

# Découverte d'un ORM PHP

# Table des matières

<b>I. Contexte</b>	<b>3</b>
<b>II. Les bases de Doctrine : apprendre à utiliser un ORM</b>	<b>3</b>
A. Les bases de Doctrine : apprendre à utiliser un ORM .....	3
B. Exercice : Quiz .....	9
<b>III. Gestion de base de données avec ORM dans PHP</b>	<b>10</b>
A. Gestion de base de données avec ORM dans PHP .....	10
B. Exercice : Quiz .....	23
<b>IV. Essentiel</b>	<b>24</b>
<b>V. Auto-évaluation</b>	<b>25</b>
A. Exercice .....	25
B. Test .....	25
<b>Solutions des exercices</b>	<b>26</b>

## I. Contexte

**Durée :** 1 H

**Environnement de travail :** REPL.IT

**Prérequis :** aucun

### Contexte

L'ORM est une technique qui permet de faire correspondre les données d'une base de données relationnelle à des objets en programmation orientée objet. Il permet de simplifier l'accès aux données en cachant la complexité du SQL derrière une interface objet. Cependant, lorsque l'on s'intéresse au PHP, Doctrine est un exemple d'ORM à utiliser. Ce dernier est un ORM pour PHP qui permet de simplifier l'interaction avec une base de données. Il permet de travailler avec des objets plutôt que des tables. Doctrine offre une abstraction de la couche de persistance et une manière de gérer les requêtes sans écrire de SQL. Il est intégré dans de nombreux frameworks PHP, et est utilisable avec les frameworks les plus populaires, notamment Symfony et Laravel. Il est donc important de comprendre concrètement comment se présente l'ORM Doctrine dans PHP.

## II. Les bases de Doctrine : apprendre à utiliser un ORM

### A. Les bases de Doctrine : apprendre à utiliser un ORM

#### Définition

#### Qu'est-ce qu'un ORM ?

Un ORM (Object-Relational-Mapper) est un outil qui sert à créer une connexion entre les paradigmes du modèle de base de données relationnelles et la programmation orientée objet. Les enregistrements de données et les clés étrangères stockés dans les tables sont en grande partie automatiquement convertis en objets, et références lorsqu'ils sont lus à partir de la base de données. Lors de l'écriture dans la base de données, une conversion a lieu dans le sens opposé. Cela peut réduire considérablement l'effort de programmation lors de l'intégration d'une base de données SQL dans le développement backend.

De plus, de nombreuses bibliothèques ORM offrent la possibilité de construire par programmation des requêtes SQL. Cela a pour effet secondaire positif que les lacunes de sécurité dues à l'injection SQL sont pratiquement impossibles et contribuent de manière significative au développement Web sécurisé.

#### Exemple

```
1 <?php
2 $maNews = new News();
3
4 // On définit les caractéristiques de la news.
5 $maNews->titre = 'La première news du site';
6 $maNews->auteur = 'Xavier';
7 $maNews->contenu = 'Bienvenue sur mon site, j\'espère qu\'il vous plaira !';
8
9 // Puis, on sauvegarde le tout dans la base de données.
10 $maNews->save();
11
12 ?>
```

## Quels sont les avantages et inconvénients d'un ORM ?

Avantages	Inconvénients
Possibilité d'écrire dans le même langage que vous codez	L'utilisation d'un ORM peut paraître assez complexe.
Pas d'écriture en SQL	Installation et configuration de l'ORM. Tous les ORM ne sont pas compatibles avec l'ensemble des SGBD.
Fonctionnalités avancées, telles que la prise en charge des transactions, la mise en commun des connexions, les migrations, les graines, les flux, etc.	Prise en main nécessaire d'un langage interne comme le DQL (Doctrine) ou query builder (Eloquent) pour des requêtes avancées.
Complexité réduite, ce qui simplifie sensiblement le travail	L'ORM doit être maintenu à jour.
Risques de conflit de requêtes supprimés	Les erreurs sont plus récurrentes via l'utilisation d'un ORM (multi-requêtes, mauvaises jointures, etc.).

### Définition Doctrine

Doctrine est une bibliothèque de mappage d'objet-relationnel (ORM) qui permet à une application écrite en PHP de stocker des objets dans une base de données relationnelle. Cet ORM est basé sur une puissante couche d'abstraction de base de données (DBAL), qui unifie la connexion des applications à n'importe quelle base de données, telles que Symfony ou Laravel.

### Méthode Tutoriel pour maîtriser les bases de Doctrine

Maintenant que vous avez compris ce qu'est un ORM et son utilité, abordons à présent la façon de l'utiliser concrètement.

Pour son bon fonctionnement, vous devez indiquer à Doctrine la structure de vos tables. Voici ci-dessous la création d'une classe héritant de la classe Doctrine record.

```
1 <?php
2 // Nous allons travailler sur un module de news, donc la classe s'appellera News.
3 class News extends Doctrine_Record
4 {
5 }
```

Indiquons maintenant à Doctrine le nom de table, les différents champs, les types et pour finir les spécificités que la table doit renfermer. Pour ce travail, nous allons utiliser `setTableName` et `hasColumn`.

```

1 <?php
2 class News extends Doctrine_Record
3 {
4     public function setTableDefinition()
5     {
6         // On définit le nom de notre table : « news ».
7         $this->setTableName('news');
8
9         // Puis, tous les champs
10        $this->hasColumn('id', 'integer', 8, ['primary' => true,
11            'autoincrement' => true]);
12        $this->hasColumn('titre', 'string', 100);
13        $this->hasColumn('auteur', 'string', 100);
14        $this->hasColumn('contenu', 'string', 4000);
15    }
16 }
17 ?>

```

Maintenant, la classe doit être enregistrée dans le fichier : *modèles/News.class.php*.

Pour terminer, vous serez en mesure de générer votre table de la manière suivante :

```

1 <?php
2
3 try {
4     $table = Doctrine_Core::getTable('News'); // On obtient l'objet représentant la table.
5     $connexion->export->createTable($table->getTableName(), $table->getColumns()); // Ensuite,
6     nous la créons.
7     echo 'La table a été créée avec succès';
8 } catch (Doctrine_Connection_Exception $e) { // Si une exception est déclenchée.
9     echo $e->getMessage(); // Nous l'affichons.
10 }

```

Si vous voyez le message de confirmation s'afficher, cela signifie que tout s'est déroulé correctement et que la table "News" a été créée avec succès. Sinon, il y a une erreur quelque part.

En utilisant `$e->getMessages()`, vous pouvez obtenir un tableau de messages d'erreur spécifiques, qui peuvent vous fournir des détails supplémentaires sur la nature de l'erreur rencontrée. En les affichant, vous pourrez mieux comprendre la cause sous-jacente de l'erreur et prendre des mesures pour la résoudre.

### Méthode Effectuer des requêtes avec Doctrine

Bien, maintenant que tout est prêt, on va pouvoir commencer à voir comment s'exécute une requête.

Pour commencer, veuillez vider votre fichier *index.php* et ne conserver que le code suivant :

```

1 <?php
2 require_once 'lib/Doctrine.php';
3 spl_autoload_register(['Doctrine', 'autoload']);
4 require_once 'modeles/News.class.php';
5
6 // Cas à adapter selon le besoin
7 $dsn = 'mysql://root@localhost/developpement';
8 $connexion = Doctrine_Manager::connection($dsn);
9 $news->titre = 'Doctrine';
10 $news->auteur = 'author';
11 $news->contenu = 'Doctrine, what else ?';

```

Une fois cela fait, une ligne correspondant aux valeurs que vous avez indiquées sera insérée dans votre table.

### Méthode Récupération des données de la table

Maintenant, nous allons aborder la récupération des données de la table. Pour obtenir tous les enregistrements contenus dans cette table, nous utiliserons l'objet `Doctrine_Query` renvoyé par la méthode `Doctrine_Query::create()`.

```
1 <?php
2 $requete = Doctrine_Query::create();
3 $requete = Doctrine_Query::create() // création de la requête.
4     ->from('news') // sélection de la table news.
5     ->execute(); // Exécution de la requête.
6 ?>
```

Pour parcourir le résultat, vous allez juste avoir besoin d'un `foreach`.

