Persistance des données – Doctrine 3

1. Notion d'ORM

Doctrine2 est l'ORM (Object-Relation Mapper) par défaut de Symfony. L'ORM dispense des requêtes SQL pour créer les bases de données et gérer la persistance des données. La persistance est la notion qui traite de l'écriture de données sur un support informatique, et plus spécifiquement du stockage des informations dans des bases de données relationnelles. Dans une application orientée objet, la persistance permet à un objet de survivre au processus qui l'a créé.
Pour cela les données doivent être sous forme d'objets. Un objet utilisé par l'ORM pour être enregistré dans une base de données s'appelle **entité.** On parle alors de persistance d'entité. Une entité n'est rien d'autre qu'un objet enrichi d'annotations exploitées par l'ORM.

Il existe deux manières de faire. Soit la base de données existe et il suffit de faire correspondre les tables aux entités (section 4). Soit la base de données n'existe pas et on va créer les tables à partir des entités (section 3).

2. Créer une base de données avec Doctrine 3

Pour créer une base de données avec Doctrine, il faut paramétrer le fichier .env pour les versions symfony4 et lui indiquer les informations permettant de créer la base de données et les paramètres permettant d'accéder au serveur de base de données.

2.1. Fichier de configuration .env

Ce fichier permet de configurer les informations concernant la bases de données, à savoir : le nom/adresse du serveur de base de données, le nom de la base de données, l'identifiant/mot de passe permettant l'accès à la base de données.

Exemple du fichier **.env :**

```
DATABASE_URL="mysql://db_user:db_password@server_mysql:server_port/db_name"
```

Remarque :

Si le fichier ne contient pas la propriété DATABASE_URL, installer doctrine avec la commande suivante :

```
composer require doctrine
```

Travail à faire :

Dans votre projet GSB, renseigner le fichier .env avec vos paramètres de connexion à la base de données et indiquer le nom de votre base de données.

Vérifier au niveau du serveur mysql que la base de données n'est pas créée.

2.2. Création de la base de données

Si la base de données n'est pas encore créée, une fois le fichier .env personnalisé, on peut la créer à l'aide de la commande suivante :

```
php bin/console doctrine:database:create          // Sf3 & 4
```

Vous remarquerez que le nom de la base de données n'est pas indiquée dans la ligne de commande. En effet, cette commande va créer la base de données avec le nom indiqué dans le fichier .env, d'où l'importance de configurer le fichier avant d'exécuter la commande.

Pour vérifier la création de la base de données dans mysql, lancer la commande suivante en ligne de commande :

$ **`mysql -u root -p`** le système vous demande le mot de passe de l'administrateur de mysql, ensuite, une fois connecté sur le serveur de base de données, lancer la commande SQL : **`show databases ;`**

3. Générer une entité avec Doctrine2

3.1. Entité

Une fois la base de données créée, il suffit de générer les différentes entités et de les persister en base de données.

Il est possible de générer une entité à l'aide du générateur de Doctrine :

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

Le système demande le nom de l'entité. Une entité est une classe PHP au niveau de Symfony qui correspond à une table au niveau de la base de données.

Une fois le nom validé, deux fichiers seront créés :

- **Entity\NomEntite.php** ;
- **Repository\NomEntiteRepository.php** qui servira à contenir les requêtes permettant d'accéder à la table correspondante.

Doctrine crée automatiquement le champ id de la table de type entier. Ensuite, il demande si vous voulez créer d'autres champs ou si vous voulez le faire plus tard.

Pour chaque champ, Doctrine vous demande : le type, la taille (par exemple le nombre de caractères pour un string), la contrainte null/not null.

```
Note that the primary key will be added automatically (named id).

Available types: array, simple_array, json_array, object,
boolean, integer, smallint, bigint, string, text, datetime, datetimetz,
date, time, decimal, float, blob, guid.

New field name (press <return> to stop adding fields): nom
Field type [string]:
Field length [255]: 30
```

Travail à faire :

On va créer l'entité qui va correspondre à la table Visiteur (voir MCD fourni sur l'ENT).

Visualiser le contenu des deux fichiers : App\Entity\Visiteur.php et App\Repository\VisiteurRepository.php

Que contient le fichier Visiteur.php ?

Que contient le fichier VisiteurRepository.php ?

Est-ce que la table a été créée ?

