

# Les entités

# Table des matières

<b>I. Comprendre les entités</b>	<b>3</b>
A. La base de données relationnelle.....	3
B. Création d'une Entité .....	4
1. Utilisation de MakerBundle .....	4
C. Analyse d'une Entité .....	5
D. Repository .....	7
<b>II. Exercice : Quiz</b>	<b>7</b>
<b>III. Les relations</b>	<b>8</b>
A. Les différentes notions .....	8
B. Les attributs et annotations dans l'entité .....	9
C. Les types de relations : .....	10
<b>IV. Exercice : Quiz</b>	<b>11</b>
<b>V. Essentiel</b>	<b>12</b>
<b>VI. Auto-évaluation</b>	<b>12</b>
A. Exercice .....	12
B. Test .....	12
<b>Solutions des exercices</b>	<b>13</b>

# I. Comprendre les entités

**Durée :** 1 h 30

**Pré requis :**

Notion de PHP

Notion de POO

## Contexte

Les avantages d'un framework comme Symfony, même s'il demande un certain apprentissage, sont nombreux. L'un d'eux est l'organisation de sa structure qui respecte l'architecture MVC (Modèle-Vue-Contrôleur). Cette architecture est aujourd'hui omniprésente dans la POO (Programmation-Orientée-Objet) pour les applications web modernes. Avant de renvoyer une Vue, un contrôleur aura la plupart du temps besoin de récupérer des éléments qui agrémenteront d'information celle-ci, cela représente notre Modèle du MVC.

Mais qu'est-ce qu'un Modèle ?

Un modèle comprend entre autres la gestion des données, que celle-ci provienne d'un fichier, d'un objet ou encore d'une base de données. Ainsi le contrôleur pourra faire passer le Modèle à la Vue.

Pourquoi parlez-vous de Modèle dans ce cours ? C'est parce que les Entités sont un composant important du Modèle sous Symfony.

À travers ce cours nous détaillerons ce qu'est une Entité, nous la décortiquerons et même si nous installerons Doctrine qui est indispensable, nous ne nous concentrerons pas sur la manipulation des Entités par cet ORM qui travaille de concert avec Symfony. Le but étant de découvrir le rôle d'une Entité dans un projet Symfony 5.4.

## Définition Une entité

Une entité avec Symfony, c'est un objet qui sera représenté sous forme de classe. Mais cet objet est un peu particulier avec Symfony car il est géré par Doctrine qui est un ORM (Object-Relational Mapping). Un ORM a pour rôle de faire le pont entre un langage objet et un langage de base de données (type MySQL, MongoDB, etc.). Ainsi grâce à Doctrine vous n'avez en théorie plus besoin de passer une seule commande SQL (Structured Query Language) car il se chargera de faire les requêtes nécessaires à votre place via des méthodes PHP.

Dans le fonctionnement de Symfony, une entité représente une table de la base de données. Lorsqu'elle est instanciée dans la BDD (Base De Données), ses attributs deviennent les champs de la table.

## A. La base de données relationnelle.

Nous l'avons vu, une Entité représente une table de la base de données. Ainsi il convient de configurer notre application afin que Doctrine nous aide à créer cette base de données puis à terme la manipuler.

Tout d'abord, assurez-vous que Doctrine est bien installé. Sinon, tapez les lignes de commandes ci-dessous :

```
1 $ composer require symfony/orm-pack
2 $ composer require --dev symfony/maker-bundle
```

Ensuite dans votre fichier .env, il va falloir configurer votre connexion à la base de données en personnalisant vos paramètres dans la variable d'environnement DATABASE\_URL.