### Méthode Créer et utiliser des fonctions de récupération à partir du modèle

Dans ce contexte, nous emploierons la méthode `Doctrine_Core::getTable`, qui requiert le nom de notre table en tant que paramètre et renvoie un objet de la classe `News`. Cette méthode permet d'extraire l'ensemble des données contenues dans une table.

### Exemple

Récupération des news :

```
1 <?php
2 $articles = Doctrine_Core::getTable('News')->findAll();
3
4 foreach($articles as $article)
5 {
6     echo $article->titre.', par <strong>'.$article->auteur.'</strong><br />';
7 }
8
9 ?>
```

### Utilisation de l'entité Manager

À partir du moment où des objets sont mappés et reconnus par l'Entity Manager, vous pouvez les manipuler. Si vous désirez mapper des entités à des tables, vous devez utiliser les formats ci-dessous :

- Dobblocs des objets PHP
- Fichiers XML
- Fichiers PHP de configuration
- Fichiers YAML

Cependant, on note une dépréciation du format YAML. Celui-ci risque de ne plus être accepté dans les futures versions de Doctrine ORM.

### Méthode Création d'une entité doctrine

Nous allons prendre l'exemple d'une classe `Article` qui aura les propriétés : `id`, `content` et `date`.

```
1 <?php
2
3 use Doctrine\ORM\Mapping as ORM;
4
5 /**
6  * @ORM\Entity
```

```
7 * @ORM\Table(name="my_entity")
8 */
9 class MyEntity
10 {
11     /**
12      * @ORM\Id
13      * @ORM\GeneratedValue
14      * @ORM\Column(type="integer")
15      */
16     private $id;
17
18     /**
19      * @ORM\Column(type="text")
20      */
21     private $content;
22
23     /**
24      * @ORM\Column(type="datetime")
25      */
26     private $date;
27
28     public function getId(): ?int
29     {
30         return $this->id;
31     }
32
33     public function getContent(): ?string
34     {
35         return $this->content;
36     }
37
38     public function setContent(string $content): void
39     {
40         $this->content = $content;
41     }
42
43     public function getDate(): ?\DateTimeInterface
44     {
45         return $this->date;
46     }
47
48     public function setDate(\DateTimeInterface $date): void
49     {
50         $this->date = $date;
51     }
52 }
```

Dans cet exemple, nous avons une entité appelée **My Entity** qui contient trois propriétés : id, content et date. L'annotation **ORM\Entity** est utilisée pour déclarer que cette classe est une entité Doctrine et **ORM\Table** est utilisée pour les spécifications de la table associée à l'entité.

Quant à l'annotation **id**, elle permet d'identifier la clé de base de la table. Avec l'annotation **@GeneratedValue**, elle permet de déléguer les responsabilités de l'unicité de l'id au système de persistance.

Avec l'annotation **@ORM\Column**, vous pouvez mapper n'importe quelle propriété PHP à une colonne qui se trouve dans la base de données.

**Remarque** PHP 8, nouvelle syntaxe possible

Depuis PHP 8, vous pouvez utiliser les attributs, aussi appelés annotations syntaxiques, comme `#[ORM\Entity]`, pour définir les métadonnées de vos entités Doctrine. Ces attributs offrent une syntaxe plus moderne et plus propre que les annotations traditionnelles. Voici comment vous pourriez réécrire votre entité en utilisant les attributs PHP 8 :

```
1 <?php
2
3 namespace App\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6
7 #[ORM\Entity]
8 #[ORM\Table(name: 'my_entity')]
9 class MyEntity
10 {
11     #[ORM\Id, ORM\GeneratedValue, ORM\Column(type: 'integer')]
12     private ?int $id;
13
14     #[ORM\Column(type: 'text')]
15     private string $content;
16
17     #[ORM\Column(type: 'datetime')]
18     private \DateTimeInterface $date;
19
20     public function getId(): ?int
21     {
22         return $this->id;
23     }
24
25     public function getContent(): string
26     {
27         return $this->content;
28     }
29
30     public function setContent(string $content): void
31     {
32         $this->content = $content;
33     }
34
35     public function getDate(): \DateTimeInterface
36     {
37         return $this->date;
38     }
39
40     public function setDate(\DateTimeInterface $date): void
41     {
42         $this->date = $date;
43     }
44 }
```



**Complément** Les différents types de mapping Doctrine

L'ensemble des types définis dans les annotations Doctrine sont différents de ceux de PHP et de ceux de la base de données, même s'ils sont mappés aux deux. En voici quelques-uns :

- **String** : il permet de mapper une propriété de type string à une colonne de type VARCHAR dans la base de données.
- **Integer** : il permet de mapper une propriété de type integer à une colonne de type INT dans la base de données.
- **Float** : il permet de mapper une propriété de type float à une colonne de type Float dans la base de données.
- **Decimal** : il permet de mapper une propriété de type decimal à une colonne de type DÉCIMAL dans la base de données.
- **Boolean** : il permet de mapper une propriété de type boolean à une colonne de type BOOLEAN.

**Méthode** Comment installer Doctrine ?

Doctrine peut être installé via Composer, le gestionnaire de dépendances pour PHP. Voici les étapes pour l'installation de Doctrine avec Composer :

Ouvrez le terminal et allez dans le répertoire de votre projet puis tapez la commande :

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

**B. Exercice : Quiz**

[solution n°1 p.27]

## Question 1

Un ORM sert à créer une connexion entre les paradigmes du modèle de base de données relationnelle et la programmation orientée objet.

- ☐ Vrai
- ☐ Faux

## Question 2

À quoi Doctrine peut-il servir ?

- ☐ Création d'un modèle
- ☐ Effectuer des requêtes
- ☐ Récupérer des données de table
- ☐ Création d'entités

## Question 3

L'ORM ne présente aucun inconvénient.

- ☐ Vrai
- ☐ Faux

## Question 4

Pour récupérer les enregistrements contenus dans une table, il faut manipuler l'objet Doctrine Query.

- ☐ Vrai
- ☐ Faux

#### Question 5

Comment installe-t-on Doctrine avec Composer ?

- ☐ En tapant la commande "composer install doctrine/orm"
- ☐ En tapant la commande "composer add doctrine/orm"
- ☐ En tapant la commande "composer require doctrine/orm"

## III. Gestion de base de données avec ORM dans PHP

### A. Gestion de base de données avec ORM dans PHP

#### Utilisation d'un ORM dans les frameworks symfony 5.4.x et Laravel 10

Laravel utilise Eloquent, un ORM basé sur Active Record où le Modèle est en charge à la fois de représenter une entité et de prendre en charge la persistance des données. Par défaut, Symfony utilise quant à lui le Doctrine ORM construit sur le principe du Data Mapper, qui sépare les concepts d'entité (un objet représentant des données), de référentiel (un objet utilisé pour collecter des entités) et de gestionnaire (objet en charge de la persistance).

#### Exemple

```

1 /**
2  * Laravel 10
3  */
4  //update
5  $post = Post::find (1);
6  $post-> name = " Marc";
7  $post-> save ();
8  //create
9  $post2 = Post::create ([ "name" => "Jean" ]);
10 //destroy
11 $post3-> destroy ();
12
13 /**
14  * Symfony 5.4.x
15  */
16 $em = $this-> getDoctrine()-> getManager(); //entity Manager
17 //update
18 $post = $em-> getRepository ("AppBundle:Post")-> find (1);
19 $post-> setName ("Marc");
20 //create
21 $post2 = new Post ();
22 $post2-> setName ("Jean");
23 $em-> persist ($post2);
24 //destroy
25 $em-> remove ($post3);
26 //On persiste toutes les opérations précédentes
27 $ em->flush ();

```

Ces deux codes sur les frameworks Laravel 10 et Symfony 5.4.x permettent de mieux expliquer leurs fonctionnements avec l'ORM.