Insérer le contenu du fichier App\Entity\Visiteur.php et App\Repository\VisiteurRepository.php dans ce document de cours.

3.2. Annotations de l'entité

Les annotations sont placées entre **/** … */**, à ne pas confondre avec les commentaires sur plusieurs lignes /* …. */

Annotations générées par la commande	Explication
@ORM\Table(name="nomTable")	Cette annotation n'est indispensable que si le nom de la table est différent du nom de l'entité Par défaut, le nom de la table est déduit du

	nom de l'entité, le tout en minuscule. Cette annotation, si elle existe, doit être **placée avant la définition de la classe.**
@ORM\Entity(repositoryClass="`App\Repository\EtatRepository`**")**	Indique le nom de l'entité repository qui va servir à récupérer les enregistrements depuis la BDD. Cette annotation est indiquée avant la définition de la table.
@ORM\Id() **@ORM\GeneratedValue()** **@ORM\Column(type="integer")**	Indique que le champ qui sera défini à la suite est un id (unique et non null) et correspond donc à la clé primaire au niveau de la BDD La clé primaire sera créée automatiquement par la BDD (auto-increment) La colonne sera de type entier
/** *** @var string** ***** *** @ORM\Column([name="libelle",] type="string", length=255)** ***/**	Définition d'une colonne. L'annotation indique : - le nom de la colonne si elle est différente du nom de l'attribut - le type de la colonne - la longueur du champ Ici, il s'agit de définir la colonne libelle avec un varchar(255)

L'annotation Column

Elle s'applique sur les attributs de classe et correspondent aux noms de colonnes de la table qui sera créée. Elle définit les caractéristiques de la colonne comme :

- le nom si on souhaite le modifier **@ORM\Column(name="nomColonne")**,
- le type : les types de colonnes définis en annotation sont des types Doctrine. Ces types sont sensibles à la casse et ont des correspondances en SQL : string (VARCHAR), integer (INT) smallint, bigint, boolean , decimal, date ou datetime (DATETIME), time, float... ;
- la génération des valeurs automatique ou non
- les contraintes null/not null : **@ORM\Column(nullable=true|false)**
- la précision d'un nombre à virgule (nombre de chiffres en tout) : (precision=5)
- le nombre de chiffres après la virgule (decimal) (scale=2)
- les contraintes d'unicité : **@ORM\Column(unique=true|false)**

3.3. Persistance des entités

La persistance des entités consiste à créer les tables correspondantes aux entités dans la base de donnés. Doctrine travaille avec de la gestion des versions du schéma de la base de données. Cela permet de pouvoir revenir sur version précédente de la BDD si besoin.

Après chaque création et/ou modification des entités, il faudra générer une nouvelle migration. Pour cela, on utilise la commande :

```
php bin/console make:migration          (1)
```

Un fichier de migration sql sera créé dans le projet. Il est horodaté. A ce stade, rien n'est modifié dans la base de données.

Pour persister en base de données, il faut lancer la commande suivante :

```
php bin/console doctrine:migrations:migrate     (2)
```

Si on veut annuler et revenir à la version précédente, il faut lancer la commande :

```
php bin/console doctrine:migrations:execute --down 'DoctrineMigrations\<Version_num>
```

Pour visualiser la liste des versions :

```
php bin/console doctrine:migrations:list
```

Pour visualiser les requêtes SQL qui vont être exécutées avant de les effectuer réellement à l'aide de la commande :

```
php bin/console doctrine:schema:[create | update] --dump-sql
```

Travail à faire :

1. Lancer la commande (1) dans votre projet GSB
2. Visualiser le contenu du fichier de migration. Quelle est la requête SQL qui sera exécutée en base de données ?
3. Lancer la commande (2).
4. Vérifier le contenu de la base de données depuis le serveur mySQL.
5. Créer l'entité Secteur
6. Générer une nouvelle migration.
7. Visualiser le contenu du fichier de migration. Quelle est la requête SQL qui sera lancée ?
8. Persister l'entité en BDD
9. Insérer le code de l'entité Secteur.
10. Insérer la capture écran du résultat obtenu en BDD (description de la table).
11. Créer l'entité Specialite
12. Répéter les opérations 6 et 8.

3.4. Entité avec relations

Les bases de données relationnelles définissent des relations entre les tables. On a 4 types de relations qui sont traduits en Doctrine avec des annotations.

