

# Découverte d'un ORM PHP

# Table des matières

<b>I. Contexte</b>	<b>3</b>
<b>II. Doctrine</b>	<b>3</b>
<b>III. Exercice : Appliquez la notion</b>	<b>5</b>
<b>IV. L'EntityManager</b>	<b>5</b>
<b>V. Exercice : Appliquez la notion</b>	<b>8</b>
<b>VI. Les entités</b>	<b>8</b>
<b>VII. Exercice : Appliquez la notion</b>	<b>14</b>
<b>VIII. Essentiel</b>	<b>14</b>
<b>IX. Auto-évaluation</b>	<b>14</b>
A. Exercice final .....	14
B. Exercice : Défi.....	15
<b>Solutions des exercices</b>	<b>16</b>

## I. Contexte

**Durée** : 1 h

**Environnement de travail** : Local

**Pré-requis** : Connaître PDO, savoir manipuler une base de données, savoir utiliser Composer

### Contexte

PDO permet de se connecter à toutes les bases de données de la même manière : il a été créé dans le but d'unifier les accès aux bases, quel que soit le SGBD. Cependant, PDO à lui seul ne permet pas à notre application d'être compatible avec tous les moteurs de base de données. En effet, il reste une différence majeure entre les différents SGBD : la syntaxe des requêtes SQL.

Dans ce cours, nous allons apprendre à utiliser des outils qui permettent de générer automatiquement les requêtes SQL pour un SGBD donné : les **ORM**.

## II. Doctrine

### Objectifs

- Comprendre la notion d'ORM
- Découvrir Doctrine
- Installer Doctrine sur un projet

### Mise en situation

Un ORM est un outil très puissant permettant de grandement faciliter les manipulations de données dans une application.

### Définition ORM

Un ORM, pour *Object Relational Mapping*, est un outil permettant d'abstraire une base de données, c'est-à-dire de simplifier leur manipulation en proposant une surcouche qui unifie toutes les opérations. Un ORM permet de représenter les données d'une base sous la forme d'objets, appelés entités : une instance d'une entité représente une ligne de données dans une table, et toute manipulation sur une entité pourra être répercutée automatiquement sur la base de données. Ainsi, un développeur n'a plus à rédiger les requêtes pour assurer le CRUD de ses données : elles seront générées par l'ORM.

L'avantage d'un ORM est qu'il permet d'uniformiser la manipulation des données en base en s'occupant du dernier aspect qui pouvait différer entre les moteurs : les requêtes SQL. En effet, PDO se chargeait déjà d'uniformiser les accès aux bases de données, mais il fallait malgré tout rester vigilant quant à la syntaxe des requêtes SQL, qui pouvait changer d'un SGBD à l'autre.

Grâce à un système de drivers similaire à celui de PDO, un ORM est capable de générer des requêtes pour n'importe quel moteur. Cela permet à notre application de s'affranchir complètement du SGBD : elle devient compatible avec n'importe quelle base de données.

### Définition Doctrine

Doctrine est l'un des ORM les plus populaires. Créé en 2006, il a été téléchargé plus d'1,5 milliard de fois. Il est intégré dans énormément de frameworks populaires, comme Symfony, CodeIgniter ou encore ZendFramework.

### Remarque

Doctrine est séparé en deux parties distinctes : Doctrine DBAL, pour *Database Abstraction & Access Layer*, et Doctrine ORM.

- Doctrine DBAL est la surcouche directe de PDO qui ajoute des fonctionnalités de gestion de base de données qui n'existent pas dans PDO. C'est lui qui s'occupe, par exemple, de générer des requêtes SQL compatibles sur tous les SGBD grâce à ses drivers.
- Doctrine ORM, quant à lui, est l'outil permettant de lier des classes aux tables correspondantes et de répercuter les manipulations faites sur les entités dans la base de données.

Doctrine ORM utilise Doctrine DBAL pour communiquer avec la base de données, mais il est possible d'utiliser Doctrine DBAL seul. Lorsque l'on parle de « Doctrine », on parle généralement de Doctrine ORM.

Avant de pouvoir utiliser Doctrine, il faut d'abord l'installer et le configurer pour son projet. La solution la plus simple est d'utiliser Composer.

### Rappel

Composer est un outil de gestion des dépendances en PHP. Il vous permet de déclarer les bibliothèques dont dépend votre projet et il les gèrera (installation/mise à jour/configuration) pour vous.

### Méthode

Pour installer Doctrine via Composer, il faut utiliser la ligne de commande `composer require doctrine/orm`, puis faire un `composer install` si besoin. Cela va installer Doctrine et toutes ses dépendances.

Puis, comme pour toutes les dépendances gérées par Composer, il va falloir utiliser l'*autoloading* de Composer dans notre application, via la ligne de code `require_once 'vendor/autoload.php';`.