Ici, l'ORM Eloquent est basé sur une syntaxe assez courte. Cependant, cette apparence de simplicité crée rapidement des « *fat models* » puisque toute la logique sera stockée au même endroit. Doctrine ORM permet naturellement une séparation plus juste, mais pourra relativement s'avérer verbeux pour des cas simples. Il n'y a donc pas de bons ou de mauvais choix.

Pour l'ORM Eloquent, vous pouvez gérer un code qui « grossit » par la séparation de la logique de récupération en Repository afin de garder un code plus simple. Pour Doctrine, la complexité de la structure est mitigée par des générateurs qui créent le code de base à la place du programmeur.

## Présentation de 8,1 PHP

8,1 PHP apporte à nouveau quelques améliorations de performances. Des améliorations de performances ont également été apportées à l'opcache dans la version 8.1. « *Inheritance Cache* », par exemple, relie toutes les classes clairement dépendantes (par exemple, les méthodes, les traits, les interfaces) qui devaient auparavant être compilées séparément. Ceux-ci peuvent désormais être mis en cache dans la mémoire partagée Opcache, ce qui optimise les performances. Vous pouvez utiliser cette fonction sans changer le code du programme.

Par exemple, les experts ont constaté une augmentation de 3,5 % de la vitesse de WordPress avec 8,1 PHP par rapport à la version 8.0.

## Nouvelle fonction `array_is_list()`

En PHP, vous pouvez facilement mapper des structures complexes avec un tableau. Cela vous permet d'ajouter de nouvelles clés de tableau et de modifier celles existantes pendant l'exécution. Mais attention, car cette flexibilité amène avec elle des problèmes de performances.

Vous pouvez optimiser un tableau PHP si vous numérotez les clés du tableau : de 0 au nombre d'éléments. Cependant, cette optimisation est actuellement difficile à mettre en œuvre, car la seule vérification des clés est très chronophage.

Avec la fonction « `array_is_list` », vous pouvez vérifier les clés de tableau contenues, en commençant par la valeur 0, pour voir si un ordre numérique peut être identifié.

### Méthode

```
1 array_is_list([]); // vrai
2 array_is_list(['pomme', 2, 3]); // vrai
3 array_is_list([0 => 'pomme', 'orange']); // vrai
4
5 // Le tableau ne commence pas par l'indice 0
6 array_is_list([1 => 'pomme', 'orange']); // faux
7
8 // Les clés ne sont pas ordonnées de manière séquentielle
9 array_is_list([1 => 'pomme', 0 => 'orange']); // faux
10
11 // Aucune intégration de clés séquentielles
12 array_is_list([0 => 'pomme', 'motCle' => 'valeur']); // faux
13
14 // Les clés ne sont pas consécutives
15 array_is_list([0 => 'pomme', 2 => 'valeur']); // faux
```

### Complément

Avec l'implémentation native de la fonction, vous pouvez implémenter le code encore plus efficacement, car il n'est plus nécessaire de supposer que les clés du tableau auront une valeur non numérique. PHP deviendra ainsi encore plus performant à l'avenir.

Également, une liste de tableaux PHP comprenant des clés non ordonnées est une source potentielle d'erreurs, car le strict respect de l'exigence de la liste est l'alpha et l'oméga. La fonction `array_is_list` est donc un excellent moyen de garder votre code sans bogue lorsqu'il s'agit de tableaux.

## Utilisation de Doctrine ORM avec Symfony 5

Dans la majorité des projets Symfony, on utilise l'ORM Doctrine pour gérer les bases de données. L'ORM est divisée en deux objets, à savoir l'Entity Manager et le Data Mapper. Chacun des deux possède une fonction ou un rôle bien spécifique.

Dans la première partie de ce cours, nous avons déjà eu à créer une première Doctrine ORM et nous avons présenté les types de mapping. Ces choses sont importantes à savoir pour ce qui va suivre.

## Créez et mettez à jour votre base de données

Grâce à des informations tirées du mapping des entités, Doctrine peut créer et mettre à jour une base de données.

Pour configurer Doctrine ORM dans votre application Symfony, vous devrez ajouter des informations au fichier `.env`.

```
1 #fichier .env
2 ###> doctrine/doctrine-bundle ###
3 ## Connexion à la BDD
4 DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
5 ###< doctrine/doctrine-bundle ###
```

Il est courant que vous ayez simplement à spécifier le `DATABASE_URL`, dont la structure typique peut ressembler à ce qui suit et qui est susceptible de varier selon les paramètres et le type de base de données employé :

`mysql://utilisateur:mot_de_passe@adresse_ip:port/nom_de_la_base_de_données.`

Symfony intègre nativement Doctrine ORM, enrichissant ainsi votre expérience en mettant à votre disposition un éventail de commandes accessibles *via* la console. Par exemple, pour initialiser le schéma de votre base de données, la commande suivante est à votre service : `php bin/console doctrine:schema:create`.

En cas de nécessité de mise à jour du schéma, veuillez utiliser la commande suivante : `doctrine:schema:update`.

## Créez des relations entre vos entités

Dans le développement de bases de données avec PHP, en particulier lorsqu'on utilise un ORM (Object-Relational Mapping) tel que Doctrine, il est crucial de comprendre les différents types de relations entre les entités. Les relations 1-1 (un-à-un) signifient qu'une entité est liée à une seule instance d'une autre entité. Par exemple, dans une application de gestion de personnel, un employé pourrait avoir un seul profil. Dans Doctrine, cette relation peut être représentée en utilisant les annotations `@OneToOne`.

Ensuite, nous avons la relation 1-n (un-à-plusieurs), qui indique qu'une seule entité peut être associée à plusieurs instances d'une autre entité. Par exemple, un département peut avoir plusieurs employés. Dans Doctrine, cette relation peut être mise en œuvre en utilisant l'annotation `@OneToMany`. Il est important de souligner qu'avec les relations un-à-plusieurs, il y a souvent un corollaire, à savoir les relations plusieurs-à-un (n-1). Par exemple, alors qu'un département peut avoir plusieurs employés (relation 1-n), un employé particulier appartient généralement à un seul département (relation n-1). Dans Doctrine, la relation plusieurs-à-un peut être représentée en utilisant l'annotation `@ManyToOne`. Ces deux types de relations sont généralement utilisés en tandem pour décrire les interactions complexes entre les entités d'une manière qui reflète fidèlement la structure sous-jacente des données.

Enfin, la relation n-n (plusieurs-à-plusieurs) est utilisée quand plusieurs instances d'une entité sont associées à plusieurs instances d'une autre entité. Par exemple, dans une application de gestion de bibliothèque, un livre peut être emprunté par plusieurs lecteurs et un lecteur peut emprunter plusieurs livres. Doctrine permet de gérer ce type de relation avec l'annotation `@ManyToMany`.

## Les relations de type 1-1

Les relations de type un-à-un (1-1) sont relativement peu communes en développement. Prenons comme exemple la situation où nous avons un objet nommé “Order” (Commande) qui est associé à un objet “Cart” (Panier) dans le scénario d'une boutique de commerce électronique. Dans ce contexte, une commande est essentiellement la concrétisation d'un panier que le client a approuvé. Il est important de noter que cette relation est exclusivement caractérisée dans l'une des deux entités, et il incombe au développeur de déterminer laquelle en fonction de la logique métier spécifique.

