Web - TP Symfony n^o 5
G. Subrenat, R. Zrour

Licence Informatique L3 - 2020-2021
UFR SFA - Université de Poitiers

Doctrine

Table des matières

1 Résumé

Dans ce TP nous verrons comment interagir avec la base de données grâce au framework Doctrine :
- connexion au serveur
- modélisations des tables
- modélisation des relations entre les tables
- manipulation des données

2 Les entités

2.1 ORM, requêtes et objets

2.1.1 Un peu de théorie

Point de cours

Un *ORM* (*Object-Relation Mapper* ou *Object-Relational Mapping*) est un framework qui permet une interface objet (au sens POO) entre le programmeur et la base de données.

L'*ORM* par défaut de Symfony est Doctrine.

Un premier intérêt est de s'abstraire du SGBD physique (Oracle, PostgresSQL, MySQL, MariaDB, ...) : il n'est plus utile de connaître les différences entre les divers SGBD (noms de type, variations syntaxiques, ...).

Et même on ne manipule plus (enfin presque plus) les tables et on ne fait plus de requêtes SQL : on ne manipule que des objets avec leurs méthodes.

2.1.2 Un peu de théorie ... plus pratique

Point de cours

Pour insérer un film dans la table, en SQL le code ressemblera à :

```
INSERT INTO films (id, nom, annee, prix) VALUES (NULL, 'Terminator', '1984', '24.99');
```

Avec Doctrine :

```
$film = new Film('Terminator', 1984, 24.99);
$orm->save($film);
```

Pour récupérer l'année d'un film, en SQL le code ressemblera à :

```
SELECT annee FROM films WHERE id = 4;
```

Avec Doctrine :

```
$film->getAnnee();
```

Et plus intéressant, pour récupérer toutes les critiques (qui sont dans une autre table) d'un film, en SQL le code ressemblera à :

```
SELECT F.nom, C.* FROM films AS F, critiques AS C WHERE F.id = 1 AND F.id = C.id_film;
```

Avec Doctrine :

```
$film->getCritiques();
```

Avec un *ORM*, un objet représentant un enregistrement se nomme une *entité* (*entity* en anglais).

Point de cours

Une entité représentant une table est une classe avec généralement un attribut par colonne, ainsi que les *setters* et les *getters*. Il est possible d'ajouter d'autres méthodes pour les opérations de plus haut niveau (requêtes multi-tables par exemple) comme la méthode *getCritiques()* vue ci-dessus.

Mais il est nécessaire que l'ORM Doctrine connaissent l'architecture de la base de données :

- quelle est la clé primaire d'une table,
- quel est le type d'une colonne,
- quel champ est une clé étrangère et vers quelle table,
- quelle est la cardinalité d'une relation, et le cas échéant avec quelle table de jointure,
- ...

Comme l'ORM ne peut pas le deviner, le programmeur doit lui dire sous forme de commentaires (on parle d'annotations [a]).
Il n'est pas commun que des commentaires contiennent une information nécessaire à l'exécution du programme [b], aussi la syntaxe sera particulière. Voici à quoi ressemble une classe *entité* :

Film.php

```php
<?php

namespace Lic\SandboxBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table(name="film")
 * @ORM\Entity(repositoryClass="Lic\SandboxBundle\Repository\FilmRepository")
 */
class Film
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(name="nom", type="string", length=255)
     */
    private $nom;

    /**
     * @ORM\Column(name="annee", type="integer")
     */
    private $annee;


    // les getters
    // et les setters
}
```

Notez les commentaires commençant par `/**` ainsi que la notation `@ORM`.

Symfony fournit un script pour générer les classes entité.

a. Les annotations sont le format recommandé par Symfony pour décrire les relations entre les entités. Mais pour Doctrine nous aurions pu choisir un autre format comme *YAML*, *PHP* ou *XML*.
b. Mais nous l'avons déjà vu pour décrire les routes.

2.1.3 De la pratique pas théorique

Pour utiliser Doctrine, il faut installer le composant *orm* mais normalement cele a déjà été fait dans les premiers TP.
Sous Windows :

```
serveur$ composer req orm
```

Sous Linux (par défaut) :

```
serveur$ composer.phar req orm
```

Les classes entités sont syntaxiquement pénibles et de fait il est préférable de les générer en mode console. Si ce n'est pas déjà fait, il faut installer le composant *maker* :

```
serveur$ composer req maker
```

Rappel pour lancer un script de configuration du site Symfony :

```
serveur$ php bin/console <commande>
```

ou

```
serveur$ symfony console <commande>
```

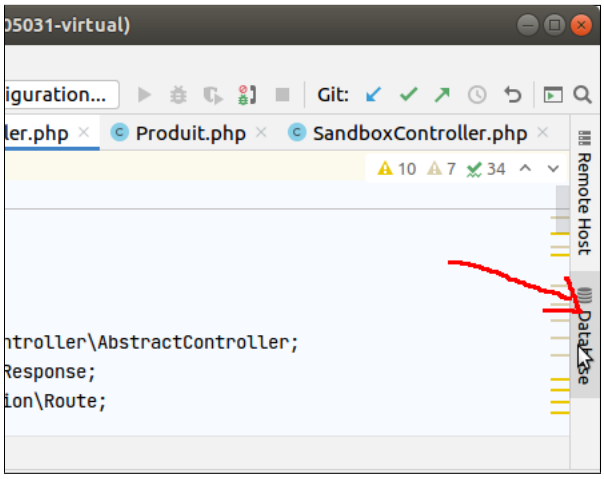
Notamment le composant *maker* laisse entrevoir 3 possibilités intéressantes. La commande suivante permet de filtrer toutes les commandes autorisées par la console :

```
serveur$ symfony console | grep -i doctrine
```

Et voici les commandes *a priori* intéressantes :

`doctrine:database:create`	→	Creates the configured database
`doctrine:migrations:diff`	→	Generate a migration by comparing your current database to your mapping information
`doctrine:migrations:latest`	→	Outputs the latest version
`doctrine:migrations:list`	→	Display a list of all available migrations and their status
`doctrine:migrations:migrate`	→	Execute a migration to ... the latest available version
`doctrine:schema:create`	→	Executes (or dumps) the SQL needed to generate the database schema
`doctrine:schema:update`	→	Executes (or dumps) the SQL needed to update the database schema to match the current mapping metadata
`doctrine:schema:validate`	→	Validate the mapping files
`make:entity`	→	Creates or updates a Doctrine entity class
`make:fixtures`	→	Creates a new class to load Doctrine fixtures

PhpStorm comporte un composant pour manipuler la base de données sans passer un programme externe (comme *phpmyadmin* ou *adminer*). Sur la barre verticale à droite de l'IDE, il y a un bouton "Database". Et en l'occurrence nous en aurons besoin.



2.2 Création et connexion à la base

2.2.1 Création et connexion

La connexion à la base de données se paramètre dans le fichier *.env* en renseignant la variable *DATABASE_URL*.

Comme indiqué précédemment, Doctrine permet de s'interfacer avec plusieurs SGBD. Dans le fichier *.env* il y a 3 exemples de syntaxe pour les SGBD *PostgresSQL*, *MySQL* et *SQLite*.

Dans ce cours nous choisissons *SQLite*. *SQLite* permet de stocker la base de données dans un unique fichier et de ne pas nécessiter la présence d'un serveur de base de données (Wamp ou Xamp en intègrent un nativement).
En outre ce fichier sera stocké dans l'arborescence du site ce qui vous facilitera la tâche lorsque vous ferez une archive d'un futur projet.
En revanche *SQLite* se prête très mal à des accès concurrents (chutes de performances). Mais dans notre cas nous voulons juste nous faciliter l'accès à la base de données.

Dans notre cas, nous allons créer un répertoire dédié, à la racine du site, pour stocker notre base de données : *database*. Et le fichier de la base s'appellera *mybase.db*.
Le fichier *.env* contient alors :

.env

```
28 DATABASE_URL="sqlite:///%kernel.project_dir%/database/mybase.db"
```

N'oubliez pas de commenter les autres affectations de *DATABASE_URL*.

Note : pour MySql la configuration est un peu plus compliquée car il faut fournir les identifiants de connexion.

Le mode *console* fournit un script pour créer la base de données : `doctrine:database:create`

Lancez les commandes (sur le serveur) :

```
serveur$ mkdir database
serveur$ php bin/console doctrine:database:create
Created database /home/subrenat/public_html/site_tp/database/mybase.db for connection named default
serveur$ ls -l database/mybase.db
-rw-r--r-- 1 subrenat subrenat 0 mars 10 17:16 database/mybase.db
```

Et vous devez avoir un message vous indiquant la réussite de l'opération. On remarque au passage que le fichier généré est vide.

Attention, le fichier *.sql* fourni n'est là que pour visualiser ce que nous souhaitons que Doctrine fasse. Mais il ne sert absolument pas dans le code du site[1] (notamment, il n'a rien à voir avec le fichier *database/mybase.sql*).

Comme nous allons laisser Doctrine créer les tables de la base de données, la démarche est la suivante :

- on créer une classe entité
- on demande à Symfony de créer la table correspondante dans la base de données (SQLite pour nous).