### Syntaxe À retenir

- Un ORM permet de gérer les données d'une base de données au travers d'objets, appelés entités. Le but d'un ORM est de permettre au développeur de ne manipuler que les entités : c'est lui qui va s'occuper de générer les requêtes SQL permettant de reproduire les changements en base de données.
- Doctrine est l'un des ORM les plus populaires. Il peut être installé dans un projet avec Composer via la ligne de commande `composer require doctrine/orm`.

### Complément

Site officiel de Doctrine<sup>1</sup>

Mapping objet relationnel sur Wikipédia<sup>2</sup>

<sup>1</sup> <https://www.doctrine-project.org>

<sup>2</sup> [https://fr.wikipedia.org/wiki/Mapping\\_objet-relationnel](https://fr.wikipedia.org/wiki/Mapping_objet-relationnel)

### III. Exercice : Appliquez la notion

#### Question

[solution n°1 p.17]

Cet exercice va nous servir de base pour tous les exercices suivants : créez un nouveau projet en local et installez-y Doctrine. Ajoutez ensuite un fichier **index.php** et mettez le code nécessaire au bon fonctionnement de Composer.

Pour tester le bon fonctionnement de Doctrine, ajoutez ce code au fichier **index.php** :

```
1 <?php
2
3 use Doctrine\ORM\Tools\Setup;
4
5 $config = Setup::createAnnotationMetadataConfiguration(['entities'], false);
```

`createAnnotationMetadataConfiguration` est une méthode statique de la classe `Setup` qui configure Doctrine.

Si l'installation s'est bien passée, ce code devrait afficher une page blanche. Dans le cas contraire, une erreur PHP devrait apparaître.

Cette méthode peut prendre 5 arguments. Nous en utiliserons 2 et laisseront les autres par défaut.

Le premier argument représente un tableau des entités qui se trouve dans le projet.

Le second argument est un booléen qui définit si nous travaillons en mode développement ou non.

#### Indice :

N'oubliez pas d'importer l'*autoloader* de Composer avant d'ajouter le code fourni.

### IV. L'EntityManager

#### Objectifs

- Comprendre l'utilité de l'`EntityManager`
- Créer un `EntityManager`

#### Mise en situation

Doctrine permet de manipuler des entités, sous la forme d'objets PHP. Pour pouvoir se connecter à la base de données et gérer ces entités, Doctrine fournit un outil appelé `EntityManager`.

#### Définition **EntityManager**

La première étape pour travailler avec Doctrine est de créer un `EntityManager`. L'`EntityManager` est le point central de Doctrine : c'est lui qui fait le lien entre les entités et la base de données, qui va s'assurer que la base de données correspond bien aux entités de l'application et qui va répercuter toutes les modifications apportées à une entité sur la base.

#### Méthode

La création d'un `EntityManager` se fait via la méthode statique `EntityManager::create`. Cette méthode prend deux paramètres : un tableau contenant les informations de connexion à la base de données et une configuration.

Le tableau d'informations peut avoir deux formes différentes :

- Un tableau possédant simplement un champ `url` ayant pour valeur le DSN de la base de données. Il est également possible de fournir une instance de PDO déjà créée grâce au champ `pdo`.
- Un tableau possédant un champ `driver` ayant pour valeur le nom du driver à utiliser, et d'autres champs correspondant aux attributs du driver spécifié. Le nom du driver doit correspondre à l'extension PHP utilisée et la liste des attributs doit permettre de reconstituer le DSN de la base de données. La liste des drivers supportés est disponible sur la documentation de Doctrine<sup>1</sup>. Il est également possible de fournir une classe de l'application en tant que driver personnalisé grâce à `driverClass`.

La configuration est un objet `Configuration` de Doctrine. Il peut être créé en utilisant la méthode statique `Setup::createAnnotationMetadataConfiguration`. Cette méthode prend deux paramètres : un tableau de chemins pointant sur les dossiers qui contiendront les entités de notre application, et un booléen indiquant si on souhaite utiliser le mode développeur ou non. Le mode développeur permet de gérer la mise en cache : il est recommandé de le mettre à `true` pour le développement, mais de le laisser à `false` une fois l'application en production.

### Exemple

Pour créer un `EntityManager` connecté à une base de données MySQL locale sans mot de passe et dont le nom d'utilisateur est `root` dans une application où les entités sont stockées dans un dossier `/entities`, il faut écrire le code suivant :

```
1 <?php
2
3 require_once 'vendor/autoload.php';
4
5 use Doctrine\ORM\Tools\Setup;
6 use Doctrine\ORM\EntityManager;
7
8 // Paramètres de connexion à la base de données
9 $dbParams = [
10     'driver' => 'pdo_mysql',
11     'user' => 'root',
12     'password' => '',
13 ];
14
15 // On crée la configuration pour gérer les entités dans le dossier /entities, en mode dev
16 $config = Setup::createAnnotationMetadataConfiguration(['entities'], true);
17
18 // On crée l'EntityManager associé
19 $entityManager = EntityManager::create($dbParams, $config);
```

### Remarque

Bien qu'aucune erreur ne soit levée au moment de l'exécution de ce code, il faut également penser à créer le dossier `/entities`, en le laissant vide pour le moment.