En analysant cela de manière pragmatique, il apparaît que, dans un environnement d'administration, il y a peu d'intérêt à examiner un panier qui a déjà été approuvé, car les informations pertinentes sont encapsulées dans l'objet de la commande. En revanche, en consultant le panier, nous pouvons obtenir un aperçu détaillé des produits qui ont été achetés, ce qui est précieux pour comprendre le contenu de la commande. Cela suggère qu'il est davantage judicieux de formaliser la relation au sein de la classe Order, qui servirait de point central pour accéder aux informations pertinentes, en consolidant les données d'une manière qui soutient efficacement la logique métier.

```
1 < ? php
2
3 namespace App\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6
7 class Command
8 {
9     /**
10      * Un panier par commande
11      * @ORM\OneToOne (targetEntity="App\Entity\Cart»)
12      * @ORM\JoinColumn (name="cart_id", referencedColumnName="id")
13      */
14     private $cart;
15
16     //...
17 }
18
19 class Cart
20 {
21     //...
22 }
```

Dans cet exemple, nous avons employé deux annotations de Doctrine, à savoir OneToOne et JoinColumn :

- OneToOne est utilisée pour établir la relation entre les deux entités en autorisant la spécification d'une entité associée grâce au paramètre targetEntity.
- JoinColumn, en revanche, est une annotation optionnelle qui donne la possibilité de déterminer la clé étrangère qui sert de référence au sein de la table.

## Les relations de type 1-n

Les relations un-à-plusieurs (1-n ou 1 à n) sont extrêmement courantes ! Revisitons notre exemple antérieur : supposons qu'un client ait rencontré un problème avec sa commande, qu'il n'a jamais reçue. Il est possible qu'il y ait un souci concernant son adresse de livraison ou de facturation.

Voici comment nous pourrions modéliser cette relation de manière "bidirectionnelle" (ce qui signifie que chaque entité est consciente de l'existence de la relation) :

```

1 < ? php
2
3 namespace App\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6 use Doctrine\Common\Collections\ArrayCollection;
7
8 /**
9  * Objet qui définit un client
10 */
11 class Customer
12 {
13     /**
14      * Un client a potentiellement plusieurs adresses
15      * @ORM\OneToMany (targetEntity="App\Entity\Address", mappedBy="customer")
16      */
17     private $addresses;
18
19     //...
20
21     public function __construct () {
22         $this-> addresses = new ArrayCollection();
23     }
24 }
25
26 /**
27  * Objet qui définit une adresse
28 */
29 class Address
30 {
31     //...
32
33     /**
34      * Les adresses sont liées à un client
35      * @ORM\ManyToOne (targetEntity="App\Entity\Customer", inversedBy="addresses")
36      * @ORM\JoinColumn (name="customer_id", referencedColumnName="id")
37      */
38     private $customer;
39 }

```

C'est l'opportunité pour vous d'explorer les deux attributs nommés `mappedBy` et `inversedBy`. En réalité, dans toute relation entre entités, l'une d'elles assume le rôle de « maître » de la relation. Il est essentiel de souligner que dans une relation `ManyToOne`, la clé étrangère réside invariablement du côté de la portion 'Many' de la relation, ce qui justifie qu'elle soit le maître de la relation. Dans ce scénario spécifique, on peut envisager que c'est l'entité client qui assume le rôle de maître de la relation. Effectivement, une adresse dite « P » ne saurait exister sans un client correspondant.

L'entité qui est maître de la relation doit spécifier l'attribut « `mappedBy` ». Celui-ci fait référence à la propriété de l'objet « subordonné » qui établit le lien entre les deux entités impliquées.

Quant à l'entité qui est possédée, elle doit spécifier l'attribut « `nversedBy` ». Cet attribut fait référence à la propriété de l'objet « maître » qui établit le lien entre les deux entités en question.

### Méthode Les relations de type n-n

Revenons à notre exemple du panier d'un client. Ce panier contient des produits, et ces mêmes produits peuvent apparaître dans plusieurs paniers. Ainsi, il est logique de mettre en place une relation de type ManyToMany entre l'entité Product et l'entité Cart.

Lorsque vous définissez une relation ManyToMany, Doctrine se charge automatiquement de créer la table de jointure pour vous. La table de jointure créée par Doctrine dans une relation ManyToMany contient généralement au moins deux colonnes, chacune faisant référence à la clé primaire d'une des deux tables qui sont liées par la relation ManyToMany.

### Exemple

Si vous avez une relation ManyToMany entre des entités Product et Category, Doctrine pourrait créer une table de jointure appelée `product_category` (le nom exact de la table peut varier en fonction de la configuration de Doctrine).

Cette table de jointure `product_category` contiendra au moins les colonnes suivantes :

- `product_id` : cette colonne stocke l'identifiant du produit pour chaque lien entre un produit et une catégorie. Elle est une clé étrangère qui fait référence à la clé primaire de la table `product`.
- `category_id` : cette colonne stocke l'identifiant de la catégorie pour chaque lien entre un produit et une catégorie. Elle est une clé étrangère qui fait référence à la clé primaire de la table `category`.

Chaque ligne dans cette table de jointure représente une association entre un produit et une catégorie. Donc, si un produit appartient à plusieurs catégories, il y aura plusieurs lignes pour ce produit dans la table de jointure.

Il est important de noter que Doctrine gère automatiquement la création et la maintenance de cette table de jointure pour vous. Vous n'avez généralement pas besoin de vous soucier de son contenu ou de son fonctionnement interne.

```

1 < ? php
2 namespace App\Entity;
3
4 use Doctrine\ORM\Mapping as ORM;
5 use Doctrine\Common\Collections\ArrayCollection;
6
7 /**
8  * Objet qui définit un produit
9  */
10 class Product
11 {
12     /**
13      * Un produit peut être mis dans plusieurs paniers
14      * @ORM\ManyToMany (targetEntity="App\Entity\Cart", inversedBy="products")
15      * @JoinTable (name="products_carts")
16      */
17     private $carts;
18
19     //...
20
21     public function __construct () {
22         $this-> carts = new ArrayCollection();
23     }
24 }
25
26 /**
27  * Objet qui définit un panier
28  */
29 class Cart

```

```

30 {
31     //...
32
33     /**
34      * Les produits sont liés à un panier
35      * @ORM\ManyToMany (targetEntity="App\Entity\Products", mappedBy="carts")
36      */
37     private $products;
38
39     //...
40
41     public function __construct () {
42         $this-> products = new ArrayCollection();
43     }
44 }

```

### Méthode Manipulation d'objets en base

Comme nous l'avons observé, l'Entity Manager joue un rôle clé dans la gestion de nos entités. C'est pour cette raison qu'il est nécessaire de l'injecter dans nos contrôleurs afin de « *persister* » et de « *flusher* » nos entités. Procédons à la mise à jour de notre formulaire de création d'articles que nous avons élaboré dans la section consacrée aux formulaires :

```

1 <? php
2 //src/Controller/FormController.php
3
4 /**
5  * @Route ("/form/new")
6  */
7 public function new (Request $request)
8 {
9     $article = new Article ();
10    $article-> setTitle ("Hello World");
11    $article->setContent ("Un très court article. ") ;
12    $article-> setAuthor ("Léa");
13
14    $form = $this-> createForm (ArticleType::class, $article);
15
16    $form-> handleRequest ($request);
17
18    if ($form-> isSubmitted() && $form-> isValid()) {
19        $em = $this-> getDoctrine()-> getManager();
20
21        $em-> persist ($article);
22        $em-> flush ();
23    }
24
25    return $this-> render ("default/new.html.twig", [
26        "form" => $form-> createView(),
27    ]);
28 }

```

La création d'un nouvel article est aussi simple que cela ! Lorsque vous appelez la méthode « *flush* », l'objet article sera automatiquement converti en une requête SQL de type INSERT. Vous n'avez donc pas besoin de vous préoccuper des détails techniques de cette conversion, car c'est précisément le rôle de l'ORM de s'en charger pour vous.



Lorsque vous appelez `flush()`, Doctrine examine tous les objets qu'il suit, détermine ce qui a changé, et génère les requêtes SQL nécessaires pour synchroniser ces changements avec la base de données.

Quand vous créez un nouvel objet, Doctrine ne sait rien à son sujet. Il ne le suit pas et ne sait donc pas qu'il doit l'insérer dans la base de données lorsque vous appelez `flush()`. C'est pourquoi vous devez appeler `persist()` sur le nouvel objet. Cela indique à Doctrine qu'il doit commencer à suivre cet objet et qu'il doit être inséré dans la base de données lors du prochain `flush()`.

