Symfony 5 : Doctrine

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Symfony

Object-Relational Mapping (lien objet-relation)

- est une couche d'abstraction à la base de données
- est une classe qui permet à l'utilisateur d'utiliser les tables d'une base de données comme des objets
- consiste à associer :
  - une ou plusieurs classes à chaque table
  - un attribut de classe à chaque colonne de la table

Symfony

Object-Relational Mapping (lien objet-relation)

- est une couche d'abstraction à la base de données
- est une classe qui permet à l'utilisateur d'utiliser les tables d'une base de données comme des objets
- consiste à associer :
  - une ou plusieurs classes à chaque table
  - un attribut de classe à chaque colonne de la table

Plusieurs ORM proposés pour chaque Langage de POO.

Symfony

Quel choix pour PHP ?

- **Doctrine**
- pdoMap
- RedBean
- FoxORM
- ...

Symfony

Doctrine ?

- un ORM pour **PHP**
- proposé en 2006 par Konsta Vesterinen (2.0 fin 2010)
- utilisé par **Symfony** depuis la version 1.3 (et autres comme Zend Framework, CodeIgniter...)
- inspiré par **Hibernate** : ORM **Java**

Symfony

Doctrine ?

- un ORM pour **PHP**
- proposé en 2006 par Konsta Vesterinen (2.0 fin 2010)
- utilisé par **Symfony** depuis la version 1.3 (et autres comme Zend Framework, CodeIgniter...)
- inspiré par **Hibernate** : ORM **Java**

Doctrine est composé de (deux) couches

- Doctrine (ORM) qui se base sur Doctrine (DBAL)
- Doctrine (DBAL) (DataBase Abstraction Layer ou couche d'abstraction de base de données) qui se base aussi sur PDO (PHP Data Objets) pour l'abstraction d'accès aux données

Symfony

Doctrine (DBAL)

- ajoute des fonctionnalités à PDO
- permet de manipuler les bases de données avec des fonctions prédéfinies (pas d'utilisation du concept objet)

Symfony

Doctrine (DBAL)

- ajoute des fonctionnalités à PDO
- permet de manipuler les bases de données avec des fonctions prédéfinies (pas d'utilisation du concept objet)

Doctrine (ORM)

- définit le lien entre DBAL et le monde objet
- permet de manipuler les éléments d'une base de données comme des objets

Symfony

Téléchargement (pas besoin avec Symfony)

- Aller dans
  `http://www.doctrine-project.org/projects/orm.html`
- Télécharger la dernière version stable

Symfony

Objectif

Ne plus écrire des requêtes **SQL**

Symfony

Objectif

Ne plus écrire des requêtes **SQL**

Si on n'a pas choisi la version complète à la création du projet, exécutez

- `composer require symfony/orm-pack`
- `composer require --dev symfony/maker-bundle`

Symfony

Préparation de la chaîne de connexion

- Allez dans le fichier `.env`
- Cherchez la ligne `DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7`
- Remplacez la par `DATABASE_URL=mysql://root:@127.0.0.1:3308/courssymfony?serverVersion=5.7` puis enregistrez

Symfony

Préparation de la chaîne de connexion

- Allez dans le fichier `.env`
- Cherchez la ligne `DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7`
- Remplacez la par `DATABASE_URL=mysql://root:@127.0.0.1:3308/courssymfony?serverVersion=5.7` puis enregistrez

Pour créer la base de données

Exécuter la commande `php bin/console doctrine:database:create`
ou `php bin/console d:d:c`

Symfony

Préparation de la chaîne de connexion

- Allez dans le fichier `.env`
- Cherchez la ligne `DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7`
- Remplacez la par `DATABASE_URL=mysql://root:@127.0.0.1:3308/courssymfony?serverVersion=5.7` puis enregistrez

Pour créer la base de données

Exécuter la commande `php bin/console doctrine:database:create`
ou `php bin/console d:d:c`

Résultat

```
Created database `courssymfony` for connection named default
```

Symfony

Définition

- correspond à une table d'une base de données relationnelle
- est un objet contenant quelques informations indispensables pour le mapping (faire le lien) avec la base de données

Symfony

Définition

- correspond à une table d'une base de données relationnelle
- est un objet contenant quelques informations indispensables pour le mapping (faire le lien) avec la base de données

Informations indispensables : les annotations

- permettent de décrire les méta-données de l'entité
- sont des commentaires spéciaux (qui peuvent être générés par **Symfony** sans les écrire)

Symfony

3 étapes pour créer ou modifier une table associée à une entité

- créer ou modifier une entité
- créer une migration $\Rightarrow$ générer le script **SQL**
- appliquer la migration $\Rightarrow$ exécuter le script

Symfony

Pour créer une entité

Exécuter la commande `php bin/console make:entity`

Symfony

Pour créer une entité

Exécuter la commande `php bin/console make:entity`

Répondre aux questions suivantes

- `Class name of the entity to create or update` par `Personne`
- `New property name` par `nom`
- `Field type` par `string`
- `Can this field be null in the database (nullable)` par `no`
- Refaire la même chose pour `prenom` et ensuite pour un attribut `sexe` de longueur 1

Symfony

Remarques

- Les types Doctrine sont sensibles à la casse
- La liste complète des types :
  https://www.doctrine-project.org/projects/doctrine-orm/en/current/reference/basic-mapping.html#doctrine-mapping-types

L'attribut `$_id` sera générée automatiquement

```php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="
    App\Repository\
    PersonneRepository")
 */
class Personne
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

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

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

    /**
     * @ORM\Column(type="string",
        length=1)
     */
    private $sexe;


    // + les getters et setters
}
```

L'attribut `$_id` sera générée automatiquement

```php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="
    App\Repository\
    PersonneRepository")
 */
class Personne
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

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

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

    /**
     * @ORM\Column(type="string",
        length=1)
     */
    private $sexe;


    // + les getters et setters
}
```

Allez aussi vérifier la création de `PersonneRepository` dans `src/Repository`

Symfony

Les différents types de Doctrine

Annotation	**désignation**
`@ORM\Entity`	marque qu'une classe PHP est une entité
`@ORM\Table`	décrit la table d'une entité persistante
`@ORM\Column`	définit les caractéristiques d'une colonne
`@ORM\Id`	marque l'identifiant de l'entité
`@ORM\GeneratedValue`	utilisée pour générer des identifiants annotés par @Id
`@ORM\OneToOne`	entité en relation avec une seule entité
`@ORM\OneToMany`	entité en relation avec plusieurs entités
`@ORM\ManyToMany`	entités en relation avec plusieurs entités

Symfony

Les différents types de Doctrine

Annotation	désignation
`@ORM\Entity`	marque qu'une classe PHP est une entité
`@ORM\Table`	décrit la table d'une entité persistante
`@ORM\Column`	définit les caractéristiques d'une colonne
`@ORM\Id`	marque l'identifiant de l'entité
`@ORM\GeneratedValue`	utilisée pour générer des identifiants annotés par @Id
`@ORM\OneToOne`	entité en relation avec une seule entité
`@ORM\OneToMany`	entité en relation avec plusieurs entités
`@ORM\ManyToMany`	entités en relation avec plusieurs entités

La liste complète d'annotations

https://www.doctrine-project.org/projects/doctrine-orm/en/current/reference/annotations-reference.html

Symfony

Les annotations **Doctrine 2** peuvent avoir des attributs.