## Lignes de commande et EntityManager

Doctrine dispose de nombreuses lignes de commande permettant d'automatiser diverses tâches, comme la création de la base de données et des tables. Toutes les lignes de commande Doctrine sont sous la forme `php vendor/bin/doctrine [commande]`. Taper simplement `php vendor/bin/doctrine` permet d'afficher la liste des commandes disponibles.

<sup>1</sup> <https://www.doctrine-project.org/projects/doctrine-dbal/en/current/reference/configuration.html#driver>

Pour pouvoir fonctionner, ces lignes de commande ont besoin d'une connexion à une base de données, et donc d'un EntityManager.

#### Remarque

Selon le système d'exploitation, sa configuration et le terminal utilisé, il faut parfois omettre le « php » de la commande pour qu'elle fonctionne : `php vendor/bin/doctrine [commande]` devient alors `vendor/bin/doctrine [commande]`.

#### Méthode

Les lignes de commande de Doctrine se basent sur un fichier `cli-config.php`, placé à la racine du projet, pour fonctionner. C'est dans ce fichier que doit être déclaré l'EntityManager qui sera utilisé. Il doit retourner le résultat de la méthode statique `ConsoleRunner::createHelperSet`, qui permet de fournir l'instance de l'EntityManager. Le fichier doit donc être sous la forme :

```
1 <?php
2
3 use Doctrine\ORM\Tools\Console\ConsoleRunner;
4
5 $entityManager = ...; // Création de l'EntityManager
6
7 return ConsoleRunner::createHelperSet($entityManager);
```

#### Exemple

Ce fichier permet aux lignes de commande de se connecter à une base de données locale appelée `doctrine_database`.

```
1 <?php
2
3 require_once 'vendor/autoload.php';
4
5 use Doctrine\ORM\Tools\Setup;
6 use Doctrine\ORM\EntityManager;
7 use Doctrine\ORM\Tools\Console\ConsoleRunner;
8
9 $dbParams = [
10     'driver' => 'pdo_mysql',
11     'user' => 'root',
12     'password' => '',
13     'dbname' => 'doctrine_database',
14 ];
15
16 $config = Setup::createAnnotationMetadataConfiguration(['entities'], true);
17
18 $entityManager = EntityManager::create($dbParams, $config);
19
20 // On donne l'EntityManager à la ligne de commande
21 return ConsoleRunner::createHelperSet($entityManager);
```

#### Attention

Il peut arriver que l'exécutable de PHP utilisé en ligne de commande ne soit pas le même que celui de votre serveur web. Assurez-vous que les extensions PDO à utiliser sont bien présentes dans le `php.ini` utilisé par la ligne de commande.

### Syntaxe À retenir

- L'EntityManager gère le CRUD des entités. Il est créé grâce à la méthode statique `EntityManager::create`.
- Doctrine propose des commandes permettant d'automatiser certaines tâches. Pour les utiliser, il faut créer un fichier `cli-config.php` à la racine du projet et fournir un EntityManager à la méthode `ConsoleRunner::createHelperSet`.

### Complément

Installer Doctrine<sup>1</sup>

## V. Exercice : Appliquez la notion

### Question

[solution n°2 p.17]

Dans cet exercice, nous allons créer un EntityManager qui se connectera à notre base locale et lancer notre première commande Doctrine. Pour le moment, cette commande ne fera aucune action, mais permettra de confirmer la bonne configuration de notre projet.

- Créez un fichier `cli-config.php` à la racine de votre projet et paramétrez la connexion à une base de données appelée `doctrine_exercice`.
- Puis, créez une base de données `doctrine_exercice` dans votre SGBD.
- Enfin, lancez la commande `php vendor/bin/doctrine orm:validate-schema` pour vous assurer que Doctrine fonctionne : deux [OK] devraient apparaître.

### Indice :

Selon les systèmes d'exploitation, la commande `php vendor/bin/doctrine orm:validate-schema` peut échouer. Dans ce cas, enlevez le « `php` » pour exécuter la commande `vendor/bin/doctrine orm:validate-schema`.

### Indice :

Si le message d'erreur « *An exception occurred in driver: could not find driver* » s'affiche, c'est que l'extension PDO n'est pas activée dans le `php.ini` de l'exécutable PHP utilisé par la console. Pour l'activer, ouvrez le fichier **php.ini** correspondant, cherchez la ligne de l'extension voulue et décommentez-la en enlevant le point-virgule devant.

Par exemple, pour `pdo_mysql`, la ligne `;extension=pdo_mysql` doit devenir `extension=pdo_mysql`.

### Indice :

Le dossier contenant les entités doit exister. Si votre EntityManager est configuré pour chercher les entités dans un dossier **/entities**, alors créez ce dossier, même si, pour le moment, il ne contiendra aucun fichier.

### Indice :

Si le message d'erreur « *An exception occurred in driver: SQLSTATE[HY000] [1049] Unknown database 'doctrine\_exercice'* » s'affiche, c'est que la base de données `doctrine_exercice` n'a pas été créée.