Lorsque vous modifiez un objet qui a déjà été récupéré à partir de la base de données (et qui donc est déjà suivi par Doctrine), vous n'avez pas besoin d'appeler `persist()` à nouveau. Doctrine sait déjà qu'il doit suivre cet objet, il détectera donc automatiquement que l'objet a changé lorsque vous appelez `flush()`, et il générera une requête SQL UPDATE pour synchroniser ce changement avec la base de données.

Pour la mise à jour, c'est donc tout aussi simple :

```
1 < ? php
2 //src/Controller/FormController.php
3
4 /**
5  * @Route (" /form/edit/{id<d+>} »)
6  */
7 public function edit (Request $request, Article $article)
8 {
9     $form = $this->createForm (ArticleType::class, $article);
10
11     $form->handleRequest ($request);
12
13     if ($form->isSubmitted() && $form->isValid()) {
14         //va effectuer la requête d'UPDATE en base de données
15         $this->getDoctrine()->getManager()->flush ();
16     }
17
18     return $this->render ("default/new.html.twig",[
19         "form" => $form->createView(),
20     ]);
21 }
```

### Définition Qu'est-ce qu'un repository ?

Pour simplifier, envisagez un repository comme un objet spécialisé dont la tâche principale est de rassembler et de gérer un ensemble d'objets, souvent appelés entités.

Les repositories ont accès principalement à deux composants importants : l'EntityManager, avec lequel vous êtes peut-être déjà familier, et le QueryBuilder qui est un outil pour construire des requêtes de manière plus précise et efficace, permettant ainsi de fouiller de manière approfondie parmi l'ensemble des entités que vous avez.

Il est important de savoir que pour chaque entité, si vous n'avez pas créé un repository spécifique pour celle-ci, Doctrine ORM met à disposition un repository par défaut. Ce repository par défaut peut être utilisé aisément à travers tous les contrôleurs dans vos applications.

### Exemple

```
1 < ? php
2
3 $repository = $this->getDoctrine()->getRepository (Article::class);
4
5 //Récupère l'objet en fonction de l'@Id (généralement appelé $ id)
6 $ article = $ repository->find ($ id);
```

```

7
8 //Recherche d'un seul article par son titre
9 $ article = $ repository->findOneBy (['title' => 'Amazing article']);
10
11 //Ou par titre et nom d'auteur
12 $ article = $ repository->findOneBy ([
13     «title» => «Amazing Article»,
14     «price» => «Léa»,
15 ]);
16
17 //Recherche de tous les articles en fonction de multiples conditions
18 $ articles = $ repository->findBy (
19     [«author» => «Léa»],
20     [«title» => «ASC»], //le deuxième paramètre permet de définir l'ordre
21     10, //le troisième la limite
22     2 //et à partir duquel on récupère [«OFFSET» au sens MySQL]
23 );
24
25 //Retrouver tous les articles
26 $articles = $repository-> findAll();

```

Avec toutes ces fonctions, 99 % des besoins de vos applications devraient être couverts. Si vous souhaitez créer des fonctions spécifiques, il faudra déclarer et créer vos propres repositories :

```

1 < ? php
2
3 namespace App\Repository;
4
5 use App\Entity\Article;
6 use Doctrine\Common\Persistence\ManagerRegistry;
7 use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
8
9 class ArticleRepository extends ServiceRepository
10 {
11     public function __construct (ManagerRegistry $registry)
12     {
13         parent::__construct ($registry, Article::class);
14     }
15
16     public function findLastArticles()
17     {
18         return $this-> findBy ([], [“publicationDate” => “DESC”]);
19     }
20 }
21
22
23 //dans un contrôleur
24
25 public function getLastArticles (ArticleRepository $repository)
26 {
27     $articles = $repository-> findLastArticles(); //ArrayCollection
28
29     //...

```

## Gérez les migrations de votre base de données

Abordons à présent un sujet sensible qui se présente lorsqu'on travaille sur une application en constante évolution et déjà déployée en production. Imaginez que votre client vous sollicite pour effectuer des changements qui affectent vos entités et, par conséquent, votre base de données. Vous devez actualiser la base de données, mais cela s'avère être une opération périlleuse.

C'est ici que les fichiers de migration entrent en jeu, en tant que méthode fiable pour actualiser la base de données, que ce soit localement ou sur un serveur en production.

Au lieu d'utiliser la commande `doctrine:schema:update` ou de modifier directement la base de données avec des scripts SQL, les fichiers de migration offrent une façon sécurisée d'effectuer les modifications nécessaires. L'un des avantages majeurs est qu'ils permettent non seulement d'appliquer ces changements, mais aussi de les révoquer sans compromettre l'intégrité de la base de données.

Suite à la modification du schéma de la base de données, les fichiers de migration permettent de générer une migration automatiquement. Cette migration contient deux parties : la partie « *Up* » (mise à jour) qui décrit les actions à effectuer pour mettre à jour le schéma de la base de données et la partie « *Down* » (rollback) qui décrit les actions à effectuer pour revenir à l'état précédent de la base de données.

Le bundle DoctrineMigration fournit un ensemble d'outils pour gérer ces fichiers de migration. Il permet notamment de générer des fichiers de migration en fonction des changements apportés aux entités, de les exécuter pour mettre à jour la base de données et de les annuler en cas de besoin.

Il est recommandé d'utiliser les fichiers de migration pour gérer les modifications de la base de données en production, afin de minimiser les risques d'erreur et de corruption de la base de données.

Cependant, notez que l'utilisation de fichiers de migration peut nécessiter une certaine expertise en matière de gestion de bases de données et de migrations. Il est donc recommandé de se former à l'utilisation de ce bundle avant de l'utiliser en production.

### Complément Installation et configuration

Pour installer cette extension dans votre projet, vous aurez besoin de composer :

```
1 $ composer require doctrine/doctrine-migrations-bundle « ^2.0 »
```

Doctrine Migrations permet de gérer les versions de la base de données de manière efficace et programmatique.

Si l'installation a été réalisée avec succès, l'utilisateur se verra accorder l'accès à un ensemble de commandes supplémentaires au sein de la console Symfony. Ces commandes sont spécifiques à la gestion des migrations de base de données avec Doctrine. Voici un aperçu plus détaillé de ces commandes :

- `doctrine:migrations:diff` - Cette commande est utilisée pour générer automatiquement un fichier de migration. Elle compare la structure actuelle de la base de données avec les schémas définis dans votre code (via les métadonnées de mapping) et crée un script SQL pour synchroniser la base de données avec le schéma de votre code.
- `doctrine:migrations:execute` - Permet d'exécuter un fichier de migration spécifique manuellement. C'est utile si vous avez besoin d'appliquer des changements spécifiques à la base de données sans exécuter l'ensemble des migrations.
- `doctrine:migrations:generate` - Crée un fichier de classe de migration vide. Cela est utile si vous souhaitez écrire votre propre script de migration SQL personnalisé.
- `doctrine:migrations:migrate` - Cette commande exécute les fichiers de migration en attente. Elle met à jour la base de données en appliquant les migrations jusqu'à la plus récente, ou jusqu'à celle spécifiée.
- `doctrine:migrations:status` - Affiche des informations sur l'état des migrations. Cela inclut des informations sur les migrations déjà exécutées, celles qui ne sont pas encore exécutées, etc.

- doctrine:migrations:version (ou version en abrégé) - Permet d'ajouter ou de supprimer manuellement des versions de migration. Cela est utile pour gérer l'état de vos migrations sans exécuter réellement les scripts SQL.