OneToOne : traduit une relation unique entre deux entités. Elle se traduit par l'ajout d'une référence dans la classe entité qui correspond à la table concernée dans le modèle logique. Le nom par défaut attribuée à la colonne est : nomPropriete_id.

L'entité qui porte la propriété faisant référence à l'entité liée est appelée entité **propriétaire**. L'entité liée est nommé entité **inverse**, elle est définie par l'option **targetEntity**. L'entité propriétaire, sous Doctrine 2, sera celle depuis laquelle on veut récupérer l'entité liée. C'est ce qui va déterminer le choix de l'entité propriétaire.

Exemple :

Entité Salarié : quand on persiste en BDD, la table Salarie comportera la colonne badge_id définie comme clé étrangère.

```
/**
 * @ORM\OneToOne(targetEntity="App\Entity\Badge")
 */
private $badge ;          // nom du champs qui permet de faire le lien avec l'entité Badge.
```

Si la relation est bidirectionnelle, il faut définir l'annotation @OneToOne dans chacune des entités concernées par la relation avec un paramètre supplémentaire :

- **mappedBy** dans l'entité **inverse** (ne contiendra pas la clé étrangère)
- **inversedBy** dans l'entité **propriétaire**. La table (SQL) correspondante à l'entité propriétaire contiendra la **clé étrangère.**

Dans notre exemple, si on veut que **Salarie** soit l'entité inverse, alors on va rajouter le code suivant dans l'entité Salarié :

```
/**
 * @ORM\OneToOne(targetEntity="App\Entity\Badge", mappedBy="salarie")
 */
// salarie sera le nom de l'attribut dans l'entité Badge
private $badge ;
```

Et dans l'entité **Badge**, donc considérée l'entité **propriétaire** (clé étrangère) :

```
/**
 * @ORM\OneToOne(targetEntity="App\Entity\Salarie", inverserdBy="badge")
 */
// badge sera le nom de l'attribut dans l'entité Salarie
```

```
private $salarie ; // champs qui permet de faire lien avec l'entité Salarie.
```

ManyToOne : traduit une relation qui permet à plusieurs entités d'être liées à une entité.

Par exemple, dans le projet GSB, un secteur (One) englobe plusieurs régions (Many) et une région est rattachée à un secteur.

L'entité propriétaire est celle qui contient l'annotation ManyToOne. Elle se traduit en BDD par la table qui contiendra la clé étrangère.

Le principe de la relation ManyToOne bidirectionnelle est le même que pour OneToOne. L'entité propriétaire (ManyToOne) sera définie avec l'annotation ManytoOne et l'attribut inversedBy. L'entité liée sera définie avec l'annotation OneToMany et la propriété mappedBy.

Exemple :

On va créer l'entité Région à l'aide de la commande **php bin/console make:entity**.

L'attribut nom sera défini avec le type String.

Par contre l'attribut secteur sera défini avec le type : relation. Il faudra alors indiquer l'entité liée, dans notre cas : Secteur.

Plusieurs relations seront affichées. Il faudra choisir celle qui sera adéquate dans notre cas.

Générer une nouvelle migration et persister en base de données.

La table Region est-elle créée ?

Comment se nomme la clé étrangère ?

Insérer le code de l'entité dans le document de cours et faire une capture écran du résultat obtenu en base de données.

Travail à faire :

Créer l'entité TypePraticien puis Praticien. On souhaite une relation unidirectionnelle.
Générer une nouvelle migration.
Vérifier dans le fichier migration les requêtes SQL qui seront exécutées
Persister en base de données
Vérifier la création des deux tables dans la BDD.
Insérer le code PHP des deux entités.
Réaliser la capture écran du résultat en BDD.

ManyToMany sans attributs : traduit une relation plusieurs à plusieurs (association 1,n – 1,n)

Dans ce cas, Doctrine crée une entité intermédiaire qui est complètement invisible dans la couche métier. Par contre, au niveau de la base de données, cela se traduira par une table portant le nom des deux entités et dont la clé primaire est la concaténation des deux clés étrangères. L'annotation ManyToMany sera définie dans l'entité à partir de laquelle on souhaite atteindre l'autre entité au niveau de la couche métier.