Symfony

Les annotations **Doctrine 2** peuvent avoir des attributs.

`@ORM\Entity`

- `repositoryClass` :
  - il permet de récupérer les entités depuis la base de données
  - il a comme valeur le nom du namespace complet du repository
  - le nom du repository est composé du nom de l'entité + `Repository` (pour notre exemple : `PersonneRepository`
- `readonly` : précise que cette entité est en lecture seule

`@ORM\Table`

- `name` : nom de la table
- `indexes` : tableau d'annotations `@index`

`@ORM\column`

- `type` : nom du type Doctrine (obligatoire)
- `name` : nom de la colonne
- `length` : longueur pour les chaînes de caractère
- `unique` : pour indiquer l'unicité des valeurs de la colonnes
- `nullable` : pour indiquer si la valeur `null` est acceptée

`@ORM\GeneratedValue`

`strategy` : nom de la stratégie (`AUTO`, `NONE`...)

Symfony

La liste complète des attributs

https://www.doctrine-project.org/projects/doctrine-orm/en/current/reference/basic-mapping.html#property-mapping

Symfony

Création d'une table à partir d'une entité

- Exécutez la commande `php bin/console make:migration` pour générer le script de création de la table
- Vérifiez le script **SQL** généré dans `src/Migrations`
- Pour créer les tables, exécutez `php bin/console doctrine:migrations:migrate` (ou `php bin/console d:m:m`)

Symfony

Création d'une table à partir d'une entité

- Exécutez la commande `php bin/console make:migration` pour générer le script de création de la table
- Vérifiez le script **SQL** généré dans `src/Migrations`
- Pour créer les tables, exécutez `php bin/console doctrine:migrations:migrate` (ou `php bin/console d:m:m`)

Vérifier la création de la table avec la console **MySQL** ou `phpMyAdmin`

Symfony

Modification d'une entité

- Ajouter un attribut
- Modifier le type d'un attribut
- Supprimer un attribut
- Ajouter/Modifier/Supprimer une/des contrainte(s) sur les attributs

Symfony

Exemple

Supprimons l'attribut `sexe` de la classe `Personne` ainsi que les getter et setter.

Symfony

Exemple

Supprimons l'attribut `sexe` de la classe `Personne` ainsi que les getter et setter.

Pour régénérer la table dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Symfony

Exemple

Supprimons l'attribut `sexe` de la classe `Personne` ainsi que les getter et setter.

Pour régénérer la table dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Vérifier les modifications avec la console `MySQL` ou `phpMyAdmin`

Symfony

Pour ajouter un nouvel attribut, on peut

- exécuter la commande `php bin/console make:entity`
- préciser le nom d'une entité existante
- préciser les nouveaux attributs (comme dans le cas d'une création)
- Faire la migration

Symfony

Comment ça marche avec **Doctrine** ?

- Pour la lecture, on utilise le `PersonneRepository`
- Pour l'écriture, on utilise `EntityManager`

Symfony

Entity Manager

- un service **Doctrine** (**Doctrine** est un service **Symfony**)
- permettant la manipulation de nos entités

Symfony

Entity Manager

- un service **Doctrine** (**Doctrine** est un service **Symfony**)
- permettant la manipulation de nos entités

Le service **Doctrine**

- `$doctrine = $this->get('doctrine');` ou
- `$doctrine = $this->getDoctrine();` (un raccourci)

Symfony

Le service **Entity Manager**

- `$em = $this->getDoctrine()->getManager();` ou
- `$em = $this->get('doctrine.orm.entity_manager');`
- ou en injectant le service `EntityManagerInterface` dans une méthode

Symfony

Le service **Entity Manager**

- `$em = $this->getDoctrine()->getManager();` ou
- `$em = $this->get('doctrine.orm.entity_manager');`
- ou en injectant le service `EntityManagerInterface` dans une méthode

Le service **Repository**

- `$em = $this->getDoctrine()->getRepository(EntityName::class);` ou
- ou en injectant le service `EntityNameRepository` dans une méthode

Symfony

Le service **Entity Manager**

- `$em = $this->getDoctrine()->getManager();` ou
- `$em = $this->get('doctrine.orm.entity_manager');`
- ou en injectant le service `EntityManagerInterface` dans une méthode

Le service **Repository**

- `$em = $this->getDoctrine()->getRepository(EntityName::class);` ou
- ou en injectant le service `EntityNameRepository` dans une méthode

Pour tester, créons un contrôleur `PersonneController`

Symfony

Contenu de `PersonneController.php`

```php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\
   AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class PersonneController extends AbstractController
{
    /**
     * @Route("/personne", name="personne")
     */
    public function index()
    {
        return $this->render('personne/index.html.twig', [
            'controller_name' => 'PersonneController',
        ]);
    }
}
```

Symfony

Pour `personne/index.html.twig`, considérons le contenu suivant

```
{% extends 'base.html.twig' %}

{% block title %}Hello PersonneController!{% endblock %}

{% block body %}
        <h1>Hello
                {{ controller_name }}!
        </h1>
        {% if personne is defined %}
                Personne
                {{ adjectif }} :
                {{ personne.id }}
                {{ personne.prenom }}
                {{ personne.nom }}
        {% endif %}
{% endblock %}
```

Symfony

Pour ajouter un tuple dans la table `Personne`

```
/**
 * @Route("/personne/add", name="personne_add")
*/
public function addPersonne()
{
    $personne = new Personne();
    $personne->setNom('Wick');
    $personne->setPrenom('John');
    $entityManager = $this->getDoctrine()->getManager();
    $entityManager->persist($personne);
    $entityManager->flush();
    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'ajoutée'
    ]);
}
```

Symfony

On peut aussi injecter le gestionnaire d'entité dans l'action

```
/**
 * @Route("/personne/add", name="personne_add")
*/
public function addPersonne(EntityManagerInterface $entityManager)
{
    $personne = new Personne();
    $personne->setNom('Wick');
    $personne->setPrenom('John');
    $entityManager->persist($personne);
    $entityManager->flush();
    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'ajoutée'
    ]);
}
```

Le namespace de `EntityManagerInterface`

```
use Doctrine\ORM\EntityManagerInterface;
```

Explication

- `$entityManager->persist($personne);` : informe **Doctrine** que l'on veut ajouter cet objet dans la base de données.
- `$em->flush();` : permet d'exécuter la requête et d'envoyer tout ce qui a été persisté avant à la base de données.