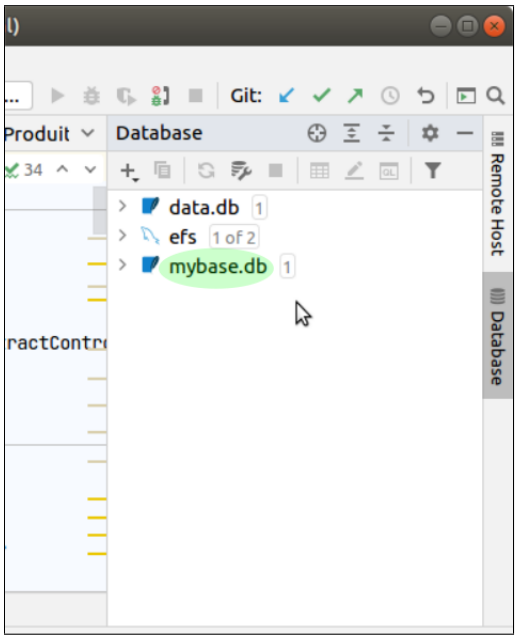
2.2.2 PhpStorm

Il faut cliquer sur le bouton "Database" mentionné ci-dessus pour faire apparaître l'interface.

Ensuite il faut initialiser la connexion entre PhpStorm et notre base SQLite :

- on clique sur *+/Data Source from Path*
- on choisit le fichier nouvellement créé
- on choisit le driver *SQLite*
- dans la nouvelle fenêtre il n'y a rien à changer ; on peut cliquer sur "Test Connection" pour une vérification

1. une version imprimée aurait pu tout à fait vous être fournie



2.3 Création d'entités

Point de cours

Il ne faut plus penser "SQL" mais penser "objet".

Prenons l'exemple d'une table *habitants* et d'une table *villes*, avec la relation *1..n* entre *habitants* et *villes*. Donc la table *habitants* possède une clé étrangère vers la table *villes*.

Le programmeur qui utilise Doctrine écrit l'entité *Habitant* représantant la table *habitants*.

La classe *Habitant* :

- N'a PAS un champ entier qui stocke la clé étrangère
- a un champ de type *Ville* qui est une référence vers un objet qui représente une ville.

2.3.1 Générer la classe

Le mode *console* fournit un script pour générer les classes entité.

Un autre script est également capable d'interagir avec le SGBD pour créer physiquement les tables décrites par nos classes entité.

Dans cet exercice, nous allons générer la classe entité qui représente la table *sb_films* qui contient les champs :

- *id* : clé primaire, entier
- *nom* : varchar
- *annee* : entier
- *enstock* : booléen
- *prix* : float
- *quantite* : entier

Et la classe se nommera *Film*.

Voici l'exécution du script qui est détaillée pas à pas. Pour les besoins de l'exercice suivez le nommage à la lettre (et au chiffre).
Le script supporte la complétion (touches haut et bas, tabulation).

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

 Class name of the entity to create or update (e.g. GrumpyChef):
 >
```

Il faut préciser le nom sans préciser d'extension. Et le nom de l'entité sera *Film*.
Le fait que le nom de la table possède un préfix (*sb_*) ne rentre pas en ligne de compte ici : nous ne voulons pas ce préfixe dans le nom de la classe.

L'entité sera créée dans le répertoire *src/Entity* du site. Comme nous n'avons pas précisé de chemin, la classe sera directement dans ce répertoire ; mais il est possible de créer des sous-répertoires pour ranger les classes.

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

 Class name of the entity to create or update (e.g. GrumpyChef):
 > Film

 created: src/Entity/Film.php
 created: src/Repository/FilmRepository.php

 Entity generated! Now let's add some fields!
 You can always add more fields later manually or by re-running this command.

 New property name (press <return> to stop adding fields):
 >
```

Deux fichiers sont déjà créés à ce stade bien que le script ne soit pas terminé. On peut regarder le fichier *Film.php* pour s'apercevoir que la clé primaire est déjà prise en compte.

À partir de ce moment, il est nécessaire de définir les colonnes de la table une à une :

- *nom* : il s'agit du nom de l'attribut de la classe ; c'est généralement le même que le nom du champ.
- *type* : il s'agit du type du champ. Il faut faire la correspondance entre les type proposés par Doctrine et les types du SGBD (par exemple *string* vs. *varchar*).
- *longueur* : dans le cas des chaînes de caractères.
- *nullable* : le champ peut-il prendre la valeur *null*.

Notez que la clé primaire est automatiquement gérée et se nommera *id*.
Que se passe-t-il si votre clé primaire ne se nomme pas ainsi ? Et bien il suffira de légèrement modifier le code généré.

Détaillons la création du champ *nom*. Pensez à bien respecter les saisies proposées, même si certaines vous semblent bizarres.

Le nom est *nom*.

```
 New property name (press <return> to stop adding fields):
 > nom

 Field type (enter ? to see all types) [string]:
 >
```

Le type est *varchar*, c'est à dire *string* pour Doctrine, il suffit de valider.

```
 Field type (enter ? to see all types) [string]:
 >

 Field length [255]:
 >
```

Nous choisissons une longueur de 200.

```
 Field length [255]:
 > 200

 Can this field be null in the database (nullable) (yes/no) [no]:
 >
```

Le champ ne peut pas être *null*, il suffit de valider.

```
Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Film.php

Add another property? Enter the property name (or press <return> to stop adding
fields):
>
```

Il reste à faire de même avec les quatre autres champs (en pensant que *quantite* peut être *null*) :

```
Add another property? Enter the property name (or press <return> to stop adding
fields):
> annee

Field type (enter ? to see all types) [string]:
> integer

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Film.php

Add another property? Enter the property name (or press <return> to stop adding
fields):
>
```

```
Add another property? Enter the property name (or press <return> to stop adding
fields):
> enstock

Field type (enter ? to see all types) [string]:
> boolean

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Film.php

Add another property? Enter the property name (or press <return> to stop adding
fields):
>
```

```
Add another property? Enter the property name (or press <return> to stop adding
fields):
> prix

Field type (enter ? to see all types) [string]:
> float

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Film.php

Add another property? Enter the property name (or press <return> to stop adding
fields):
>
```

```
Add another property? Enter the property name (or press <return> to stop adding
fields):
> quantite

Field type (enter ? to see all types) [string]:
> integer

Can this field be null in the database (nullable) (yes/no) [no]:
> yes

updated: src/Entity/Film.php

Add another property? Enter the property name (or press <return> to stop adding
fields):
>
```

Il reste à valider une dernière fois :

```
Add another property? Enter the property name (or press <return> to stop adding
fields):
>


 Success!

Next: When you're ready, create a migration with php bin/console make:migration
```

Le fichier *Entity/Film.php* a été créé avec tous les champs annotés, ainsi que les *getters* et *setters*.
Notez également la création du fichier *Repository/FilmRepository.php*.

Et n'oubliez pas d'effectuer une synchronisation avec le serveur si nécessaire.

2.3.2 Compléter la classe

Une instance d'une classe entité représente une ligne de la table correspondante, un film par exemple.

Aussi dans un premier temps il n'est pas utile de rajouter des méthodes à cette classe. Mais pour une utilisation avancée des entités, la possibilité d'ajouter des méthodes s'avère utile.

Si des membres doivent être initialisés par défaut, le constructeur est fait pour cela.

Film.php

```php
    /**
     * Film constructor.
     */
    public function __construct()
    {
        $this->enstock = true;
        $this->quantite = null;
    }
```

Il ne faut plus désormais raisonner table, requête, jointure, ... mais constructeur, accesseurs, ...

2.3.3 Notations

Voici quelqes explications succinctes sur les diverses annotations. N'hésitez pas à aller voir la documentation si vous voulez plus d'informations.

Doctrine est une bibliothèque indépendante de Symfony. Toutes les annotations, dans les classes de Symfony, sont préfixées par `ORM\`. Ce préfixe est propre à Symfony (car nous sommes dans le namespace *ORM*), et ne se trouve pas dans la documentation de Doctrine.

Ainsi lorsque dans la documentation officielle il y a l'annotation `/** @Entity */`, il faut la traduire par `/** @ORM\Entity */`.

L'annotation *Entity* se place avant la définition de la classe. Elle possède deux paramètres : *repositoryClass* et *readOnly* (*false* par défaut). Cette annotation est automatiquement mise par le script.

L'annotation *Table* (i.e. `@ORM\Table`) se place également avant la définition de la classe. Elle possède l'attribut *name* qui précise le nom physique de la table dans le SGBD.
Le nom de la table, si l'annotation *Table* n'est pas renseignée, est déduit par Symfony. À titre personnel, je préfère sytématiquement préciser le nom de la table. En l'occurrence nous n'avons pas le choix car la table a un nom préfixé par *sb_* que Symfony ne peut pas deviner.

Film.php

```php
/**
 * @ORM\Table (name="sb_films")
 * @ORM\Entity(repositoryClass=FilmRepository::class)
 */
class Film
{
```

L'annotation *Column* se place avant un attribut de classe pour préciser le lien avec le champ de la table. Voici les paramètres :

- *type* : le type du champ. Consultez la documentation ou les tutoriels pour les correspondances entre Doctrine, SQL et PHP. Il est préférable de systématiquement renseigner ce paramètre.
- *name* : le nom du champ dans la table. Très souvent le nom du membre dans la classe est le même que le nom du champ dans la table ; l'habitude est alors de ne pas préciser l'attribut *name* cf. ci-dessous).
- *length* : uniquement pour le type *string*, il définit la longueur de la chaîne. Pour une chaîne, il est préférable de systématiquement renseigner ce paramètre.
- *unique* : si le champ est unique (*false* par défaut).
- *nullable* : si le champ peut être *NULL* (*false* par défaut).
- *precision* : pour le type *decimal*.
- *scale* : pour le type *decimal*.

