NULL,
4   country VARCHAR(255) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER
5   SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;
```

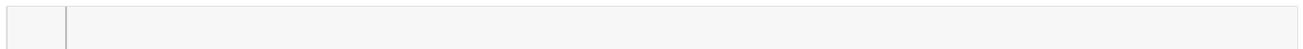
```
4 ALTER TABLE users ADD address_id INT DEFAULT NULL;
5 ALTER TABLE users ADD CONSTRAINT FK_1483A5E9F5B7AF75 FOREIGN KEY
  (address_id) REFERENCES addresses (id);
6 CREATE UNIQUE INDEX UNIQ_1483A5E9F5B7AF75 ON users (address_id);
```

5.2. Interaction avec une entité ayant une relation

Essayons maintenant d'affecter une adresse à un de nos utilisateurs :

```
1 <?php
2 ##### set-user-address.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
  'bootstrap.php']);
5
6 use Tuto\Entity\User;
7 use Tuto\Entity\Address;
8
9 $userRepo = $entityManager->getRepository(User::class);
10
11 $user = $userRepo->find(1);
12
13 $address = new Address();
14 $address->setStreet("Champ de Mars, 5 Avenue Anatole");
15 $address->setCity("Paris");
16 $address->setCountry("France");
17
18 $user->setAddress($address);
19
20 $entityManager->flush();
```

En exécutant le code, une belle erreur s'affiche :



Vous l'aurez deviné, l'entité *adresse* n'est pas encore gérée par *Doctrine*. Notre ORM ne sait donc pas quoi faire avec. Pour résoudre ce problème nous avons deux solutions que nous allons aborder ci-dessous.

5.2.1. Gestion manuelle des relations

Comme le message d'erreur le suggère, nous pouvons corriger le problème en utilisant directement la méthode `persist` sur l'entité *adresse* avant de flusher. Elle sera ainsi gérée par *Doctrine*. Le

code devient alors :

```
1 <?php
2 ##### set-user-address.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7 use Tuto\Entity\Address;
8
9 $userRepo = $entityManager->getRepository(User::class);
10
11 $user = $userRepo->find(1);
12
13 $address = new Address();
14 $address->setStreet("Champ de Mars, 5 Avenue Anatole");
15 $address->setCity("Paris");
16 $address->setCountry("France");
17
18 $user->setAddress($address);
19
20 $entityManager->persist($address);
21 $entityManager->flush();
```

En ré-exécutant le code, une adresse est créée et affectée à notre utilisateur.

id	street	city	country
1	Champ de Mars, 5 Avenue Anatole	Paris	France

FIGURE 5.2. – Adresse sauvegardée

id	firstname	lastname	role	address_id
1	First	LAST	admin	1

FIGURE 5.3. – Utilisateur avec une adresse

5.2.2. Délégation à Doctrine

Nous pouvons aussi utiliser le système de cascade de *Doctrine* pour gérer la relation entre les entités *utilisateur* et *adresse*. Avec ce système, nous pouvons définir comment *Doctrine* gère l'entité *adresse* si l'entité *utilisateur* est modifiée. Ainsi, pendant la création et la suppression de l'utilisateur, nous pouvons demander à *Doctrine* de répercuter ces changements sur son adresse.

III. Les relations avec Doctrine 2

Le paramètre à utiliser est l'attribut `cascade` de l'annotation `OneToOne`. Voici un tableau récapitulatif de quelques valeurs possibles et de leurs effets.

Cascade	Effet
<code>persist</code>	Si l'entité <i>utilisateur</i> est sauvegardée, faire de même avec l'entité adresse associée
<code>remove</code>	Si l'entité <i>utilisateur</i> est supprimée, faire de même avec l'entité adresse associée



Il faut porter une attention particulière lors de l'utilisation de la cascade d'opérations. Les performances de votre application peuvent en pâtir si cela est mal configurée.

Pour notre cas, puisqu'une adresse ne sera associée qu'à un et un seul utilisateur et vice versa, nous pouvons nous permettre de la créer ou de la supprimer suivant l'état de l'entité *utilisateur*.

L'entité *utilisateur* devient :

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10  * @ORM\Table(name="users")
11  */
12 class User
13 {
14     // ...
15
16     /**
17     * @ORM\OneToOne(targetEntity=Address::class, cascade={"persist", "remove"})
18     */
19     protected $address;
20
21     // ...
22 }
```

N'hésitez pas à valider les annotations en utilisant les commandes que *Doctrine* met à notre disposition.



Avec l'attribut `cascade`, les opérations de cascade sont effectuées par *Doctrine* au niveau applicatif. Les événements SQL de type `ON DELETE` et `ON UPDATE` sur les clés étrangères ne sont pas utilisés.

5.3. La relation OneToOne bidirectionnelle

Avec notre modèle actuel, lorsque nous avons une entité *utilisateur*, nous pouvons trouver l'adresse qui lui est associée. Par contre, lorsque nous avons une entité *adresse*, nous ne sommes pas en mesure de récupérer l'utilisateur associé. La relation que nous avons configurée est dite **unidirectionnelle** car seul un des membres de la relation peut faire référence à l'autre.

Dans certains cas, il est possible d'avoir une relation dite bidirectionnelle où chacun des deux membres peut faire référence à l'autre.

Pour ce faire, nous devons mettre à jour les annotations des entités.

Nous allons d'abord modifier l'adresse pour rajouter l'information relative à l'utilisateur.

```
1 <?php
2 ##### src/Entity/Address.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="addresses")
11 */
12 class Address
13 {
14     // ...
15
16     /**
17     * @ORM\OneToOne(targetEntity=User::class, mappedBy="address")
18     */
19     protected $user;
20
21     // ...
22 }
```

L'attribut `mappedBy` est obligatoire et elle permet de dire à *Doctrine* que cette relation est bidirectionnelle et que l'attribut utilisé dans l'autre côté de la relation est `address`.

III. Les relations avec Doctrine 2

Dans la même logique, au niveau de l'utilisateur, nous devons mettre en place un attribut `inverseBy` pour signifier à *Doctrine* que nous utilisons une relation bidirectionnelle. La valeur de ce paramètre désigne le nom de l'attribut dans l'autre côté de la relation.

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="users")
11 */
12 class User
13 {
14     // ...
15     /**
16     * @ORM\OneToOne(targetEntity=Address::class, cascade={"persist", "remove"})
17     */
18     protected $address;
19
20     // ...
21 }
```

Ces modifications n'affectent pas la configuration de la base de données mais permettent au niveau applicatif d'accéder plus facilement à certaines informations.

Pour notre exemple actuel, avoir une relation bidirectionnelle n'est pas pertinente car l'adresse ne sera jamais utilisée sans l'utilisateur. Nous pouvons donc nous en passer.

5.4. L'annotation `JoinColumn`

Si nous consultons la base de données, nous pouvons voir que la colonne qui porte la clé étrangère s'appelle `address_id`. *Doctrine* a choisi automatique ce nom en se basant sur l'attribut `address` de notre entité *utilisateur*.

De plus, avec la configuration actuelle, il est possible d'avoir un utilisateur sans adresse (colonne à `NULL`). Nous pouvons personnaliser ces informations en utilisant l'annotation `JoinColumn`.

Cette annotation nous permet entre autres :

- de modifier le nom de la colonne portant la clé étrangère avec son attribut `name` ;
- ou de rajouter une contrainte de nullité avec l'attribut `nullable`.

Avec comme configuration :

III. Les relations avec Doctrine 2

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="users")
11 */
12 class User
13 {
14     // ...
15
16     /**
17     * @ORM\OneToOne(targetEntity=Address::class, cascade={"persist", "remove"})
18     * @ORM\JoinColumn(name="address", nullable=false)
19     */
20     protected $address;
21
22     // ...
23 }
```

... nous aurions eu comme requête SQL :

```
1 ALTER TABLE users ADD address INT NOT NULL;
```

au lieu de :

```
1 ALTER TABLE users ADD address_id INT DEFAULT NULL;
```

Doctrine utilise beaucoup de valeurs par défaut mais nous laisse le choix de les personnaliser à travers les différentes annotations à notre disposition.

Maintenant que nous savons comment *Doctrine* gère les relations dans un modèle de données, nous allons découvrir d'autres types de relations comme le 1..n et le m..n.

Même si l'ORM nous facilite le travail, il faut quand même s'y connaître un minimum en modélisation de base de données pour profiter une maximum de ses fonctionnalités.

6. Relation ManyToOne et OneToMany - 1..n

Regardons plus en détails le modèle de notre sondage. Une question doit avoir plusieurs réponses au choix et chaque réponse ne peut être associée qu'à une seule question.

Dans cette partie, nous allons voir comment matérialiser ce genre de relations courantes (1..n) avec *Doctrine*.

6.1. Relation ManyToOne

Nous devons avoir en tête le schéma que nous désirons avant de pouvoir le réaliser avec *Doctrine*.

Notre modèle de données pour intégrer les questions et leurs réponses ressemblerait à :

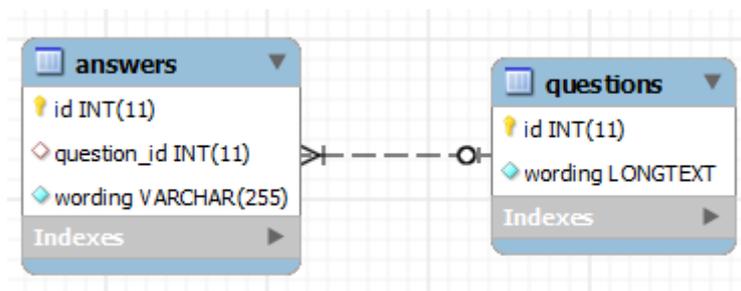


FIGURE 6.1. – Relation ManyToOne : Plusieurs réponses associées à une question

Un tel résultat peut être obtenu avec *Doctrine* en utilisant l'annotation `ManyToOne`. Prenons une réponse ayant comme attribut un libellé et la question à laquelle elle est associée. Une question aura, quant à elle, un libellé et une liste de réponses possibles.

?

Mais où placer l'annotation `ManyToOne`, dans l'entité *question* ou dans l'entité *réponse* ?

Cette annotation est souvent source de problèmes et d'incompréhensions. Il existe d'ailleurs plusieurs astuces pour l'utiliser à bon escient.

Reprenons notre exemple :

- Une réponse est liée à **une seule (one)** question ;
- Une question peut avoir **plusieurs (many)** réponses.

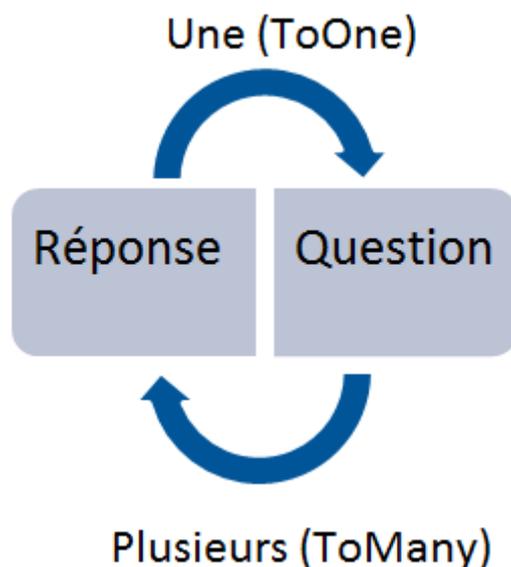


FIGURE 6.2. – Relation entre Question et Réponse

Dans une relation `ManyToOne`, le **many** qualifie l'entité qui doit contenir l'annotation. Ici le **many** qualifie les réponses. Donc, pour notre cas, l'annotation doit être dans l'entité *réponse*.

Nos deux entités doivent donc être configurées comme suit :

— la question :

```
1 <?php
2 ##### src/Entity/Question.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="questions")
11 */
12 class Question
13 {
14     /**
15     * @ORM\Id
16     * @ORM\GeneratedValue
17     * @ORM\Column(type="integer")
18     */
19     protected $id;
20
21     /**
22     * @ORM\Column(type="text")
23     */
24     protected $wording;
```

III. Les relations avec Doctrine 2

```
25
26 public function __toString()
27 {
28     $format = "Question (id: %s, wording: %s)\n";
29     return sprintf($format, $this->id, $this->wording);
30 }
31
32 // getters et setters
33 }
```

— et la réponse :

```
1 <?php
2 ##### src/Entity/Answer.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="answers")
11 */
12 class Answer
13 {
14     /**
15     * @ORM\Id
16     * @ORM\GeneratedValue
17     * @ORM\Column(type="integer")
18     */
19     protected $id;
20
21     /**
22     * @ORM\Column(type="string")
23     */
24     protected $wording;
25
26     /**
27     * @ORM\ManyToOne(targetEntity=Question::class)
28     */
29     protected $question;
30
31     public function __toString()
32     {
33         $format = "Answer (id: %s, wording: %s)\n";
34         return sprintf($format, $this->id, $this->wording);
35     }
36 }
```

```
37     // getters et setters
38 }
```

Il suffit de mettre à jour la base de données à l'aide de l'invite de commande de *Doctrine*.

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql --force
2 CREATE TABLE answers (id INT AUTO_INCREMENT NOT NULL, question_id
3   INT DEFAULT NULL, wording VARCHAR(255)
4   NOT NULL, INDEX IDX_50D0C6061E27F6BF (question_id), PRIMARY
5   KEY(id))
6   DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE =
7   InnoDB;
8 CREATE TABLE questions (id INT AUTO_INCREMENT NOT NULL, wording
9   LONGTEXT NOT NULL, PRIMARY KEY(id))
10  DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE =
11  InnoDB;
12 ALTER TABLE answers ADD CONSTRAINT FK_50D0C6061E27F6BF FOREIGN KEY
13   (question_id) REFERENCES questions (id);
```

6.2. Relation OneToMany

Dans le cas des questions et réponses, il est légitime de vouloir accéder aux réponses depuis l'entité *question* (pour afficher rapidement les réponses associées à une question par exemple).

Comme pour la relation **OneToOne**, nous pouvons rendre une relation **ManyToOne** bidirectionnelle en utilisant l'annotation *miroir OneToMany*. La configuration est assez proche de celle de l'annotation **OneToOne**. Nous pouvons même activer les opérations de cascade pour créer une question et toutes les réponses associées plus facilement.

Ainsi, à l'image de toutes les relations bidirectionnelles, chacune des deux entités doit maintenant être configurée pour faire référence à l'autre.

La configuration finale est donc :

```
1 <?php
2 ##### src/Entity/Answer.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="answers")
```

III. Les relations avec Doctrine 2

```
11 */
12 class Answer
13 {
14     // ...
15
16     /**
17     * @ORM\ManyToOne(targetEntity=Question::class, inversedBy="answers")
18     */
19     protected $question;
20
21     // ...
22 }
```

```
1 <?php
2 ##### src/Entity/Question.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9 * @ORM\Entity
10 * @ORM\Table(name="questions")
11 */
12 class Question
13 {
14     // ...
15
16     /**
17     * @ORM\OneToMany(targetEntity=Answer::class, cascade={"persist", "remove"})
18     */
19     protected $answers;
20
21     // ...
22 }
```

Testons le tout en créant une question et plusieurs réponses associées.

Pour créer une question, le code est simple.