## VI. Les entités

<sup>1</sup> <https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/reference/configuration.html#installation-and-configuration>



## Objectifs

- Comprendre la notion d'entité
- Créer des entités Doctrine

## Mise en situation

Un ORM permet de manipuler des données sous forme d'objets, appelés entités. Dans cette partie, nous allons voir comment créer des entités qui pourront être manipulées par Doctrine.

### Définition Les entités

Pour créer des entités, il faut créer une classe PHP représentant une table de notre base de données. Cette classe est similaire à celle que nous avons déjà créée pour PDO en utilisant le mode de récupération `PDO::FETCH_CLASS` : chaque propriété doit correspondre à une colonne de la table et posséder des getters et setters permettant de gérer l'accès en lecture et en écriture des données.

Cependant, une entité Doctrine possède des informations supplémentaires sous la forme d'annotations.

Une annotation est un commentaire formaté de manière particulière et qui permet d'ajouter des métadonnées à un élément. Dans Doctrine, les annotations sont utilisées pour faire correspondre une classe à une table et une propriété à un champ.

Les annotations Doctrine sont importées via l'instruction `use Doctrine\ORM\Mapping as ORM;` et doivent être utilisées dans des blocs de commentaires ouverts avec le symbole `/**` (au lieu du `/*` habituel).

Chaque annotation est sous la forme `@ORM\Nom(parametre="valeur")`.

### Méthode Annotations au niveau de la classe

Les annotations au niveau de la classe se placent au-dessus de l'instruction de déclaration de la classe :

```
1 <?php
2
3 use Doctrine\ORM\Mapping as ORM;
4
5 /**
6  * Les annotations sont placées ici
7  */
8 class MyEntity
9 {
10 // Reste de la classe...
11 }
```

Il existe deux annotations principales au niveau des classes :

- `@ORM\Entity` permet d'indiquer à Doctrine que cette classe est une entité. Elle ne possède pas de paramètres.
- `@ORM\Table(name="nomTable")` permet d'indiquer à Doctrine le nom de la table correspondant à cette entité.

### Exemple

Le code suivant permet de créer une entité `User`, qui représente la table `users` de notre base de données :

```
1 <?php
2
3 use Doctrine\ORM\Mapping as ORM;
4
5 /**
6  * @ORM\Entity
7  * @ORM\Table(name="users")
8  */
9 class User
10 {
11 }
```

### Remarque Notion de "Code First"

Au moment de commencer à développer la structure d'une application, il existe trois visions principales :

- Le Code First désigne le fait de commencer par coder les classes, puis de générer cette structure en base de données.
- Le Database First désigne au contraire le fait de commencer par construire la structure des tables, puis de générer les entités en conséquence.
- Enfin, le Model First désigne le fait d'utiliser des outils permettant de modéliser les relations de notre application (par exemple, en utilisant l'UML, un tableau Excel ou un fichier XML) et de générer la base de données et les entités en conséquence.

Il est possible d'utiliser ces trois visions avec Doctrine, mais il a été pensé à l'origine pour être Code First. Cela signifie que nous allons d'abord créer nos entités, puis Doctrine se chargera de générer les tables correspondantes en base de données.

Gardez cependant en mémoire que l'inverse est possible et que Doctrine est capable de générer des entités à partir d'une base de données existante. Cela peut être pratique pour intégrer Doctrine dans un projet existant, par exemple.

### Méthode Annotations au niveau des propriétés

Les annotations des propriétés se placent au-dessus de la déclaration de chaque propriété :

```
1 <?php
2
3 use Doctrine\ORM\Mapping as ORM;
4
5 class MyEntity
6 {
7     /**
8      * Annotations de la propriété $id
9      */
10    private int $id;
11
12    /**
13     * Annotations de la propriété $name
14     */
15    private string $name;
16 }
```

Il existe trois annotations principales au niveau des classes :

- `@ORM\Column(name="nomColonne", type="typeDeDonnees")` permet d'indiquer à Doctrine la colonne correspondant à la propriété. L'attribut `name` est facultatif si la propriété possède le même nom que la colonne, mais il est d'usage de le mettre dans tous les cas.
- `@ORM\Id` permet d'indiquer à Doctrine que cette colonne est une clé primaire.
- `@ORM\GeneratedValue(strategy="nomStrategie")` est utilisé conjointement avec `ID` pour indiquer à Doctrine comment la valeur de la clé doit être générée. Les deux valeurs les plus utilisées sont `AUTO` pour les auto-incrémentations ou les séquences (selon le SGBD) et `UUID`, mais il en existe d'autres pour les cas plus rares.

#### Exemple

Complétons notre entité `User` :

```
1 <?php
2
3 use Doctrine\ORM\Mapping as ORM;
4
5 /**
6  * @ORM\Entity
7  * @ORM\Table(name="users")
8  */
9 class User
10 {
11     /**
12      * @ORM\Column(name="id", type="string")
13      * @ORM\Id
14      * @ORM\GeneratedValue(strategy="UUID")
15      */
16     private string $id;
17
18     /**
19      * @ORM\Column(name="email", type="string")
20      */
21     private string $email;
22
23     /**
24      * @ORM\Column(name="fullname", type="string")
25      */
26     private string $fullname;
27 }
```

Ici, `$id` est un identifiant de type `UUID`, `$email` et `$fullname` sont des chaînes de caractères.