Voici un exemple :

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=mariadb-10.4.17&charset=utf8mb4"
```

- db\_user est à remplacer par votre nom d'utilisateur de la base de données.
- db\_password est à remplacer par le mot de passe pour accéder à la base de données.

- db\_name est à remplacer par le nom que vous souhaitez utiliser pour la base de données.
- 3306 et le port par défaut, mais il vous faudra vérifier que c'est bien celui-ci utiliser pour l'accès à votre base de données.
- serverVersion doit contenir la version du serveur que votre ordinateur utilise.

Les bases de données relationnelles les plus utilisées sont MySQL, Mariadb avec MySQL, PostgreSQL, SQLite, oci8.

Vous pouvez maintenant créer la base de données avec la commande :

```
1 $ php bin/console doctrine:database:create
```

Il convient également, comme pour tout projet, de préparer le design de votre base de données. Pour cela on pourra utiliser UML pour la partie objet ou imaginer notre base de données et ses tables avec Merise.

Les diagrammes de classe ou encore les modèles physiques de données vous permettront à l'avance de définir vos propriétés, vos attributs et les relations entre chacune des entités.

## B. Création d'une Entité

Il est possible de créer manuellement un fichier PHP dans le dossier Entity et de développer l'entité avec ses méthodes et ses attributs, puis de faire une migration vers la base de données. Mais, grâce à Maker-bundle que nous avons installé, Doctrine peut nous aider à créer facilement une Entité. Alors, créons une Entité « tâche » que nous appellerons Task puis détaillons notre nouvelle Entité.

### 1. Utilisation de MakerBundle

#### Méthode

MakerBundle est un composant exclusivement développé pour aider à la création de Contrôleurs, de formulaires ou d'entités. Ainsi une fois installé vous pourrez Entité avec la ligne de commande suivante :

```
1 $ php bin/console make:entity
```

Le terminal va demander le nom de l'entité que vous voulez créer. Tapons dans l'invite :

```
> Task
```

Notez que si vous rentrez le nom d'une entité déjà existante, il modifiera cette entité. Maker vient de créer une entité Task dans :

src/Entity/Task.php et un repository dans: src/repository/TaskRepository.php(voyez plus loin dans ce cours ce qu'est un Repository).

Ensuite la console demande de créer une nouvelle propriété. Saisissons :

```
> createAt
```

Maker vous demande le type que vous voulez donner à votre propriété et par défaut suppose (grâce au nom) que vous voulez le type « *datetime\_immutable* ». Mais, si vous voulez choisir un autre type, vous pouvez découvrir la liste en rentrant « ? ».Validons le type proposé à savoir « *datetime\_immutable* ».

Maintenant le terminal demande si nous voulons rendre cette propriété « *nullable* » ou pas et par défaut, il propose non. En d'autres termes, est-ce que la date est obligatoire ou facultative ? Validons cette question car effectivement nous ne voulons pas qu'une tâche ne puisse pas avoir de date de création.

Ensuite il nous demande si nous voulons créer une autre propriété. Saisissons :

```
>title
```

Validons string, 255(c'est le nombre maximum de caractères et non null.

Ajoutons la propriété :

```
>content
```

Cette fois-ci tapons « *text* » et validons non null.

Enfin, ajoutons la propriété `isDone` qui nous permettra de savoir si une tâche est validée ou pas.

`>isDone`

C'est un Boolean non nul.

Pour finir la création de notre Entité, il suffit de taper `enter` quand l'invitation propose de créer une nouvelle propriété.

Vous devriez voir le message « *Success!* » ainsi que l'invitation de taper la ligne de commande :

`php bin/console make:migration`

Cette commande va vous permettre de créer une classe qui décrit les changements nécessaires afin de mettre à jour le schéma de la base de données. Cette classe de migration permettra à votre projet d'être tout le temps synchronisé avec votre base de données et voir si celle-ci est bien à jour avec vos dernières modifications / ajouts d'entité. Vous la retrouverez dans le dossier `migrations/` à la racine de votre projet. Une fois cette ligne de commande effectuée vous devriez avoir le message « *Success* » ainsi que le chemin du nouveau fichier de migration qui vient d'être créé.

La dernière étape est mettre à jour la base de données grâce à cette migration, avec la ligne de commande qui est proposée, à savoir :

`php bin/console doctrine:migrations:migrate`

Vous avez un message d'avertissement qui vous prévient que si vous validez le schéma de la base de données étant mis à jour, vous pourriez perdre des données. Votre BDD est au courant que sa dernière structure dépend de la dernière classe de migration. Veillez donc toujours à ce que la dernière classe de migration soit bien exécutée.

## C. Analyse d'une Entité

Revenons sur notre Entité qui a été créée, qui se trouve dans le dossier `/src/Entity/Task.php`.

Voici le fichier créé :