```
1 <?php
2 ##### create-question.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
```

III. Les relations avec Doctrine 2

```
6 use Tuto\Entity\Question;
7 use Tuto\Entity\Answer;
8
9 $question = new Question();
10
11 $question->setWording("Doctrine 2 est-il un bon ORM ?");
12
13 $entityManager->persist($question);
14 $entityManager->flush();
15
16 echo $question;
```



Mais comment relier une réponse à une question ?

L'attribut `answers` de la classe `Question` représente une liste de réponses. Avec *Doctrine*, nous pouvons représenter une collection grâce à la classe `ArrayCollection`. Cette classe permettra à *Doctrine* de gérer convenablement tous les changements qui pourront survenir sur la collection.

L'API de la classe `ArrayCollection` est pratique pour manipuler des listes (ajout, suppression, recherche d'un élément).

Il est d'ailleurs possible d'utiliser le package [doctrine/collections](#) dans des projets pour profiter des fonctionnalités des collections sans installer l'ORM *Doctrine*.

Nous allons donc modifier l'entité `question` pour rajouter ces modifications.

```
1 <?php
2 ##### src/Entity/Question.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 /**
10 * @ORM\Entity
11 * @ORM\Table(name="questions")
12 */
13 class Question
14 {
15     // ...
16
17     /**
18     * @ORM\OneToMany(targetEntity=Answer::class, cascade={"persist", "remove"})
19     */
20     protected $answers;
21 }
```

III. Les relations avec Doctrine 2

```
22     public function __construct()
23     {
24         $this->answers = new ArrayCollection();
25     }
26     // ...
27
28     public function getAnswers()
29     {
30         return $this->answers;
31     }
32
33     public function addAnswer(Answer $answer)
34     {
35         $this->answers->add($answer);
36         $answer->setQuestion($this);
37     }
38 }
```

Dans la méthode `addAnswer`, nous avons rajouté une petite logique qui permet de maintenir l'application dans un état cohérent. Lorsqu'une réponse est associée à une question, cette question est elle aussi automatiquement liée à la réponse.

Cette liaison est obligatoire pour le bon fonctionnement de la cascade d'opération de *Doctrine* (vous pourrez consulter la partie annexe de ce cours pour une explication en détail - *Owning side - Inverse side*). Pour chaque relation faisant intervenir une collection, il faudra y porter une attention particulière.

Nous pouvons maintenant compléter la création de la question.

```
1 <?php
2 ##### create-question.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6
7 use Tuto\Entity\Question;
8 use Tuto\Entity\Answer;
9
10 $question = new Question();
11 $question->setWording("Doctrine 2 est-il un bon ORM ?");
12
13 $yes = new Answer();
14 $yes->setWording("Oui, bien sûr !");
15
16 $question->addAnswer($yes);
17
18 $no = new Answer();
```

```
19 $no->setWording("Non, peut mieux faire.");
20
21 $question->addAnswer($no);
22
23 $entityManager->persist($question);
24 $entityManager->flush();
25
26 echo $question;
```

Grâce à la cascade des opérations de sauvegarde, nous ne sommes pas obligés de persister les réponses. *Doctrine* regarde l'état de la collection et fait le nécessaire pour que les réponses soient sauvegardées correctement.

id	wording
1	Doctrine 2 est-il un bon ORM ?

FIGURE 6.3. – Insertion d'une question

id	question_id	wording
1	1	Oui, bien sûr !
2	1	Non, peut mieux faire.

FIGURE 6.4. – Insertion des réponses

6.3. Pratiqons : Création d'une entité sondage

6.3.1. Énoncé

Pour pratiquer ce que nous venons de voir, nous allons créer une entité *sondage* avec deux attributs :

- son titre : une chaîne de caractères qui permet de décrire le sondage ;
- sa date de création : une date PHP permettant de savoir la date de création du sondage.

Le sondage est constitué d'une liste de questions. La relation entre les deux entités doit être bidirectionnelle mais aucune opération de cascade ne doit être définie.

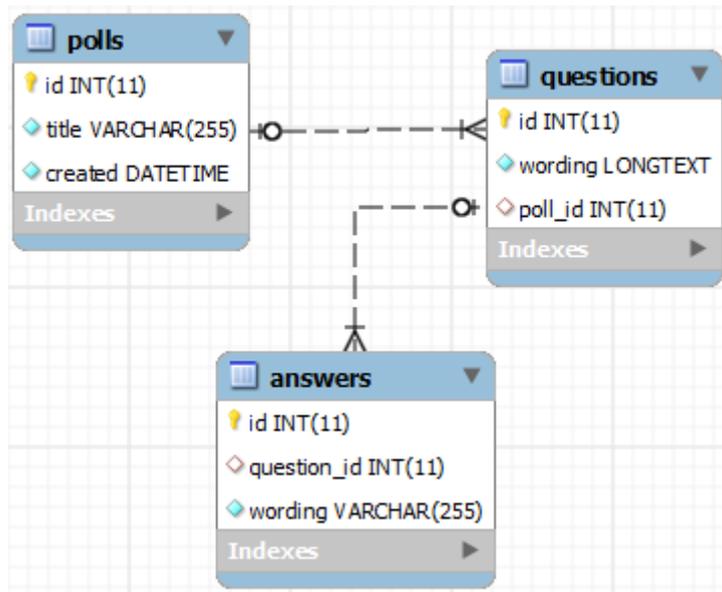


FIGURE 6.5. – Relation ManyToOne : Plusieurs questions associées à un sondage

6.3.2. Proposition de solution

Nous allons créer une entité nommée `Poll` pour représenter le sondage et éditer l'entité `question` pour le lier au sondage.

Si nous réécrivons notre exemple :

- Une question est liée à **un seul (one)** sondage ;
- Un sondage peut avoir **plusieurs (many)** questions.

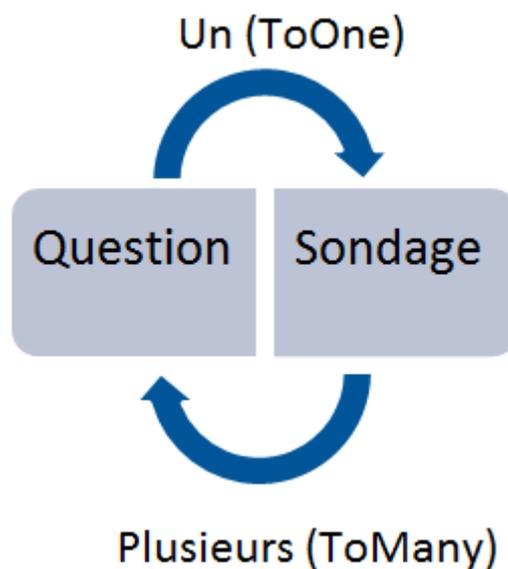


FIGURE 6.6. – Relation entre Question et Sondage

Dans une relation `ManyToOne`, le **many** qualifie l'entité qui doit contenir l'annotation.

III. Les relations avec Doctrine 2

Donc, pour notre cas, l'annotation doit être dans l'entité *question*.

La configuration des entités est donc :

```
1 <?php
2 ##### src/Entity/Poll.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 /**
10  * @ORM\Entity
11  * @ORM\Table(name="polls")
12  */
13 class Poll
14 {
15     /**
16     * @ORM\Id
17     * @ORM\GeneratedValue
18     * @ORM\Column(type="integer")
19     */
20     protected $id;
21
22     /**
23     * @ORM\Column(type="string")
24     */
25     protected $title;
26
27     /**
28     * @ORM\Column(type="datetime")
29     */
30     protected $created;
31
32     /**
33     * @ORM\OneToMany(targetEntity=Question::class, mappedBy="poll")
34     */
35     protected $questions;
36
37     public function __construct()
38     {
39         $this->questions = new ArrayCollection();
40     }
41
42     public function __toString()
43     {
44         $format = "Poll (id: %s, title: %s, created: %s)\n";
45         return sprintf($format, $this->id, $this->title,
46             $this->created->format(\DateTime::ISO8601));
47     }
48 }
```

III. Les relations avec Doctrine 2

```
46     }
47
48     // ...
49
50     public function addQuestion(Question $question)
51     {
52         // Toujours maintenir la relation cohérente
53         $this->questions->add($question);
54         $question->setPoll($this);
55     }
56 }
```

```
1 <?php
2 ##### src/Entity/Question.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 /**
10  * @ORM\Entity
11  * @ORM\Table(name="questions")
12  */
13 class Question
14 {
15     // ...
16
17     /**
18      * @ORM\ManyToOne(targetEntity=Poll::class, inversedBy="questions")
19      */
20     protected $poll;
21
22     // ...
23 }
```

Mettons à jour la base de données :

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql --force
2 CREATE TABLE polls (id INT AUTO_INCREMENT NOT NULL, title
3     VARCHAR(255) NOT NULL, created DATETIME NOT NULL,
4     PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE
5     utf8_unicode_ci ENGINE = InnoDB;
6 ALTER TABLE questions ADD poll_id INT DEFAULT NULL;
7 ALTER TABLE questions ADD CONSTRAINT FK_8ADC54D53C947C0F FOREIGN
8     KEY (poll_id) REFERENCES polls (id);
```

III. Les relations avec Doctrine 2

```
6 CREATE INDEX IDX_8ADC54D53C947C0F ON questions (poll_id);
```

Nous pouvons maintenant tester la création d'un sondage avec quelques questions. La seule nouveauté ici sera l'utilisation des dates PHP.

En effet, *Doctrine* transforme automatiquement les dates pour nous. Le seul élément à prendre en compte est le fuseau horaire. Vu que MySQL ne stocke pas le fuseau horaire dans ses dates, nous devons définir dans PHP un fuseau horaire identique pour toutes les applications qui pourraient d'utiliser les mêmes données.

Nous pouvons utiliser la fonction PHP [date_default_timezone_set](#) ou le paramètre d'initialisation [date.timezone](#) .

```
1 <?php
2 ##### create-poll.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\Poll;
7 use Tuto\Entity\Question;
8 use Tuto\Entity\Answer;
9
10 $poll = new Poll();
11
12 $poll->setTitle("Doctrine 2, ça vous dit ?");
13 $poll->setCreated(new \DateTime("2017-03-03T08:00:00Z"));
14
15 ##### Question 1
16 $questionOne = new Question();
17 $questionOne->setWording("Doctrine 2 est-il un bon ORM ?");
18
19 $yes = new Answer();
20 $yes->setWording("Oui, bien sûr !");
21 $questionOne->addAnswer($yes);
22
23 $no = new Answer();
24 $no->setWording("Non, peut mieux faire.");
25 $questionOne->addAnswer($no);
26
27 ##### Ajout de la question au sondage
28 $poll->addQuestion($questionOne);
29
30
31 ##### Question 2
32 $questionTwo = new Question();
33 $questionTwo->setWording("Doctrine 2 est-il facile d'utilisation ?");
```

III. Les relations avec Doctrine 2

```
34
35 $yesDoc = new Answer();
36 $yesDoc->setWording("Oui, il y a une bonne documentation !");
37 $questionTwo->addAnswer($yesDoc);
38
39 $yesTuto = new Answer();
40 $yesTuto->setWording("Oui, il y a de bons tutoriels !");
41 $questionTwo->addAnswer($yesTuto);
42
43 $no = new Answer();
44 $no->setWording("Non.");
45 $questionTwo->addAnswer($no);
46
47 ##### Ajout de la question au sondage
48 $poll->addQuestion($questionTwo);
49
50
51 $entityManager->persist($questionOne);
52 $entityManager->persist($questionTwo);
53 $entityManager->persist($poll);
54
55 $entityManager->flush();
56
57 echo $poll;
```

Une fois que le sondage est créé, nous pouvons le récupérer et l'afficher. Grâce aux relations que nous avons définies, *Doctrine* peut chercher toutes les informations dont nous avons besoin en se basant juste sur le sondage. Voyez donc par vous-même :

```
1 <?php
2 ##### get-poll.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\Poll;
7
8 $pollRepo = $entityManager->getRepository(Poll::class);
9
10 $poll = $pollRepo->find(1);
11
12 echo $poll;
13 foreach ($poll->getQuestions() as $question) {
14     echo "- ", $question;
15
16     foreach ($question->getAnswers() as $answer) {
17         echo "-- ", $answer;
18     }
19 }
```

III. Les relations avec Doctrine 2

```
19 }
```

Le principe des relations `ManyToOne` reste semblable à une relation `OneToOne`. *Doctrine* utilise la notion de collection afin de bien gérer nos entités.

Nous pouvons donc exploiter l'[API de cette classe](#) pour consulter, ajouter, modifier ou supprimer une entité appartenant à une relation 1..n très facilement.

Notre application reste ainsi cohérente et facile d'utilisation.



Dans les faits, une annotation `OneToMany` ne peut pas exister toute seule. Si la relation 1..n est unidirectionnelle, il faut obligatoirement avoir l'annotation `ManyToOne`. C'est une contrainte que *Doctrine* nous impose (Cf Annexe).

7. Relation ManyToMany - n..m

Notre système de sondage est presque complet. Il ne reste plus qu'à gérer les participations des utilisateurs et notre modèle sera finalisé.

Sachant qu'un utilisateur peut participer à **plusieurs** sondages différents et qu'un sondage peut avoir **plusieurs** participations, nous avons là une relation n..m.

Ce genre de relations est, sans doute, le type de relation le plus complexe mais hélas assez courant.

Doctrine les gère avec l'annotation `ManyToMany`. Comme pour les autres types de relation, une relation `ManyToMany` peut être bidirectionnelle. La configuration étant semblable, nous ne l'aborderons pas.

7.1. Relation ManyToMany simple

Nous pouvons commencer par essayer de recenser tous les utilisateurs qui ont participé à un sondage.

Pour obtenir un schéma de données permettant de gérer cette contrainte, nous sommes obligés d'avoir une table de jointure.