#### Méthode Relations entre les entités

Doctrine nous permet également de spécifier des relations entre nos entités. Cela permet d'indiquer à Doctrine quelles clés étrangères nous souhaitons mettre en place dans nos tables. Pour définir une relation entre deux entités, il faut définir une propriété qui aura le type d'une autre entité, et ajouter une annotation selon le type de relation entre les deux :

- **Many-To-One** (`@ORM\ManyToOne(targetEntity="ClassName", inversedBy="fieldName")`) est l'association la plus commune : « plusieurs X ont un Y ». La table X générée possédera une clé étrangère vers la table Y. Par exemple, plusieurs employés appartiennent à une entreprise : un champ `entreprise_id` sera créé dans la table employés.

- **One-To-Many** (`@ORM\OneToMany(targetEntity="ClassName", mappedBy="fieldName")`) est l'inverse de l'association *Many-To-One* : « un X possède plusieurs Y ». La table Y générée possédera une clé étrangère vers la table X. Par exemple, une entreprise possède plusieurs employés : un champ `entreprise_id` sera créé dans la table employés. Souvent, les entités ont une relation *One-to-Many* d'un côté, et une relation *Many-To-One* de l'autre.
- **One-To-One** (`@ORM\OneToOne(targetEntity="ClassName")`) est une variante du *Many-To-One* : « un X a un Y ». La table X générée possédera une clé étrangère vers la table Y, mais cette clé étrangère sera un index unique. Par exemple, chaque employé possède son propre badge : un champ `badge_id` sera créé dans la table employés, mais un index unique empêchera le partage d'un même badge par deux employés.
- **Many-To-Many** (`@ORM\ManyToMany(targetEntity="ClassName")`) est une association plus rare : « plusieurs X ont plusieurs Y ». Les tables générées seront X, Y, et une table associative entre X et Y. Par exemple, plusieurs employés peuvent travailler sur plusieurs projets : il y aura une table employés, une table projets et une table permettant de lier les deux.

Pour chaque relation, il est possible de spécifier le nom des colonnes grâce à l'annotation : `@JoinColumn(name="fieldName", referencedColumnName="referencedFieldName")`.

### Exemple

Imaginons que nos utilisateurs puissent appartenir à un groupe. Créons une entité `Group` et lions nos entités `User` et `Group` grâce à une relation *Many-To-One* :

`Group.php` :

```
1 <?php
2
3 use Doctrine\ORM\Mapping as ORM;
4
5 /**
6  * @ORM\Entity
7  * @ORM\Table(name="groups")
8  */
9 class Group
10 {
11     /**
12      * @ORM\Column(name="id", type="string")
13      * @ORM\Id
14      * @ORM\GeneratedValue(strategy="UUID")
15      */
16     private string $id;
17
18     /**
19      * @ORM\Column(name="name", type="string")
20      */
21     private string $name;
22 }
```

`User.php` :

```
1 <?php
2
3 use Doctrine\ORM\Mapping as ORM;
4
5 require_once 'Group.php';
6
7 /**
8  * @ORM\Entity
9  * @ORM\Table(name="users")
```

```
10 */
11 class User
12 {
13     /**
14      * @ORM\Column(name="id", type="string")
15      * @ORM\Id
16      * @ORM\GeneratedValue(strategy="UUID")
17      */
18     private string $id;
19
20     /**
21      * @ORM\Column(name="email", type="string")
22      */
23     private string $email;
24
25     /**
26      * @ORM\Column(name="fullname", type="string")
27      */
28     private string $fullname;
29
30     /**
31      * @ORM\ManyToOne(targetEntity="Group")
32      */
33     private Group $group; // On ajoute une propriété de type Group et on spécifie la relation
34 }
```

Nous aurions pu préciser l'annotation `@JoinColumn(name="group_id", referencedColumnName="id")` pour forcer le nom des colonnes, mais dans notre cas, Doctrine aurait compris sans.

La table users ainsi générée possédera un champ `group_id`, clé étrangère de la table groups.

### Syntaxe À retenir

- Les entités sont des classes dotées d'annotations. Les annotations permettent d'ajouter des métadonnées aux éléments constitutifs de l'entité. Elles permettent d'ajouter des informations sur la table au niveau de la classe, et des informations sur les colonnes au niveau des propriétés.
- Une fois les entités générées, il est possible de laisser Doctrine créer les tables correspondant aux entités grâce à la commande `php vendor/bin/doctrine orm:schema-tool:create`. En cas de doute, rajouter l'option `--dump-sql` permet de visualiser les requêtes qui seront jouées, sans faire de modification.