Explication

- `$entityManager->persist($personne);` : informe **Doctrine** que l'on veut ajouter cet objet dans la base de données.
- `$em->flush();` : permet d'exécuter la requête et d'envoyer tout ce qui a été persisté avant à la base de données.

Utilisez le profiler pour mieux comprendre le fonctionnement

- Allez à la page `http://localhost:8000/_profiler/14d964?panel=db`
- Vérifiez la présence de 3 Queries

Explication

- `$entityManager->persist($personne);` : informe **Doctrine** que l'on veut ajouter cet objet dans la base de données.
- `$em->flush();` : permet d'exécuter la requête et d'envoyer tout ce qui a été persisté avant à la base de données.

Utilisez le profiler pour mieux comprendre le fonctionnement

- Allez à la page `http://localhost:8000/_profiler/14d964?panel=db`
- Vérifiez la présence de 3 Queries

Symfony utilise les transactions pour les opérations sur une base de données

- `START TRANSACTION`
- `INSERT INTO ...`
- `COMMIT`

Pour vérifier si les valeurs sont valides avant insertion, on peut utiliser `ValidatorInterface`

```
public function addPersonne(EntityManagerInterface $entityManager,
   ValidatorInterface $validator)
{
    $personne = new Personne();
    $personne->setNom('Wick');
    $personne->setPrenom('John');
    $errors = $validator->validate($personne);
    if (count($errors) > 0) {
        return new Response((string) $errors, 400);
    }
    $entityManager->persist($personne);
    $entityManager->flush();
    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'ajoutée'
    ]);
}
```

Le namespace de `ValidatorInterface`

```
use Symfony\Component\Validator\Validator\ValidatorInterface;
```

Symfony

Si on n'a pas choisi la version complète à la création du projet, exécutez

- `composer require symfony/validator`

Symfony

Contraintes vérifiées par **Symfony**

- `Type`
- `NotNull`
- `UniqueEntity`
- `Length`

Symfony

Quatre méthodes prédéfinies pour la recherche

- `find` : cherche et retourne un seul tuple selon la clé primaire
- `findOneBy` : cherche et retourne un seul tuple selon les colonnes données en paramètre dans un tableau associatif
- `findBy` : cherche et retourne plusieurs tuples selon les colonnes données en paramètre dans un tableau associatif
- `findAll` : retourne tous les tuples de la table.

Symfony

Exemple avec `find` (la méthode est à placer après `addPersonne`)

```
/**
 * @Route("/personne/{id}", name="personne_show")
 */
public function showPersonne(int $id, PersonneRepository
   $personneRepository)
{
    $personne = $personneRepository->find($id);

    if (!$personne) {
        throw $this->createNotFoundException(
            'Personne non trouvée avec  l\'id ' . $id
        );
    }

    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'recherchée'
    ]);
}
```

Symfony

Rien à changer dans `personne/index.html.twig`

```
{% extends 'base.html.twig' %}

{% block title %}Hello PersonneController!{% endblock %}

{% block body %}
        <h1>Hello
                {{ controller_name }}!
        </h1>
        {% if personne is defined %}
                Personne
                {{ adjectif }} :
                {{ personne.id }}
                {{ personne.prenom }}
                {{ personne.nom }}
        {% endif %}
{% endblock %}
```

Exemple avec `findOneBy` (la méthode est à placer après `showPersonne`)

```
/**
 * @Route("/personne/{nom}/{prenom}", name="personne_show_one")
 */
public function showPersonneByNomAndPrenom(string $nom, string $prenom,
    PersonneRepository $personneRepository)
{
    $personne = $personneRepository->findOneBy([
        "nom" => $nom,
        "prenom" => $prenom
    ]);

    if (!$personne) {
        throw $this->createNotFoundException('Personne non trouvée');
    }

    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'recherchée'
    ]);
}
```

Symfony

Exemple avec `findAll` (la méthode est à placer après `addPersonne`)

```
/**
 * @Route("/personne/show", name="personne_show_all")
 */
public function showAllPersonne(PersonneRepository $personneRepository)
{
    $personnes = $personneRepository->findAll();

    if (!$personnes) {
        throw $this->createNotFoundException('La table est vide');
    }

    return $this->render('personne/show.html.twig', [
        'controller_name' => 'PersonneController',
        'personnes' => $personnes,
    ]);
}
```

Symfony

Contenu de `show.html.twig`

```
{% extends 'base.html.twig' %}

{% block title %}Hello PersonneController!
{% endblock %}

{% block body %}

  <h1>Hello {{ controller_name }}!</h1>
  <ul>
    {% for personne in personnes %}
      <li>{{ personne.prenom }} {{ personne.nom }}</li>
    {% endfor %}
  </ul>

{% endblock %}
```

Symfony

Pour modifier une `personne`, il faut la récupérer avant avec `personneRepository`

```
/**
 * @Route("/personne/edit/{id}", name="personne_update")
 */
public function updatePersonne(int $id, EntityManagerInterface
   $entityManager)
{
    $personne = $entityManager->getRepository(Personne::class)->find(
       $id);
    if (!$personne) {
        throw $this->createNotFoundException(
            'Personne non trouvée avec  l\'id ' . $id
        );
    }
    $personne->setNom('Travolta');
    $entityManager->flush();
    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'modifiée'
    ]);
}
```

Symfony

Symfony nous permet de récupérer l'objet `personne` dont l'identifiant est passé en paramètre dans la barre d'adresse

```
/**
 * @Route("/personne/edit/{id}", name="personne_update")
 */
public function updatePersonne(Personne $personne,
   EntityManagerInterface $entityManager)
{
    if (!$personne) {
        throw $this->createNotFoundException(
            'Personne non trouvée avec  l\'id ' .  $personne->id
        );
    }
    $personne->setNom('Abruzzi');
    $entityManager->flush();
    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'modifiée'
    ]);
}
```

Symfony

Pour supprimer une `personne`**, il faut aussi la récupérer avant avec** `personneRepository`

```
/**
 * @Route("/personne/delete/{id}", name="personne_delete")
 */
public function deletePersonne(int $id, EntityManagerInterface
   $entityManager)
{
    $personne = $entityManager->getRepository(Personne::class)
       ->find($id);
    if (!$personne) {
        throw $this->createNotFoundException(
            'Personne non trouvée avec  l\'id ' . $id
        );
    }
    $entityManager->remove($personne);
    $entityManager->flush();
    return $this->redirectToRoute("personne_show_all");
}
```

Symfony

Autres méthodes

- `clear()` : annule tous les `persist` effectués par l'`EntityManager`
- `detach($entity)` : annule le `persist` effectué par l'`EntityManager` sur `$entity`
- `refresh($entity)` : remet à jour l'entité en argument par les valeurs de la base de données. Les nouvelles modifications sur cette entité seront perdues.
- `contains($entity)` : retourne `true` si `$entity` est gérée par `EntityManager`.