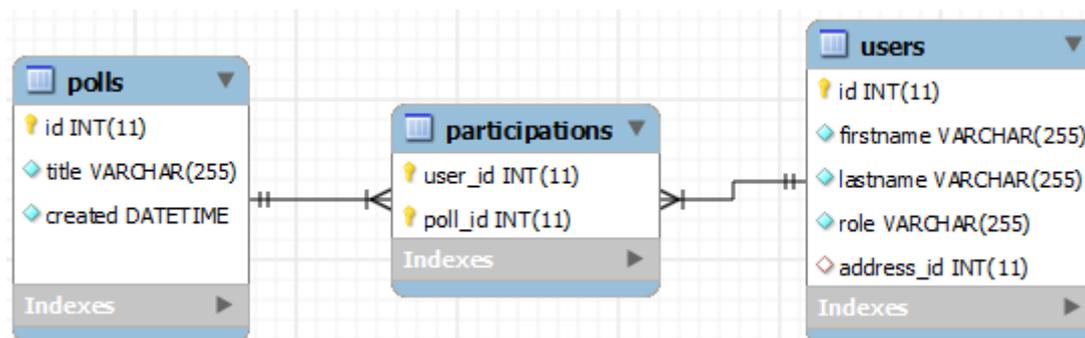


FIGURE 7.1. – Relation ManyToMany : Plusieurs utilisateurs , plusieurs sondages

La configuration de l'annotation `ManyToMany` est très simple. La seule question que nous devons nous poser est :

?

Depuis quelle entité devons-nous faire référence à l'autre ?

III. Les relations avec Doctrine 2

Dans notre cas, nous allons faire un choix **complètement arbitraire**. Depuis un utilisateur, nous serons en mesure de voir tous les sondages auxquels il a participé. L'annotation `ManyToOne` sera donc configurée sur l'entité *utilisateur*.

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 // ...
10 class User
11 {
12     //...
13
14     /**
15      * @ORM\ManyToOne(targetEntity=Poll::class)
16      */
17     protected $polls;
18
19     public function __construct()
20     {
21         $this->polls = new ArrayCollection();
22     }
23
24     // ...
25 }
```

Il faut noter que si nous voulons, par la suite, accéder aux utilisateurs depuis un sondage, nous pourrions rendre la relation bidirectionnelle.

Générons le code SQL avec la commande *Doctrine* :

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql
2
3 CREATE TABLE user_poll (user_id INT NOT NULL, poll_id INT NOT NULL,
4     INDEX IDX_FE3DB68CA76ED395 (user_id),
5     INDEX IDX_FE3DB68C3C947C0F (poll_id), PRIMARY KEY(user_id,
6     poll_id)) DEFAULT CHARACTER SET utf8
7 COLLATE utf8_unicode_ci ENGINE = InnoDB;
8 ALTER TABLE user_poll ADD CONSTRAINT FK_FE3DB68CA76ED395 FOREIGN
9     KEY (user_id)
10 REFERENCES users (id) ON DELETE CASCADE;
11 ALTER TABLE user_poll ADD CONSTRAINT FK_FE3DB68C3C947C0F FOREIGN
12     KEY (poll_id)
```

```
9 REFERENCES polls (id) ON DELETE CASCADE;
```

Le nom de la table de jointure est `user_poll`. Il n'est pas assez explicite et ne permet pas de voir de manière claire la nature de la relation. En plus, nous avons défini une convention au début de ce cours pour mettre tous les noms de table au pluriel.

Mais heureusement, nous pouvons le personnaliser en utilisant l'annotation `JoinTable`.

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 // ...
10 class User
11 {
12     // ...
13
14     /**
15      * @ORM\ManyToMany(targetEntity=Poll::class)
16      * @ORM\JoinTable(name="participations")
17      */
18     protected $polls;
19
20     public function __construct()
21     {
22         $this->polls = new ArrayCollection();
23     }
24
25     // ...
26 }
```

Le nom de la table est maintenant correct. Mais nous n'allons pas encore la créer.

7.2. Relation ManyToMany avec attributs



Comment rajouter la date à laquelle l'utilisateur a participé au sondage ?

Dans un modèle purement SQL, cette information doit se retrouver dans la table *participations*. Mais avec l'ORM, nous avons juste les entités utilisateur et sondage. La date ne peut être lié à aucunes des deux.

III. Les relations avec Doctrine 2

En effet, si la date de participation est liée à l'utilisateur, nous ne pourrions conserver qu'une seule date de participation par utilisateur (celui du dernier sondage).

Et si cette date est liée au sondage, nous ne pourrions conserver que la date de participation d'un seul utilisateur (celui du dernier utilisateur ayant participé au sondage). **Nous atteignons là une limite de la relation *ManyToMany*.**

Mais le problème peut être résolu en utilisant une double relation **ManyToOne**. Considérons que le fait de participer à un sondage est matérialisé par une entité *participation*.

Ainsi, un utilisateur peut avoir **plusieurs** participations (en répondant à plusieurs sondages). Et une participation est créée par **un seul** utilisateur.

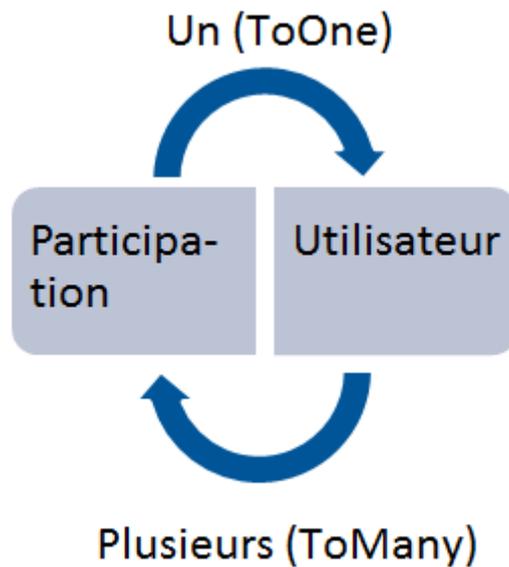


FIGURE 7.2. – Relation entre Participation et Utilisateur

Donc entre une participation et un utilisateur, nous avons une relation **ManyToOne**. Mais cela ne s'arrête pas là !

Un sondage peut avoir **plusieurs** participations (plusieurs utilisateurs peuvent répondre à un même sondage). Une participation est liée à **un** sondage.

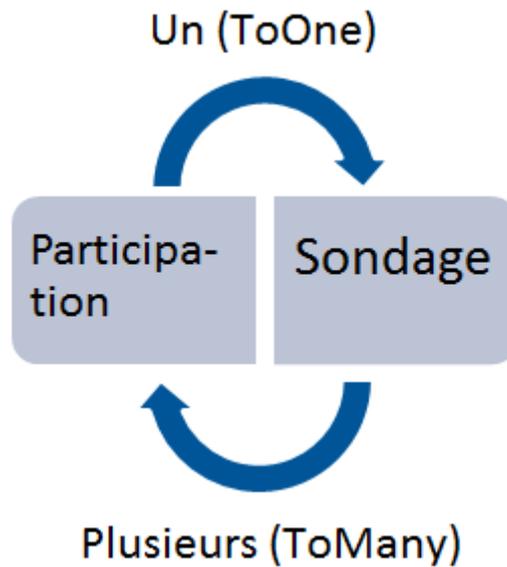


FIGURE 7.3. – Relation entre Participation et Sondage

Nous avons donc aussi une relation `ManyToOne` entre une participation et un sondage.

Nous pouvons maintenant réécrire la relation `ManyToOne`. Il faudra d'abord créer une entité *participation* qui sera liée aux entités *utilisateur* et *sondage*.

```
1 <?php
2 ##### src/Entity/Participation.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="participations",
11     uniqueConstraints={
12         @ORM\UniqueConstraint(name="user_poll_unique", columns={"user_id", "po
13     }
14 )
15 */
16 class Participation
17 {
18     /**
19     * @ORM\Id
20     * @ORM\GeneratedValue
21     * @ORM\Column(type="integer")
22     */
23     protected $id;
24
25     /**
26     * @ORM\Column(type="datetime")
```

III. Les relations avec Doctrine 2

```
27     */
28     protected $date;
29
30     /**
31     * @ORM\ManyToOne(targetEntity=User::class, inversedBy="participations")
32     */
33     protected $user;
34
35     /**
36     * @ORM\ManyToOne(targetEntity=Poll::class)
37     */
38     protected $poll;
39
40     public function __toString()
41     {
42         $format = "Participation (Id: %s, %s, %s)\n";
43         return sprintf($format, $this->id, $this->user,
44             $this->poll);
45     }
46     // ...
47 }
```

Il y a une petite subtilité dans la configuration de celle-ci. Nous avons rajouté une contrainte d'unicité sur les champs `user` et `poll` en utilisant l'annotation `UniqueConstraint`. Son utilisation est proche de celle de l'annotation `Index`.

Avec cette contrainte, nous interdisons à un utilisateur de répondre plusieurs fois à un même sondage.

Pour l'entité *utilisateur*, nous allons juste reconfigurer l'attribut `polls`.

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 // ...
10 class User
11 {
12     // ...
13
14     /**
15     * @ORM\OneToMany(targetEntity=Participation::class, mappedBy="user")
16     */
```

```
17     protected $participations;
18
19     public function __construct()
20     {
21         $this->participations = new ArrayCollection();
22     }
23
24     // ...
25 }
```



Pour l'entité *sondage*, puisque la relation est unidirectionnelle, nous avons aucune modification à faire dessus.

En relançant la commande de mise à jour de la base de données, vous pouvez voir que le code SQL généré est exactement le même que celui dans la relation `ManyToOne`.

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql
2 CREATE TABLE participations (id INT AUTO_INCREMENT NOT NULL,
   user_id INT DEFAULT NULL, poll_id INT DEFAULT NULL,
3 INDEX IDX_FDC6C6E8A76ED395 (user_id), INDEX IDX_FDC6C6E83C947C0F
   (poll_id),
4 UNIQUE INDEX user_poll_unique (user_id, poll_id),
5 PRIMARY KEY(id) DEFAULT CHARACTER SET utf8 COLLATE
   utf8_unicode_ci ENGINE = InnoDB;
6 ALTER TABLE participations ADD CONSTRAINT FK_FDC6C6E8A76ED395
   FOREIGN KEY (user_id) REFERENCES users (id);
7 ALTER TABLE participations ADD CONSTRAINT FK_FDC6C6E83C947C0F
   FOREIGN KEY (poll_id) REFERENCES polls (id);
```



Toute relation `ManyToOne` peut ainsi être décomposée en deux relations `ManyToOne`.

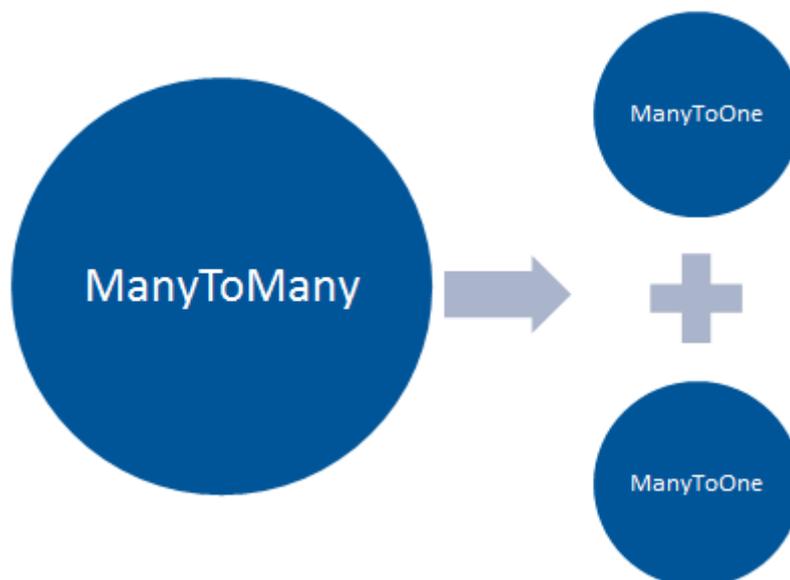


FIGURE 7.4. – Décomposition de la relation ManyToMany

Revenons maintenant à notre problème initial : Comment rajouter la date à laquelle l'utilisateur a participé à un sondage ?

Avec notre nouveau modèle de données, la réponse est simple. Il suffit de rajouter la date de participation à l'entité *participation*.

```
1 <?php
2 ##### src/Entity/Participation.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10  * @ORM\Table(name="participations",
11  *           uniqueConstraints={
12  *             @ORM\UniqueConstraint(name="user_poll_unique", columns={"user_id", "po
13  *           })
14  * }
15  */
16 class Participation
17 {
18     /**
19     * @ORM\Id
20     * @ORM\GeneratedValue
21     * @ORM\Column(type="integer")
22     */
23     protected $id;
```

```
24
25     /**
26     * @ORM\Column(type="datetime")
27     */
28     protected $date;
29
30     /**
31     * @ORM\ManyToOne(targetEntity=User::class, inversedBy="participations")
32     */
33     protected $user;
34
35     /**
36     * @ORM\ManyToOne(targetEntity=Poll::class)
37     */
38     protected $poll;
39
40     public function __toString()
41     {
42         $format = "Participation (Id: %s, %s, %s)\n";
43         return sprintf($format, $this->id, $this->user,
44             $this->poll);
45     }
46     // ...
47 }
```

Nous pouvons maintenant mettre à jour la base de données.

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql --force
2 CREATE TABLE participations (id INT AUTO_INCREMENT NOT NULL,
3   user_id INT DEFAULT NULL, poll_id INT DEFAULT NULL,
4   date DATETIME NOT NULL, INDEX IDX_FDC6C6E8A76ED395 (user_id),
5   INDEX IDX_FDC6C6E83C947C0F (poll_id),
6   UNIQUE INDEX user_poll_unique (user_id, poll_id), PRIMARY KEY(id))
7   DEFAULT CHARACTER SET utf8
8   COLLATE utf8_unicode_ci ENGINE = InnoDB;
9 ALTER TABLE participations ADD CONSTRAINT FK_FDC6C6E8A76ED395
10   FOREIGN KEY (user_id) REFERENCES users (id);
11 ALTER TABLE participations ADD CONSTRAINT FK_FDC6C6E83C947C0F
12   FOREIGN KEY (poll_id) REFERENCES polls (id);
```

7.3. Création des participations

Nous allons tester notre configuration en créant une participation au sondage « Doctrine 2, ça vous dit ? ».

```
1 <?php
2 ##### create-participation.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\Participation;
7 use Tuto\Entity\Poll;
8 use Tuto\Entity\User;
9
10 $participation = new Participation();
11
12 $participation->setDate(new \DateTime("2017-03-03T09:00:00Z"));
13
14 $pollRepo = $entityManager->getRepository(Poll::class);
15 $poll = $pollRepo->find(1);
16
17 $userRepo = $entityManager->getRepository(User::class);
18 $user = $userRepo->find(4);
19
20 $participation->setUser($user);
21 $participation->setPoll($poll);
22
23 $entityManager->persist($participation);
24 $entityManager->flush();
25
26 echo $participation;
```



Grâce à la clé primaire que nous avons définie, un utilisateur ne pourra pas participer deux fois au même sondage. Si vous ré-exécutez l'extrait de code, vous aurez une erreur fatale (Fatal error : Uncaught PDOException : SQLSTATE[23000] : Integrity constraint violation : 1062 Duplicata du champ '4-1' pour la clef 'user_poll_unique').