### Complément

Installer Doctrine<sup>1</sup>

<sup>1</sup> <https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/reference/configuration.html#installation-and-configuration>

## VII. Exercice : Appliquez la notion

### Question

[solution n°3 p.17]

Un collectionneur de bandes dessinées aimerait informatiser sa collection.

Il souhaiterait stocker le titre et le nombre de pages de toutes ses bandes dessinées.

Créez une entité `Comic` permettant de gérer ces informations et générez la base de données correspondante.

## VIII. Essentiel

## IX. Auto-évaluation

### A. Exercice final

#### Exercice 1

[solution n°4 p.18]

Exercice

À quoi sert un ORM ?

- ☐ Se connecter à n'importe quelle base de données
- ☐ Manipuler une base de données au travers d'objets, plutôt que de requêtes SQL
- ☐ Lancer des requêtes SQL depuis une ligne de commande

Exercice

Qu'est-ce que Doctrine ?

- ☐ Un des ORM les plus populaires
- ☐ Une version de PDO permettant de créer des ORM
- ☐ Une interface commune implémentée par tous les ORM

Exercice

Quelle commande permet d'installer Doctrine via Composer ?

- ☐ `composer require doctrine`
- ☐ `composer require doctrine/orm`
- ☐ `composer require doctrine/dbal`

Exercice

Quel est le rôle de l'`EntityManager` ?

- ☐ Installer Doctrine
- ☐ Exécuter des commandes dans la console
- ☐ Faire le lien entre les entités et la base de données

Exercice

Au moment de créer un `EntityManager`, il faut lui donner un tableau indiquant le driver à utiliser. D'où vient le nom du driver ?

- ☐ C'est le nom de l'extension PHP correspondante
- ☐ C'est le nom du préfixe du DSN correspondant
- ☐ C'est le nom du SGBD correspondant

#### Exercice

Comment envoyer une commande Doctrine depuis une console ouverte à la racine de son projet ?

- ☐ `php bin/vendor [commande]`
- ☐ `php vendor/bin/doctrine [commande]`
- ☐ `php doctrine [commande]`

#### Exercice

Sur quel fichier est-ce que Doctrine se base pour récupérer l'`EntityManager` à utiliser ?

- ☐ `cli-config.php`
- ☐ `doctrine-config.php`
- ☐ `entity-manager.php`

#### Exercice

Qu'est-ce qu'une entité ?

- ☐ Un tableau associatif correspondant à une table de notre base de données
- ☐ Une classe correspondant à une table de notre base de données
- ☐ Un fichier de configuration décrivant la structure d'une table de notre base de données

#### Exercice

Comment est-ce que l'`EntityManager` fait le lien entre une entité et une table ?

- ☐ Grâce à des annotations
- ☐ Grâce à un fichier de configuration
- ☐ Grâce au nom des classes, qui correspond à celui de la table correspondante

#### Exercice

Quel est le meilleur moyen pour créer la base de données correspondant à nos entités ?

- ☐ En recréant à la main les tables correspondant à nos entités
- ☐ En exécutant les requêtes retournées par la commande `php vendor/bin/doctrine orm:schema-tool:create --dump-sql`
- ☐ En exécutant la commande `php vendor/bin/doctrine orm:schema-tool:create`

### B. Exercice : Défi

Dans ce défi, vous allez devoir utiliser Doctrine pour gérer une table complexe permettant de stocker les informations de musiques.

**Question 1**

[solution n°5 p.20]

Un producteur de musique souhaiterait avoir une application permettant de visualiser toutes les musiques qu'il produit. Il a besoin de stocker les informations suivantes :

- Le titre de la musique
- Le nom du compositeur
- La durée (en secondes) de la musique
- La date de création
- Un champ permettant de savoir si la musique est terminée ou non

Ces informations doivent être stockées dans une table créée dans une base de données MariaDB, appelée `music_prod`.

En ajoutant autant de champs que nécessaire et en déterminant le meilleur type pour chaque champ, utilisez Doctrine pour générer la table correspondante. Vous pouvez vous aider de la documentation expliquant tous les types utilisables dans Doctrine<sup>1</sup>.

**Question 2**

[solution n°6 p.22]

Comparez les types de la table générée avec ceux présents dans Doctrine. Y a-t-il une différence ?

**Indice :**

Avez-vous utilisé le type `boolean` de Doctrine pour savoir si une musique était terminée ou non ?