```

1 <?php
2
3 namespace App\Entity;
4
5 use App\Repository\TaskRepository;
6 use Doctrine\DBAL\Types\Types;
7 use Doctrine\ORM\Mapping as ORM;
8
9 #[ORM\Entity(repositoryClass: TaskRepository::class)]
10 class Task
11 {
12     #[ORM\Id]
13     #[ORM\GeneratedValue]
14     #[ORM\Column]
15     private ?int $id = null;
16
17     #[ORM\Column]
18     private ?\DateTimeImmutable $createdAt = null;
19
20     #[ORM\Column(length: 255)]
21     private ?string $title = null;
22
23     #[ORM\Column(type: Types::TEXT)]
24     private ?string $content = null;
25
26     #[ORM\Column]
27     private ?bool $isDone = null;

```

```
28
29 public function getId(): ?int
30 {
31     return $this->id;
32 }
33
34 public function getCreatedAt(): ?\DateTimeImmutable
35 {
36     return $this->createdAt;
37 }
38
39 public function setCreatedAt(\DateTimeImmutable $createdAt): self
40 {
41     $this->createdAt = $createdAt;
42
43     return $this;
44 }
45
46 public function getTitle(): ?string
47 {
48     return $this->title;
49 }
50
51 public function setTitle(string $title): self
52 {
53     $this->title = $title;
54
55     return $this;
56 }
57
58 public function getContent(): ?string
59 {
60     return $this->content;
61 }
62
63 public function setContent(string $content): self
64 {
65     $this->content = $content;
66
67     return $this;
68 }
69
70 public function isIsDone(): ?bool
71 {
72     return $this->isDone;
73 }
74
75 public function setIsDone(bool $isDone): self
76 {
77     $this->isDone = $isDone;
78
79     return $this;
80 }
81 }
```

Décortiquons ce code PHP :

C'est un fichier PHP et donc commence par <?PHP. L'entité est une classe qui se nomme Task. Elle a un Namespace comme toutes les classes avec Symfony et utilise les classes TaskRepository, ainsi que deux autres classes de Doctrine qui servent au mapping ainsi qu'au typage.

Vous avez certainement remarqué l'attribut ci-dessous qui est positionné avant que la classe ne commence :