Symfony

Repository

- une classe PHP
- contenant les méthodes de récupération de données relatives à nos entités
- pouvant être utilisé pour
  - définir des nouvelles méthodes
  - personnaliser des méthodes existantes

Symfony

La méthode `findBy()`

Elle peut prendre plusieurs paramètres

- un tableau de contraintes pour le `where`, obligatoire
- un tableau de contraintes pour le `orderBy`, par défaut
- une valeur pour `limit`, par défaut
- une valeur pour `offset`, par défaut

Exemple avec `findBy()`

```
/**
 * @Route("/personne/show/{nom}/{prenom}/{number}", name="
   personne_show_some")
 */
public function showSomePersonne(string $nom, string $prenom, int
   $number, PersonneRepository $personneRepository)
{
    $personnes = $personneRepository->findBy(
        [
            "nom" => $nom,
            "prenom" => $prenom
        ],
        ["nom" => "ASC"],
        $number,
        1
    );
    if (!$personnes) {
        throw $this->createNotFoundException('Aucun résultat trouvé');
    }
    return $this->render('personne/show.html.twig', [
        'controller_name' => 'PersonneController',
        'personnes' => $personnes,
    ]);
}
```

Symfony

Les méthodes magiques

- `findByAttribut($valeur)` : retourne un tableau de tous les tuples dont `Attribut` a comme valeur `$valeur)`
- `findOneByAttribut($valeur)` : retourne un seul tuple dont `Attribut` a comme valeur `$valeur`

Symfony

Les méthodes magiques

- `findByAttribut($valeur)` : retourne un tableau de tous les tuples dont `Attribut` a comme valeur `$valeur)`
- `findOneByAttribut($valeur)` : retourne un seul tuple dont `Attribut` a comme valeur `$valeur`

Pour notre entité `Personne`, on peut avoir

- `findByNom()`
- `findByPrenom()`
- `findOneByNom()`
- `findOneByPrenom()`

Symfony

Query Builder

- Dans la classe `PersonneRepository`, on définit une méthode qui
  - utilise un objet `QueryBuilder` : on l'obtient avec la méthode `createQueryBuilder()` de l'`EntityManager` et on l'utilise pour construire la requête
  - récupère l'objet `Query` de `QueryBuilder`
  - récupère les résultats de la `Query`
- retourne le résultat

Symfony

Définissons notre méthode dans la classe `PersonneRepository`

```
public function findOneByNomAndPrenom(string $nom,
   string $prenom)
{
    $queryBuilder = $this->createQueryBuilder('p')
        ->where('p.nom = :nom')
        ->setParameter('nom', $nom)
        ->andWhere('p.prenom = :prenom')
        ->setParameter('prenom', $prenom);
    $query = $queryBuilder->getQuery();
    $result = $query->setMaxResults(1)->
       getOneOrNullResult();
    return $result;
}
```

Symfony

Modifions `showPersonneByNomAndPrenom` **dans** `PersonneController`

```
/**
 * @Route("/personne/{nom}/{prenom}", name="personne_show_one")
 */
public function showPersonneByNomAndPrenom(string $nom, string $prenom,
    PersonneRepository $personneRepository)
{
    $personne = $personneRepository->findOneByNomAndPrenom($nom,
       $prenom);

    if (!$personne) {
        throw $this->createNotFoundException('Personne non trouvée');
    }

    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'recherchée'
    ]);
}
```

Symfony

Autres méthodes de `Query`

- `getResult()` : Exécute la requête et retourne le résultat sous forme d'un tableau d'objets (même quand il s'agit d'un seul objet)
- `getArrayResult()` : Exécute la requête et retourne le résultat sous forme d'un tableau de tableaux
- `getScalarResult()` : Exécute la requête et retourne le résultat sous forme d'une valeur (à utiliser lorsque la requête retourne une unique valeur)
- `getOneOrNullResult()` : Exécute la requête et retourne un seul objet ou une valeur `null`
- Plusieurs autres : `getSingleResult()`, `getSingleScalarResult()`...
- `execute()` : à utiliser pour exécuter des requêtes `insert`, `update`, `delete` ou `select`

Symfony

DQL

- Langage de requêtes adapté à **Doctrine**
- Contrairement à `QueryBuilder`, **DQL** permet d'écrire des requêtes sous forme de chaînes de caractères

Symfony

DQL

- Langage de requêtes adapté à **Doctrine**
- Contrairement à `QueryBuilder`, **DQL** permet d'écrire des requêtes sous forme de chaînes de caractères

Remarques

- Pas besoin de `QueryBuilder` pour construire les requêtes
- Par contre, on doit passer par la méthode `createQuery()`
- Et on a toujours besoin de `Query` pour récupérer les résultats

Symfony

Modifions la méthode `findOneByNomAndPrenom` **de la** `PersonneRepository` **et utilisons** `DQL`

```
public function findOneByNomAndPrenom(string $nom, string
   $prenom)
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery(
            'SELECT p
            FROM App\Entity\Personne p
            WHERE p.nom = :nom
            and p.prenom = :prenom'
    )->setParameter('nom', $nom)
        ->setParameter('prenom', $prenom);
    $result = $query->setMaxResults(1)->getOneOrNullResult();
    return $result;
}
```

Symfony

La méthode `showPersonneByNomAndPrenom` de `PersonneController` reste inchangée

```
/**
 * @Route("/personne/{nom}/{prenom}", name="personne_show_one")
 */
public function showPersonneByNomAndPrenom(string $nom, string $prenom,
    PersonneRepository $personneRepository)
{
    $personne = $personneRepository->findOneByNomAndPrenom($nom,
       $prenom);

    if (!$personne) {
        throw $this->createNotFoundException('Personne non trouvée');
    }

    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'recherchée'
    ]);
}
```

Symfony

Tester une requête DQL avec la Console

Exécuter `php bin/console doctrine:query:dql "requêteDQL"`

Symfony

Tester une requête DQL avec la Console

Exécuter `php bin/console doctrine:query:dql "requêteDQL"`

Exemple

Exécuter

```
php bin/console doctrine:query:dql "SELECT p FROM
App\Entity\Personne p"
```

Symfony

Remarques

- La requête **DQL** précédente permet de sélectionner un objet
- Il est tout de même possible de sélectionner seulement quelques attributs d'un objet
- Dans ce cas, le résultat est un tableau contenant les champs sélectionnés
- Cependant, il est impossible de modifier (ou supprimer) les valeurs de ces attributs sélectionnées

Symfony

Modifions la méthode `findOneByNomAndPrenom` de la `PersonneRepository` et utilisons `SQL`

```
public function findOneByNomAndPrenom(string $nom, string
   $prenom)
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->getConnection()->prepare(
        'SELECT *
            FROM personne
            WHERE nom = :nom
            and prenom = :prenom'
    );
    $query->execute(array('nom' => $nom, "prenom" => $prenom));
    $result = $query->fetch();
    return $result;
}
```