Travail à faire :

On va traduire l'association travailler. On souhaite que la relation soit bidirectionnelle.
Modifier l'entité Visiteur pour répondre au besoin.
Générer une nouvelle migration.
Persister en BDD.
Vérifier la création de la table VisiteurRegion en BDD.
Insérer le code PHP qui a été ajouté dans l'entité Visiteur et dans l'entité Region.
Insérer la capture écran du résultat obtenu en BDD

ManyToMany avec attributs : Pour créer ce type de relation, il faut créer une entité intermédiaire avec les attributs concernées et des attributs correspondants aux relations ManyToOne vers les entités concernées par la relation ManyToMany.

Doctrine crée un attribut id pour référencer les tuples. Cela ne correspond pas tout à fait à l'association N-N porteuse de propriétés mais facilite les requêtes à la BDD et les opérations CRUD. Il est possible de supprimer cet id dans l'entité et de faire en sorte que les attributs du ManyToOne soient des clés primaires.

Exemple :

On va créer la relation PraticienSpecialite qui traduit l'association posseder dans le MCD de notre projet GSB. L'attribut praticien traduira la relation ManyToOne vers l'entité Praticien et l'attribut traduira la relation ManyToOne vers l'entité Specialite. Ensuite, on définira les attributs diplome et coef_prescription.

Travail à faire :

Insérer le code PHP de l'entité PraticienSpecialite
Réaliser une migration puis persister en BDD.
Insérer une capture écran du résultat obtenu dans la BDD.

On souhaite que la table praticien_specialite ait comme clé primaire la concaténation des clés primaires praticien_id et specialite_id.

Travail à faire :

Dans l'entité PraticienSpecialite, ajouter l'annotation @ORM\Id avant les annotations @ManyToOne.
Supprimer l'annotation @ORM\Id qui définit l'attribut id.
Réaliser une migration puis persister en BDD.
Insérer une capture écran du résultat en BDD.

Il est possible de garder l'id généré automatiquement par Doctrine d'une relation ManytoMany. L'inconvénient réside dans le fait que l'unicité des tuples ne peut plus être assurée par la contrainte de clé primaire générée par la concaténation des clés étrangères et doit donc être réalisée dans la couche métier. Si on souhaite néanmoins que cette unicité soit assurée par la BDD, il faudrait ajouter dans l'entité la contrainte d'unicité sur les deux attributs.

Travail à faire :

Dans l'entité PraticienSepcialite, dans l'annotation de l'entité, ajouter ceci :

```
/**
 * @ORM\Entity(repositoryClass=PraticienSpecialiteRepository::class)
 * @UniqueEntity(
 *     fields={"praticien", "specialite"},
 *     message="Cet enregistrement existe déjà"
 *)
```

Réaliser une migration puis persister en BDD.
Insérer une capture écran du résultat en BDD.
Insérer dans la table TypePraticien l'enregistrement suivant :

```
code : MH, libelle : Médecin Hospitalier, Lieu : Hopital ou Clinique
```

Insérer le praticien qui a les caractéristiques suivantes dans la table Praticien :

```
Nom : Notini,Prénom : Alain, Adresse : 114 r Authie, CP : 85000,
Ville : LA ROCHE SUR YON, coef_notoriete : 290.03, praticien_id : 1
```

Insérer le tuple suivant dans la table Specialite :

```
Code : ACP, libelle : Anatomie et Cytologie Pathologique
```

Insérer le tuple suivant dans la table praticien_specialite :

```
praticien id : 1, specialite id : 1, diplome : DES, coef_prescription : 15
```

Insérer à nouveau ce tuple. Que se passe-t-il ?

4. Générer les objets depuis une base de données existante

Il est possible également d'effectuer du reverse engineering, c'est à dire de créer les entités depuis une base de données existante.

```
php bin/console doctrine:mapping:import "App\Entity" annotation --path=src/Entity
```

Cette commande va générer autant d'entités que de tables présentes dans la base de données. Ces entités décrivent la correspondance entre les tables de la base de données et le code PHP.

Il est également possible de générer des fichiers XML qui décrivent la base de données. Il suffit de remplacer dans la commande

précédente « annotation » par « xml ».

Et pour générer les getters/setters : `php bin/console make:entity --regenerate App`

5. L'entity Manager

L'entity manager va permettre de persister les objets, à savoir exécuter toutes les opérations d'insertion, de modification ou de suppression d'un enregistrement. Un enregistrement en BDD correspond à un objet d'une entité.