```
#[ORM\Entity(repositoryClass: TaskRepository::class)]
```

PS : voir le bloc information sur Annotations et attributs.

Cet attribut permet de préciser à Doctrine le Repository qui est relié avec cette Entité. Il s'applique sur la classe. Voilà pourquoi il est situé juste avant celle-ci.

Ensuite nous avons en private les variables suivantes :

- \$id
- \$createAt
- \$title
- \$content
- \$isDone

Ici, les attributs précisent également les métadonnées nécessaires à Doctrine (voir bloc information : annotations et attributs dans la suite du cours). Vous avez certainement remarqué la présence de \$id que nous n'avons pas demandé lors de la création de notre Entité. En effet, Doctrine crée automatiquement et systématiquement une colonne ou champ id.

Ensuite, vous pouvez constater la mise en place des getter et des setter qui vous permettront d'avoir un accès aux variables dans votre code et de les modifier.

## D. Repository

Lorsque nous avons créé une Entité Task avec Maker-bundle, Doctrine a également créé un fichier TaskRepository.php dans le dossier Repository. L'ORM a créé dans ce fichier un objet qui a la responsabilité de récupérer une collection d'objets. En utilisant EntityManager et QueryBuilder le repository manipule les entités liées au repository et maîtrise les requêtes vers la base de données. Vous approfondissez le rôle du repository en étudiant la manipulation des Entités dans d'autres cours.

## Exercice : Quiz

[solution n°1 p.15]

### Question 1

Avec Symfony il n'est pas possible de se connecter avec une base de données de type sqlite.

- ☐ Vrai
- ☐ Faux

### Question 2

Une Entité est représentée par une classe sous Symfony.

- ☐ Vrai
- ☐ Faux

### Question 3

Une Entité représente une table de la base de données.

- ☐ Vrai
- ☐ Faux

#### Question 4

Si on crée une Entité « à la main », il n'est pas nécessaire de faire de migration.

- ☐ Vrai
- ☐ Faux

#### Question 5

Le fichier qu'il est nécessaire de configurer pour accéder à la base de données se nomme :

- ☐ .env
- ☐ .security
- ☐ .entity

### III. Les relations

Il y a deux notions à comprendre pour relier des entités entre elles. La notion de propriété et d'inverse et la notion d'unidirectionnalité et de bidirectionnalité.

#### A. Les différentes notions

##### Notion de propriété et d'inverse :

Cette notion est abstraite, mais essentielle à comprendre. En fait, il y a toujours une entité dite propriétaire et une dite inverse. L'entité propriétaire est celle qui contient la référence à l'autre entité. Illustrons cela :

Puisque nous avons créé une Entité Task, admettons que nous voulons que nos tâches soient reliées à un utilisateur dans notre application web. Vous pouvez par exemple créer une propriété `user_id` dans votre entité Task. C'est donc la tâche qui est propriétaire de la relation car elle contient la colonne de liaison `user_id`.

Notez tout de même qu'il n'y aura pas besoin de créer à la main `user_id`. C'est Doctrine qui vous aidera à entretenir cette relation.

##### Notion d'unidirectionnalité et de bidirectionnalité

Cela veut dire qu'une relation peut être à sens unique ou à double sens. Pour les relations uniques, il existera dans notre exemple de relation entre tâche et user, une méthode dans notre entité tâche `getUser()` qui ira simplement chercher l'utilisateur associé via sa clé étrangère. Dans le cadre d'une relation bidirectionnelle, il faut expliciter la relation aussi dans l'entité inverse pour que Doctrine, dans cet exemple, aille chercher de lui-même toutes les tâches d'un utilisateur. Il exécutera implicitement une requête pour cela lorsque vous appellerez depuis un User `getTasks()`.

#### Annotations et attributs

Une des nouveautés de PHP 8 est l'intégration des attributs « *attributes* » en anglais.

L'attribut est une fonctionnalité qui permet de définir les métadonnées dans le code ensuite celles-ci seront lues grâce à l'API de Reflection de PHP. En fait, un attribut c'est une annotation, mais en natif. Cela signifie qu'avec les annotations il y a besoin d'un parseur tiers pour extraire les métadonnées du PHP alors que les attributs sont interprétés en même temps que le reste du code. La syntaxe est aussi légèrement différente.



**Exemple****Annotations :**

```
/* *
 * @assert\All ({
 *     @Assert\NotBlank,
 *     @Assert\length(min=3)
 * })
/*
private string $title;
```

**Exemple****Attributs :**

```
#[Assert\NotBlank]
#[Assert\length(min: 3)
private string $title;
```

**B. Les attributs et annotations dans l'entité**

Pour spécifier les relations, il est nécessaire d'utiliser les annotations ou attributs. Si vous créez vos relations avec Maker, il les ajoutera directement sur la colonne ou champ qui utilisera cette relation.

**Exemple**

Dans une Entité Task, on crée une relation qui se nomme « *author* » en ManyToOne avec User. Ainsi une tâche peut avoir qu'un seul utilisateur, mais un utilisateur plusieurs tâches.

Voici ce qui a été ajouté dans l'entité Task :