Symfony

Quatre (ou trois) relations possibles

- `OneToOne` : chaque objet d'une première classe est en relation avec un seul objet de la deuxième classe
- `OneToMany` : chaque objet d'une première classe peut être en relation avec plusieurs objets de la deuxième classe (la réciproque est `ManyToOne`)
- `ManyToMany` : chaque objet d'une première classe peut être en relation avec plusieurs objets de la deuxième classe et inversement

Symfony

Pour la suite

- Créons une entité `Adresse` avec la commande `php bin/console make:entity`
- Cette entité a trois attributs :
  - `rue` (`string` de taille 30),
  - `codePostal` (`string` de taille 5) et
  - `ville` (`string` de taille 30)

Symfony

Contenu de l'entité `Adresse`

```php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="
    App\Repository\
    AdresseRepository")
 */
class Adresse
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string",
        length=30)
     */
    private $rue;

    /**
     * @ORM\Column(type="string",
        length=5)
     */
    private $codePostal;

    /**
     * @ORM\Column(type="string",
        length=30)
     */
    private $ville;
```

Symfony

Pour ajouter `Adresse` dans `Personne`

- exécutez la commande `php bin/console make:entity`
- répondez à `Class name of the entity to create or update` par `Personne`
- répondez à `New property name` par `adresse`
- répondez à `Field type` par `OneToOne`
- répondez à `What class should this entity be related to?` par `Adresse`
- répondez à `Is the Personne.adresse property allowed to be null (nullable)?` par `yes`
- répondez à `Do you want to add a new property to Adresse so that you can access/update Personne` par `no`
- cliquez sur `entrée` pour répondre à `Add another property?`

Nouveau contenu de `Personne`

```php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\PersonneRepository")
 */
class Personne
{
   ...
   /**
    * @ORM\OneToOne(targetEntity="App\Entity\Adresse", cascade={"remove
       "})
    */
   private $adresse;
   ...
}
```

Nouveau contenu de `Personne`

```php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\PersonneRepository")
 */
class Personne
{
   ...
   /**
    * @ORM\OneToOne(targetEntity="App\Entity\Adresse", cascade={"remove
       "})
    */
   private $adresse;
   ...
}
```

Notation

- `Personne` : entité propriétaire
- `Adresse` : entité inverse

Symfony

```
/**
 * @ORM\OneToOne(targetEntity="App\Entity\Adresse", cascade={"remove
    "})
 */
```

Symfony

```
/**
 * @ORM\OneToOne(targetEntity="App\Entity\Adresse", cascade={"remove
    "})
 */
```

Explication

- `targetEntity` : namespace complet vers l'entité liée.
- `cascade` : permet de `cascader` les opérations comme `persist`, `remove` qu'on peut faire de l'entité propriétaire à l'entité inverse.

Symfony

```
/**
 * @ORM\OneToOne(targetEntity="App\Entity\Adresse", cascade={"remove
    "})
 */
```

Explication

- `targetEntity` : namespace complet vers l'entité liée.
- `cascade` : permet de `cascader` les opérations comme `persist`, `remove` qu'on peut faire de l'entité propriétaire à l'entité inverse.

On peut aussi ajouter

```
* @ORM\JoinColumn(nullable=false)
```

- Pour indiquer que chaque personne doit avoir une adresse. Par défaut, c'est facultatif.

Symfony

Pour régénérer la table dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Symfony

Pour régénérer la table dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Vérifier les modifications avec la console `MySQL` ou `phpMyAdmin`

Modifions `addPersonne` pour ajouter une personne avec une adresse

```
/**
 * @Route("/personne/add", name="personne_add")
*/
function addPersonne(EntityManagerInterface $entityManager)
{
    $adresse = new Adresse();
    $adresse->setRue('paradis');
    $adresse->setVille('Marseille');
    $adresse->setCodePostal('13015');
    $entityManager->persist($adresse);
    $personne = new Personne();
    $personne->setNom('Wick');
    $personne->setPrenom('John');
    $personne->setAdresse($adresse);
    $entityManager->persist($personne);
    $entityManager->flush();
    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'ajoutée'
    ]);
}
```

Symfony

Modifions l'entité `Personne` pour éviter de persister les adresses avant

```
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\
    PersonneRepository")
 */
class Personne
{
   ...
   /**
    * @ORM\OneToOne(targetEntity="App\Entity\Adresse", cascade
       ={"remove", "persist"})
    */
   private $adresse;
   ...
}
```

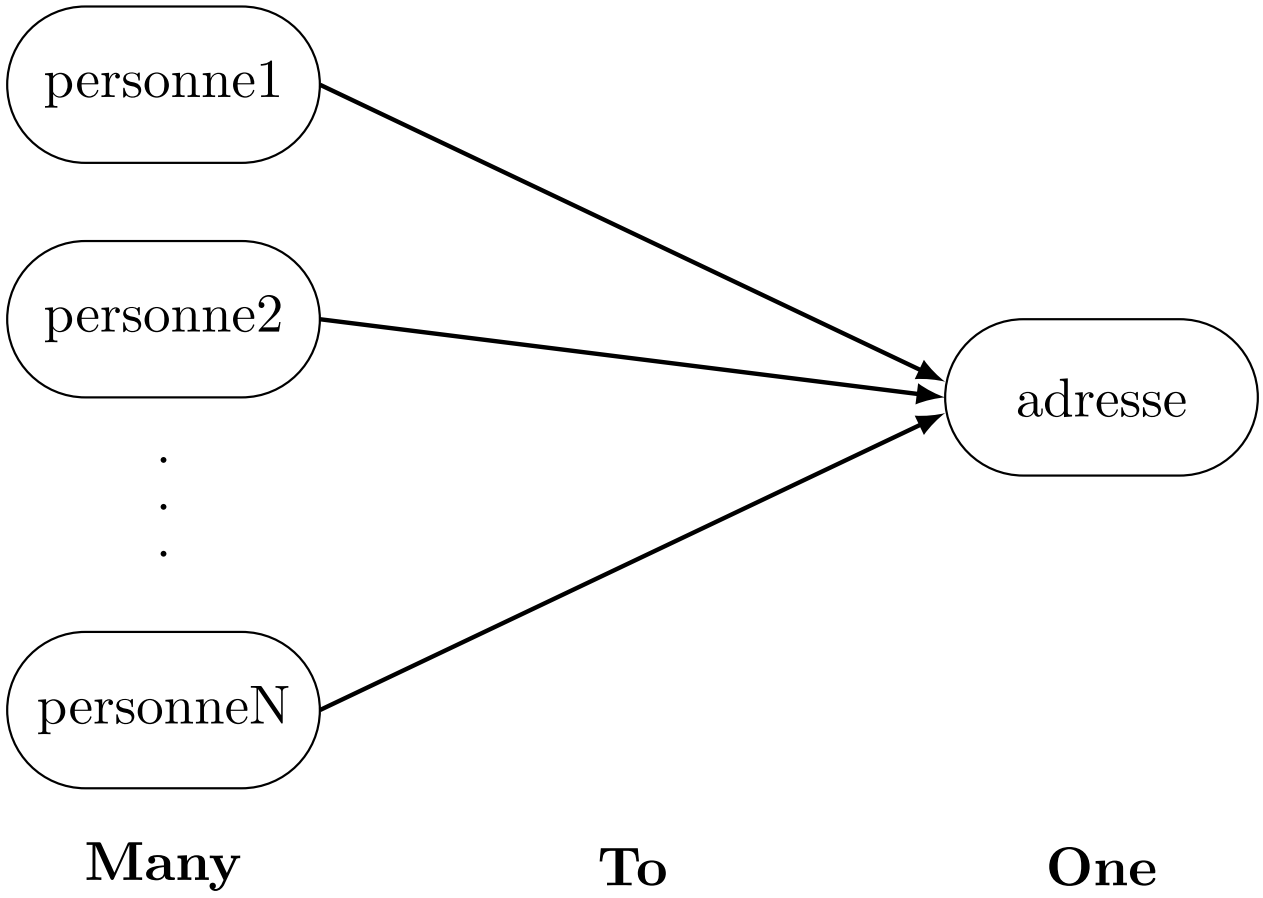
Ainsi pour ajouter une personne

```
/**
 * @Route("/personne/add", name="personne_add")
*/
function addPersonne(EntityManagerInterface $entityManager)
{
    $adresse = new Adresse();
    $adresse->setRue('paradis');
    $adresse->setVille('Marseille');
    $adresse->setCodePostal('13015');
    $personne = new Personne();
    $personne->setNom('Wick');
    $personne->setPrenom('John');
    $personne->setAdresse($adresse);
    $entityManager->persist($personne);
    $entityManager->flush();
    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => 'ajoutée'
    ]);
}
```