**Solutions des exercices**

---

<sup>1</sup> <https://www.doctrine-project.org/projects/doctrine-dbal/en/2.10/reference/types.html>



**p. 5 Solution n°1**

L'installation de Doctrine se fait via la commande `composer require doctrine/orm`. Le fichier **index.php** final est le suivant :

```
1 <?php
2
3 // Importation de l'autoloader de Composer
4 require_once 'vendor/autoload.php';
5
6 use Doctrine\ORM\Tools\Setup;
7
8 $config = Setup::createAnnotationMetadataConfiguration(['entities'], false);
```

**p. 8 Solution n°2**

Fichier **cli-config.php** :

```
1 <?php
2
3 // Importation de l'autoloader de Composer
4 require_once 'vendor/autoload.php';
5
6 use Doctrine\ORM\Tools\Setup;
7
8 $config = Setup::createAnnotationMetadataConfiguration(['entities'], false);
```

Pour créer la base de données :

```
1 CREATE DATABASE doctrine_exercice;
```

La commande `php vendor/bin/doctrine orm:validate-schema` devrait alors afficher :

```
1 Mapping
2 -----
3
4 [OK] The mapping files are correct.
5
6 Database
7 -----
8
9 [OK] The database schema is in sync with the mapping files.
```

**p. 14 Solution n°3**

L'entité `Comic` est définie ainsi :

```
1 <?php
2
3 use Doctrine\ORM\Mapping as ORM;
4
5 /**
6  * @ORM\Entity
7  * @ORM\Table(name="comics")
8  */
9 class Comic
10 {
```

```

11  /**
12   * @ORM\Column(name="id", type="string")
13   * @ORM\Id
14   * @ORM\GeneratedValue(strategy="UUID")
15   */
16  private string $id;
17
18  /**
19   * @ORM\Column(name="title", type="string")
20   */
21  private string $title;
22
23  /**
24   * @ORM\Column(name="pages", type="integer")
25   */
26  private int $pages;
27 }

```

La commande `php vendor/bin/doctrine orm:schema-tool:create --dump-sql` indique :

```

1 The following SQL statements will be executed:
2
3      CREATE TABLE comics (id VARCHAR(255) NOT NULL, title VARCHAR(255) NOT NULL, pages INT
      NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE `utf8_unicode_ci` ENGINE =
      InnoDB;

```

En lançant la commande `php vendor/bin/doctrine orm:schema-tool:create`, la table `comics` est bien créée.

Il aurait également été possible d'utiliser un entier auto-incrémenté pour l'identifiant de `Comics`. Dans ce cas, la propriété `$id` aurait été :

```

1  /**
2   * @ORM\Column(name="id", type="integer")
3   * @ORM\Id
4   * @ORM\GeneratedValue(strategy="AUTO")
5   */
6  private int $id;

```

## Exercice p. 14 Solution n°4


### Exercice

À quoi sert un ORM ?

- ☐ Se connecter à n'importe quelle base de données
- ☒ Manipuler une base de données au travers d'objets, plutôt que de requêtes SQL
- ☐ Lancer des requêtes SQL depuis une ligne de commande
- ☒ Un ORM permet de faire le lien entre des classes et des tables, et est capable de générer des requêtes SQL lui-même : cela permet au développeur de n'avoir à manipuler que des objets.

### Exercice


Qu'est-ce que Doctrine ?

- ☒ Un des ORM les plus populaires
- ☐ Une version de PDO permettant de créer des ORM
- ☐ Une interface commune implémentée par tous les ORM
-  Doctrine est un ORM, qui est intégré dans de nombreux frameworks.

### Exercice

---


Quelle commande permet d'installer Doctrine via Composer ?

- ☐ `composer require doctrine`
- ☒ `composer require doctrine/orm`
- ☐ `composer require doctrine/dbal`
-  La commande permettant d'installer Doctrine est `composer require doctrine/orm`.  
`Doctrine/dbal` est une dépendance qui sera automatiquement installée par Composer.

### Exercice

---


Quel est le rôle de l'EntityManager ?

- ☐ Installer Doctrine
- ☐ Exécuter des commandes dans la console
- ☒ Faire le lien entre les entités et la base de données
-  L'EntityManager se place entre les entités et la base de données et fait le lien entre les deux. Bien qu'il soit requis pour lancer des commandes, ce n'est pas lui qui les exécute : il n'est qu'un composant permettant de le faire.

### Exercice

---


Au moment de créer un EntityManager, il faut lui donner un tableau indiquant le driver à utiliser. D'où vient le nom du driver ?

- ☒ C'est le nom de l'extension PHP correspondante
- ☐ C'est le nom du préfixe du DSN correspondant
- ☐ C'est le nom du SGBD correspondant
-  Doctrine se base sur le nom de l'extension PHP nécessaire au bon fonctionnement de la connexion. Par exemple, pour utiliser le SGBD MySQL, il faut indiquer `pdo_mysql`, qui est le nom de l'extension PHP.

### Exercice

---


Comment envoyer une commande Doctrine depuis une console ouverte à la racine de son projet ?

- ☐ `php bin/vendor [commande]`
- ☒ `php vendor/bin/doctrine [commande]`
- ☐ `php doctrine [commande]`
-  Pour envoyer une commande Doctrine, il faut pointer vers le binaire Doctrine situé dans le dossier `vendor/bin/doctrine`, ce qui donne la commande : `php vendor/bin/doctrine [commande]`.

### Exercice

Sur quel fichier est-ce que Doctrine se base pour récupérer l'EntityManager à utiliser ?

- ☒ cli-config.php
- ☐ doctrine-config.php
- ☐ entity-manager.php

 Les commandes Doctrine vont récupérer l'EntityManager dans le fichier cli-config.php placé à la racine du projet.