### Méthode

Commençons par regarder le statut des migrations de notre application :

```
1 bin/console doctrine:migrations:status
2
3 == Configuration
4
5 >> Name: Application Migrations
6 >> Database Driver: pdo_sqlite
7 >> Database Name: /var/www/html/var/data/blog.twis
8 >> Configuration Source: manually configured
9 >> Version Table Name: migration_versions
10 >> Version Column Name: version
11 >> Migrations Namespace: App\Migrations
12 >> Migrations Directory: /var/www/html/migrations
13 >> Previous Version: Already at first version
14 >> Current Version: 0
15 >> Next Version: Already at latest version
16 >> Latest Version: 0
17 >> Executed Migrations: 0
18 >> Executed Unavailable Migrations: 0
19 >> Available Migrations: 0
20 >> New Migrations: 0
```

Nous venons de procéder à l'installation de l'extension, et à ce stade, nous n'avons ni fichier de migration ni version de migration appliqué à notre base de données. Comme nous avons travaillé avec un nouvel objet appelé Article tout au long du cours, notre base de données n'est pas à jour par rapport à cet objet.

### Élaborez votre premier fichier de migration

Afin de créer votre tout premier fichier de configuration, utilisez la commande `doctrine:migrations:generate`. Voici comment l'utiliser :

```
bin/console doctrine:migrations:generate
```

```
Generated new migration class to
"/var/www/html/migrations/Version20181028033516.php"
```

En ouvrant le fichier correspondant, vous constaterez qu'il n'est pas très utile.

```
1 < ? php
2 namespace App\Migrations;
3 use Doctrine\DBAL\Migrations\AbstractMigration,
4 Doctrine\DBAL\Schema\Schema;
5
6 class Version20181028033516 extends AbstractMigration
7 {
8     public function up (Schema $schema)
9 {
10
11     }
12
13     public function down (Schema $schema)
14 {
```

```

15
16     }
17 }

```

Pourtant, si l'on utilise la commande `doctrine:migrations:status`, il y aura du changement.

```

1 bin/console doctrine:migrations:status
2
3 == Configuration
4
5 >> Name: Application Migrations
6 >> Database Driver: pdo_sqlite
7 >> Database Name: /var/www/html/var/data/blog. twist
8 >> Configuration Source: manually configured
9 >> Version Table Name: migration_versions
10 >> Version Column Name: version
11 >> Migrations Namespace: App\Migrations
12 >> Migrations Directory: /var/www/html/migrations
13 >> Previous Version: Already at first version
14 >> Current Version: 0
15 >> Next Version: 2018-10-28 3:35:16 am (20,181,028,033,516)
16 >> Latest Version: 2018-10-28 3:35:16 am (20,181,028,033,516)
17 >> Executed Migrations: 0
18 >> Executed Unavailable Migrations: 0
19 >> Available Migrations: 1
20 >> New Migrations: 1

```

Un fichier de migration (vide!) a été créé, il est donc disponible. Pour preuve, utilisons la commande de migration pour appliquer nos modifications !

```

1 bin/console doctrine:migrations:migrate 20,181,028,033,516
2
3 Application Migrations
4
5 WARNING! You are about to execute a database migration that could result in schema changes and
6 data loss. Are you sure you wish to continue? (y/n) y
7 Migrating up to 20,181,028,033,516 from 0
8
9 ++ migrating 20,181,028,033,516
10
11 Migration 20,181,028,033,516 was executed but did not result in any SQL statements.
12
13 ++ migrated (0.03s)
14
15 -----
16 ++ finished in 0.03s
17 ++1 migrations executed
18 ++0 sql queries

```

Doctrine ORM est capable de générer automatiquement les fichiers de migration, ce qui constitue l'un des principaux avantages de l'utilisation d'un ORM plutôt que de gérer manuellement les migrations.

#### Méthode

Voyons maintenant comment procéder ensemble pour générer automatiquement ces fichiers de migration.

Dans ce cours, nous avons créé une nouvelle entité appelée Article, qui a été correctement mappée et reconnue par Doctrine ORM. Utilisons cette fois la commande `doctrine:migrations:diff`:

→ `bin/console doctrine:migrations:migrate 20,181,028,033,516`

Generated new migration class to  
"/var/www/html/migrations/Version20181028040934.php" from schema differences.

Cela ne vous aidera pas beaucoup. Regardons donc le fichier généré par le bundle Doctrine Migrations, cette fois :

```
1 < ? php
2
3 namespace App\Migrations;
4
5 use Doctrine\DBAL\Migrations\AbstractMigration,
6 Doctrine\DBAL\Schema\Schema;
7
8 class Version20181028033516 extends AbstractMigration
9 {
10     public function up (Schema $schema)
11     {
12         //this up () migration is auto-generated, please modify it to your needs
13 $this-> abortIf ($this-> connection-> getDatabasePlatform[]-> getName[]! == "sqlite",
14 "Migration can only be executed safely on \'sqlite\'");
15
16 $this-> addSql ("CREATE TABLE article [id INT NOT NULL, title VARCHAR(255) NOT NULL,
17 content TEXT NOT NULL, date DATETIME NOT NULL, PRIMARY KEY (id)]");
18
19     public function down (Schema $schema)
20     {
21         //this down () migration is auto-generated, please modify it to your needs
22 $this-> abortIf ($this-> connection-> getDatabasePlatform[]-> getName[]! == "sqlite",
23 "Migration can only be executed safely on \'sqlite\'");
24
25 $this-> addSql ("DROP TABLE article");
26     }
27 }
```

L'extension a automatiquement créé le fichier de migration que vous pourrez appliquer en toute sécurité en local, comme sur votre serveur de production. En cas de problèmes, appliquez la migration précédente.

## Présentation de Laravel 9

Laravel 9 prend en charge PHP à partir de la version 8.1.

Tout d'abord, Laravel coupe les vieilles habitudes : 8,0 PHP et les versions antérieures ne sont plus pris en charge dans Laravel. Afin de pouvoir se concentrer pleinement sur le développement ultérieur de PHP, le framework sera désormais basé sur les nouvelles versions de PHP et adaptera les fonctionnalités actuelles et futures à 8,1 PHP et supérieures. À titre d'exemple, Laravel cite les propriétés readonly.

### Complément Déclarations de type natif dans le squelette de code Laravel

Certaines déclarations de type supplémentaires ont été ajoutées dans les versions récentes de PHP, que Laravel prend désormais entièrement en charge. Ceux-ci incluent les indications de type, les types de retour et les types d'union. La dernière version de Laravel intègre ces types dans son framework. Cependant, les applications qui utilisent des versions plus anciennes et leur système de type peuvent perdre leur fonctionnalité en conséquence.

**Complément** Comment configurer Laravel ?

Pour pouvoir parfaitement utiliser Laravel, il faut d'abord comprendre son écosystème. Pour son installation, rendez-vous sur **getcomposer.org**. La seconde chose à faire est la création des migrations avec la console Artisan et la dernière est de créer des modèles Eloquent. Pour plus d'approfondissement sur le sujet, vous pouvez vous rendre sur « Seed » qui est la base de données.

**Complément** Création et exécution d'une migration

Un fichier de migration doit d'abord être créé pour chaque table de base de données à utiliser avec l'ORM Eloquent. Entre autres choses, la structure de la nouvelle table est définie dans ce fichier. Il peut également être généré directement lors de la création du modèle. Un nouveau fichier de migration doit également être créé dans le cas où une table existante doit être modifiée. La commande suivante est requise pour créer réellement la nouvelle table ou pour modifier une table existante.

Pour créer une migration, on utilise la commande : `php artisan make : migration create_users_table`. Ensuite, celui-ci doit être placé dans un dossier **database/migration**.

Pour l'exécution, on utilise la commande : `php artisan migrate`. Pour créer la migration avec un modèle en même temps, il faut utiliser : `php artisan make:model - - migration NomModel`.

**Création et exécution d'un modèle avec Eloquent ORM**

Créer une nouvelle classe de modèle pour une application, cela correspond généralement à une table BD. Si nous voulons créer un contrôleur pour le nouveau modèle en même temps, nous pouvons écrire la commande comme ceci :

```
1 $ php artisan make:model MainMenusModel --c
```

Si nous travaillons avec Eloquent ORM et que le modèle doit également être préparé pour la base de données en même temps, nous pouvons étendre la commande :

```
1 $ php artisan make:model MainMenusModel --c --m
```

Cela permet d'économiser beaucoup de temps de travail.