Le membre *$id*, qui représente la clé primaire, est un peu particulier :

- il est généré automatiquement par le script
- il est préférable [2] de lui laisser ce nom
- si le nom du champ dans la table diffère de *id*, on passe alors par l'attribut *name* de l'annotation *Column*.

Si les conventions de nommage dans SQL et PHP diffèrent, alors l'attribut *name* de l'annotation *Column* devient utile. Par exemple un champ dans la table s'appelle *diametre_roue*, et dans l'entité il s'appellera *diametreRoue*.
Dans ce cas il conseillé, lors de l'exécution du script `make:entity`, d'indiquer les noms des membres de la classe plutôt que les noms des champs de la table.

Film.php

```
26      /**
27       * @ORM\Column(type="integer")
28       * l'attribut 'name' n'est pas précisé, le nom du champ est supposé être "annee"
29       */
30      private $annee;
31
32      /**
33       * @ORM\Column(name="enstock", type="boolean")
34       * l'attribut 'name' est inutile ici
35       */
36      private $enstock;
```

2.3.4 Vérification

La commande suivante permet de vérifier vos classes entités :

```
serveur$ symfony console doctrine:schema:validate --skip-sync
```

Et la commande suivante vérifie en plus que les dernières modifications des entités ont bien été injectées dans la base de données :

```
serveur$ php bin/console doctrine:schema:validate
```

Précisons bien ici l'enchainement des opérations :

- une classe entité (PHP) correspond à une table dans la base (SQL)
- on crée la classe avec le script
- on modifie le code de la classe (cf. ci-dessous)

Jusque là la table dans la base n'est pas créée ou mise à jour. C'est normal car c'est une opération très dangereuse et elle doit être faite explicitement pas le programmeur.
C'est pour cela qu'il y a une "erreur" de synchronisation : l'état de l'entité n'est pas le même que celui de la table dans la base (la table n'existe pas encore).

Vérifiez et si nécessaire corrigez votre classe *Film*.

2. voire obligatoire ?

2.3.5 Compléter la classe (bis)

Dans le script *.sql* fourni on remarque que le champ *enstock* a comme valeur par défaut la valeur 1 (i.e. *true*).
Il est possible de le préciser dans les annotations (notez que la valeur par défaut est *true* est non 1) :

```
@ORM\Column(name="enstock", type="boolean", options={"default"=true})
```

En l'occurrence il y a redondance avec le code du constructeur.

De même le champ *annee* est associé à un commentaire que l'on peut également préciser dans les annotations. Pour le membre *$annee* cela donne :

```
@ORM\Column(type="integer", options={"comment"="année de sortie"})
```

Faites les changements et vérifiez la syntaxe.

2.3.6 Compléter la classe (ter)

Si l'on doit rajouter des membres dans l'entiré (i.e. des champs dans la table), il suffit de relancer le script `make:entity`. Ce dernier gère le fait que l'entité existe déjà.

Pour retirer un memblre, il faut le faire à la main.

Mais en l'occurrence il n'y a rien à faire comme manipulation dans cette section : on indique juste la démarche pour un changement ultérieur.

2.4 Synchroniser Doctrine et le SGBD

Il s'agit maintenant de répercuter l'état des entités vers la base de données (ajout/suppression de tables, modification de tables).

On rappelle que modifier la classe entité n'a aucune répercussion automatique sur le fichier de la base de données : lorsqu'on est satisfait de la classe, alors on demande explicitement d'effectuer les changements dans la base.

Il existe plusieurs scripts pour effectuer ces opérations.

Nous allons utiliser les deux scripts suivants :

- `doctrine:migrations:diff`
- `doctrine:migrations:migrate`

Ces scripts permettent de garder un historique des différentes versions de la base de données et de revenir en arrière dans les versions.

La commande :

```
serveur$ php bin/console doctrine:migrations:diff
```

génère le code (mais ne l'exécute pas) pour mettre à jour la base de données par rapport à la dernière migration.
Le fichier *migrations/Version<datetime>* [3] est généré et contient la méthode *up* pour effectuer la mise à jour, et la méthode *down* pour défaire la mise à jour.

Et pour appliquer la méthode *up* de la dernière mise à jour, la commande est :

```
serveur$ symfony console doctrine:migrations:migrate
```

Avec un avertissement sur les risques de pertes de données.

Attention : les commandes *diff* et *migrate* vont ensemble ; il ne faut pas lancer plusieurs fois de suite la commande *diff*.

Chaque fois que l'on modifie des entités, il faut relancer ces deux commandes pour une mise à jour de la base de données.

3. *datetime* sera *20210312170437* si on lance le script le 12 mars 2021 à 17h04m37s

Note 1 : on remarque qu'il ne faut pas faire deux `doctrine:migrations:diff` dans la même seconde.

Note 2 : ces scripts injectent une table propre dans la base de données pour gérer les versions.

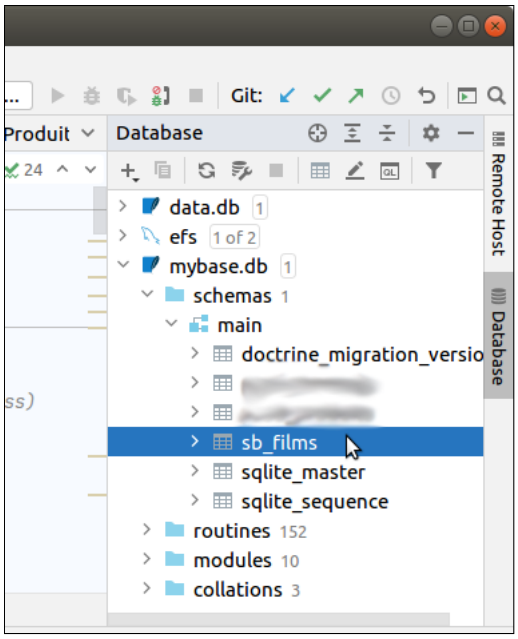
Note 3 : il existe un autre script (`doctrine:schema:update`) qui permet les mises à jour, mais sans versionnement [4].

Note 4 : en case de catastrophe :

- on peut supprimer le fichier contenant la base de données
- on peut supprimer tous les fichiers du répertoire *migrations*
- recréer la base de données
- refaire une migration
- mais on perd tout l'historique
- et surtout ON PERD TOUTES LES DONNÉES
- seule la structure des tables est concservée
- bref à éviter absolument

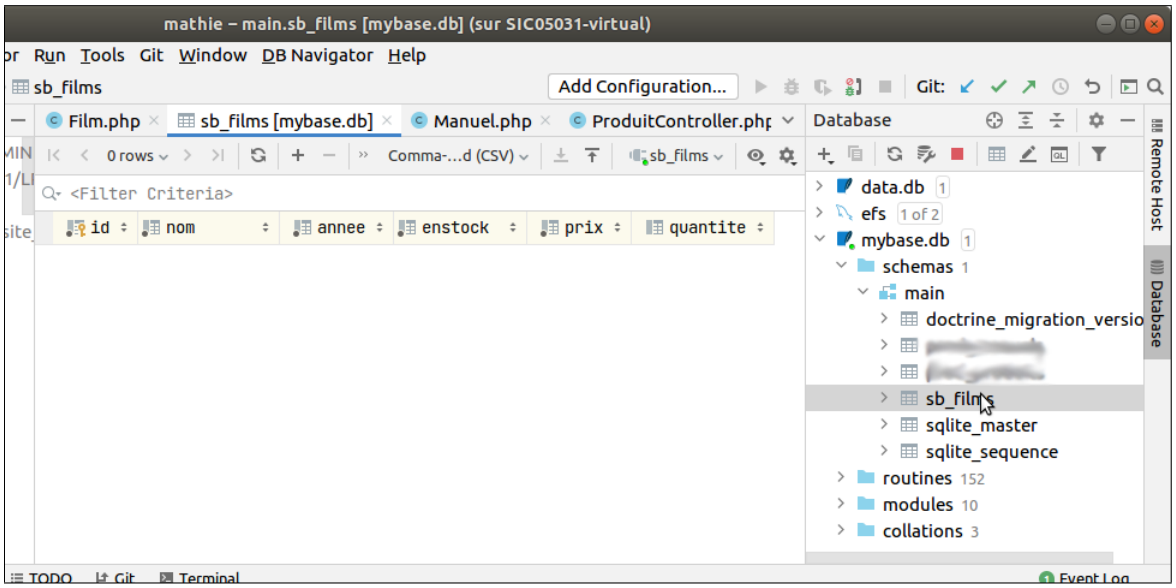
Faites ces commandes et avec le composant de PhpStorm vérifiez que la base de données a été mise à jour :

- déplier *mybase.db*
- déplier *schemas*
- déplier *main*
- la table *sb_films* apparaît



Un double-click sur le nom de la table permet d'afficher son contenu (vide pour l'instant).