Chacune des annotations que nous venons de voir comporte un ensemble de paramètres qui permettent de les personnaliser. N'hésitez donc pas à consulter [la documentation officielle](#) sur les associations pour approfondir le sujet.

La référence complète est disponible sur [le site de Doctrine](#).

8. TP : Finir la modélisation du système de sondage

Rien ne vaut la pratique pour bien saisir tout ce que nous venons de voir. Nous allons donc finaliser la modélisation de notre application en rajoutant la gestion des choix des utilisateurs qui répondent aux sondages.

Il y aura une proposition de solution pour le problème posé mais il est utile de rappeler qu'il est préférable d'essayer par soi-même avant de la consulter.

Servez-vous des indices pour avancer progressivement.

8.1. Pratiquons

8.1.1. Énoncé

Lorsqu'un utilisateur participe à un sondage, il choisit une réponse pour chacune des questions de celui-ci.

Notre objectif est de stocker tous les choix que l'utilisateur a faits pour chacune de ses participations aux sondages et de pouvoir retrouver rapidement ces choix à partir d'un utilisateur.

Pour élargir le système de sondage, nous considérons aussi qu'il est possible d'avoir des questions avec plusieurs réponses possibles (question à choix multiple).

8.1.2. Indices

- Les choix des utilisateurs doivent être matérialisés par une entité.
- Un choix fait intervenir une question et une ou plusieurs réponses.
- Les participations des utilisateurs sont matérialisées grâce à l'entité *participation*. Il faut donc la mettre à jour pour rajouter les choix.

8.2. Proposition de solution

8.2.1. Définition et implémentation du schéma

Nous allons comment par modéliser l'élément le plus élémentaire dans la gestion des choix des utilisateurs : un choix pour une question donnée.

III. Les relations avec Doctrine 2

Un choix représente le ou les réponses que l'utilisateur a choisies pour une question donnée. De cette description, nous pouvons déduire deux informations :

- un choix est en relation avec **une seule** question du sondage (la question à laquelle l'utilisateur répond) ;
- et elle est en relation avec **une ou plusieurs** réponses (les réponses sont liées à la question à laquelle l'utilisateur répond).

Pour décrire les relations avec *Doctrine*, nous pouvons dire que :

- Un choix est lié à **une** question.
- Une question est liée à **plusieurs** choix (plusieurs utilisateurs peuvent répondre à une même question).

Entre l'entité *choix* et *question*, nous avons une relation **ManyToOne**.

```
1 <?php
2 #####src/Entity/Choice.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity
10 * @ORM\Table(name="choices")
11 */
12 class Choice
13 {
14     /**
15     * @ORM\Id
16     * @ORM\GeneratedValue
17     * @ORM\Column(type="integer")
18     */
19     protected $id;
20
21     /**
22     * @ORM\ManyToOne(targetEntity=Question::class)
23     */
24     protected $question;
25
26     public function __toString()
27     {
28         $format = "Choice (id: %s)\n";
29         return sprintf($format, $this->id);
30     }
31
32     // ...
33 }
```

III. Les relations avec Doctrine 2

Dans la même logique,

- Un choix peut contenir **plusieurs** réponses (pour les questions à choix multiple).
- Une réponse est liée à **plusieurs** choix (plusieurs utilisateurs peuvent choisir les mêmes réponses pour une question donnée).

Nous avons donc une relation **ManyToMany** entre les entités *choix* et *réponse* .

```
1 <?php
2 #####src/Entity/Choice.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 /**
10  * @ORM\Entity
11  * @ORM\Table(name="choices")
12  */
13 class Choice
14 {
15     /**
16      * @ORM\Id
17      * @ORM\GeneratedValue
18      * @ORM\Column(type="integer")
19      */
20     protected $id;
21
22     /**
23      * @ORM\ManyToOne(targetEntity=Question::class)
24      */
25     protected $question;
26
27     /**
28      * @ORM\ManyToMany(targetEntity=Answer::class)
29      * @ORM\JoinTable(name="selected_answers")
30      */
31     protected $answers;
32
33     public function __construct()
34     {
35         $this->answers = new ArrayCollection();
36     }
37
38     public function __toString()
39     {
40         $format = "Choice (id: %s)\n";
41         return sprintf($format, $this->id);
42     }
43 }
```

```
43
44     public function addAnswer(Answer $answer)
45     {
46         if ($answer->getQuestion() == $this->question) {
47             $this->answers->add($answer);
48         }
49     }
50
51     public function getId()
52     {
53         return $this->id;
54     }
55
56     public function setId($id)
57     {
58         $this->id = $id;
59     }
60
61     public function getQuestion()
62     {
63         return $this->question;
64     }
65
66     public function setQuestion($question)
67     {
68         $this->question = $question;
69     }
70
71     public function getAnswers()
72     {
73         return $this->answers;
74     }
75
76     public function getParticipation()
77     {
78         return $this->participation;
79     }
80
81     public function setParticipation($participation)
82     {
83         $this->participation = $participation;
84     }
85 }
```

L'annotation `JoinTable` sur l'attribut `answers` nous permet de nommer la table qui contiendra les réponses choisies : `selected_answers`.

Maintenant que le choix est modélisé, nous devons le relier au sondage. En répondant à un sondage, l'utilisateur fait un choix pour chaque question. Pour rappel, nous avons déjà une entité *participation* qui nous permet déjà de relier un utilisateur et le sondage auquel il participe.

III. Les relations avec Doctrine 2

La participation a un sondage peut donc être complétée en enregistrant tous les choix de l'utilisateur.

- Une participation contient **plusieurs** choix.
- Un choix est lié à **une** participation.

La relation entre l'entité *participation* et *choix* est une relation *ManyToOne*. Nous allons la rendre bidirectionnelle pour pouvoir récupérer tous les choix depuis la participation d'un utilisateur.

```
1 <?php
2 ##### src/Entity/Participation.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 // ...
10 class Participation
11 {
12     // ...
13
14     /**
15      * @ORM\OneToMany(targetEntity=Choice::class, mappedBy="participation")
16      */
17     protected $choices;
18
19     public function __construct()
20     {
21         $this->choices = new ArrayCollection();
22     }
23
24     // ...
25
26     public function addChoice(Choice $choice)
27     {
28         $this->choices->add($choice);
29         $choice->setParticipation($this);
30     }
31 }
```

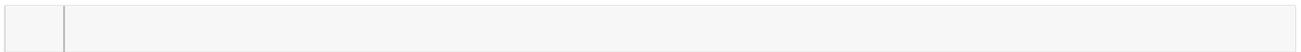
La mise à jour de la base de données donne :

```
1 -- vendor/bin/doctrine orm:schema-tool:update --dump-sql --force
2 CREATE TABLE choices (id INT AUTO_INCREMENT NOT NULL, question_id
3   INT DEFAULT NULL,
4   participation_id INT DEFAULT NULL, INDEX IDX_5CE96391E27F6BF
5   (question_id),
```

```
4 INDEX IDX_5CE96396ACE3B73 (participation_id), PRIMARY KEY(id)
  DEFAULT CHARACTER SET utf8
5 COLLATE utf8_unicode_ci ENGINE = InnoDB;
6 CREATE TABLE selected_answers (choice_id INT NOT NULL, answer_id
  INT NOT NULL, INDEX IDX_315F9F94998666D1 (choice_id),
7 INDEX IDX_315F9F94AA334807 (answer_id), PRIMARY KEY(choice_id,
  answer_id)) DEFAULT CHARACTER SET utf8 COLLATE
8 utf8_unicode_ci ENGINE = InnoDB;
9 ALTER TABLE choices ADD CONSTRAINT FK_5CE96391E27F6BF FOREIGN KEY
  (question_id) REFERENCES questions (id);
10 ALTER TABLE choices ADD CONSTRAINT FK_5CE96396ACE3B73 FOREIGN KEY
  (participation_id) REFERENCES participations (id);
11 ALTER TABLE selected_answers ADD CONSTRAINT FK_315F9F94998666D1
  FOREIGN KEY (choice_id) REFERENCES choices (id)
12 ON DELETE CASCADE;
13 ALTER TABLE selected_answers ADD CONSTRAINT FK_315F9F94AA334807
  FOREIGN KEY (answer_id) REFERENCES answers (id)
14 ON DELETE CASCADE;
```

8.2.2. Tests

Nous allons tester notre implémentation répondant au sondage que nous avons créé dans le chapitre précédent. Pour rappel, voici le contenu du sondage :



Pour la première question, l'utilisateur va répondre « Oui, bien sûr ! » et pour la seconde, il choisira les deux réponses « Oui ».

```
1 <?php
2 #####create-full-participation.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
  'bootstrap.php']);
5
6 use Tuto\Entity\Participation;
7 use Tuto\Entity\Poll;
8 use Tuto\Entity\User;
9 use Tuto\Entity\Choice;
10
11 $participation = new Participation();
12
13 $participation->setDate(new \DateTime("2017-03-04T10:00:00Z"));
14 $entityManager->persist($participation);
```

III. Les relations avec Doctrine 2

```
15 $entityManager->flush(); // Nous devons flusher une première fois
    pour avoir un identifiant pour la participation
16
17 $pollRepo = $entityManager->getRepository(Poll::class);
18 $poll = $pollRepo->find(1);
19
20 $userRepo = $entityManager->getRepository(User::class);
21 $user = $userRepo->find(3);
22
23
24 $participation->setUser($user);
25 $participation->setPoll($poll);
26
27 /*
28 Poll (id: 1, title: Doctrine 2, ça vous dit ?, created:
    2017-03-03T08:00:00+0000)
29 - Question (id: 2, wording: Doctrine 2 est-il un bon ORM ?)
30 -- Answer (id: 3, wording: Oui, bien sûr !)
31 -- Answer (id: 4, wording: Non, peut mieux faire.)
32 - Question (id: 3, wording: Doctrine 2 est-il facile d'utilisation
    ?)
33 -- Answer (id: 5, wording: Oui, il y a une bonne documentation !)
34 -- Answer (id: 6, wording: Oui, il y a de bons tutoriels !)
35 -- Answer (id: 7, wording: Non.)
36 */
37
38 $questions = $poll->getQuestions();
39
40 // Choix pour la première question
41 $questionOne = $questions->get(0);
42 $answers = $questionOne->getAnswers();
43
44 $choice = new Choice();
45 $choice->setQuestion($questionOne);
46 $choice->addAnswer($answers->get(0));
47
48 $entityManager->persist($choice);
49 $participation->addChoice($choice);
50
51 // Choix pour la deuxième question
52 $questionTwo = $questions->get(1);
53 $answers = $questionTwo->getAnswers();
54
55 $choice = new Choice();
56 $choice->setQuestion($questionTwo);
57 $choice->addAnswer($answers->get(0));
58 $choice->addAnswer($answers->get(1));
59
60 $entityManager->persist($choice);
61 $participation->addChoice($choice);
```

III. Les relations avec Doctrine 2

```
62
63 $entityManager->flush();
64
65 echo $participation;
```

Une fois la participation créée, nous pouvons maintenant afficher toutes les participations de l'utilisateur (avec un peu de formatage pour une meilleure lisibilité).

```
1 <?php
2 ##### get-user-participations.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7
8 $userRepo = $entityManager->getRepository(User::class);
9 $user = $userRepo->find(3);
10
11 $participations = $user->getParticipations();
12
13 foreach ($participations as $participation) {
14     echo $participation;
15     $choices = $participation->getChoices();
16     foreach ($choices as $choice) {
17         echo "- ", $choice;
18         $answers = $choice->getAnswers();
19         echo "-- ", $choice->getQuestion();
20         foreach ($answers as $answer) {
21             echo "--- ", $answer;
22         }
23     }
24 }
```

Voilà donc un exemple d'implémentation qui permet d'avoir les participations de chaque utilisateur.

Le schéma complet de notre base de données ressemble maintenant à :

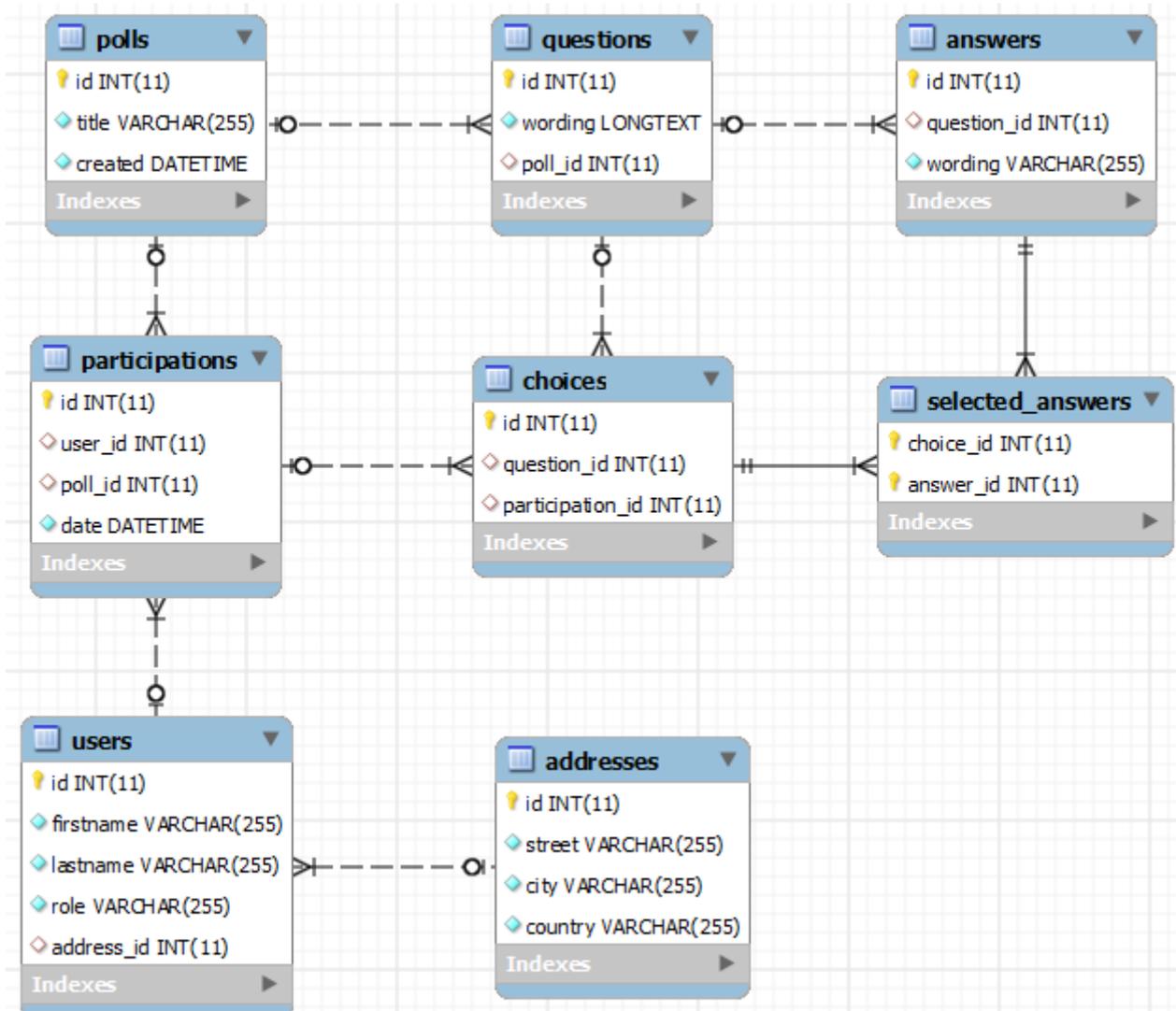


FIGURE 8.1. – Schéma complet de la base de données

8.3. Idées d'amélioration

Bien que le système soit fonctionnel, il est toujours possible de l'améliorer.