Symfony

Exemple

Si plusieurs personnes pouvaient avoir la même adresse.



Symfony

Il faut juste changer

```
/**
 * @ORM\ManyToOne(targetEntity=Adresse::class,
    cascade={"remove", "persist"})
 * @ORM\JoinColumn(nullable=false)
 */
```

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

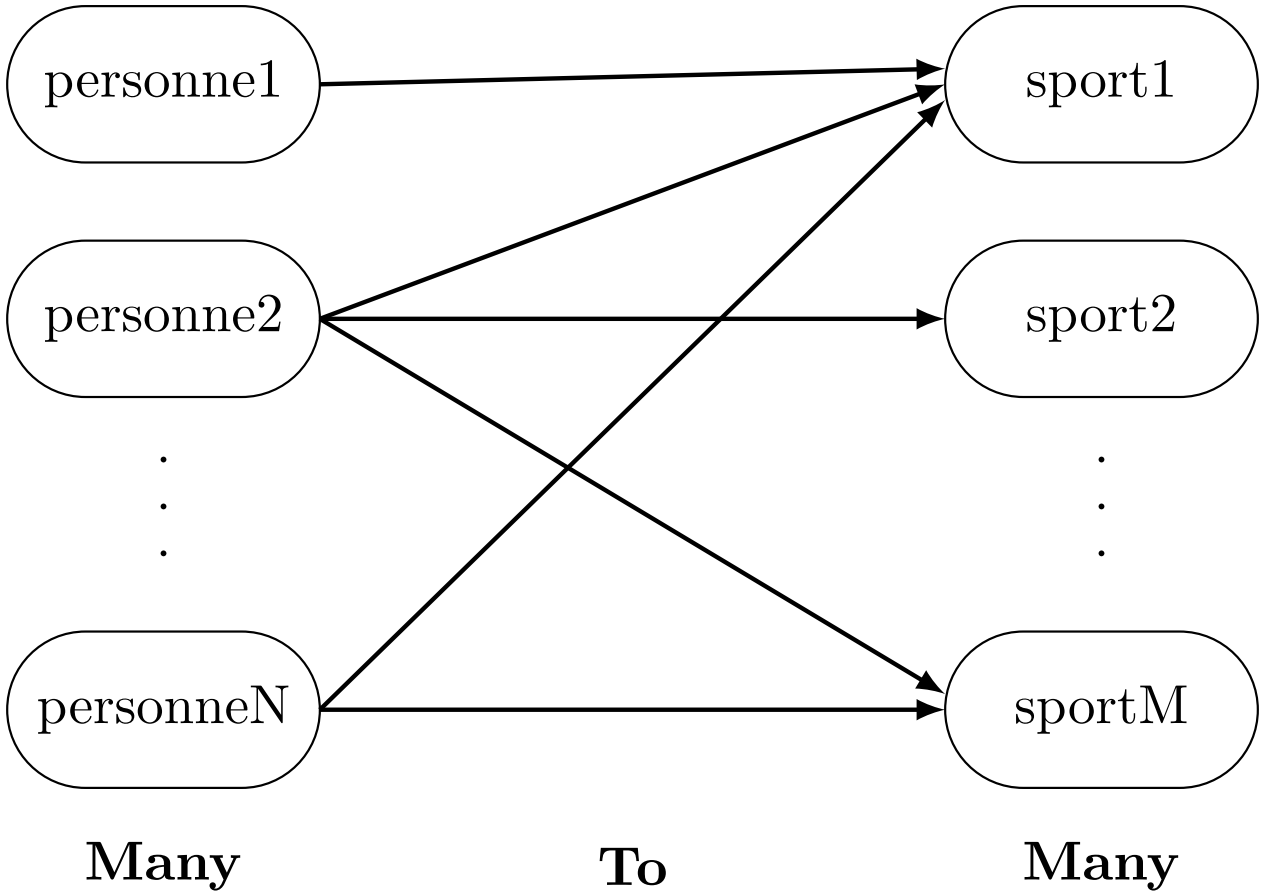
Vérifier les modifications avec la console `MySQL` ou `phpMyAdmin`

Pour tester

```
/* adresse */
$adresse = new Adresse();
$adresse->setRue('paradis');
$adresse->setVille('Marseille');
$adresse->setCodePostal('13015');

/* première personne */
$personne = new Personne();
$personne->setNom('Cohen');
$personne->setPrenom('Sophie');
$personne->setAdresse($adresse);

/* deuxième personne */
$personne2 = new Personne();
$personne2->setNom('Wolf');
$personne2->setPrenom('Bob');
$personne2->setAdresse($adresse);

/* persistance de données */
$entityManager->persist($personne);
$entityManager->persist($personne2);
$entityManager->flush();
```

Exemple

- Une personne peut pratiquer plusieurs sports
- Un sport peut être pratiqué par plusieurs personnes



Symfony

Démarche

- On commence par créer une entité `Sport` avec un seul attribut `name`
- On définit la relation `ManyToMany` (exactement comme pour les deux relations précédentes) soit dans `Personne` soit dans `Sport`

Contenu de l'entité `Sport`

```php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\SportRepository")
 */
class Sport
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

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

    //  + getters et setters
}
```