4. à déconseiller



3 Manipuler les entités

3.1 Contrôleur de test

Dans le cette section on demande uniquement un squelette : les vues seront quasi-vides, et les actions ne font qu'appeler la vue. La manipulation de la base de données sera faite dans un second temps et expliquée des les sections suivantes.

Dans le controller *Sandbox*, créez les actions (avec les routes et vues associées) :

- *list* : pas de paramètre, le but est d'afficher tous les films de manière condensée. (le nom interne de la route sera `sandbox_doctrine_list`)
- *view* : un paramètre qui est l'id du film, le but est d'afficher en détails les informations sur le film. (le nom interne de la route sera `sandbox_doctrine_view`)
- *delete* : un paramètre qui est l'id du film, le but est de supprimer le film avec toutes ses critiques. (le nom interne de la route sera `sandbox_doctrine_delete`) ; il n'y a pas de vue associée, mais une redirection sur l'action *list*

D'autres triplets route/action/vue seront nécessaires.

3.2 Ajouter des enregistrements avec l'*EntityManager*

3.2.1 Un peu de théorie

Point de cours

Service Doctrine : le service se récupère dans un contrôleur de deux façons quasi-équivalentes :

- *$this->get('doctrine') ;*
- *$this->getDoctrine() ;*

Pour l'instant, la seule chose qui nous intéresse est que ce service nous fournit l'*EntityManager*. C'est lui qui va gérer tous nos objets et générer automatiquement le code SQL.
La seule chose qu'il fait difficilement, c'est récupérer les entités présentes dans la base de données ; pour cela nous passerons par les *Repository* comme nous le verrons dans le chapitre suivant.

Nos contrôleurs commencerons fréquemment avec le code suivant :

```
$em = $this->getDoctrine()->getManager();
```

La démarche pour ajouter un enregistrement dans la table des films est la suivante :

- on crée un objet de type *Film*.
  À ce stade, le film est indépendant de Doctrine.
- on l'enregistre dans l'*EntityManager*.
  On dit juste à Doctrine qu'il en est responsable, mais encore aucune action sur la base de données.
- on indique à ce dernier de l'injecter dans la base de données.
  La modification de la base de données se fait toujours de manière explicite.

Dans une action, il est aussi possible de récupérer l'*EntityManager* par injection de dépendance (i.e. comme paramètre de l'action) :

```
public function exempleAction(EntityManagerInterface $em): Response
{
    // on peut utiliser $em ici
```

3.2.2 Création par la pratique

Dans cette section il n'y a rien à coder, juste à lire et comprendre. La mise en pratique se fera immédiatement après.

Voici le code pour ajouter un nouveau film dans un contrôleur :

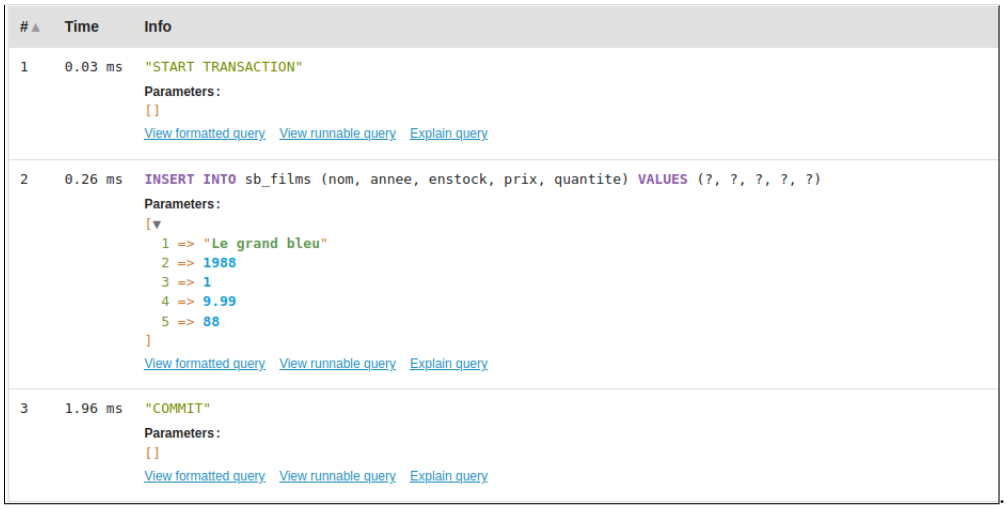
SandboxController.php

```
5   use App\Entity\Film;

18  class SandboxController extends AbstractController
19  {

485     /**
486      * @Route(
487      *     "/doctrine/ajouterendur",
488      *     name = "sandbox_doctrine_ajouterendur"
489      * )
490      */
491     public function doctrineAjouterendurAction(): Response
492     {
493         $em = $this->getDoctrine()->getManager();
494
495         $film = new Film();         // le film est encore indépendant de Doctrine
496         $film->setNom('Le grand bleu')
497             ->setAnnee(1988)
498             ->setEnstock(true)      // inutile : valeur par défaut
499             ->setPrix(9.99)
500             ->setQuantite(88);
501         dump($film);
502
503         $em->persist($film);   // Doctrine devient responsable du film
504         $em->flush();          // injection physique dans la BD
505         dump($film);
506
507         return $this->redirectToRoute('sandbox_doctrine_view', ['id' => $film->getId()]);
508     }
```

- *ligne 5* : ne pas oublier le *use*,
- *ligne 493* : récupération de l'*EntityManager*,
- *lignes 495 à 500* : création de l'objet sans se préoccuper de la base de données, À cet instant Doctrine ne connaît pas cet objet.
- *ligne 501* : l'objet n'est pas encore en base de données, il n'a pas encore d'*id*,
- *ligne 503* : Doctrine prend en charge l'objet (l'objet est *persisté)* mais ne l'injecte pas encore dans la base,
- *ligne 504* : Doctrine fait les requêtes SQL nécessaires avec les objets qu'il *persiste*,
- *ligne 505* : le film a été inséré (*flush*) dans la base de données, il a donc enfin un *id*,

- *ligne 507* : et une redirection avec le nom interne de la route et le nouvel *id* en paramètre.

Pour voir les requêtes effectuées, on peut cliquer sur l'icône à droite de la toolbar.



Notez, si vous n'avez pas paramétré Symfony, qu'une redirection dans le code du contrôleur semble invalider la consultation des requêtes dans la toolbar.

Si plusieurs entités sont persistées, alors, lors du *flush*, les requêtes sont incluses dans une transaction.

Notez que Doctrive gère de manière transparente les entités créées comme les entités récupérées (et modifiées) par un *Repository*. Il n'y pas pas à faire la différence entre un *INSERT INTO* et un *UPDATE*.

3.2.3 Exercice

Rajoutez dans le contrôleur le code de la section précédente [5], et associez-y une route.
Au fait pourquoi n'y a-t-il pas de vue associée ?

Exécutez le code et vérifiez (avec le composant "Database" de PhpStorm par exemple) que la base de données a bien été mise à jour.
Note : sous Linux vous pouvez avoir une erreur indiquant que la base est *read-only*. Une manière violente de régler le problème est [6] :

```
serveur$ chmod 777 database
serveur$ chmod 777 database/mybase.db
```

Il peut être utile de consulter, via la toolbar, les requêtes SQL exécutées par Doctrine. Or lorsqu'on clique sur la toolbar, l'entrée *Doctrine* est peut-être désactivée.
Cela est vraisemblablement dû à la redirection ; en effet lors d'une redirection Symfony refait un traitement complet depuis le début, et de fait a "oublié" ce qu'il a fait avant.

Pour pallier ce problème, il y a deux solutions :

- Lorsque vous avez cliqué sur la toolbar, en haut à gauche il y a un bouton nommé "Last 10". Symfony garde en mémoire les informations sur les 10 dernières actions [7]. Vous pouvez cliquer sur ce bouton et accéder à l'avant-dernière (ou peut-être antépénultième) action.
- Il est également possible de dire à Symfony de faire une pause avant d'exécuter une redirection. Pour cela il faut modifier le fichier *web_profiler.yaml* du répertoire *config/packages/dev* en rem-

5. Évitez les copier-coller à partir d'un pdf, cela se passe rarement bien.
6. 'L'utilisation de *setfacl* serait bien plus élégante.
7. On comprend pourquoi le mode "dev" est moins performant que le mode "prod".

plaçant la ligne :
```
intercept_redirects: false
```
par
```
intercept_redirects: true
```

Essayez les deux méthodes et regardez les requêtes SQL générées par Symfony.

Dans l'action *ajoutendur*, avec les deux lignes `dump($film);`, regardez le résultat.

Si vous avez choisi la méthode qui intercepte les redirections, Après l'exécution de l'action, on accède au contenu de l'entité sous forme arborescente (que l'on peut développer avec la souris) grâce à la petite icône en forme de cible apparaît dans la toolbar et il suffit de passer la souris dessus.

Si vous avez utilisé le bouton "Last 10", alors dans le menu à gauche l'entrée "Debug" (avec la même petite cible) devient active.

On remarque bien que dans le premier affichage l'*id* est à *null* alors qu'il est renseigné dans le deuxième affichage.
C'est l'appel à *flush* qui modifie physiquement la base de données et donc donne un *id* à l'enregistrement nouvellement créé.