5.1. Le service Doctrine 2

La persistance des données va être gérée par le service Doctrine 2. Ce service est accessible depuis le contrôleur :

```
<?php $doctrine = $this->get('doctrine') ; ?> ou
<?php $doctrine = $this->getDoctrine() ; ?>
```

Ce service gère deux choses :

- ⇨ les différentes connexions à différentes bases de données : $doctrine->getConnection($name) => Couche DBAL (Database Abstraction Layer)
- ⇨ les différents gestionnaires d'entité : EntityManager : $doctrine->getManager($name) => Couche ORM. La méthode getManager() permet de récupérer l'EntityManager.

5.2. Le service EntityManager

L'EntityManager demande à Doctrine de persister un objet, donc c'est lui qui est en charge d'exécuter les requêtes SQL qui permettent de manipuler les lignes des tables correspondantes aux entités PHP. Par contre l'EntityManager ne sait pas récupérer les entités depuis la base de données, il faut passer par les repositories. Il existe plusieurs manières de récupérer l'EntityManager.

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

Les méthodes suivantes permettent de manipuler des objets d'entité en les persistant dans la base de données.

Pour persister un objet d'entité, il faut appeler les méthodes persist() : **$em->persist($entite)** et flush() : **$em->flush()**.

`$em->persist($objetEntite)`	Prépare l'objet pour être persisté en base de données dans la table correspondante. Cela revient à exécuter une commande SQL INSERT/UPDATE mais l'opération ne sera effective en BDD que lors de l'appel de la méthode flush()
`$em->flush()`	Exécute l'opération en base de données (insert/update/delete)
`$em->detach($objetEntite)`	Annule le persist() effectué sur l'entité en argument
`$em->contains($objetEntite)`	Retourne vrai si l'entité est gérée par l'EM (persist() au préalable)
`$em->clear()`	Annule tous les persist() effectués (avant flush())
`$em->remove($objetEntite)`	Supprime la ligne correspondante à l'objet de la BDD. Effectif au prochain flush()

Travail à faire :

Créer le contrôleur Secteur.
Créer une méthode pour insérer le secteur qui aura comme libellé : Nord
Vérifier en base de données si le tuple a été enregistré.

5.3. Les repositories

Les repositories permettent de récupérer les entités (commande SQL SELECT). Il en existe un par entité. Il faut donc préciser le repository de l'entité concernée en argument.

```
$em = $this->getDoctrine()->getManager() ;
$rp = $em->getRepository('NomEntite::class');
```

Une fois le repository récupéré, on peut utiliser les méthodes définies dans le repository, par exemple find($critereRecherche) pour retourner l'entité correspondant au critère de recherche. Par exemple, pour chercher un visiteur avec son identifiant :

```
$visiteur = $rp->find($id) ; // $id contient la valeur de l'identifiant
if ($visiteur === NULL) {
      throw $this->createNotFoundException('Visiteur non trouvé');
}
else {
      return $this->render('visiteur/show.html.twig',
                  array('visiteur'=>$visiteur));
}
```

Travail à faire :

Créer la méthode show($id) qui va afficher le secteur dont l'identifiant est passé en paramètre depuis l'URL.
Créer la méthode showSecteurs() qui va permettre d'afficher tous les enregistrements de la table secteur.
Créer la méthode edit($id) qui va modifier le libellé du secteur dont l'identifiant est passé en paramètre.
Vérifier que la mise à jour dans la base de données.

Créer un autre enregistrement de secteur avec le même libellé.
Afficher la liste des enregistrements de l'entité Secteur.
Supprimer le 2ème doublon.

En résumé :

Le rôle d'un ORM est de gérer la persistance des données, c'est à dire leur stockage en base de données. L'ORM fourni par défaut avec Symfony est Doctrine (la version actuelle est 3.3.0) . C'est une bibliothèque totalement indépendante de Symfony, d'où le préfixe ORM\ des annotations.

Une entité est une simple classe du point de vue PHP complété avec des informations de mapping qui permettent d'enregistrer l'objet instancié de l'entité en base de données.

L'EntityManager est récupéré depuis le contrôleur via $this->getDoctrine()->getManager(). Il permet, grâce à des méthodes, de persister les objets d'entité en bases de données en réalisant l'équivalent des requêtes SQL (INSERT, UPDATE, DELETE).

Les repositories sont utilisées pour récupérer les données des entités depuis une base de données. On verra l'essentiel des requêtes possibles pour extraire les données depuis l'application sans passer par des requêtes SQL.