```
#[ORM\ManyToOne(inversedBy: 'tasks')]
private ?User $author = null;
...

public function getAuthor(): ?User
{
    return $this->author;
}

public function setAuthor(?User $author): self
{
    $this->author = $author;

    return $this;
}
```

Et dans l'entité User :

```
@ORM(OneToMany(mappedBy = 'author', targetEntity = Task::class))
private Collection $tasks;

public function addTask(Task $task): self
{
    if (!$this->tasks->contains($task)) {
        $this->tasks->add($task);
        $task->setAuthor($this);
    }
    return $this;
}

public function removeTask(Task $task): self
{
    if ($this->tasks->removeElement($task)) {
        // set the owning side to null (unless already changed)
        if ($task->getAuthor() === $this) {
            $task->setAuthor(null);
        }
    }
    return $this;
}
```

- mappedBy : cet attribut spécifie le nom de la propriété sur la cible Entité qui est le côté propriétaire de cette relation.
- inversedBy : cet attribut désigne le champ de l'entité qui est le côté inverse de la relation.

Notez aussi les méthodes qui ont été ajoutées qui permettent une bidirectionnalité.

## C. Les types de relations :

### 1. OneToOne :

La relation 1..1 ou One to One est une relation classique. Comme son nom l'indique, elle définit une dépendance unique entre deux entités.

Imaginons que pour chaque utilisateur nous voulions lui associer une image. Il est également possible de le rajouter en attribut à l'entité User. Cette relation n'est pas toujours utilisée, mais dans certains cas de figure il peut être préférable de créer une entité à part, plutôt que d'ajouter un attribut à l'Entité existante. Nous pouvons donc créer une entité Image et la relier à notre entité User en OneToOne car nous voulons qu'un User puisse n'avoir qu'une seule image et qu'une image soit reliée à un seul utilisateur. Ainsi on aura une Entité Image avec un id, un nom et une relation.

Une relation OneToOne est le fait de n'avoir la relation entre 2 entités qu'une fois (soit les clés non duplicables en table).

### 2. ManyToOne

La relation Many To One ou n..1 permet à plusieurs Entité A d'établir une relation avec une Entité B. Imaginons que vous ayez un blog avec des articles et des commentaires. Il sera possible d'ajouter plusieurs commentaires sur un même article. Ainsi l'entité commentaire sera lié à l'entité article en relation ManyToOne. Car chaque commentaire est relié qu'à un seul article et qu'un article peut avoir plusieurs commentaires. En BDD, cela correspond pour ligne de la table commentaire, une foreign-key de la table article.

### 3. OneToMany

La relation One To Many ou 1..n permet d'établir à une entité A une relation avec plusieurs Entités B. En fait, on se positionne du côté inverse c'est-à-dire que dans l'exemple de blog mentionné dans la relation ManyToOne on effectue la relation à partir de l'entité article et non de commentaire. La foreign key en BDD sera donc TOUJOURS du côté many.

### 4. ManyToMany

La relation Many To Many ou n..n permet à plein d'objets d'être en relation avec plein d'autres. Cette relation a pour particularité de créer une table de jointure entre les deux entités liées. Cette table ne doit pas être modifiée, elle est implicitement créée par doctrine pour retenir les foreign keys de chacune des entités pour chaque relation entre elles.

Par exemple, si vous créez un blog qui contient des articles et des commentaires. Vous pouvez également ajouter des catégories. Vous pouvez aussi donner la possibilité qu'un article soit rattaché à deux catégories. Nous rentrons ici dans la relation ManyToMany.

Précisons l'exemple, nous pourrions créer des catégories d'article suivant :