3.2.4 Quelques méthodes de l'*EntityManager*

- *persist* : un objet de type "entité" est pris en compte et géré par Doctrine. Un objet récupéré par Doctrine (cf. utilisaton des *Repositories*) n'a pas besoin d'être persisté.
- *detach* : l'objet désigné n'est plus persisté.
- *flush* : tous les objets persistés et ayant subi des modifications (création, modification ou suppression) depuis le dernier *flush* génèrent des requêtes SQL.
- *contains* : indique si un objet est persisté.
- *clear* : appelle *detach* sur toutes les entités persistées ou bien uniquement sur un type précis (les *Film* par exemple).
- *refresh* : si un objet a un état différent de celui de la base de données, alors il est réinitialisé avec les informations de la base de données.
- *remove* : supprime un objet de la base de données (effectif lors de l'appel à *flush*).

3.3 Récupérer les entités via les *Repository*

3.3.1 Utilisation de base

Pour récupérer des entités existant dans la base de données, l'*EntityManager* n'est pas l'outil le plus simple[8]. Il est préférable de passer par les *Repository* (que l'on obtient avec l'*EntityManager*!).

Il existe un repository par classe entité. Voici comment on récupère le repository de la classe *Film* :

```
$em = $this->getDoctrine()->getManager();
$filmRepository = $em->getRepository('App:Film');
```

Il existe une autre méthode moins élégante mais indispensable si vous n'avez pas respecté le nommage des espaces de noms :

```
$em = $this->getDoctrine()->getManager();
$filmRepository = $em->getRepository('App\Entity\Film');
```

Et si vous vous demandez (vous devriez d'ailleurs) comment Symfony fait le lien entre la classe *Film* et la classe *FilmRepository*, vous avez la réponse dans les annotations en tête de la classe *Film*.

La méthode *find($id)* permet de récupérer une entité via son id. La méthode retourne un objet de type *Entity* (*Film* par exemple), ou *null* si l'id n'existe pas.

```
$film = $filmRepository->find(5);
```

8. et dans ce cours nous ne l'utiliserons pas à cette fin.

Il est inutile de persister une entité récupérée par un *Repository*, Doctrine le fait automatiquement. Autrement dit, si vous modifiez une entité récupérée, le prochain *flush* injectera les modifications dans la base de données.

La méthode *findAll()* récupère tous les enregistrements de la table. La méthode retourne un tableau d'objets de type *Entity* (*Film* par exemple).

```
$films = $filmRepository->findAll();
```

Notez également la méthode *remove($film)* de l'*EntityManager* (et non pas du *Repository*) qui supprimera l'entité de la base de données (*DELETE*) lors du prochain *flush*.

3.3.2 Exercices

Écrivez une méthode *modifierendur* qui modifie un film dont l'id est précisé en dur dans le code. Ne modifiez pas tous les champs et vérifiez si Doctrine optimise la requête grâce à la toolbar. Cette action fait une redirection vers la route *view*.

Écrivez une méthode *effacerendur* qui efface un film dont l'id est précisé en dur dans le code. Cette action fait une redirection vers la route *list*.

Écrivez la méthode (*listAction*) qui liste tous les films (uniquement l'id et le nom) dans la vue associée. Dans la vue faites un *dump* pour voir la structure, puis affichez les données avec une boucle et des balises HTML

Écrivez la méthode (*viewAction*) qui affiche toutes les informations d'un film dont l'id est passé en paramètre. Attention, le film peut ne pas exister.
De même faites un *dump* avant un affichage HTML.

Écrivez la méthode (*deleteAction*) qui supprime le film dont l'id est passé en paramètre. Cette action fait une redirection vers la route *list*.
Si le film n'existe pas, l'action doit lever une exception 404.

Dans la vue de l'action *list*, pour chaque film, mettez deux liens :
- un vers l'action *view* pour ce film
- un vers l'action *delete* pour ce film

Dans la vue de l'action *view*, mettez un lien vers l'action *list*.

Indication : en Twig la fonction *path* prend en paramètres le nom interne d'une route et d'éventuels arguments pour cette route, et génère l'URL associée.

3.4 Plus loin avec les *Repository*

Voici des pistes de recherches si vous êtes intréressés :
- les méthodes *findBy* et *findOneBy*
- les méthodes magiques (par exemple pour l'entité *Film* utiliser la méthode *findByEnstock*)
- créer ses propres méthodes avec le *QueryBuilder*
- créer ses propres méthodes avec le langage DQL
- la pagination avec l'objet *Paginator*

3.5 Exercice

On revient au site de vente.

Une remarque : dans le controller *Sandbox* il y a beaucoup (trop) d'actions et de vues ; ce n'est pas la bonne démarche [9].

Dans la réalité, il y a souvent un controller par entité, avec en plus quelques controllers généralistes (comme pour gérer l'accueil de l'internaute par exemple).

9. C'est comme si un programme C ne comportait qu'un seul fichier de plusieurs millers de lignes.

3.5.1 Entités

En vous référant au fichier *.sql* fourni, créez l'entité *Manuel*. Vous devez respecter à la lettre[10] les indications du fichier *.sql* (nom de la table, types, valeurs par défaut, commentaires, ...).
On rappelle que le fichier *.sql* n'est fourni que pour décrire la base de données et ne doit pas être utilisé dans le code.

Utilisez les migrations pour créer la table *prod_manuels* dans la base de données.

De même créez l'entité *Produit* : si ce n'est que pour l'instant on ne s'occupe pas de la clé étrangère *id_manuel*[11].
Note : le membre dans la classe doit s'appeler *dateCreation* alors que le champ doit s'appeler *date_creation*.

Puis faites une nouvelle migration.

3.5.2 Alimentation

Dans le controller *Produit*, créez une action pour alimenter les deux tables avec des données en dur ; par exemple 4 produits et 2 manuels.
Note : les *fixtures* sont un moyen plus élégant pour peupler une base.

3.5.3 Actions

Complétez les actions suivantes du controller *Produit* :

- *list* : affiche uniquement la dénomination de chaque produit. Pour chaque produit sont proposés un lien pour visualiser le produit et un lien pour l'effacer.
  En tête de page, mettez un lien pour ajouter un produit.
- *view* : en tête de page, mettez un lien pour afficher la liste des produits ; puis affichez toutes les informations du produit.
  Si le produit n'existe pas, faîte une redirection vers l'action *list* avec un message flash indiquant l'erreur.
- *delete* : si le produit n'existe pas, levez une exception 404, sinon après la suppression, faîtes une redirection vers la route *list*.
  Dans tous les cas, mettez un message flash adéquat.

4 Les relations entre entités

4.1 Présentation

Les relations entre tables (cardinalité, clé étrangère, contrainte d'intégrité) doivent être renseignées dans Doctrine. C'est ainsi que ce dernier est capable de faire des jointures automatiquement. De fait, via les annotations, on décrit le schéma entités-associations dans les entités.

On distingue plusieurs types de relations directement liées aux cardinalités :

- *OneToOne* : par exemple un produit est lié à un et seul manuel d'utilisation (ou éventuellement aucun), et un manuel est lié à un et un seul produit.
- *ManyToOne* (et *OneToMany*) : par exemple un produit peut avoir plusieurs images, et une image est liée à un et un seul produit. De même un film peut avoir plusieurs critiques, et une critique est liée à un seul film.
- *ManyToMany* : par exemple un produit peut être distribué dans plusieurs pays, et un pays peut distribuer plusieurs produits.

10. le mieux possible du moins
11. Rappelez-vous qu'on ne stockera pas un entier dans l'entité *Produit*, mais directement une référence à une instance de l'entité *Manuel*

- *ManyToMany avec attributs* : par exemple un produit est vendu dans plusieurs magasins et un magasin vend plusieurs produits ; pour un couple produit/magasin les attributs sont le prix unitaire et la quantité en stock.

Le *propriétaire* d'une relation est celui qui techniquement possède la clé étrangère (sauf pour une relation *ManyToMany*[12] où le choix est libre). C'est donc la classe *Critique* qui est propriétaire de la relation avec la classe *Film*. L'autre classe est dite *inverse*.

Attention, lors de la création de la classe entité (*Critique* pour nous), il ne faut pas créer la colonne *id_film* qui sera gérée directement par Doctrine.
De manière générale, dans Symfony on ne manipule pas directement une clé étrangère : si c'est le cas il y a vraisemblablement une erreur de conception.
Soyons plus précis :

- Si dans la classe *Critique* il y a un champ *$id_film* (ou *$idFilm*) de type entier, c'est une erreur : vous êtes en train de recoder le fonctionnement d'une base de données alors que c'est le rôle de Doctrine.
- En revanche, toujours dans *Critique* il doit y avoir un champ *$film* de type *Film* (i.e. la classe entité *Film*). Nous allons détailler tout cela dans la suite.

Une relation est par défaut *unidirectionnelle* dans le sens "propriétaire vers inverse". Le propriétaire connait son inverse, mais la réciproque est fausse.
On peut faire *$proprietaire->getInverse()* (dans notre cas *$critique->getFilm()*).