Nous pouvons par exemple pour chaque question, définir le type de réponses attendus (choix multiple, choix unique, etc.). Cela permettrait par exemple lors de l'affichage de la question sur un site web de faire le choix entre des cases à cocher (pour le choix multiple) ou des boutons radio (pour le choix unique du style *Vrai ou Faux*).

Un utilisateur pourrait mettre en favoris un sondage pour pouvoir retrouver plus tard le résultat de celui-ci (sans y participer).

N'hésitez donc pas à le modifier pour bien saisir les différentes subtilités que peut cacher *Doctrine* (tester les attributs des annotations utilisées, tester les opérations de cascade, rajouter des contraintes, etc.).

III. Les relations avec Doctrine 2

Nous avons maintenant toutes les bases et les rudiments pour concevoir le modèle de données d'une application PHP complète.

Bien qu'il soit possible de personnaliser *Doctrine*, il faut toujours essayer de garder les conventions par défaut afin d'avoir un code facilement compréhensible et maintenable.

En réalité, nous n'avons pour l'instant exploré qu'une petite partie des possibilités que nous offre *Doctrine*.

Dans la partie suivante, nous allons voir comment tirer le meilleur parti d'une base de données grâce aux outils que *Doctrine* met à notre disposition.

Voici un résumé de cette partie fourni par [artragis](#) sous licence [CC-BY](#).

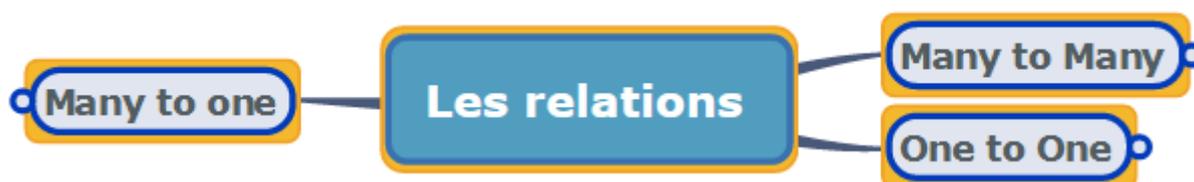


FIGURE 8.2. – Résumé des relations avec Doctrine 2

Vous pouvez consulter le [lien interactif](#) pour plus de détails.

Quatrième partie

Exploiter une base de données avec Doctrine 2

IV. Exploiter une base de données avec Doctrine 2

À quoi bon avoir un modèle de données complet si nous ne sommes pas en mesure de l'exploiter de manière optimale ?

Doctrine permet non seulement de concevoir un modèle complet mais il fournit aussi un ensemble d'outils nous permettant d'en tirer le meilleur. Nous avons entre autres :

- la construction des requêtes complexes profitant de la puissance du langage SQL ;
- les jointures pour optimiser nos requêtes ;
- ou encore les transactions ;

Dans cette dernière partie, nous aborderons donc toutes les notions nécessaires à une utilisation optimale de Doctrine pour une application pleinement fonctionnelle.

9. À la rencontre du QueryBuilder

Le langage SQL possède un grand nombre d'opérations. En une requête, nous pouvons calculer des sommes, des moyennes, faire des jointures entre différentes tables, gérer des transactions, etc.

À ce stade, il nous manque beaucoup de fonctionnalités du langage SQL que les méthodes simples de la famille *findXXX* ne peuvent pas fournir.

Dans cette partie, nous allons voir un ensemble de concepts qui nous permettront de rendre une application utilisant *Doctrine* fonctionnelle, efficace et performante.

9.1. Le QueryBuilder

Le *QueryBuilder* ⁶ est une classe permettant de créer des requêtes en utilisant le langage PHP. Pour avoir un aperçu de cette classe, voici un extrait issu de [la documentation officielle de Doctrine](#) ⁷ de son API.

```
1 <?php
2 class QueryBuilder
3 {
4     public function select($select = null);
5
6     public function addSelect($select = null);
7
8     public function delete($delete = null, $alias = null);
9
10    public function update($update = null, $alias = null);
11
12    public function set($key, $value);
13
14    public function from($from, $alias, $indexBy = null);
15
16    public function join($join, $alias, $conditionType = null,
17        $condition = null, $indexBy = null);
18
19    public function innerJoin($join, $alias, $conditionType = null,
20        $condition = null, $indexBy = null);
21
22    public function leftJoin($join, $alias, $conditionType = null,
23        $condition = null, $indexBy = null);
24 }
```

```
21
22     public function where($where);
23
24     public function andWhere($where);
25
26     public function orWhere($where);
27
28     public function groupBy($groupBy);
29
30     public function addGroupBy($groupBy);
31
32     public function having($having);
33
34     public function andHaving($having);
35
36     public function orHaving($having);
37
38     public function orderBy($sort, $order = null);
39
40     public function addOrderBy($sort, $order = null);
41 }
```

À la lecture de l'API de cette classe, nous pouvons retrouver pas mal de mots clés SQL dans les méthodes disponibles (SELECT, UPDATE, DELETE, GROUP BY, HAVING, etc.). En effet, cette classe rajoute une couche d'abstraction entre notre code PHP et les requêtes SQL effectives que nous voulons exécuter.

Doctrine se charge ensuite de traduire la requête dans la bonne syntaxe SQL en prenant bien sûr en compte le moteur de base de données que nous utilisons (MySQL, PostgreSQL, MSSQL, Oracle, etc.).

9.1.1. Création d'un *QueryBuilder*

La meilleure façon d'appréhender le mode de fonctionnement du *QueryBuilder* est d'utiliser des exemples.

```
1 <?php
2 ##### query-builder.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7
8 $userRepo = $entityManager->getRepository(User::class);
9
10 $queryBuilder = $entityManager->createQueryBuilder();
```

```
11
12 $queryBuilder->select('u')
13     ->from(User::class, 'u')
14     ->where('u.firstname = :firstname')
15     ->setParameter('firstname', 'First');
16
17 $query = $queryBuilder->getQuery();
18
19 echo $query->getDQL(), "\n";
20 echo $query->getOneOrNullResult();
```

Une fois n'est pas de coutume. L'*entity manager* est utilisé pour créer notre *QueryBuilder*. Ensuite, nous construisons une requête SQL avec du code PHP.

La première ligne du résultat a des airs de famille avec le SQL. Au lieu d'avoir le nom du table, nous avons le nom d'une entité. L'alias « u » est utilisé pour récupérer tous les attributs de l'entité (équivalent du `SELECT *`). Le langage utilisé s'appelle le **DQL** : *Doctrine Query Langage*.

C'est un langage de requête conçu à l'image de *Hibernate Query Language* (HQL), un ORM pour Java, qui permet de faire des requêtes en utilisant nos classes PHP.

Il faut voir le DQL comme étant une couche par-dessus le SQL qui utilise nos entités comme des tables et les attributs de ceux-ci comme des noms de colonne. Il permet de réaliser ainsi presque toutes les requêtes que supporte le SQL natif en se basant exclusivement sur notre modèle objet.

La configuration du *QueryBuilder* supporte le système de paramètres nommés à l'image de PDO. Son utilisation est très grandement recommandée pour éviter les injections SQL.

Il est aussi intéressant de souligner que pour gérer les paramètres nommés passés au *QueryBuilder*, nous avons le choix entre des entiers qui doivent être préfixés par le caractère « ? », ou des chaînes de caractères qui doivent être préfixées par le caractère « : ».

9.1.2. Exemples de requêtes avec le *QueryBuilder*

Pour mieux appréhender son comportement, nous allons voir quelques exemples de requêtes basées sur le *QueryBuilder*.

9.1.2.1. Suppression d'un utilisateur

```
1 <?php
2 $queryBuilder = $entityManager->createQueryBuilder();
3 $queryBuilder->delete(User::class, 'u')
4     ->where('u.id = :id')
5     ->setParameter('id', 5);
6
7 $query = $queryBuilder->getQuery();
8
9 echo $query->getDQL(), "\n";
10 echo $query->execute();
```

Résultat :

--

9.1.2.2. Recherche d'un utilisateur

```
1 <?php
2 $queryBuilder = $entityManager->createQueryBuilder();
3 $queryBuilder->select('u')
4     ->from(User::class, 'u')
5     ->where('u.firstname LIKE :firstname')
6     ->andWhere('u.lastname = :lastname')
7     ->setParameter('firstname', 'First %')
8     ->setParameter('lastname', 'LAST 3');
9
10 $query = $queryBuilder->getQuery();
11
12 echo $query->getDQL(), "\n";
13 foreach ($query->getResult() as $user) {
14     echo $user;
15 }
```

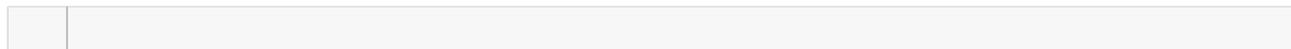
Résultat :

--

9.1.2.3. Recherche de plusieurs utilisateurs

```
1 <?php
2 $queryBuilder = $entityManager->createQueryBuilder();
3 $queryBuilder->select('u')
4     ->from(User::class, 'u')
5     ->where('u.id IN (:ids)')
6     ->setParameter('ids', [3,4]);
7
8 $query = $queryBuilder->getQuery();
9
10 echo $query->getDQL(), "\n";
11 foreach ($query->getResult() as $user) {
12     echo $user;
13 }
```

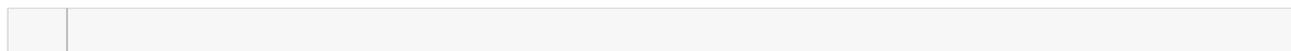
Résultat :



9.1.2.4. Recherche de plusieurs utilisateurs avec des limitations

```
1 <?php
2 $queryBuilder = $entityManager->createQueryBuilder();
3 $queryBuilder->select('u')
4     ->from(User::class, 'u')
5     ->setFirstResult(5)
6     ->setMaxResults(3);
7
8 $query = $queryBuilder->getQuery();
9
10 echo $query->getDQL(), "\n";
11 foreach ($query->getResult() as $user) {
12     echo $user;
13 }
```

Ici, nous affichons aussi le SQL pour bien voir l'utilisation des limitations (LIMIT et OFFSET).



9.1.2.5. Mise à jour d'un utilisateur

```
1 <?php
2 $address = new Address();
3 $address->setStreet("Place d'Armes");
4 $address->setCity("Versailles");
5 $address->setCountry("France");
6 $entityManager->persist($address);
7 $entityManager->flush();
8
9 $queryBuilder = $entityManager->createQueryBuilder();
10 $queryBuilder->update(User::class, 'u')
11     ->set('u.address', '?1')
12     ->where('u.id = ?2')
13     ->setParameter(1, $address->getId())
14     ->setParameter(2, 8);
15
16 $query = $queryBuilder->getQuery();
17
18 echo $query->getDQL(), "\n";
19 echo $query->execute(), "\n";
20 echo $userRepo->find(8), "\n";
```

Résultat :

9.1.2.6. Récupérer le nombre d'utilisateurs dans la base

```
1 <?php
2 $queryBuilder = $entityManager->createQueryBuilder();
3
4 $queryBuilder->select('COUNT(u.id)')
5     ->from(User::class, 'u');
6
7 $query = $queryBuilder->getQuery();
8
9 echo $query->getDQL(), "\n";
10 echo $query->getSingleScalarResult();
```

Résultat :

IV. Exploiter une base de données avec Doctrine 2

Sans le *QueryBuilder*, nous étions obligés de récupérer une entité *utilisateur* avant de pouvoir la modifier ou la supprimer. Ce qui impliquait l'exécution d'au moins deux requêtes.

Maintenant, nous avons tous les outils nécessaires pour éditer une ou plusieurs entités en une seule requête, limiter le nombre de résultat avec les méthodes `setFirstResult` et `setMaxResults` ou encore faire des jointures à l'image du langage SQL.

i

Il est possible d'exécuter directement du DQL en utilisant la méthode `createQuery` de l'*entity manager*. Mais nous n'aborderons pas ce cas d'usage. La [documentation officielle](#) comporte beaucoup d'exemples sur ce sujet.

9.1.3. Personnaliser les réponses du *QueryBuilder*

À chaque requête, nous avons le choix du type de réponses que nous souhaitons. Ainsi, suivant la méthode utilisée, *Doctrine* peut nous envoyer une entité, une liste, un tableau ou même un scalaire.