- Sportif
- Informatique
- Jeux vidéo
- Politique
- Économique

Nous pouvons ajouter autant de catégories que l'on souhaite. Il suffit de créer un objet catégorie à partir de l'entité Catégorie. Mais imaginons qu'un article se rapporte sur un nouveau device de Google ou Apple. Il sera possible de classer cet article dans la catégorie économique et dans la partie Informatique. On parle de relation entre les entités Article et Catégorie de ManyToMany car un article peut avoir plusieurs catégories et qu'une catégorie peut avoir plusieurs articles.

## Exercice : Quiz

[solution n°2 p.15]

### Question 1

Dans le cas de relation unidirectionnelle entre Entités, il y a toujours une Entité propriétaire.

- ☐ Vrai
- ☐ Faux

### Question 2

Quel type de relation est décrit par la règle suivante ? Pour une instance de l'entité A, il existe zéro, une ou plusieurs instances de l'entité B ; et pour une instance de l'entité B, il existe zéro, une ou plusieurs instances de l'entité A.

- ☐ 1..n
- ☐ n..1
- ☐ n..n

### Question 3

Une annotation c'est pareil qu'un attribut.

- ☐ Vrai
- ☐ Faux

### Question 4

Dans le cas de relation bidirectionnelle, il y a une annotation ou un attribut sur chaque entité qui explicite la relation.

- ☐ Vrai
- ☐ Faux

### Question 5

Vous avez une Entité User et une Entité Image vous voulez les relier entre elles si c'est Image qui est propriétaire qu'elle est la bonne réponse :

- ☐ L'entité User aura la propriété Image\_Id
- ☐ L'Entité Image aura la propriété User\_id
- ☐ Ni l'une ni l'autre il n'y a pas de propriété supplémentaire

## V. Essentiel

Dans ce cours nous avons vu un élément important dans Symfony à savoir l'Entité. Les Entités sont des objets qui seront gérés par l'ORM Doctrine. Ces Objets ont des propriétés ou attributs ainsi que des méthodes. Chaque Entité a un Repository associé afin de permettre à Doctrine de persister des données et de personnaliser des requêtes.

En fait les Entités sont un élément du module qu'on appelle Modèle dans le MVC. Le modèle ou logique métier a pour rôle de travailler afin de fournir les informations de la base de données (dans les exemples que nous avons vus) afin de permettre au Contrôleur de faire passer ses informations à la Vue.

Les Entités peuvent être reliées entre elles grâce à ce qu'on appelle des relations, nous avons décrit les différents types de relations utilisés à savoir OneToOne, OneToMany, ManyToOne et ManyToMany, ce qui permet de créer des bases de données relationnelles très simple ou très complexe. Alors, n'oubliez jamais de créer vos diagrammes UML avant de commencer à développer vos entités. Votre diagramme de classes peut véritablement vous aider à ne pas vous tromper dans le positionnement de vos relations.

Maintenant, il vous reste à maîtriser la manipulation des Entités avec Doctrine, c'est un chapitre à part entière que nous vous encourageons à approfondir.

## VI. Auto-évaluation

### A. Exercice

Vous êtes un développeur full-stack expérimenté dans une agence web où vous gérer de nombreuses applications web. On vous demande de créer un blog pour un client avec le framework Symfony 5.4.

#### Question 1

[solution n°3 p.17]

Donnez un exemple d'entité que vous pourriez utiliser avec leurs propriétés et expliquez quels outils vous allez utiliser.

#### Question 2

[solution n°4 p.17]

Pour ce même projet et avec ces entités, quelles relations allez-vous utiliser. Expliquer les relations et les champs qui seront créés à l'aide de Maker.

### B. Test

#### Exercice 1 : Quiz

[solution n°5 p.17]

##### Question 1

J'utilise Maker pour construire mes entités. Si je fais une relation entre une Entité User qui est Propriétaire et une Entité Adresse.