Mais le contraire n'est pas vrai : un film ne peut pas récupérer directement la liste de ses commentaires (i.e. *$film->getCritiques()*). Si on veut avoir ce comportement il y a deux possibiltés :

- on crée une relation bidirectionnelle (et *getCritiques()* est utilisable)
- on passe par les *Repositories*

Attention, Doctrine lance des requêtes SQL au dernier moment (on parle de *lazy loading*). Par exemple examinons le pseudo-code suivant :

```
$critiques = $critiqueRepository->findAll();
foreach ($critiques as $critique)
....$film = $critique->getFilm();
```

Sous l'hypothèse que l'on a 50 critiques :

- la première ligne fait une requête qui rappatrie toutes les critiques, mais pour chacune le film n'est pas rappatrié.
- la troisième ligne est exécutrée 50 fois. Chaque fois Doctrine s'aperçoit que le film n'est pas présent et il déclenche une requête SQL.
- soit un total de 51 requêtes.

Remarques sur le *lazy loading* :

- si on n'a pas besoin des films associés aux critiques, le *lazy loading* est le bienvenu,
- dans le cas contraire (cf. code ci-dessus) c'est une catastrophe car les performances chutent drastiquement.

Pour éviter les requêtes multiples, on peut écrire notre propre méthode dans le *repository* de *Critique* pour rappatrier l'ensemble des informations en une seule requête.

4.2 Les relations *One/Many-To-One/Many*

Le but de cette section est d'avoir un panorama détaillé des notations. La mise en pratique se fera dans la section suivante.

Nous allons étudier les annotations nécessaires pour tous ces types de relations.

La commande `make:entity` est capable de générer automatiquement les annotations, les getters et setters correspondants, et même un constructeur dans certains cas.
Il est cependant indispensable de bien comprendre chaque directive.

12. qui, nous le verrons par la suite pour des raisons techniques, est assez rare dans Doctrine

Rappel :

- relation unidirectionnelle : la classe propriétaire connaît la classe inverse, et la réciproque est fausse ; seule la classe propriétaire à un membre référençant la classe inverse.
- relation bidirectionnelle : les deux classes se connaissent mutuellement, et chacune a un membre référençant l'autre.

4.2.1 *OneToOne*

Relation unique entre deux tables : une ligne d'une table est reliée à une (ou zéro) et une seule ligne de l'autre table et vice-versa.

Prenons l'exemple de l'entité *Habitant* et de l'entité *Permis* (de conduire).

Il faut choisir l'entité propriétaire car les deux sont éligibles. Nous choisissons *Habitant* car on peut supposer qu'il est plus fréquent qu'un habitant accède à son permis que le contraire. Mais l'autre choix est tout à fait valable.

Dans le cas d'une relation unidirectionnelle, seule la classe propriétaire (*Habitant*) déclare un membre dont le type est la classe inverse (*Permis*).
On réinsiste sur le fait que la classe *Habitant* déclare un membre de type *Permis* et non pas un entier pour stocker la clé étrangère.

Voici le résultat généré par le script :

Habitant.php

```
8  /**
9   * @ORM\Entity(repositoryClass=HabitantRepository::class)
10  */
11 class Habitant
12 {
```

```
18     /**
19      * @ORM\OneToOne(targetEntity=Permis::class, cascade={"persist", "remove"})
20      */
21     private $permis;
```

Quelques explications et remarques :

- le getter et le setter sont générés automatiquement par le script
- il manque éventuellement des annotations (cf. ci-dessous)
- la classe inverse est explicitement mentionnée
- le membre *permis* peut être *null* (comportement par défaut, sinon ce serait explicitement mentionné, cf. ci-dessous)
- annotation *cascade* :
  - aucun rapport avec le *CASCADE* des SGBD
  - *persist* : indique que si on persiste une entité *Habitant* alors l'entité *Permis* liée est automatiquement persistée
  - *remove* : indique que si on supprime (via la méthode *remove* de l'*entityManager*) une entité *Habitant* alors l'entité *Permis* liée subit automatiquement le même ordre.
  - il y a un risque important, à utiliser avec modération.
- Quelle est le nom de la clé étrangère ? Par défaut ce sera *permis_id* (cf. ci-dessous pour en choisir un autre).
- Quelle est le nom de la table inverse ? Symfony ira voir l'entité *Permis* dans laquelle le nom de la table est indiquée.

Voici les annotations complétées :

Habitant.php

```
20     /**
21      * @var Permis
22      * @ORM\OneToOne(targetEntity=Permis::class, cascade={"persist", "remove"})
23      * @ORM\JoinColumn(
24      *     name="id_permis",
25      *     nullable=true,
26      *     referencedColumnName="id"
27      * )
```

```
28      */
29     private $permis;
```

Quelques explications et remarques :

- ligne 21 : aucun rapport avec Doctrine, elle permet à PhpStorm de connaître le type du membre
- ligne 23 : indique à Doctrine comment configurer la clé étrangère
- ligne 24 : indispensable si on veut un autre nom que le nom par défaut pour la clé étrangère (c'est le cas ici)
- ligne 25 : inutile ici car c'est la valeur par défaut
- ligne 26 : indique le nom de la clé primaire de la classe inverse ; inutile ici car "id" est la valeur par défaut.

Pour mettre à jour physiquement la base de données, ne pas oublier le triptyque :

- `symfony console doctrine:schema:validate`
- `symfony console doctrine:migrations:diff`
- `symfony console doctrine:migrations:migrate`

Et si on veut que la classe inverse connaisse la classe propriétaire, il faut déclarer un membre annoté (la commande `make:entity` le gère également) dans la classe inverse.
Voici le résultat dans la classe *Permis*

Permis.php

```
20     /**
21      * @ORM\OneToOne(targetEntity=Habitant::class, mappedBy="permis", cascade={"persist", "remove"})
22      */
23     private $habitant;
```

- la classe propriétaire est indiquée (*targetEntity*)
- ainsi que le nom du membre dans la classe propriétaire (*mappedBy*)

Et il y a un changement dans la classe *Habitant*

Habitant.php

```
20     /**
21      * @var Permis
22      * @ORM\OneToOne(targetEntity=Permis::class, inversedBy="habitant", cascade={"persist", "remove"})
23      * @ORM\JoinColumn(
24      *     name="id_permis",
25      *     nullable=true,
26      *     referencedColumnName="id"
27      * )
28      */
29     private $permis;
```

- On note le champ *inversedBy* qui précise le nom du membre dans la classe inverse.

4.2.2 *ManyToOne*

Une ligne de le première table ne peut être liée qu'à au plus une ligne de la seconde ; en revanche une ligne de la seconde table peut être liée à plusieurs lignes de la première.

Prenons l'exemple de l'entité *Habitant* et de l'entité *Ville*. Et on considère qu'un habitant peut n'habiter aucune ville.

Pour le choix de l'entité propriétaire il n'y a justement pas le choix car la clé étrangère ne peut être que dans la table *habitants*. Donc *Habitant* et la classe propriétaire.

Cf. ci-dessus pour les notions de relations unidirectionnelle ou bidirectionnelle.

Le côté *Many* est la classe *Habitant* et le *One* concerne la *Ville* : *plusieurs* habitants sont dans *une* ville. Une autre manière de dire est que le *Many* est toujours du côté de la clé étrangère et donc de la classe propriétaire, et le *One* toujours du côté de la classe inverse.

Voici le résultat du script avec un choix d'une relation bidirectionnelle et des ajouts à la main. Pour la classe propriétaire *Habitant* :

Habitant.php

```
31     /**
32      * @var Ville
33      * @ORM\ManyToOne(targetEntity=Ville::class, inversedBy="habitants")
34      * @ORM\JoinColumn(
35      *     name="id_ville",
36      *     nullable=true,
37      *     referencedColumnName="id"
38      * )
39      */
40     private $ville;
```

- À part le *ManyToOne*, c'est exactement les mêmes annotations que pour un *OneToOne*. En revanche certaines méthodes du *Repository* s'adapteront au fait qu'on ait une relation *Many*.
- Le getter et le setter sont aussi classiques.
- Nous avons choisi de ne pas mettre la propriété *cascade* notamment pour le *remove* : en effet supprimer un habitant ne doit pas entraîner la suppression de la ville.

Et pour la classe inverse *Ville* :

Ville.php

```
22     /**
23      * @var Collection
24      * @ORM\OneToMany(targetEntity=Habitant::class, mappedBy="ville")
25      */
26     private $habitants;
```

- On note le type du membre : *Collection*
- On note que le nom du membre est au pluriel[13] : c'est très important car ce membre est une collection d'habitants.
- Les getter et setters sont plus complexes car la suppression (ou ajout) d'un habitant à la collection implique de modifier les données de l'habitant également.

Et ne pas oublier la migration.

4.2.3 *OneToMany*

Cette relation est dans une classe inverse et intimement liée au *ManyToOne* de la classe propriétaire.

4.2.4 *ManyToMany* sans attribut

Une ligne de le première table peut être liée à plusieurs lignes de la seconde ; et réciproquement une ligne de la seconde table peut être liée à plusieurs lignes de la première.

Prenons l'exemple de l'entité *Habitant* et de l'entité *Nationalite*.

Comme pour le *OneToOne* il faut choisir l'entité propriétaire : nous prenons de nouveau *Habitant*.

Il faut une table de jointure et Doctrine la gère de manière transparente. Le nom de la table de jointure et les noms des deux clés étrangères peuvent être générés automatiquement par Symfony ; mais il est possible de les imposer via des annotations.