Symfony

Pour ajouter `Sport` dans `Personne`

- exécutez la commande `php bin/console make:entity`
- répondez à `Class name of the entity to create or update` par `Personne`
- répondez à `New property name` par `sports`
- répondez à `Field type` par `ManyToMany`
- répondez à `What class should this entity be related to?` par `Sport`
- répondez à `Is the Personne.sports property allowed to be null (nullable)?` par `yes`
- répondez à `Do you want to add a new property to Sport so that you can access/update Personne` par `no`
- cliquez sur `entrée` pour répondre à `Add another property?`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Vérifier les modifications avec la console `MySQL` ou `phpMyAdmin`

Symfony

Pour tester

```
$sport = new Sport();
$sport->setName('Football');

$sport2 = new Sport();
$sport2->setName('Tennis');

$personne = new Personne();
$personne->setNom('Dalton');
$personne->setPrenom('Jack');
$personne->addSport($sport);
$personne->addSport($sport2);

$personne2 = new Personne();
$personne2->setNom('Benamar');
$personne2->setPrenom('Karim');
$personne2->addSport($sport);

$entityManager->persist($personne);
$entityManager->persist($personne2);
$entityManager->flush();
```

Symfony

Si l'association est porteuse de données

- Par exemple : la relation (`ArticleCommande`) entre `Commande` et `Article`
- Pour chaque article d'une commande, il faut préciser la quantité commandée.

Symfony

Si l'association est porteuse de données

- Par exemple : la relation (`ArticleCommande`) entre `Commande` et `Article`
- Pour chaque article d'une commande, il faut préciser la quantité commandée.

Solution

- Créer trois entités `Article`, `Commande` et `ArticleCommande`
- Définir la relation `OneToMany` entre `Article` et `ArticleCommande`
- Définir la relation `ManyToOne` entre `ArticleCommande` et `Commande`
- La relation `OneToMany` est l'inverse de `ManyToOne`

Symfony

Remarques

- Les relations, qu'on a étudiées, sont unidirectionnelles
- C'est à dire on peut faire `$personne->getAdresse()`
- Mais on ne peut faire `$adresse->getPersonne()`

Symfony

Remarques

- Les relations, qu'on a étudiées, sont unidirectionnelles
- C'est à dire on peut faire `$personne->getAdresse()`
- Mais on ne peut faire `$adresse->getPersonne()`

Solution

Rendre les relations bidirectionnelles

Symfony

Remarques

- Les relations, qu'on a étudiées, sont unidirectionnelles
- C'est à dire on peut faire `$personne->getAdresse()`
- Mais on ne peut faire `$adresse->getPersonne()`

Solution

Rendre les relations bidirectionnelles

Avant de commencer

Supprimons tout ce qui concerne `Adresse` dans `Personne`.

Symfony

Démarche

- Exécutez la commande `php bin/console make:entity`
- `Class name of the entity to create or update` : Adresse
- `New property name` : personnes
- `Field type` : OneToMany
- `What class should this entity be related to` : Personne
- `New field name inside Personne` : adresse
- Cliquez deux fois sur `entrez`

Contenu d'`Adresse.php`

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\AdresseRepository")
 */
class Adresse
{
   ...
     /**
      * @ORM\OneToMany(targetEntity="App\Entity\Personne", mappedBy="
         adresse")
      */
     private $personnes;

     public function __construct()
     {
         $this->personnes = new ArrayCollection();
     }
   ...
}
```

`mappedBy` fait référence à l'attribut `adresse` dans la classe `Personne`

Symfony

Contenu de `Personne.php`

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\PersonneRepository")
 */
class Personne
{
    ...

    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Adresse", inversedBy="
        personnes")
     */
    private $adresse;

    ...
}
```

`inversedBy` fait référence à l'attribut `personnes` dans la classe `Adresse`

Symfony

Ainsi, on peut faire :

```
$adresse = new Adresse();
$adresse->setRue('10 rue de Lyon');
$adresse->setVille('Marseille');
$adresse->setCodePostal(13015);
$personne = new Personne();
$personne->setNom('Wick');
$personne->setPrenom('John');
$personne->setAdresse($adresse);
$adresse->getPersonnes();
```

Symfony

Exercice

Écrire un code qui permet d'insérer une personne dans la base de données avec deux adresses

- la première est une adresse qui existe déjà
- la deuxième est une nouvelle qui n'existait pas

Symfony

Trois possibilités avec l'héritage

- `SINGLE_TABLE`
- `JOINED`

Symfony

Trois possibilités avec l'héritage

- `SINGLE_TABLE`
- `JOINED`

Exemple

- Une classe mère `Personne`
- Deux classes filles `Etudiant` et `Enseignant`

Symfony

Démarche

- On commence par créer deux entités `Etudiant` et `Enseignant`
- L'entité `Etudiant` a un seul attribut `niveau` de type `string` (de longueur 30)
- L'entité `Enseignant` a un seul attribut `salaire` de type `integer`

Symfony

Pour indiquer comment transformer les classes mère et filles en tables

Il faut utiliser l'annotation `@InheritanceType`

Symfony

Pour indiquer comment transformer les classes mère et filles en tables

Il faut utiliser l'annotation `@InheritanceType`

Il faut aussi indiquer la solution choisie pour l'héritage

Dans la classe mère on ajoute

```
@ORM\InheritanceType("SINGLE_TABLE")
```

Symfony

Exemple

Et pour distinguer étudiant, enseignant et personne

- `@DiscriminatorColumn(name="type", type="string")` dans la classe mère,
- `@DiscriminatorMap("personne" = "Personne", "etudiant" = "Etudiant", "enseignant" = "Enseignant")`

Symfony

Exemple

Et pour distinguer étudiant, enseignant et personne

- `@DiscriminatorColumn(name="type", type="string")` dans la classe mère,
- `@DiscriminatorMap("personne" = "Personne", "etudiant" = "Etudiant", "enseignant" = "Enseignant")`

Dans la table `personne`, on aura une colonne `type` qui aura comme valeur soit `personne`, soit `etudiant` soit `enseignant`.