- ☐ Je crée une propriété Adresse\_id dans l'entité User
- ☐ Je crée une propriété User\_id dans l'entité Adresse
- ☐ Je crée une relation OneToOne dans l'entité User

##### Question 2

Quelle variable d'environnement faut-il configurer pour accéder à la base de données ?

- ☐ DATABASE\_URL
- ☐ BASE\_DE\_DONNEES
- ☐ DATA\_SQL\_URL

Question 3

Les annotations et les attributs sont :

- ☐ Des métadonnées qui font les relations
- ☐ Des relations vers la base de données
- ☐ Des métadonnées que Doctrine va utiliser

Question 4

Maker permet de créer les guetteurs et les setteurs.

- ☐ Vrai
- ☐ Faux

Question 5

Cette écriture «  $n..n$  » représente-t-elle la relation ManyToMany ?

- ☐ Vrai
- ☐ Faux

## Solutions des exercices




**Exercice p. 7 Solution n°1****Question 1**

Avec Symfony il n'est pas possible de se connecter avec une base de données de type sqlite.

☐ Vrai

☒ Faux


 Symfony propose dans le fichier .env, une ligne de code commenté sqlite préconfigurée prête à être paramétrée.

**Question 2**

Une Entité est représentée par une classe sous Symfony.

☒ Vrai

☐ Faux


 Une classe est une structure d'objets qui permet de créer des objets. La classe qui représente une entité permettra de créer des objets qui seront stockés dans la base de données.

**Question 3**

Une Entité représente une table de la base de données.

☒ Vrai

☐ Faux


 C'est également vrai. Par exemple, une Entité User aura une table User dans la BDD qui contiendra tous les utilisateurs créés.

**Question 4**

Si on crée une Entité « à la main », il n'est pas nécessaire de faire de migration.

☐ Vrai

☒ Faux

 Faux, l'objectif de Symfony c'est que Doctrine gère votre base de données. Il vous sera nécessaire de taper la ligne de commande \$php bin/console Doctrine:migration pour prévenir Doctrine des modifications qui sont nécessaires de prendre en compte.


**Question 5**

Le fichier qu'il est nécessaire de configurer pour accéder à la base de données se nomme :

☒ .env

☐ .security


☐ .entity

 Le fichier .env que l'on nomme dotenv est un fichier qui contient des variables d'environnement telles que DATABASE\_URL qui permet de configurer la connexion à la base de données.

**Exercice p. 11 Solution n°2**


### Question 1

Dans le cas de relation unidirectionnelle entre Entités, il y a toujours une Entité propriétaire.

- ☒ Vrai
- ☐ Faux
-  Effectivement, il y a une entité dite propriétaire et l'autre dite inverse.


### Question 2

Quel type de relation est décrit par la règle suivante ? Pour une instance de l'entité A, il existe zéro, une ou plusieurs instances de l'entité B ; et pour une instance de l'entité B, il existe zéro, une ou plusieurs instances de l'entité A.

- ☐ 1..n
- ☐ n..1
- ☒ n..n
-  C'est la relation ManyToMany qui permet à un objet d'avoir de 0 à plusieurs relations avec l'objet B et à l'objet B d'avoir de 0 à plusieurs relations avec l'objet A.


### Question 3

Une annotation c'est pareil qu'un attribut.

- ☐ Vrai
- ☒ Faux
-  Les attributs peuvent remplacer les annotations car ce sont aussi des métadonnées qui sont depuis PHP 8 natifs. Les attributs ont une syntaxe propre.


### Question 4

Dans le cas de relation bidirectionnelle, il y a une annotation ou un attribut sur chaque entité qui explicite la relation.

- ☒ Vrai
- ☐ Faux
-  L'annotation ou l'attribut explicite la relation à chaque entité. Dans le cas d'une relation unidirectionnelle, c'est simplement l'entité propriétaire qui utilisera cette métadonnée.

### Question 5

Vous avez une Entité User et une Entité Image vous voulez les relier entre elles si c'est Image qui est propriétaire qu'elle est la bonne réponse :