Pour définir une propriété `$primaryKey` il faut utiliser cette convention : **`public $primaryKey = 'ncin'`**.

Il convient de préciser que la convention pour définir le nom de la clé primaire dans un modèle Eloquent est la suivante : il faut définir la propriété `$primaryKey` avec le nom de la clé souhaitée. Par défaut, Eloquent suppose que la clé primaire de la table est la colonne "id".

Quant aux timestamps, la propriété `$timestamps` est utilisée pour indiquer à Eloquent si la table contient ou non des colonnes "created\_at" et "updated\_at". Par défaut, Eloquent suppose que la table possède ces deux colonnes et `$timestamps` est donc initialisé à `true`. Si la table ne possède pas ces colonnes, il faut définir `$timestamps` à `false`.

Pour ce qui est des timestamps, il faut : **`public $timestamps = false`**

**B. Exercice : Quiz**

[solution n°2 p.28]

## Question 1

Quel ORM est préconisé avec le framework Laravel ?

- ☐ Doctrine
- ☐ Eloquent
- ☐ Propel

## Question 2

Quel est l'objet en charge de la persistance des données dans Doctrine ORM ?

- ☐ Entity Manager
- ☐ Repository
- ☐ Data Mapper

Question 3

Comment configure-t-on Doctrine ORM dans une application Symfony ?

- ☐ En modifiant le fichier .htaccess
- ☐ En définissant le DATABASE\_URL dans le fichier .env ou .env.local
- ☐ En créant un fichier de configuration YAML

Question 4

Quelle est la différence entre Eloquent et Doctrine ORM ?

- ☐ Eloquent est basé sur Active Record tandis que Doctrine ORM est basé sur le principe du Data Mapper
- ☐ Eloquent est basé sur le principe du Data Mapper tandis que Doctrine ORM est basé sur Active Record
- ☐ Eloquent et Doctrine ORM sont basés sur le principe du Data Mapper

Question 5

Quel est l'objectif de la fonction "persist" dans Doctrine ORM ?

- ☐ Mettre à jour une entité dans la base de données
- ☐ Supprimer une entité de la base de données
- ☐ Ajouter une nouvelle entité à la base de données

## IV. Essentiel

Nous avons vu dans ce cours comment l'utilisation de l'ORM se présentait avec les frameworks de PHP que sont : Symfony et Laravel. Dans un premier temps nous avons donc présenté les bases de Doctrine afin de comprendre et de maîtriser comment on utilise un ORM. Ainsi, l'ORM qui est une abréviation de Object-Relational-Mapper permet de créer une connexion entre les paradigmes du modèle de base de données relationnelles et la programmation orientée objet.

Comme avantage, l'ORM permet d'écrire dans une langue que l'on maîtrise déjà. Il fait aussi abstraction du système de base de données et, en fonction de l'ORM que vous utilisez, vous pouvez bénéficier de nombreuses fonctionnalités avancées comme la prise en charge des transactions, les migrations, les graines, etc. Pour que votre travail soit simplifié, la complexité est totalement réduite.

Parlant de Doctrine, par définition, c'est une bibliothèque de Mappage Objet-Relationnel (ORM) qui permet aux applications écrites en PHP de stocker leurs objets dans une base de données relationnelles. On peut utiliser Doctrine pour créer un modèle, effectuer une requête, récupérer des données de table, etc.

Dans un second temps, nous avons vu comment l'ORM fonctionne réellement avec PHP, notamment ces frameworks Symfony et Laravel.

Laravel utilise un ORM Eloquent qui se base sur Active Record où le model est chargé de représenter une entité et aussi prend en charge la persistance des données. Symfony, quant à lui, est axé sur un ORM appelé Doctrine. Ce dernier est construit sur le principe du Data Mapper qui permet de séparer les concepts d'entité, de gestionnaire ainsi que de référentiel.



## V. Auto-évaluation

### A. Exercice

Vous travaillez en tant que développeur dans une agence de développement web. Un client potentiel vous contacte pour discuter de son projet de création d'une application web. Au cours de la discussion, le client mentionne qu'il souhaite utiliser un ORM pour gérer les interactions avec la base de données de son application. Le client est au courant qu'il existe des ORM pour PHP, mais il ne connaît pas les différences entre les deux frameworks les plus populaires pour le développement web en PHP : Laravel et Symfony.

#### Question 1

[solution n°3 p.29]

Pouvez-vous expliquer, au client à l'aide d'exemple de code, la différence de syntaxe entre l'utilisation de l'ORM dans Laravel et dans Symfony ?

#### Question 2

[solution n°4 p.30]

Le client a compris qu'il y a une différence de syntaxe pour mettre à jour un objet entre Symfony et Laravel. Mais il vous interroge sur d'autres fonctionnalités d'un ORM, comme la gestion des relations entre les objets. Est-ce que la syntaxe est similaire entre les deux frameworks pour ces fonctionnalités, par exemple la relation many-to-many ? Répondez à cette question en utilisant des exemples de code.

### B. Test

#### Exercice 1 : Quiz

[solution n°5 p.31]

##### Question 1

Que produit un "flush" ?

- ☐ Extraction d'une entité
- ☐ Enregistrement de l'ensemble des entités supervisées par Doctrine ORM
- ☐ Actualisation d'une entité dans la base de données

##### Question 2

Qu'est-ce que l'opcode en PHP ?

- ☐ Un compilateur de code PHP
- ☐ Un cache de code PHP compilé
- ☐ Un outil de débogage pour PHP

##### Question 3

Que produit un "find" au sein d'Eloquent ?

- ☐ Extraire une entité en utilisant son identifiant
- ☐ Actualiser une entité dans la base de données
- ☐ Effacer une entité de la base de données

##### Question 4

Quelle est la fonction de la commande "doctrine:schema:update" dans Symfony ?

- ☐ Créer la base de données à partir des entités définies
- ☐ Mettre à jour la base de données en fonction des changements apportés aux entités
- ☐ Supprimer la base de données

## Question 5

Qu'est-ce qu'un repository ?

- ☐ Un objet qui permet de récupérer une collection d'objets
- ☐ Une méthode pour stocker des objets dans une base de données relationnelle
- ☐ Une bibliothèque de Mappage Objet-Relationnel (ORM)


**Solutions des exercices**

**Exercice p. 9 Solution n°1****Question 1**

Un ORM sert à créer une connexion entre les paradigmes du modèle de base de données relationnelle et la programmation orientée objet.

☒ Vrai

☐ Faux

 L'ORM permet une conversion automatique des enregistrements et des clés étrangères stockés dans les tables en objets et références lors de la lecture depuis la base de données. De même, lors de l'écriture dans la base de données, une conversion automatique a lieu dans le sens contraire.

**Question 2**


À quoi Doctrine peut-il servir ?

☒ Création d'un modèle

☒ Effectuer des requêtes

☒ Récupérer des données de table

☒ Création d'entités


 Toutes ces réponses sont correctes. Doctrine peut être utilisé pour toutes ces tâches. Doctrine permet la création d'un modèle en définissant des entités qui sont des classes PHP correspondant aux tables de la base de données. On peut effectuer des requêtes en utilisant le langage DQL (Doctrine Query Language), qui est très similaire à SQL mais fonctionne avec les objets et les classes PHP au lieu de tables et de colonnes de base de données. Les données peuvent être récupérées à partir de la base de données en utilisant ces requêtes, qui sont ensuite automatiquement converties en instances des classes d'entité correspondantes. En outre, la création d'entités est une tâche courante lors de l'utilisation de Doctrine, permettant de décrire et de gérer la structure des données.

**Question 3**

L'ORM ne présente aucun inconvénient.

☐ Vrai

☒ Faux


 L'utilisation de l'ORM ne présente pas que des avantages. En effet, sa configuration initiale n'est pas facile. Vous devrez fournir un certain effort afin d'utiliser l'ORM. Pour finir, on peut facilement avoir des requêtes plus performantes en les évacuant soi-même avec une maîtrise du SQL.

**Question 4**

Pour récupérer les enregistrements contenus dans une table, il faut manipuler l'objet Doctrine Query.