Méthode appelée	Retour
<code>\$query->getResult()</code>	Une liste d'entités ou le nombre de lignes mis à jour par la requête (mise à jour ou suppression)
<code>\$query->getSingleResult()</code>	Une seule entité (lève une exception si le résultat est nul ou contient plusieurs entités)
<code>\$query->getOneOrResult()</code>	Une seule entité (renvoie null si la réponse est vide, et lève une exception si elle contient plusieurs entités)
<code>\$query->getArrayResult()</code>	Une liste de tableaux
<code>\$query->getScalarResult()</code>	Une liste de scalaires
<code>\$query->getSingleScalarResult()</code>	Un seul scalaire (lève une exception si le résultat est nul ou contient plusieurs scalaires)

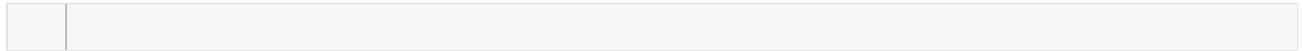
Même si le besoin est rare, nous pouvons quand même personnaliser encore plus les réponses du *QueryBuilder*. Les méthodes `getResult`, `getSingleResult`, `getOneOrResult` acceptent un paramètre `hydrationMode` qui permet de choisir entre une entité (*choix par défaut*) ou un tableau. Pour récupérer, par exemple, les informations d'un seul utilisateur en tant que tableau, nous pouvons écrire :

```
1 <?php
2 $queryBuilder = $entityManager->createQueryBuilder();
3
4 $queryBuilder->select('u')
```

IV. Exploiter une base de données avec Doctrine 2

```
5     ->from(User::class, 'u')
6     ->where('u.id = :id')
7     ->setParameter('id', 1);
8
9     $query = $queryBuilder->getQuery();
10
11    var_dump($query->getSingleResult(Doctrine\ORM\Query::HYDRATE_ARRAY));
```

Nous obtenons un tableau associatif avec comme clé les noms des attributs.



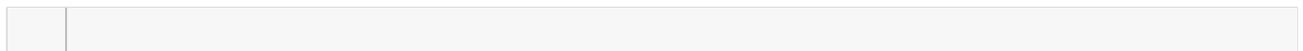
9.1.4. Les expressions

Dans tous nos exemples, nous avons utilisé des mots clés SQL comme *LIKE*, *COUNT*, etc. *Doctrine* nous permet de nous affranchir de cela en utilisant les expressions du *QueryBuilder*.

Voici un exemple d'utilisation pour le *COUNT* :

```
1 <?php
2 $queryBuilder = $entityManager->createQueryBuilder();
3
4 $queryBuilder->select($queryBuilder->expr()->count('u.id'))
5     ->from(User::class, 'u');
6
7 $query = $queryBuilder->getQuery();
8
9 echo $query->getDQL(), "\n";
10 echo $query->getSingleScalarResult(), "\n";
```

Même si le code change, la requête générée reste néanmoins la même :



Vous pouvez consulter l'[API de Doctrine](#) pour voir toutes les méthodes disponibles.

Les expressions sont souvent qualifiées de complexes pour le bénéfice qu'il rajoute au code. Il est donc courant de voir des applications utilisant *Doctrine* sans ce composant.

?

Nous avons créé beaucoup de requêtes qui pourraient être utiles dans différentes parties de notre code. Comment faire pour réutiliser les requêtes avancées que nous venons de créer ?

Nous allons répondre à cette question en explorant les *repositories* personnalisés de *Doctrine*.

9.2. Les repositories personnalisés

9.2.1. Configuration

Lorsque nous avons fait appel à l'*entity manager* pour récupérer un *repository*, *Doctrine* avait créé pour nous un *repository* qui dispose nativement d'un ensemble de méthodes utilitaires pour faire des requêtes de base.

Cependant, nous avons la possibilité, pour chaque entité, de personnaliser le *repository* que *Doctrine* va utiliser.

La configuration de celui-ci se fait en deux temps :

1. Pour commencer, il faut créer une classe qui étend la classe `EntityRepository` ;
2. Ensuite, il faut demander à *Doctrine* d'utiliser ce *repository*.

Nous allons tester cette configuration avec les utilisateurs. Le *repository* ressemble donc à :

```
1 <?php
2 ##### src/Repository/UserRepository.php
3
4 namespace Tuto\Repository;
5
6 use Doctrine\ORM\EntityRepository;
7
8 class UserRepository extends EntityRepository
9 {
10 }
```

Pour l'utiliser, nous devons mettre à jour les annotations sur l'entité *utilisateur*. Vous l'aurez peut-être remarqué, les entités représente la glu entre notre code et *Doctrine*. Toutes les configurations se situent sur celles-ci. Cela permet de centraliser tout ce qui est lié à l'ORM.

Pour déclarer un *repository*, il suffit juste d'utiliser l'attribut `repositoryClass` de l'annotation `Entity`.

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Tuto\Repository\UserRepository;
```

6. Le terme *QueryBuilder* signifie littéralement « constructeur de requêtes ».

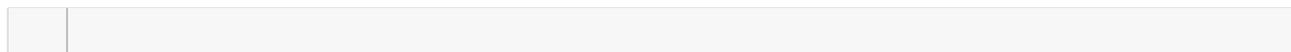
IV. Exploiter une base de données avec Doctrine 2

```
7 use Doctrine\Common\Collections\ArrayCollection;
8 use Doctrine\ORM\Mapping as ORM;
9
10 /**
11  * @ORM\Entity(repositoryClass=UserRepository::class)
12  * // ...
13  */
14 class User
15 {
16     // ...
17 }
```

Nous pouvons maintenant tester cette configuration.

```
1 <?php
2 ##### custom-repository.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7 use Tuto\Entity\Poll;
8
9 $userRepo = $entityManager->getRepository(User::class);
10 echo get_class($userRepo), "\n";
11
12 $pollRepo = $entityManager->getRepository(Poll::class);
13 echo get_class($pollRepo), "\n";
```

Résultat :



Notre *repository* est bien récupéré par la méthode `getRepository`. Et puisque nous étendons la classe `EntityRepository`, les méthodes déjà vues durant ce cours sont tous valides.

9.2.2. Utilisation des *repositories* personnalisés

Ce *repository* peut maintenant être utilisé à la place du *repository* natif de *Doctrine*. Dans l'API de celui, nous disposons de plusieurs méthodes qui nous permettent d'accéder indirectement à l'*entity manager*.

Toutes les requêtes basées sur le *QueryBuilder* peuvent être ainsi encapsulées dans le *repository*. Le code utilisé reste identique à ce que nous avons déjà utilisé pour illustrer l'API du *QueryBuilder*.

IV. Exploiter une base de données avec Doctrine 2

```
1 <?php
2 ##### src/Repository/UserRepository.php
3
4 namespace Tuto\Repository;
5
6 use Doctrine\ORM\EntityRepository;
7 use Tuto\Entity\User;
8
9 class UserRepository extends EntityRepository
10 {
11     public function searchByFirstname($firstname)
12     {
13         $queryBuilder = $this->_em->createQueryBuilder()
14             ->select('u')
15             ->from(User::class, 'u')
16             ->where('u.firstname = :firstname')
17             ->setParameter('firstname', $firstname);
18
19         $query = $queryBuilder->getQuery();
20
21         return $query->getOneOrNullResult();
22     }
23
24     public function deleteById($id)
25     {
26         $queryBuilder = $this->_em->createQueryBuilder();
27         $queryBuilder->delete(User::class, 'u')
28             ->where('u.id = :id')
29             ->setParameter('id', $id);
30
31         $query = $queryBuilder->getQuery();
32
33         return $query->execute();
34     }
35 }
```

Maintenant à chaque fois que nous accédons au *repository*, nous pouvons utiliser ces méthodes.

```
1 <?php
2 ##### custom-repository.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6
7 use Tuto\Entity\User;
8
9 $userRepo = $entityManager->getRepository(User::class);
```

IV. Exploiter une base de données avec Doctrine 2

```
9  
10 echo $userRepo->searchByFirstname("First");
```

N'hésitez pas à implémenter les autres méthodes pour vous entraîner.

i

Il est certes possible d'utiliser directement l'*entity manager* pour créer des *QueryBuilder*. Mais en les mettant dans des *repositories* personnalisés, nous pouvons centraliser toutes les requêtes et les réutiliser sans duplication de code.

Le *QueryBuilder* offre une API puissante qui nous permet de tirer un maximum de profit des fonctionnalités de notre base de données. Son principe d'utilisation est simple et proche du SQL natif. Vous ne devriez donc pas être trop perdu en l'utilisant dans vos applications.

L'ORM ne représente malheureusement pas la solution miracle et il existe des cas où utiliser des requêtes SQL natives sera inévitable.

Donc, si ce qui est présenté ne remplit pas entièrement votre besoin, il faut savoir qu'il est possible avec *Doctrine* d'[exécuter des requête SQL natives](#) et même d'associer le résultat à un objet.

10. Optimiser l'utilisation de Doctrine

L'un des principaux reproches que les développeurs font aux ORMs est leurs performances modestes.

Rajouter une couche d'abstraction à la base de données ne se fait pas sans perte. Si notre application faisait des insertions ou lectures massives de données (des millions sur un temps très court), le choix d'un ORM serait effectivement très discutable.

Mais pour la plupart des applications, *Doctrine* peut grandement faire l'affaire. En plus, il est tout à fait possible de faire cohabiter *Doctrine* avec des requêtes SQL natives.

Nous allons donc voir les optimisations possibles pour qu'une application PHP utilisant *Doctrine* soit la plus performante possible.

10.1. Tracer les requêtes Doctrine

Avant d'aborder les méthodes d'optimisation, nous allons configurer la connexion afin de tracer toutes les requêtes exécutées. Ainsi, nous pourrions voir en détail les résultats des optimisations que nous effectuerons. Pour ce faire, il suffit de créer une classe qui implémente l'interface `Doctrine\DBAL\Logging\SQLLogger`.

Si nous avons déjà un système de log, nous aurions pu le configurer pour supporter *Doctrine* grâce à cette interface. Mais pour notre cas, nous allons utiliser une classe fournie par défaut par *Doctrine*. La configuration de *Doctrine* ressemble maintenant à :

```
1 <?php
2 ##### bootstrap.php
3
4 require_once join(DIRECTORY_SEPARATOR, [__DIR__, 'vendor',
5     'autoload.php']);
6
7 use Doctrine\ORM\Tools\Setup;
8 use Doctrine\ORM\EntityManager;
9 use Doctrine\DBAL\Logging\DebugStack;
10
11 $entitiesPath = [
12     join(DIRECTORY_SEPARATOR, [__DIR__, "src", "Entity"])
13 ];
14 $isDevMode = true;
15 $proxyDir = null;
```

IV. Exploiter une base de données avec Doctrine 2

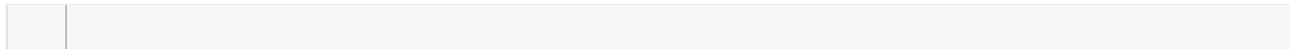
```
16 $cache = null;
17 $useSimpleAnnotationReader = false;
18
19 // Connexion à la base de données
20 $dbParams = [
21     'driver' => 'pdo_mysql',
22     'host'   => 'localhost',
23     'charset' => 'utf8',
24     'user'   => 'root',
25     'password' => '',
26     'dbname' => 'poll',
27 ];
28
29 $config = Setup::createAnnotationMetadataConfiguration(
30     $entitiesPath,
31     $isDevMode,
32     $proxyDir,
33     $cache,
34     $useSimpleAnnotationReader
35 );
36
37 // Logger fourni nativement par Doctrine
38 $config->setSQLLogger(new DebugStack());
39
40 $entityManager = EntityManager::create($dbParams, $config);
41
42 return $entityManager;
```

Pour tester cette modification, nous allons réutiliser la méthode de récupération des participations d'un utilisateur.

```
1 <?php
2 ##### get-user-participations.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6
7 use Tuto\Entity\User;
8
9 $time = microtime(true);
10
11 $userRepo = $entityManager->getRepository(User::class);
12 $user = $userRepo->find(3);
13
14 $participations = $user->getParticipations();
15
16 foreach ($participations as $participation) {
17     echo $participation;
```

```
17     $choices = $participation->getChoices();
18     foreach ($choices as $choice) {
19         echo "- ", $choice;
20         $answers = $choice->getAnswers();
21         echo "-- ", $choice->getQuestion();
22         foreach ($answers as $answer) {
23             echo "--- ", $answer;
24         }
25     }
26 }
27
28 $time = microtime(true) - $time;
29 $logger = $entityManager->getConfiguration()->getSQLLogger();
30
31 echo "\nPerformances:\n";
32 echo "Nombre de requêtes : ", count($logger->queries), "\n";
33 echo "Durée d'exécution totale : ", $time , " ms\n";
34
35 foreach ($logger->queries as $query) {
36     echo "- ", json_encode($query), "\n";
37 }
```

Avec la réponse, nous avons en détails toutes les requêtes exécutées et leur durées :



Nous sommes actuellement à huit (8) requêtes. Voyons donc comment faire pour réduire ce nombre.

10.2. Fetch mode (Extra-Lazy, Lazy, Eager) et lazy-loading

Prenons un exemple simple pour commencer.

```
1 <?php
2 ##### get-user.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7
8 $userRepo = $entityManager->getRepository(User::class);
9 $logger = $entityManager->getConfiguration()->getSQLLogger();
10
11 $user = $userRepo->find(1);
```

IV. Exploiter une base de données avec Doctrine 2

```
12 $address = $user->getAddress();
13
14 echo "Find User\n";
15 echo "Firstname : ", $user->getFirstName(), "\n";
16
17 foreach ($logger->queries as $query) {
18     echo "- ", json_encode($query), "\n";
19 }
20
21 echo "City :", $address->getCity(), "\n";
22
23 echo "After Print address\n";
24
25 echo "\nPerformances:\n";
26 echo "Nombre de requêtes : ", count($logger->queries), "\n";
27
28 foreach ($logger->queries as $query) {
29     echo "- ", json_encode($query), "\n";
30 }
```

Lorsque que nous récupérons les informations sur un utilisateur avec une méthode `find`, *Doctrine* exécute une seule requête SQL pour récupérer les informations liées à notre utilisateur.

```
1 SELECT t0.id AS id_1, t0.firstname AS firstname_2, t0.lastname AS
   lastname_3, t0.role AS role_4,
2 t0.address_id AS address_id_5 FROM users t0 WHERE t0.id = ?
```

Lorsque nous appelons la méthode `getAddress` *Doctrine* nous renvoie une entité *adresse* avec une grande subtilité : seul l'identifiant de celui est connu. C'est lorsque nous essayons d'afficher une information différente de l'identifiant de l'adresse (ici la ville) que *Doctrine* effectue une autre requête pour récupérer les informations de l'adresse.