### Exercice

Qu'est-ce qu'une entité ?


- ☐ Un tableau associatif correspondant à une table de notre base de données
- ☒ Une classe correspondant à une table de notre base de données
- ☐ Un fichier de configuration décrivant la structure d'une table de notre base de données

 Une entité est une classe PHP dont chaque propriété correspond à une colonne d'une table de notre base.

### Exercice

Comment est-ce que l'EntityManager fait le lien entre une entité et une table ?

- ☒ Grâce à des annotations
- ☐ Grâce à un fichier de configuration
- ☐ Grâce au nom des classes, qui correspond à celui de la table correspondante


 Doctrine utilise des annotations pour savoir à quelle table correspond quel champ. Il est techniquement possible d'utiliser un fichier de configuration, mais cette solution, bien que mentionnée dans la documentation, est dépréciée.

### Exercice

Quel est le meilleur moyen pour créer la base de données correspondant à nos entités ?

- ☐ En recréant à la main les tables correspondant à nos entités
- ☐ En exécutant les requêtes retournées par la commande `php vendor/bin/doctrine orm:schema-tool:create --dump-sql`

☒ En exécutant la commande `php vendor/bin/doctrine orm:schema-tool:create`

 Bien que les trois solutions fonctionnent, la plus pratique reste de laisser Doctrine créer les tables pour nous grâce à la commande `php vendor/bin/doctrine orm:schema-tool:create`.

### p. 16 Solution n°5

La première étape est de créer la base de données :

```
1 CREATE DATABASE music_prod;
```

Il faut ensuite ajouter Doctrine au projet, via Composer :

```
1 composer require doctrine/orm
```

Pour pouvoir utiliser les lignes de commande Doctrine, un fichier `cli-config.php` doit être ajouté à la racine du projet :

```

1 <?php
2
3 require_once 'vendor/autoload.php';
4
5 use Doctrine\ORM\Tools\Setup;
6 use Doctrine\ORM\EntityManager;
7 use Doctrine\ORM\Tools\Console\ConsoleRunner;
8
9 $dbParams = [
10     'driver' => 'pdo_mysql',
11     'user' => 'root',
12     'password' => '',
13     'dbname' => 'music_prod',
14 ];
15
16 $config = Setup::createAnnotationMetadataConfiguration(['entities'], true, null, null,
17     false);
18 $entityManager = EntityManager::create($dbParams, $config);
19 return ConsoleRunner::createHelperSet($entityManager);

```

Puis, un dossier `entities` doit être créé (le nom n'a pas d'importance, il doit simplement correspondre à celui mentionné dans le fichier `cli-config.php`). Dans ce dossier, on y crée le fichier `Music.php` :

```

1 <?php
2
3 use Doctrine\ORM\Mapping as ORM;
4
5 /**
6  * @ORM\Entity
7  * @ORM\Table(name="musics")
8  */
9 class Music
10 {
11     /**
12      * @ORM\Column(name="id", type="string")
13      * @ORM\Id
14      * @ORM\GeneratedValue(strategy="UUID")
15      */
16     private string $id;
17
18     /**
19      * @ORM\Column(name="title", type="string")
20      */
21     private string $title;
22
23     /**
24      * @ORM\Column(name="composer", type="string")
25      */
26     private string $composer;
27
28     /**
29      * @ORM\Column(name="length", type="integer")
30      */
31     private int $length;
32
33     /**


```

```

34     * @ORM\Column(name="creation_date", type="date")
35     */
36     private int $creationDate;
37
38     /**
39     * @ORM\Column(name="is_finished", type="boolean")
40     */
41     private bool $isFinished;
42 }

```

Enfin, la commande `vendor/bin/doctrine orm:schema-tool:create` permet de générer la table :

	#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut
<input type="checkbox"/>	1	<b>id</b> 	varchar(255)	utf8_unicode_ci		Non	Aucun(e)
<input type="checkbox"/>	2	<b>title</b>	varchar(255)	utf8_unicode_ci		Non	Aucun(e)
<input type="checkbox"/>	3	<b>composer</b>	varchar(255)	utf8_unicode_ci		Non	Aucun(e)
<input type="checkbox"/>	4	<b>length</b>	int(11)			Non	Aucun(e)
<input type="checkbox"/>	5	<b>creation_date</b>	date			Non	Aucun(e)
<input type="checkbox"/>	6	<b>is_finished</b>	tinyint(1)			Non	Aucun(e)

#### p. 16 Solution n°6

Le type `boolean` de Doctrine correspond à un `tinyint(1)`. Cela peut surprendre, mais le type booléen n'existe pas en MariaDB : Doctrine les gère donc sous forme d'entier (0 pour « false » et 1 pour « true »).

Cependant, cela est transparent pour le développeur : le type du champ `$isFinished` est bien un booléen. Lorsque nous manipulerons une instance de la classe `Music`, nous pourrons ainsi utiliser librement le type booléen, sans nous demander si la base de données comprendra ou non. Doctrine s'en chargera pour nous.