☒ Vrai

☐ Faux

 Pour manipuler les enregistrements d'une table, il faut passer par la méthode `Doctrine_Query::create()` :. Cette dernière exige la manipulation de l'objet `Doctrine_Query`.

### Question 5


Comment installe-t-on Doctrine avec Composer ?

- ☐ En tapant la commande “composer install doctrine/orm”
- ☐ En tapant la commande “composer add doctrine/orm”
- ☒ En tapant la commande “composer require doctrine/orm”
-  Composer utilise le mot “require” pour installer une dépendance et “remove” pour la supprimer.

## Exercice p. 23 Solution n°2


### Question 1

Quel ORM est préconisé avec le framework Laravel ?

- ☐ Doctrine
- ☒ Eloquent
- ☐ Propel
-  Il est également possible d'utiliser d'autres ORM tel que Doctrine ORM en installant les packages appropriés. Cependant, l'utilisation d'Eloquent est fortement recommandée pour profiter pleinement des fonctionnalités de Laravel.


### Question 2

Quel est l'objet en charge de la persistance des données dans Doctrine ORM ?

- ☒ Entity Manager
- ☐ Repository
- ☐ Data Mapper
-  L'Entity Manager est l'objet en charge de la persistance des données dans Doctrine ORM. Il permet de gérer les entités et de les persister dans la base de données.

### Question 3

Comment configure-t-on Doctrine ORM dans une application Symfony ?

- ☐ En modifiant le fichier .htaccess
- ☒ En définissant le DATABASE\_URL dans le fichier .env ou .env.local
- ☐ En créant un fichier de configuration YAML
-  Pour configurer Doctrine ORM dans une application Symfony, il faut définir le DATABASE\_URL dans le fichier .env en production ou dans le fichier .env.local en cas d'un développement en local. Cette configuration permet à Symfony de communiquer avec la base de données.

### Question 4

Quelle est la différence entre Eloquent et Doctrine ORM ?

- ☒ Eloquent est basé sur Active Record tandis que Doctrine ORM est basé sur le principe du Data Mapper
- ☐ Eloquent est basé sur le principe du Data Mapper tandis que Doctrine ORM est basé sur Active Record
- ☐ Eloquent et Doctrine ORM sont basés sur le principe du Data Mapper
- ☐ Eloquent est un ORM basé sur Active Record, tandis que Doctrine ORM est basé sur le principe du Data Mapper. La différence entre les deux réside dans la façon dont ils gèrent la persistance des données.

### Question 5

Quel est l'objectif de la fonction "persist" dans Doctrine ORM ?

- ☐ Mettre à jour une entité dans la base de données
- ☐ Supprimer une entité de la base de données
- ☒ Ajouter une nouvelle entité à la base de données
- ☐ La fonction "persist" dans Doctrine ORM permet d'ajouter une nouvelle entité à la base de données.

### p. 25 Solution n°3

```

1 /**
2
3 •   Laravel
4
5 **/
6
7 // update
8
9 $post = Post ::find(1) ;
10
11 $post->name = « Marc » ;
12
13 $post->save() ;
14
15 // create
16
17 $post2 = Post ::create(['name' => 'Jean']) ;
18
19 // destroy
20
21 $post3->destroy() ;
22
23 /**
24
25 •   Symfony
26
27 **/
28
29 $em = $this->getDoctrine()->getManager() ; // entity Manager
30
31 // update
32
33 $post = $em->getRepository('AppBundle :Post')->find(1) ;
34
35 $post->setName('Marc') ;

```

```

12
13 // create
14
15 $post2 = new Post() ;
16
17 $post2->setName('Jean') ;
18
19 $em->persist($post2) ;
20
21 // destroy
22
23 $em->remove($post3) ;
24
25 // On persiste toutes les opérations précédentes
26
27 $em->flush() ;

```

Dans Laravel, l'ORM Eloquent utilise une syntaxe concise et facile à comprendre qui permet de réaliser des opérations de base comme la mise à jour et la suppression des données avec peu de code. En revanche, dans Symfony, l'utilisation de l'ORM Doctrine nécessite une syntaxe plus verbeuse. Par exemple, pour mettre à jour un objet Post dans Doctrine, on utilise la méthode find() pour récupérer l'objet, on modifie ses attributs, on appelle la méthode persist() pour informer l'EntityManager que l'objet doit être mis à jour, et enfin on appelle la méthode flush() pour enregistrer les modifications dans la base de données.

#### p. 25 Solution n°4

En effet, la syntaxe pour gérer les relations entre les objets est également différente entre Laravel et Symfony. Par exemple, si vous avez une relation "Many-to-Many" entre les objets Post et Tag, vous pouvez la gérer de la manière suivante avec Laravel :

```

1 // Ajouter un tag à un post
2 $post = Post::find(1);
3 $post->tags()->attach($tagId);
4
5 // Supprimer un tag d'un post
6 $post->tags()->detach($tagId);
7
8 // Récupérer tous les posts pour un tag donné
9 $posts = Tag::find($tagId)->posts;

```

En revanche, la syntaxe pour gérer les relations "Many-to-Many" avec Doctrine dans Symfony est la suivante :

```

1 // Ajouter un tag à un post
2 $post = $em->getRepository(Post::class)->find($postId);
3 $tag = $em->getRepository(Tag::class)->find($tagId);
4 $post->addTag($tag);
5
6 // Supprimer un tag d'un post
7 $post->removeTag($tag);
8
9 // Récupérer tous les posts pour un tag donné
10 $tag = $em->getRepository(Tag::class)->find($tagId);
11 $posts = $tag->getPosts();


```

La syntaxe pour gérer les relations "Many-to-Many" est bien différente. Cependant, les deux frameworks offrent des fonctionnalités complètes pour gérer les relations entre les objets, y compris les relations "One-to-One", "One-to-Many" et "Many-to-Many".

**Exercice p. 25 Solution n°5****Question 1**

Que produit un "flush" ?


- ☐ Extraction d'une entité
- ☒ Enregistrement de l'ensemble des entités supervisées par Doctrine ORM
- ☐ Actualisation d'une entité dans la base de données

 Dans Doctrine, ainsi que dans beaucoup d'ORMs, la méthode "flush" est utilisée pour sauvegarder les modifications effectuées sur les objets qui sont sous la gestion de l'EntityManager, en les enregistrant dans la base de données. Lorsque vous appelez la méthode "flush", les modifications effectuées sur les entités sont traduites en requêtes SQL appropriées (INSERT, UPDATE, DELETE, etc.) et exécutées au niveau de la base de données pour refléter ces changements.

**Question 2**

Qu'est-ce que l'opcache en PHP ?


- ☐ Un compilateur de code PHP
- ☒ Un cache de code PHP compilé
- ☐ Un outil de débogage pour PHP

 L'opcache en PHP est un cache de code PHP compilé qui permet d'optimiser les performances de l'application.

**Question 3**

Que produit un "find" au sein d'Eloquent ?


- ☒ Extraire une entité en utilisant son identifiant
- ☐ Actualiser une entité dans la base de données
- ☐ Effacer une entité de la base de données

 La méthode "find" est employée afin de retrouver une entité en utilisant sa clé primaire (ID).

**Question 4**

Quelle est la fonction de la commande "doctrine:schema:update" dans Symfony ?

- ☐ Créer la base de données à partir des entités définies
- ☒ Mettre à jour la base de données en fonction des changements apportés aux entités
- ☐ Supprimer la base de données

 La commande "doctrine:schema:update" dans Symfony permet de mettre à jour la base de données en fonction des changements apportés aux entités.

**Question 5**

Qu'est-ce qu'un repository ?

- Ⓐ Un objet qui permet de récupérer une collection d'objets
- Une méthode pour stocker des objets dans une base de données relationnelle
- Une bibliothèque de Mappage Objet-Relationnel (ORM)
- Q Les repositories Doctrine ORM ont accès à deux objets principalement, à savoir l'EntityManager et un QueryBuilder.