```
1 SELECT t0.id AS id_1, t0.street AS street_2, t0.city AS city_3,
   t0.country AS country_4, t5.id AS id_6,
2 t5.firstname AS firstname_7, t5.lastname AS lastname_8, t5.role AS
   role_9, t5.address_id AS address_id_10
3 FROM addresses t0 LEFT JOIN users t5 ON t5.address_id = t0.id WHERE
   t0.id = ?
```

Cette technique est appelée le *lazy-loading*. Pour économiser les requêtes SQL, *Doctrine* utilise dès que possible cette technique. Cependant dans certains cas, cette optimisation n'est pas utile et entraîne même des pertes de performances. *Doctrine* nous laisse le choix de modifier ce comportement à notre guise.

Ainsi, pour les annotations permettant de gérer les relations, il y a un attribut `fetch` qui peut prendre trois valeurs.

IV. Exploiter une base de données avec Doctrine 2

LAZY est la valeur par défaut et son comportement est celui que nous venons de décrire.

EAGER permet d'utiliser une jointure et récupérer les deux entités en relation avec une seule requête.

Testons cette configuration.

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Tuto\Repository\UserRepository;
7 use Doctrine\Common\Collections\ArrayCollection;
8 use Doctrine\ORM\Mapping as ORM;
9
10 // ...
11 class User
12 {
13     // ...
14
15     /**
16      * @ORM\OneToOne(targetEntity=Address::class, cascade={"persist", "remove"})
17      */
18     protected $address;
19
20     // ...
21 }
```

En ré-exécutant, notre exemple *Doctrine* utilise une seule requête pour récupérer l'utilisateur et son adresse.

```
1 SELECT t0.id AS id_1, t0.firstname AS firstname_2, t0.lastname AS
   lastname_3, t0.role AS role_4,
2 t0.address_id AS address_id_5, t6.id AS id_7, t6.street AS
   street_8, t6.city AS city_9, t6.country AS country_10
3 FROM users t0 LEFT JOIN addresses t6 ON t0.address_id = t6.id WHERE
   t0.id = ?
```

Nous pouvons dès à présent rajouter cette configuration dans les entités *choix* et *utilisateur* pour optimiser le nombre de requêtes exécutées par *Doctrine*.

```
1 <?php
2 ##### src/Entity/Choice.php
3
4 namespace Tuto\Entity;
```

IV. Exploiter une base de données avec Doctrine 2

```
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 // ...
10 class Choice
11 {
12     // ...
13
14     /**
15      * @ORM\ManyToOne(targetEntity=Question::class, fetch="EAGER")
16      */
17     protected $question;
18
19     // ...
20 }
```

```
1 <?php
2 ##### src/Entity/User.php
3
4 namespace Tuto\Entity;
5
6 use Tuto\Repository\UserRepository;
7 use Doctrine\Common\Collections\ArrayCollection;
8 use Doctrine\ORM\Mapping as ORM;
9
10 // ...
11 class User
12 {
13     // ...
14
15     /**
16      * @ORM\OneToOne(targetEntity=Address::class, cascade={"persist", "remove"})
17      */
18     protected $address;
19
20     /**
21      * @ORM\OneToMany(targetEntity=Participation::class, mappedBy="user", fetch="EAGER")
22      */
23     protected $participations;
24
25     // ...
26 }
```

Le nombre de requêtes est maintenant passé à cinq (5).

Et pour finir, pour les relations `ManyToMany` et `OneToMany`, il existe une option `EXTRA_LAZY`. Cette option permet d'interagir avec une collection mais en évitant au maximum de la charger entièrement en mémoire.

L'appel aux méthodes `contains($entity)`, `containsKey($key)`, `count()`, `get($key)`, `slice($offset, $length = null)`, `add($entity)` et `offsetSet($key, $entity)` de la collection ne déclenchera pas une requête de chargement de toutes les entités.

Si par exemple, nous avons la relation entre les utilisateurs et les participations en mode `EXTRA_LAZY`, un appel à la méthode `$user->getParticipations()->count()` se traduirait par une requête `SELECT COUNT(*)`.

Ce mode est utile si nous traitons une collection trop grandes et que nous voulons éviter de la charger en mémoire.

10.3. Les jointures

Il est aussi possible de faire des jointures en utilisant le `QueryBuilder`. Comme pour le `fetch mode`, ces jointures permettent d'améliorer les performances de notre application.

Lorsque nous utilisons le paramètre `fetch`, **toutes les requêtes** utiliseront cette jointure. Mais dans certains cas, il n'est pas nécessaire d'utiliser une jointure.

Imaginons, par exemple, une page pour éditer les informations personnelles des utilisateurs. La jointure pour récupérer les participations de celui-ci nous est d'aucune utilité dans ce cas.

En utilisant le `QueryBuilder` en conjonction avec le `repository` personnalisé, nous pouvons définir des méthodes pour faire des jointures sur les informations qui nous intéressent et les appeler qu'au cas par cas selon le contexte de l'application.

Pour récupérer un utilisateur, son adresse et ses participations, nous pouvons avoir :

```
1 <?php
2 ##### src/Repository/UserRepository.php
3
4 namespace Tuto\Repository;
5
6 use Doctrine\ORM\EntityRepository;
7 use Tuto\Entity\User;
8
9 class UserRepository extends EntityRepository
10 {
11     public function find($id)
12     {
13         $queryBuilder = $this->_em->createQueryBuilder()
```

IV. Exploiter une base de données avec Doctrine 2

```
14         ->select(['u', 'a', 'p']) // récupération des alias
           obligatoire pour que la jointure soit effective
15         ->from(User::class, 'u')
16         ->leftJoin('u.address', 'a')
17         ->leftJoin('u.participations', 'p')
18         ->where('u.id = :id')
19         ->setParameter('id', $id);
20
21         $query = $queryBuilder->getQuery();
22
23         return $query->getOneOrNullResult();
24     }
25
26     // ...
27 }
```

Ici, nous avons surchargé la méthode `find` native de *Doctrine*. Ainsi en rajoutant la jointure, nous savons maintenant que **toutes les requêtes** `find` profiteront de cette optimisation.

Nous pouvons aussi bien utiliser des `LEFT JOIN` comme des `INNER JOIN` suivant le résultat que nous souhaitons avoir.

Il faut préférer les jointures manuelles avec des méthodes de *repository* personnalisées au *fetch mode Eager*. Ainsi, vous pourrez garder un contrôle total sur toutes les requêtes que *Doctrine* va exécuter et choisir, selon le contexte, les jointures nécessaires.

Les moyens pour optimiser l'utilisation de *Doctrine* sont très nombreux. Le plus simple et intuitif reste les jointures que vous avez sûrement déjà eu l'occasion d'utiliser dans plusieurs contextes.

Il existe bien d'autres moyens d'optimiser les performances de *Doctrine* mais il faut éviter de tomber dans le piège de [l'optimisation prématurée](#) .

Donc avant de songer à appliquer d'autres techniques pour améliorer les performances de votre application ([objets partiels](#) , [entités en lecture seule](#) , etc.) assurez vous que les jointures sont bien configurées et essayez de répondre à deux questions :

- Le programme est-il réellement trop lent ?
- Les mesures de performances ont-ils décelé un point dans *Doctrine* améliorable ?

Si la réponse à ces deux questions est « oui », alors il est envisageable d'aller plus loin dans l'optimisation de *Doctrine*.

11. Configurer Doctrine pour la production

Maintenant que notre application est performante, nous allons voir la configuration optimale à mettre en place avant de la mettre en production.

Nous allons voir dans ce chapitre les différents paramètres de configuration recommander pour que *Doctrine* fonctionne de manière optimale.

Lorsque nous avons mis en place la configuration de *Doctrine*, nous avons passé sous silence une bonne partie de ses mécanismes internes. Le fichier de configuration a été écrit pour un environnement de développement.

Pour une mise en production, nous devons revoir un certain nombre de points.

11.1. La gestion du cache



Quelles sont les informations mises en cache par *Doctrine* ?

Pour son bon fonctionnement, *Doctrine* gère trois (3) types de données :

- les réponses des requêtes exécutées ;
- les requêtes SQL (une fois le DQL traduit en SQL) ;
- et enfin les annotations sur les entités (une fois parsées).

La mise en cache de ses éléments est nécessaire pour le bon fonctionnement de *Doctrine*. Avec notre configuration actuelle, ces informations sont juste stockées dans un tableau PHP. Donc à chaque exécution de nos scripts, le cache doit être régénéré.

En production, un tel comportement n'est pas envisageable. Dès lors, *Doctrine* fournit nativement le support de plusieurs systèmes de cache plus persistant qu'un simple tableau PHP. On peut citer entre autres :

Type de cache	Implémentation Doctrine	Description
APCu - APC User Cache ↗	Doctrine\Common\Cache\ApcuCache	Extension PHP de cache mémoire
redis ↗	Doctrine\Common\Cache\RedisCache	Serveur de cache mémoire
MongoDB ↗	Doctrine\Common\Cache\MongoDBCache	Base de données NoSQL

Si vous avez des doutes sur le choix à faire, l'extension APCu fera largement l'affaire si vous utilisez PHP 7.

C'est un usage moins courant mais il est aussi possible de créer soi-même un système de cache en implémentant l'interface `Doctrine\Common\Cache\Cache`.

11.2. Les proxies

Lorsque nous récupérons une classe depuis les *repositories*, *Doctrine* peut utiliser en interne des classes intermédiaires qui étendent les nôtres. Ces classes sont appelées des proxies. C'est d'ailleurs grâce à ces proxies que le système de lazy-loading peut fonctionner. Et c'est l'une de leur principale utilité.

Pour s'en rendre compte, nous allons modifier la configuration de *Doctrine*. Dans l'entité *utilisateur*, supprimons tous les fetch EAGER que nous avons configurés et exécutons le code suivant.

```
1 <?php
2 ##### get-user-class.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7
8 $userRepo = $entityManager->getRepository(User::class);
9
10 $user = $userRepo->find(1);
11 echo get_class($user->getAddress());
```

La réponse est :

```
1 DoctrineProxies\__CG__\Tuto\Entity\Address.
```

Sans nos optimisations, la classe `Address` est chargée en mode lazy-loading. *Doctrine* utilise donc un proxy.

Cette classe est déclarée dans le dossier temporaire de votre système (`/tmp` sur linux, `C:\Users\votre_utilisateur\AppData\Local\Temp` sur Windows). Vous pouvez trouver ce dossier en utilisant la fonction PHP `sys_get_temp_dir`.

Voici un extrait de son code :

```

1 <?php
2 namespace DoctrineProxies\__CG__\Tuto\Entity;
3
4 /**
5  * DO NOT EDIT THIS FILE - IT WAS CREATED BY DOCTRINE'S PROXY GENERATOR
6  */
7 class Address extends \Tuto\Entity\Address implements
8     \Doctrine\ORM\Proxy\Proxy
9 {
10     // ...
11
12     /**
13      * {@inheritDoc}
14      */
15     public function getStreet()
16     {
17         $this->__initializer__ &&
18             $this->__initializer__->__invoke($this, 'getStreet',
19             []);
20
21         return parent::getStreet();
22     }
23 }

```

La méthode `getStreet` comme toutes les méthodes de notre entité, ont été réécrites par *Doctrine*. Le secret du lazy-loading réside dans ce code.



Mais pourquoi parle-t-on des proxies ?

Avec la configuration actuelle, *Doctrine* est obligé de régénérer ces classes à chaque exécution. Les performances d'une application en production en pâtiraient énormément. En production, nous devons désactiver cette génération automatique et créer les proxies manuellement grâce à la commande *Doctrine* appropriée : `vendor/bin/doctrine orm:generate-proxies`.

11.3. Exemple de configuration

Un exemple valant mieux qu'un long discours, voici une configuration qui utilise APCu pour le système de cache et où les proxies sont stockés dans le dossier *proxies* de notre projet.

```

1 <?php
2 ##### bootstrap.php
3

```

IV. Exploiter une base de données avec Doctrine 2

```
4 require_once join(DIRECTORY_SEPARATOR, [__DIR__, 'vendor',
   'autoload.php']);
5
6 use Doctrine\ORM\Tools\Setup;
7 use Doctrine\Common\Cache\ApcuCache;
8 use Doctrine\Common\Proxy\AbstractProxyFactory;
9 use Doctrine\ORM\EntityManager;
10 use Doctrine\DBAL\Logging\DebugStack;
11
12 $entitiesPath = [
13     join(DIRECTORY_SEPARATOR, [__DIR__, "src", "Entity"])
14 ];
15
16 $isDevMode = false;
17 $proxyDir = join(DIRECTORY_SEPARATOR, [__DIR__, "proxies"]);
18 $cache = null;
19 if ($isDevMode == false) {
20     $cache = new ApcuCache();
21 }
22
23 $useSimpleAnnotationReader = false;
24
25 // Connexion à la base de données
26 $dbParams = [
27     'driver' => 'pdo_mysql',
28     'host' => 'localhost',
29     'charset' => 'utf8',
30     'user' => getenv("TUTO_DATABASE_USER"),
31     'password' => getenv("TUTO_DATABASE_PASSWORD"),
32     'dbname' => 'poll',
33 ];
34
35 $config = Setup::createAnnotationMetadataConfiguration(
36     $entitiesPath,
37     $isDevMode,
38     $proxyDir,
39     $cache,
40     $useSimpleAnnotationReader
41 );
42
43 if ($isDevMode) {
44     $config->setSQLLogger(new DebugStack());
45 }
46
47 if ($isDevMode == false) {
48     $config->setAutogenerateProxyClasses(AbstractProxyFactory::AUTOGENERATE_NEVER);
49 }
50
51 $entityManager = EntityManager::create($dbParams, $config);
```

IV. Exploiter une base de données avec Doctrine 2

```
52  
53 return $entityManager;
```

La variable `$isDevMode` permet de revenir sur la configuration en mode développement facilement.

En essayant d'exécuter le script `get-user-class.php`, une erreur fatale est affichée avec un message du style : *Fatal error : require() : Failed opening required '/dossier/proxies/___CG___TutoEntityAddress.php'*.

Nous devons générer manuellement les proxies :

Notre application est maintenant prête pour une mise en production. Les configurations présentées ici sont **obligatoires** si vous voulez avoir une application performante.

Surtout n'oubliez pas de régénérer les proxies et à vider le système de cache **à chaque nouvelle mise en production** pour éviter les bugs et incohérences.

12. Annexes

Les sujets à couvrir sur *Doctrine* sont très nombreux et variés. Nous allons aborder dans ce chapitre certains mécanismes et concepts qui peuvent se montrer très utiles dans beaucoup d'occasions.

i

Il n'existe pas de relations particulières entre les sujets qui sont abordés dans ce chapitre.