- ☐ L'entité User aura la propriété Image\_Id
- ☒ L'Entité Image aura la propriété User\_id
- ☐ Ni l'une ni l'autre il n'y a pas de propriété supplémentaire
-  C'est toujours l'entité propriétaire qui contient la propriété donc dans notre exemple c'est Image qui contient User\_id.



**p. 12 Solution n°3**


Avant même de commencer à programmer il vous faudra créer vos diagrammes UML et pour vous aider à la création de vos entités avec leurs propriétés le diagramme de classes vous sera particulièrement utile. Vous devriez créer au minimum trois entités que nous pourrions nommer User, Comment, Article. Pour la création de ceux-ci l'utilisation de maker vous sera appréciable, vous n'aurez pas besoin de créer un champ Id, maker le fera par défaut. Voici les propriétés que vous pourriez donner: User aura email (string 255), password(string 255), firstname(string 255), lastname(string 255) et admin(boolean). Article possédera title (string 255), image (string 255), content (text), date (datetime) et chapo (string 255). Comment possédera date-commentaire (datetime), commentaire (text), checked (boolean).

**p. 12 Solution n°4**

Vous pourriez ajouter les relations suivantes avec Maker. Taper la ligne de commande \$php bin/console make:entity et choisissez Comment, et User comme nom de propriété, dans le choix du type : entrer relation et choisissez User comme entité de relation avec la relation ManyToOne car nous souhaitons que plusieurs commentaire soit associé à un seul utilisateur, puis validez non nullable et n'ajoutons pas d'autre propriété vers User. Puisque nous sommes toujours dans Comment ajoutons aussi la propriété Article avec la relation ManyToOne, non nullable et sans autre propriété. Une fois tout validé, ajoutons aussi avec la même démarche à l'entité Article la propriété User en relation ManyToOne vers User, non nullable avec aucune autre propriété. Une fois fini il faudra dans le terminal ajouter les commandes suivantes : \$php bin/console make:migration et php bin/console doctrine:migrations:migrate. Voilà nous avons nos relations entre les entités.


**Exercice p. 12 Solution n°5****Question 1**

J'utilise Maker pour construire mes entités. Si je fais une relation entre une Entité User qui est Propriétaire et une Entité Adresse.

- ☐ Je crée une propriété Adresse\_id dans l'entité User
- ☐ Je crée une propriété User\_id dans l'entité Adresse
- ☒ Je crée une relation OneToOne dans l'entité User
-  Avec Maker en créant une relation, il créera lui-même la propriété nécessaire, il suffit de déclarer la relation dans l'entité propriétaire.

**Question 2**


Quelle variable d'environnement faut-il configurer pour accéder à la base de données ?

- ☒ DATABASE\_URL
- ☐ BASE\_DE\_DONNEES
- ☐ DATA\_SQL\_URL
-  Avec symfony il suffit de décommenter la variable d'environnement DATABASE\_URL qui convient à la bonne base de données et de configurer les paramètres personnels.

**Question 3**

---


Les annotations et les attributs sont :

- ☐ Des métadonnées qui font les relations
- ☐ Des relations vers la base de données
- ☒ Des métadonnées que Doctrine va utiliser
-  Les métadonnées sont lues par l'API de Réflexion de PHP, elles peuvent concerner les relations, mais pas uniquement.

**Question 4**

---


Maker permet de créer les guetteurs et les setteurs.

- ☒ Vrai
- ☐ Faux
-  Pour toutes les propriétés, Maker ajoutera des guetteurs et des setters, mais rien n'empêche de les modifier.

**Question 5**

---

Cette écriture « *n..n* » représente-t-elle la relation ManyToMany ?

- ☒ Vrai
- ☐ Faux
-  Symbole utilisé dans les diagrammes de classes représente effectivement ManyToMany, en français plusieurs vers plusieurs.