Symfony

La classe `Personne`

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\PersonneRepository")
 * @ORM\InheritanceType("SINGLE_TABLE")
 * @ORM\DiscriminatorColumn(name="type", type="string")
 * @ORM\DiscriminatorMap({"personne" = "Personne", "etudiant" = "Etudiant", "enseignant" = "
   Enseignant"})
 */
class Personne {

  // + tout le code précédent

}
```

La classe `Etudiant`

```
/**
 *
 * @ORM\Entity(repositoryClass="App\
   Repository\EtudiantRepository")
 */
class Etudiant extends Personne
{
    // le contenu ne change pas
}
```

La classe `Enseignant`

```
/**
 * @ORM\Entity(repositoryClass="App\
   Repository\EnseignantRepository")
 */
class Enseignant extends Personne
{
    // le contenu ne change pas
}
```

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Vérifier les modifications avec la console `MySQL` ou `phpMyAdmin`

Symfony

Ainsi, on peut faire :

```
$personne = new Personne();
$personne->setNom('Wick');
$personne->setPrenom('John');

$etudiant = new Etudiant();
$etudiant->setNom('Maggio');
$etudiant->setPrenom('Carol');
$etudiant->setNiveau('master');

$enseignant = new Enseignant();
$enseignant->setNom('Baggio');
$enseignant->setPrenom('Roberto');
$enseignant->setSalaire(2000);

$entityManager->persist($personne);
$entityManager->persist($etudiant);
$entityManager->persist($enseignant);
$entityManager->flush();
```

Symfony

Allons voir la base de données

- une seule table `Personne` a été créée
- cette table a les colonnes `id`, `nom`, `prenom`, `salaire`, `niveau` et `type`
- la personne `Wick John` a la valeur `null` dans `salaire` et `niveau` et la valeur `personne` dans `type`
- l'étudiant `Maggio Carol` a la valeur `null` dans `salaire` et la valeur `etudiant` dans `type`
- l'enseignant `Baggio Roberto` a la valeur `null` dans `niveau` et la valeur `enseignant` dans `type`

Symfony

Remplaçons `SINGLE_TABLE` **par** `JOINED` **dans la classe** `Personne`

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\PersonneRepository")
 * @ORM\InheritanceType("JOINED")
 * @ORM\DiscriminatorColumn(name="type", type="string")
 * @ORM\DiscriminatorMap({"personne" = "Personne", "etudiant" = "Etudiant", "enseignant" = "
   Enseignant"})
 */
class Personne
{

  // + tout le code précédent

}
```

Pas de changement `Etudiant`

```
/**
 *
 * @ORM\Entity(repositoryClass="App\
   Repository\EtudiantRepository")
 */
class Etudiant extends Personne
{
    // le contenu ne change pas
}
```

Pas de changement `Enseignant`

```
/**
 * @ORM\Entity(repositoryClass="App\
   Repository\EnseignantRepository")
 */
class Enseignant extends Personne
{
    // le contenu ne change pas
}
```

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Vérifier les modifications avec la console `MySQL` ou `phpMyAdmin`

Symfony

Allons voir la base de données

- trois tables : créées `Personne`, `Etudiant` et `Enseignant`
- une table `Personne` avec les colonnes `id`, `nom`, `prenom` et `type`
- une table `Etudiant` avec les colonnes `id` et `niveau`
- une table `Enseignant` avec les colonnes `id` et `salaire`

Symfony

Ainsi, on peut faire :

```
$personne = new Personne();
$personne->setNom('Wick');
$personne->setPrenom('John');

$etudiant = new Etudiant();
$etudiant->setNom('Maggio');
$etudiant->setPrenom('Carol');
$etudiant->setNiveau('master');

$enseignant = new Enseignant();
$enseignant->setNom('Baggio');
$enseignant->setPrenom('Roberto');
$enseignant->setSalaire(2000);

$entityManager->persist($personne);
$entityManager->persist($etudiant);
$entityManager->persist($enseignant);
$entityManager->flush();
```

Symfony

Cycle de vie d'une entité

Le cycle de vie de chaque objet d'une entité passe par trois événements principaux

- création (avec `persist()`)
- mise à jour (avec `flush()`)
- suppression (avec `remove()`)

Symfony

Une méthode `callback`

- Une méthode `callback` est une méthode qui sera appelée avant ou après un évènement survenu sur une entité
- On utilise les annotations pour spécifier quand la méthode callback sera appelée

Symfony

Une méthode `callback`

- Une méthode `callback` est une méthode qui sera appelée avant ou après un évènement survenu sur une entité
- On utilise les annotations pour spécifier quand la méthode callback sera appelée

C'est comme les triggers en SQL

Symfony

Les méthodes `callback`

- `@PrePersist` : avant qu'une nouvelle entité soit persistée.
- `@PostPersist` : après l'enregistrement de l'entité dans la base de données.
- `@PostLoad` : après le chargement d'une entité de la base de données.
- `@PreUpdate` : avant que la modification d'une entité soit enregistrée en base de données.
- `@PostUpdate` : après que la modification d'une entité est enregistrée en base de données.
- `@PreRemove` : avant qu'une entité soit supprimée de la base de donnée.
- `@PostRemove` : après qu'une entité est supprimée de la base de donnée.

Symfony

La classe `Personne`

```
class Personne{
    ...
    /**
     * @ORM\Column(name="nbrMAJ", type="integer")
     */
    private $nbrMAJ = 0;
    ...
    public function setNbrMAJ($nbrMAJ)     {
        $this->nbrMAJ = $nbrMAJ;
        return $this;
    }
    public function getNbrMAJ()     {
        return $this->nbrMAJ;
    }
}
```

Symfony

La classe `Personne`

```
class Personne{
    ...
    /**
     * @ORM\Column(name="nbrMAJ", type="integer")
     */
    private $nbrMAJ = 0;
    ...
    public function setNbrMAJ($nbrMAJ)     {
        $this->nbrMAJ = $nbrMAJ;
        return $this;
    }
    public function getNbrMAJ()     {
        return $this->nbrMAJ;
    }
}
```

On utilise l'attribut `nbrMAJ` pour compter le nombre de modifications d'une entité

Symfony

Démarche

- Tout d'abord, on doit indiquer à `Doctrine` que notre entité utilise une fonction `callback` avec l'annotation `HasLifecycleCallbacks`
- Ensuite, on va créer une méthode qui sera appelée avant chaque modification
- Cette méthode doit incrémenter chaque fois le nombre de mise-à-jour (`nbrMAJ`)

Symfony

Dans la classe Personne

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\
   PersonneRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Personne{
    ...
    /**
     * @ORM\PreUpdate
     */
    public function updateNbrMAJ()    {
        $this->setNbrMAJ($this->getNbrMAJ() + 1);
    }
    ...
}
```

Dans le contrôleur

```
/**
 * @Route("/personne/event", name="personne_event")
 */
function event(EntityManagerInterface $entityManager)
{
    $personne = new Personne();
    $personne->setNom('Wick');
    $personne->setPrenom('John');

    $entityManager->persist($personne);
    $entityManager->flush();

    $personne->setNom('Travolta');
    $entityManager->flush();

    $personne->setNom('Abruzzi');
    $entityManager->flush();
    return $this->render('personne/index.html.twig', [
        'controller_name' => 'PersonneController',
        'personne' => $personne,
        'adjectif' => $personne->getNbrMAJ()
    ]);
}
```

Symfony

Pour générer les entités à partir d'une base de données existante

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

Symfony

Pour générer les entités à partir d'une base de données existante

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

Remarque

Les attributs des entités générées n'ont pas de getters/setters.

Symfony

Pour générer les entités à partir d'une base de données existante

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

Remarque

Les attributs des entités générées n'ont pas de getters/setters.

Pour générer les getters/setters

exécutez la commande `php bin/console make:entity --regenerate App`