12.1. Support des transactions

Beaucoup de moteur de base de données fournissent le mécanisme de transaction afin d'assurer l'atomicité d'un ensemble de requêtes.

i

L'atomicité est une propriété qui désigne une opération ou un ensemble d'opérations en mesure de s'exécuter complètement sans interruption.

Un ensemble de requêtes sera considéré comme atomique si nous avons la garantie qu'elles seront toutes exécutées ensemble. Pour obtenir un tel résultat, *Doctrine* fournit une API nous permettant d'exploiter le mécanisme de transaction des bases de données relationnelles.

12.1.1. Transactions implicites

Lorsque que nous utilisons l'*entity manager*, *Doctrine* crée automatiquement une transaction afin d'assurer que nos opérations se déroulent correctement. Ainsi, toutes les opérations déclarées avant l'appel de la méthode `flush` de l'*entity manager* sont exécutées dans une même transaction. Nous pouvons ainsi effectuer sereinement toutes les modifications que nous voulons sur notre modèle de données avant d'effectuer les requêtes SQL associées.

```
1 <?php
2 ##### create-user-with-address.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 $logger = $entityManager->getConfiguration()->getSQLLogger();
7 use Tuto\Entity\User;
```

```
8 use Tuto\Entity\Address;
9
10 // Instanciation de l'utilisateur
11
12 $user = new User();
13 $user->setFirstname("First with Address");
14 $user->setLastname("Lastname");
15 $user->setRole("admin");
16
17 // Création de l'adresse
18 $address = new Address();
19 $address->setStreet("Place Charles de Gaulle");
20 $address->setCity("Paris");
21 $address->setCountry("France");
22
23 $user->setAddress($address);
24
25 // Gestion de la persistance
26 /*
27 Le premier persist n'est pas obligatoire car il y a une opération
28   de cascade
29 sur l'adresse
30 */
31 $entityManager->persist($address);
32 $entityManager->persist($user);
33 $entityManager->flush();
34
35 // Vérification du résultats
36 echo $user;
37
38 echo "Nombre de requêtes : ", count($logger->queries), "\n";
39
40 foreach ($logger->queries as $query) {
41     echo "- ", json_encode($query), "\n";
42 }
```

En exécutant ce code, *Doctrine* exécute quatre (4) requêtes dont une pour démarrer la transaction (START TRANSACTION) et une autre pour la valider (COMMIT).

12.1.2. Transactions explicites

Il est toujours possible d'activer les transactions à la demande. Si par exemple nous voulons supporter la suppression d'un sondage. Il serait intéressant de pouvoir faire toutes les suppressions nécessaires (le sondage, les questions, les réponses, les participations, les choix des utilisateurs liés à ce sondage) dans une même transaction.

IV. Exploiter une base de données avec Doctrine 2

Pour créer une transaction, nous devons utiliser directement la connexion à la base de données.

```
1 <?php
2 // Démarrage de la transaction
3 $connection = $entityManager->getConnection();
4 $connection->beginTransaction();
5 try {
6     // Suppression du sondage et de tous les éléments liés
7     $connection->commit();
8 } catch (Exception $e) {
9     // Si il y a un problème, nous annulons la transaction
10    $connection->rollBack();
11 }
```



Lorsque nous démarrons manuellement une transaction, *Doctrine* désactive automatiquement les transactions implicites. L'appel à la méthode `flush` ne créera plus de transactions.

Il y a aussi une fonction utilitaire de l'*entity manager* permettant d'utiliser les transactions sans passer par la connexion.

```
1 <?php
2 $entityManager->transactional(function($entityManager) {
3     // opérations
4 });
```

Toutes les opérations dans la fonction anonyme passée à la méthode `transactional` seront exécutées dans une même transaction.

12.2. Les références (ou comment économiser une requête)

Supposons que nous avons dans notre base de données l'adresse *Address* (*id* : 10, *street* : 6 Parvis Notre-Dame - Pl. Jean-Paul II, *city* : Paris, *country* : France). N'hésitez pas à la créer pour tester par vous-même.

Nous connaissons l'identifiant de l'adresse (10) et nous voulons l'associer à un utilisateur (Id : 7) déjà existant.

Nous avons le choix entre faire une requête pour récupérer l'adresse et une autre pour récupérer l'utilisateur avant de sauvegarder nos modifications. Ou alors nous pouvons utiliser un type d'objet appelé : référence.

Les références sont des objets qui permettent d'avoir une instance d'une entité dont on connaît l'identifiant sans accéder à la base de données.

Ce sont des proxies *Doctrine* qui permettent d'utiliser facilement une entité dans une relation.

```
1 <?php
2 ##### set-user-address.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\User;
7 use Tuto\Entity\Address;
8
9 $userRepo = $entityManager->getRepository(User::class);
10 $logger = $entityManager->getConfiguration()->getSQLLogger();
11
12 $user = $userRepo->find(7);
13
14 $address = $entityManager->getReference(Address::class, 10);
15
16 $user->setAddress($address);
17
18 $entityManager->flush();
19
20 echo "\nPerformances:\n";
21 echo "Nombre de requêtes : ", count($logger->queries), "\n";
22
23 foreach ($logger->queries as $query) {
24     echo "- ", json_encode($query), "\n";
25 }
```

Les références sont construites autour du principe de *lazy-loading*. Dès que nous essayerons d'accéder ou de modifier un de ses attributs, *Doctrine* chargera toutes ses informations depuis la base de données. Mais vous remarquerez en consultant les logs des requêtes SQL qu'en aucun cas l'adresse n'est récupérée pour mettre à jour l'utilisateur.

12.3. Owning side - Inverse side : gestion des relations

12.3.1. Définition

La gestion des relations bidirectionnelles par *Doctrine* n'est pas une tâche anodine.

Si nous récupérons un sondage de la base de données et que nous modifions la liste des questions qui lui sont associées (en retirant une question par exemple), à la sauvegarde des modifications, la question retirée sera toujours liée au sondage.

```
1 <?php
2 ##### owning-side-inverse-side.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6
7 $logger = $entityManager->getConfiguration()->getSQLLogger();
8
9 use Tuto\Entity\Poll;
10
11 $pollRepo = $entityManager->getRepository(Poll::class);
12
13 $poll = $pollRepo->find(1);
14
15 echo $poll;
16
17 $questions = $poll->getQuestions();
18
19 $questions->remove(1);
20
21 $entityManager->flush();
22
23 echo "\nPerformances:\n";
24 echo "Nombre de requêtes : ", count($logger->queries), "\n";
25
26 foreach ($logger->queries as $query) {
27     echo "- ", json_encode($query), "\n";
28 }
```

Aucune requête de mise à jour n'a été exécutée.

En réalité, dans toutes les relations, *Doctrine* ne vérifie les changements que d'un seul côté pour mettre à jour la relation. Ce côté **responsable** de la relation est appelé l'*owning side*. Réciproquement, l'autre côté de la relation est appelé l'*inverse side*.

12.3.2. Identification de l'*owning side*

?

Comment identifier l'*owning side* d'une relation ?

Selon le type de relation bidirectionnelle utilisé, l'*owning side* peut être soit imposé par *Doctrine* soit choisi par nous-même.

Pour une relation *ManyToOne - OneToMany*, l'*owning side* sera toujours l'entité contenant l'annotation *ManyToOne*. Dans ce cas-ci, *Doctrine* impose le choix.

IV. Exploiter une base de données avec Doctrine 2

Par contre, pour les relations *OneToOne* et *ManyToOne* bidirectionnelles, nous devons nous-même choisir l'*owning side* en spécifiant l'attribut `inversedBy`.

Ainsi pour supprimer une question d'un sondage, nous devons modifier la relation en nous basant sur l'entité *question* qui est l'*owning side* de la relation car elle contient l'annotation `ManyToOne`.

La suppression ressemblerait donc à :

```
1 <?php
2 ##### owning-side-inverse-side.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 $logger = $entityManager->getConfiguration()->getSQLLogger();
7
8 use Tuto\Entity\Question;
9
10 $questionRepo = $entityManager->getRepository(Question::class);
11 $question = $questionRepo->find(2);
12
13 echo $question;
14
15 $question->setPoll(null);
16
17 $entityManager->flush();
18
19 echo "\nPerformances:\n";
20 echo "Nombre de requêtes : ", count($logger->queries), "\n";
21
22 foreach ($logger->queries as $query) {
23     echo "- ", json_encode($query), "\n";
24 }
```

Notez la présence de la requête `UPDATE questions SET poll_id = ? WHERE id = ?` qui permet ainsi de dissocier la question au sondage.

12.3.3. Contraintes

Ce mode de fonctionnement interne de *Doctrine* introduit quelques contraintes d'utilisation de l'ORM. Dans toutes les relations unidirectionnelles, seul l'*owning side* doit être configuré.

Ainsi, pour une relation 1..n unidirectionnelle, seul l'annotation `ManyToOne` doit être configurée.

IV. Exploiter une base de données avec Doctrine 2

En d'autres termes, l'annotation `OneToMany` ne peut être utilisée toute seule. À chaque annotation `OneToMany`, il **doit** y avoir une annotation `ManyToOne` correspondante.

Dans la même logique, pour une relation 1..1 unidirectionnelle, une seule entité contient l'annotation `OneToOne`. *Doctrine* créera forcément la clé étrangère dans cette entité.

C'est le cas pour notre relation entre un utilisateur et une adresse. La clé étrangère est dans la table `users` car l'entité *utilisateur* est l'*owning side*.

i

De manière générale, en plus d'être complexes, les relations bidirectionnelles sont souvent une source de problèmes de performance. Avant de les utiliser, il faut toujours évaluer la réelle nécessité d'une telle configuration.

12.4. Les événements Doctrine

12.4.1. Les types d'événements

Pour nous permettre d'agir sur les entités durant leur cycle de vie, l'*entity manager* de *Doctrine* génère des événements pour chaque opération qu'il effectue sur une entité. Ainsi, nous avons à notre disposition un ensemble d'événements et nous pouvons citer entre autres :

Événement	Description
<code>preRemove</code>	L'événement déclenché avant que l' <i>entity manager</i> supprime l'entité.
<code>postRemove</code>	L'événement déclenché après la suppression effective de l'entité.
<code>prePersist</code>	L'événement déclenché avant que l' <i>entity manager</i> sauvegarde l'entité.
<code>postPersist</code>	L'événement déclenché après la sauvegarde effective de l'entité.
<code>preUpdate</code>	L'événement déclenché avant la mise à jour de l'entité.
<code>postUpdate</code>	L'événement déclenché après la mise à jour effective de l'entité.
<code>postLoad</code>	L'événement déclenché une fois qu'une entité a été chargée depuis la base de données (ou après l'appel de la méthode <code>refresh</code> de l' <i>entity manager</i>).
<code>preFlush</code>	L'événement déclenché juste avant la mise à jour effective de l'entité.
<code>postFlush</code>	L'événement déclenché après la mise à jour effective de l'entité.

x

Ces événements ne sont pas déclenchés pour les requêtes DQL.

12.4.2. Les méthodes de rappel (*callbacks*)

Nous pouvons modifier la gestion de nos entités en nous basant sur ces événements grâce aux fonctions de rappels. Ces fonctions seront appelées automatiquement par *Doctrine*.

Pour la création d'un sondage, nous avons dû rajouter la date de création manuellement. Avec les événements, nous pouvons avoir une méthode simple qui modifie la date de création lorsque l'événement `prePersist` est déclenché.

Pour cela, nous devons d'abord spécifier que l'entité *sondage* utilise les événements de *Doctrine* avec l'annotation `HasLifecycleCallbacks`. Ensuite, nous pourrons utiliser les annotations pour désigner les méthodes qui seront appelées si un événement spécifique est déclenché. Ainsi pour l'entité sondage, nous aurons :

```
1 <?php
2 ##### src/Entity/Poll.php
3
4 namespace Tuto\Entity;
5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\ORM\Mapping as ORM;
8
9 /**
10  * @ORM\Entity
11  * @ORM\Table(name="polls")
12  * @ORM\HasLifecycleCallbacks
13  */
14 class Poll
15 {
16     // ...
17
18     /**
19      * @ORM\PrePersist
20      */
21     public function prePersist()
22     {
23         $this->created = new \Datetime();
24     }
25 }
```

```
1 <?php
2 ##### create-poll-prepersist.php
3
4 $entityManager = require_once join(DIRECTORY_SEPARATOR, [__DIR__,
5     'bootstrap.php']);
6 use Tuto\Entity\Poll;
```

IV. Exploiter une base de données avec Doctrine 2

```
7
8 $poll = new Poll();
9
10 $poll->setTitle("Les événements Doctrine 2");
11
12 $entityManager->persist($poll);
13
14 $entityManager->flush();
15
16 echo $poll;
```

Notre méthode `prePersist` est appelée automatiquement par *Doctrine* et le sondage a ainsi une date de création.

Ces événements permettent de personnaliser grandement le comportement de *Doctrine*. Ils sont d'ailleurs utilisés par beaucoup d'extensions pour gérer des problématiques courantes. N'hésitez donc pas les consulter pour approfondir le sujet ([Atlantic18/DoctrineExtensions](#) [↗](#)).

Le système d'événement peut même être poussé encore plus loin en créant par exemple nos propres événements, des classes spécialisées pour gérer les événements, etc. [La documentation officielle](#) [↗](#) présente bien ces sujets.

L'objectif de ce chapitre annexe est d'explicitier au mieux le fonctionnement de *Doctrine* et d'aborder quelques sujets intéressants que nous n'avions pas pu introduire tout au long de ce cours.

Vous pouvez vous-y référer pour éclaircir rapidement certains points mais il faut garder en tête qu'il n'est pas conçu pour être complet à 100 %.

Voici un résumé de cette partie fourni par [artrags](#) [↗](#) sous licence [CC-BY](#) [↗](#).



FIGURE 12.1. – Résumé de l'exploitation d'une base de données avec Doctrine 2

Vous pouvez consulter le [lien interactif](#) [↗](#) pour plus de détails.

Cinquième partie

Conclusion

V. Conclusion

Nous avons pu voir tout au long de ce cours que *Doctrine* apporte une grande facilité dans le traitement des données, le maintien de la cohérence du modèle et la gestion des relations entre les entités.

Les utilitaires en ligne de commandes permettent d'avoir un contrôle total sur la configuration et l'état de notre base de données.

En utilisant *Doctrine*, il est donc possible de créer et d'exploiter un schéma de données complet sans écrire une seule ligne de SQL.

Retenez néanmoins qu'il existe encore beaucoup de sujets à découvrir et que [la documentation officielle](#) reste la référence la plus complète si vous voulez approfondir votre apprentissage de *Doctrine*.