Voici le résultat du script avec un choix d'une relation bidirectionnelle et des ajouts à la main. Pour la classe propriétaire *Habitant* :

Habitant.php

```
45     /**
46      * @var Collection
47      * @ORM\ManyToMany(targetEntity=Nationalite::class, inversedBy="habitants")
48      * @ORM\JoinTable(
49      *     name="asso_habitants_nationalites",
50      *     joinColumns={
51      *         @ORM\JoinColumn (name="id_habitant", referencedColumnName="id")
52      *     },
```

13. C'est une faute de programmation de ne pas mettre un pluriel à un nom de variable contenant plusieurs valeurs.

```
53      *     inverseJoinColumns={
54      *         @ORM\JoinColumn (name="id_nationalite", referencedColumnName="id")
55      *     }
56      * )
57      */
58     private $nationalites;
```

- ligne 47 : On retrouve les indications *targetEntity* et *inversedBy*
- ligne 48 : annotation indiquant que l'on paramètre de table de jointure (cette annotation est facultative : Doctrine choisirait alors des noms par défaut)
- ligne 49 : on impose le nom de la table de jointure [14]
- lignes 50 à 52 : configuration de la clé étrangère (dans la table de jointure) vers la table propriétaire
- lignes 53 à 55 : configuration de la clé étrangère (dans la table de jointure) vers la table inverse
- ligne 58 : on note à nouveau que le nom du membre est au pluriel
- la clé primaire de la table de jointure est composée de ses deux clés étrangères [15]
- un constructeur a été généré automatiquement dans la classe propriétaire
- de même pour les getter et setters

Et pour la classe inverse *Nationalite* :

Nationalite.php

```
23     /**
24      * @var Collection
25      * @ORM\ManyToMany(targetEntity=Habitant::class, mappedBy="nationalites")
26      */
27     private $habitants;
```

- cf. remarques ci-dessus

Attention il est coûteux, en terme de codage, de passer d'une relation *ManyToMany* sans attribut à une relation *ManyToMany* avec attributs.
Aussi ce choix doit-il être assumé. Dans le doute il est préférable de choisir le codage d'une relation *ManyToMany* avec attributs (cf. section suivante) même si dans un premier temps il n'y a pas d'attribut.

4.2.5 *ManyToMany* avec attributs

C'est une relation *ManyToMany* classique si ce n'est que la table de jointure contient d'autres informations que les deux clés étrangères.

Prenons l'exemple de l'entité *Habitant* et de l'entité *Etablissement* (scolaire) : la table de jointure, outre les deux clés étrangères, contient l'année d'inscription.

Doctrine ne sait pas gérer ce type de relation, la solution est de créer deux relations *ManyToOne* en créant une entité gérant la table de jointure que l'on nomme par exemple *HabitantEtablissement* :

- *HabitantEtablissement ManyToOne Habitant*
- *HabitantEtablissement ManyToOne Etablissement*
- un membre *annee* dans *HabitantEtablissement*

On retombe donc dans le cas *ManyToOne*, mais avec quelques contraintes :

- les deux clés étrangères ne peuvent pas être *null*.
- faire des relations bidirectionnelles n'a que peu d'intérêt (il est préférable de définir des méthodes d'interrogation des les *repositories* qui cacheront la présence de l'entité de jointure).
- la clé primaire de la table *asso_habitants_etablissements* devrait être le triplet (*id_habitant*, *id_etablissement*, *annee*). Mais :
- il n'est pas possible avec Doctrine d'avoir une clé primaire répartie sur plusieurs champs dans une entité.
- il faut donc ajouter une clé primaire (*id*) auto-incrémentée
- il faut créer un index *unique* sur le triplet pour éviter des incohérence dans la base de données.

14. par défaut, ça aurait été *habitant_nationalite*
15. c'est le seul cas dans Symfony où une clé primaire est composée de plusieurs champs.

Voici à quoi ressemble l'entité de jointure :

HabitantEtablissement.php

```
8  /**
9   * @ORM\Table (
10  *     name="asso_habitants_etablissement",
11  *     uniqueConstraints={
12  *         @ORM\UniqueConstraint(name="ahe_idx", columns={"id_habitant", "id_etablissement", "annee"})
13  *     }
14  * )
15  * @ORM\Entity(repositoryClass=HabitantEtablissementRepository::class)
16  */
17 class HabitantEtablissement
18 {
19     /**
20      * @ORM\Id
21      * @ORM\GeneratedValue
22      * @ORM\Column(type="integer")
23      */
24     private $id;
25
26     /**
27      * @var Habitant
28      * @ORM\ManyToOne(targetEntity=Habitant::class)
29      * @ORM\JoinColumn(name="id_habitant", nullable=false)
30      */
31     private $habitant;
32
33     /**
34      * @var Etablissement
35      * @ORM\ManyToOne(targetEntity=Etablissement::class)
36      * @ORM\JoinColumn(name="id_etablissement", nullable=false)
37      */
38     private $etablissement;
39
40     /**
41      * @ORM\Column(type="integer")
42      */
43     private $annee;
```

- lignes 11 à 13 : index unique sur ce qui aurait du être la clé primaire
- lignes 19 à 24 : clé primaire "artificielle" imposée par Doctrine
- lignes 26 à 31 : première entité de la jointure ; on note qu'elle ne peut pas être *null*.
- lignes 33 à 38 : idem pour la deuxième entité.
- lignes 40 à 43 : le fameux attribut de la table de jointure.

Les deux autres classes ne sont pas modifiées car nous nous sommes restreints à des relations unidirectionnelles.

4.3 Relation entre *Critique* et *Film*

Nous passons à la mise en pratique dans le controller *Sandbox*.

4.3.1 Entité *Critique*

Commencez par créer l'entité *Critique* (avec la commande `make:entity`) en ignorant la clé étrangère. Regardez le fichier *.sql* fourni pour voir les directives, notamment :

- la table s'appelle *sb_critiques*
- *note* est associé à un commentaire, peut être *null* et a comme valeur par défaut *null*.
- on ne s'occupe pas (encore) de *id_film*
- faites le constructeur pour initialiser le membre *note* à *null*.

Finissez par une migration

Il s'agit d'une relation *ManyToOne* de *Critique* vers *Film* dont *Critique* est propriétaire (car il possède la clé étrangère).
C'est bien *ManyToOne* car il y a plusieurs (*many*) critiques pour un (*one*) film.

C'est le script `make:entity` qui se charge de tout :

- création du nouveau membre que l'on va appeler *film*
- annotation pour la relation propriétaire
- annotation pour la relation inverse
- getters et setters
- constructeur

Attention (on insiste lourdement) l'attribut ajouté ne représente pas la clé étrangère [16] mais directement le film visé ; donc l'attribut sera bien de type *Film*.

```
serveur$ symfony console make:entity

 Class name of the entity to create or update (e.g. GrumpyPopsicle):
 > Critique

 Your entity already exists! So let's add some new fields!

 New property name (press <return> to stop adding fields):
 >
```

- le script a bien détecté que la classe existe et va la compléter.

```
 New property name (press <return> to stop adding fields):
 > film

 Field type (enter ? to see all types) [string]:
 >
```

- il est classique (et lisible) d'appeler le membre comme le nom de l'entité ; et on rappelle une dernière (et cinquante-septième) fois que le membre est de type *Film* et non pas un entier.

```
 Field type (enter ? to see all types) [string]:
 > ManyToOne

 What class should this entity be related to?:
 >
```

- le type *ManyToOne* est proposé et va nous aider à configurer la relation.

```
 What class should this entity be related to?:
 > Film

 Is the Critique.film property allowed to be null (nullable)? (yes/no) [yes]:
 >
```

- l'entité inverse est demandée

```
 Is the Critique.film property allowed to be null (nullable)? (yes/no) [yes]:
 > no

 Do you want to add a new property to Film so that you can access/update Critiqu
e objects from it - e.g. $film->getCritiques()? (yes/no) [yes]:
 >
```

- la réponse est non : une critique est obligatoirement liée à un film.

```
 Do you want to add a new property to Film so that you can access/update Critiqu
e objects from it - e.g. $film->getCritiques()? (yes/no) [yes]:
 >

 A new property will also be added to the Film class so that you can access the
related Critique objects from it.

 New field name inside Film [critiques]:
 >
```

- à la question d'ajouter un membre à *Film* (l'entité inverse) la réponse est *yes* car nous voulons une relation bidirectionnelle. Il suffit de valider car la réponse *yes* est la réponse par défaut.

16. En fait c'est le cas, mais c'est transparent.

```
New field name inside Film [critiques]:
>

Do you want to activate orphanRemoval on your relationship?
A Critique is "orphaned" when it is removed from its related Film.
e.g. $film->removeCritique($critique)

NOTE: If a Critique may *change* from one Film to another, answer "no".

Do you want to automatically delete orphaned App\Entity\Critique objects (orpha
nRemoval)? (yes/no) [no]:
>
```

- de même il suffit de valider car le nom du membre proposé par défaut nous convient. On note à nouveau que le nom du champ est au pluriel.

```
Do you want to automatically delete orphaned App\Entity\Critique objects (orpha
nRemoval)? (yes/no) [no]:
>

updated: src/Entity/Critique.php
updated: src/Entity/Film.php

Add another property? Enter the property name (or press <return> to stop adding
fields):
>
```

- nous choisisson la réponse *non* : avec un choix positif, la supression d'un film pourrait entraîner la suppression de plusieurs dizaines de critiques ; c'est trop dangereux.

```
Add another property? Enter the property name (or press <return> to stop adding
fields):
>


 Success!

Next: When you're ready, create a migration with php bin/console make:migration

serveur$
```

- une dernière validation pour terminer le script.

Il reste à préciser le nom de la clé étrangère dans le nouveau membre et la migration peut être faite.

Le résultat doit être le suivant pour l'entité *Critique* :

Critique.php

```
8  /**
9   * @ORM\Table (name="sb_critiques")
10  * @ORM\Entity(repositoryClass=CritiqueRepository::class)
11  */
12 class Critique
13 {
```

```
31     /**
32      * @var Film
33      * @ORM\ManyToOne(targetEntity=Film::class, inversedBy="critiques")
34      * @ORM\JoinColumn(name="id_film", nullable=false)
35      */
36     private $film;
37
38     /**
39      * Critique constructor.
40      */
41     public function __construct()
42     {
43         $this->note = null;
44     }
```

Et pour l'entité *Film* :

Film.php

```
10 /**
```

```
11  * @ORM\Table (name="sb_films")
12  * @ORM\Entity(repositoryClass=FilmRepository::class)
13  */
14 class Film
15 {
```

```
50     /**
51      * @var Collection
52      * @ORM\OneToMany(targetEntity=Critique::class, mappedBy="film")
53      */
54     private $critiques;
55
56     /**
57      * Film constructor.
58      */
59     public function __construct()
60     {
61         $this->enstock = true;
62         $this->quantite = null;
63         $this->critiques = new ArrayCollection();
64     }
```

4.3.2 Création d'entités

Tout d'abord un exemple pour ajouter un film et deux critiques. Notez que le code serait similaire pour ajouter une critique à un film existant.

SandboxController.php

```
575     /**
576      * @Route(
577      *     "/doctrine/critique/ajouterendur",
578      *     name = "sandbox_doctrine_critique_ajouterendur"
579      * )
580      */
581     public function doctrineCritiqueAjouterendurAction(): Response
582     {
583         $em = $this->getDoctrine()->getManager();
584
585         $film = new Film();
586         $film->setNom('Le grand bleu')
587             ->setAnnee(1988)
588             ->setEnstock(true)      // inutile : valeur par défaut
589             ->setPrix(9.99)
590             ->setQuantite(88);
591         $em->persist($film);
592
593         $critique1 = new Critique();
594         $critique1->setNote(5)
595             ->setAvis("sa a changer tout ma vi")
596             ->setFilm($film);
597         $em->persist($critique1);
598
599         $critique2 = new Critique();
600         $critique2->setNote(0)
601             ->setAvis("Le grand vide plutôt !")
602             ->setFilm($film);
603         $em->persist($critique2);
604
605         $em->flush();
606
607         dump($film);
608
609         return $this->redirectToRoute('sandbox_doctrine_critique_view1', ['id' => $film->getId()]);
610         //return $this->redirectToRoute('sandbox_doctrine_critique_view2', ['id' => $film->getId()]);
611     }
```

- *lignes 585 à 590* : création d'un film ; on aurait aussi pu récupérer un film via le *Repository*,
- *lignes 593 à 596* : création de la première critique ; notez comment on y associe le film sans s'occuper de la clé étrangère,
- *lignes 599 à 602* : idem pour la deuxième critique,

- *lignes 591, 597, 603, 605* et on n'oublie pas les *persist* et le *flush*.

Codez cette action pour vérifiez son bon fonctionnement. N'oubliez pas la route associée.
Dans la toolbar, regardez les requêtes SQL générées.

4.3.3 Récupération des entités (via le repository)

L'action *view1* affiche le détail d'un film avec ses critiques.

Nous ne nous servons pas de la relation inverse (ce qui dommage bien entendu) pour utiliser le *Repository* des critiques.

Voici le code :

SandboxController.php

```
614     /**
615      * @Route(
616      *     "/doctrine/critique/view1/{id}",
617      *     name = "sandbox_doctrine_critique_view1",
618      *     requirements = {"id" = "[1-9]\d*"}
619      * )
620      */
621     public function doctrineCritiqueView1Action($id): Response
622     {
623         $em = $this->getDoctrine()->getManager();
624         $filmRepository = $em->getRepository('App:Film');
625         $critiqueRepository = $em->getRepository('App:Critique');
626 
627         /** @var Film $film */
628         $film = $filmRepository->find($id);
629         if (is_null($film))
630             throw NotFoundHttpException('Le film ' . $id . ' n\'existe pas;');
631 
632         $critiques = $critiqueRepository->findBy(array('film' => $film));
633 
634         $args = array(
635             'film' => $film,
636             'critiques' => $critiques,
637         );
638         return $this->render('Sandbox/doctrine_critique_view1.html.twig', $args);
639     }
```

La seule nouveauté est la fonction *findBy* d'un repository qui permet de faire une recherche selon un tableau de critères.

Via la toolbar, regardez les requêtes SQL exécutées. On remarque qu'il y'en a deux, ce qui n'est pas optimisé.

La vue ressemble à :

doctrine_critique_view1.html.twig

```
7  {% block vue %}
8      <p><a href="{{ path('sandbox_doctrine_list') }}">Retour vers la liste complète</a>.</p>
9  
10     {{ dump(film) }}
11     {{ dump(critiques) }}
12 
13     <h2>Film</h2>
14     {% if film is null %}
15         <p>Le film inexistant</p>
16     {% else %}
17         <table>
18             <tr><th>id</th><td>{{ film.id }}</td></tr>
19             <tr><th>nom</th><td>{{ film.nom }}</td></tr>
20             <tr><th>annee</th><td>{{ film.annee }}</td></tr>
21             <tr><th>enstock</th><td>{{ film.enstock }}</td></tr>
22             <tr><th>prix</th><td>{{ film.prix }}</td></tr>
23             <tr><th>quantite</th><td>{{ film.quantite }}</td></tr>
24         </table>
25     {% endif %}
26 
27     <h2>Critiques</h2>
```

```
28      <ul>
29          {% for critique in critiques %}
30              <li>{{ critique.note }} : {{ critique.avis }}</li>
31          {% endfor %}
32      </ul>
33  {% endblock %}
```

4.3.4 Récupération des entités (via le lien bidirectionnel)

L'action *view2* affiche également le détail d'un film avec ses critiques.

Mais dans cette version nous nous servons de la relation bidirectionnelle qui simplifie légèrement le code.

Voici le code :

SandboxController.php

```
642     /**
643      * @Route(
644      *     "/doctrine/critique/view2/{id}",
645      *     name = "sandbox_doctrine_critique_view2",
646      *     requirements = {"id" = "[1-9]\d*"}
647      * )
648      */
649     public function doctrineCritiqueView2Action($id): Response
650     {
651         $em = $this->getDoctrine()->getManager();
652         $filmRepository = $em->getRepository('App:Film');
653 
654         /** @var Film $film */
655         $film = $filmRepository->find($id);
656         if (is_null($film))
657             throw NotFoundHttpException('Le film ' . $id . ' n\'existe pas;');
658 
659         $args = array(
660             'film' => $film,
661         );
662         return $this->render('Sandbox/doctrine_critique_view2.html.twig', $args);
663     }
```

On ne s'occupe plus de récupérer les critiques, cela est fait automatiquement.

Via la toolbar, regardez les requêtes SQL exécutées. On remarque qu'il y'en a deux, ce qui n'est pas optimisé.

La vue ressemble à :

doctrine_critique_view2.html.twig

```
7   {% block vue %}
8       <p><a href="{{ path('sandbox_doctrine_list') }}">Retour vers la liste complète</a>.</p>
9   
10      {{ dump(film) }}
11  
12      <h2>Film</h2>
13      {% if film is null %}
14          <p>Le film inexistant</p>
15      {% else %}
16          <table>
17              <tr><th>id</th><td>{{ film.id }}</td></tr>
18              <tr><th>nom</th><td>{{ film.nom }}</td></tr>
19              <tr><th>annee</th><td>{{ film.annee }}</td></tr>
20              <tr><th>enstock</th><td>{{ film.enstock }}</td></tr>
21              <tr><th>prix</th><td>{{ film.prix }}</td></tr>
22              <tr><th>quantite</th><td>{{ film.quantite }}</td></tr>
23          </table>
24      {% endif %}
25  
26      <h2>Critiques</h2>
27      <ul>
28          {% for critique in film.critiques %}
29              <li>{{ critique.note }} : {{ critique.avis }}</li>
30          {% endfor %}
31      </ul>
32  {% endblock %}
```

4.3.5 Exercice

Le but est d'appliquer toutes les notions vues au site de vente :

- relation *Produit OneToOne Manuel* : sachant qu'un produit peut ne pas avoir de manuel d'utilisation.
- relation *Image ManyToOne Produit*
- relation *Produit ManyToMany Pays* sans attribut
- relation *Produit ManyToMany Magasin* avec attributs

Essayez de respecter les données du fichier SQL fourni : noms des tables, non des champs, types, ...