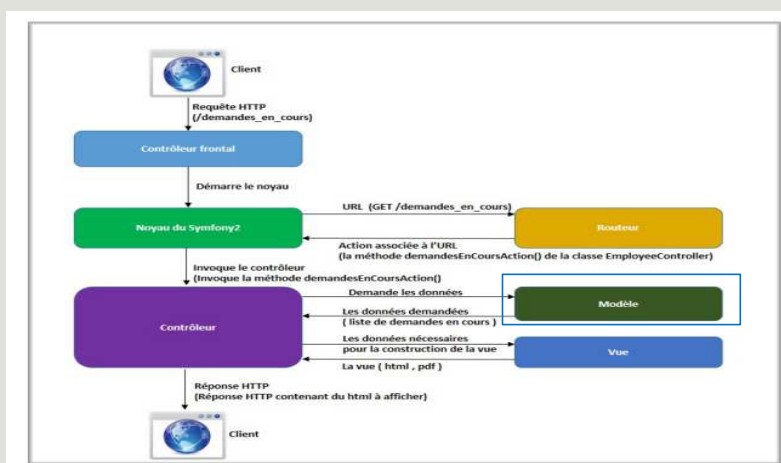


Symfony L'ORM DOCTRINE

AYMEN SELLAOUTI

1

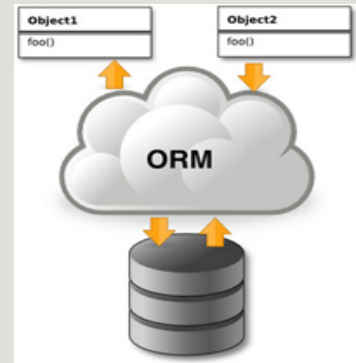
Introduction (1)



2

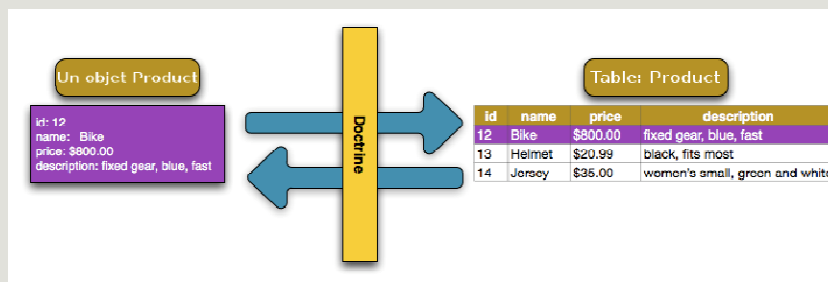
Introduction (2)

- ORM : Object Relation Mapper
- Couche d'abstraction
- Gérer la persistance des données
- Mapper les tables de la base de données relationnelle avec des objets
- Crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.
- Propose des méthodes prédéfinies



3

Introduction (3)



- Doctrine : ORM le plus utilisé avec Symfony2
- Associe des classes PHP avec les tables de votre BD (mapping)

4

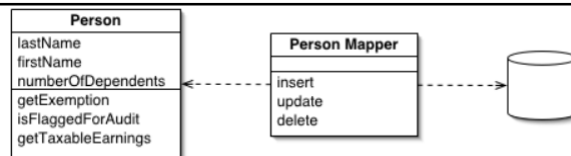
Fonctionnement de Doctrine

Doctrine utilise deux design pattern objet :

- Data Mapper
- Unit of Work

5

Data Mapper



C'est la **couche entre les objets (entités) et les tables de la base de données**.

Dans le cas de PHP elle **synchronise** les données stockées en base de données avec vos objets PHP.

Elle se charge d'**insérer** et de **mettre à jour** les données de la base en se basant sur le contenu des propriété de votre objet.

Elle peut aussi **supprimer** des enregistrement de la base de données.

Elle peut aussi **hydrater** vos objets en utilisant les informations contenues dans la base de données.

Ceci permet d'avoir une **abstraction complète** de la base de données vu que les objets sont indépendants du système de stockage. C'est le Data Mapper qui se charge de ça.

Doctrine implémente ce design pattern via l'objet **EntityManager**.

Patterns of Enterprise Application Architecture

6

Unit of Work

Unit of Work
registerNew(object)
registerDirty(object)
registerClean(object)
registerDeleted(object)
commit
rollback

Afin d'éviter une multitude de petite requêtes envoyé à votre base de données, et de garder un historique des requêtes effectuées au niveau de votre base de données, Doctrine utilise le design pattern **Unit of work**.

Pour une raison de performance et d'intégrité, La synchronisation effectuée par l'Entity Manager ne se fait pas pour chaque changement avec la base de données.

Le design pattern **Unit of Work** permet de gérer l'état des différents objets hydratés par l'Entity Manager.

Une transaction est ouverte regroupant l'ensemble des opérations. Le commit de cette transaction déclenchera l'exécution de l'ensemble de ses requêtes.

En cas d'échec l'ensemble des requêtes et annulées.

7

Les entités (1)

- Objet PHP
- Les entités représente les objets PHP équivalentes à une table de la base de données.
- Une entité est généralement composée par les attributs de la tables ainsi que leurs getters et setters
- Manipulable par l'ORM

8

Les entités (2)

Exemple

```
<?php
namespace Rt4\AsBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Etudiant
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="Rt4\AsBundle\Entity\StudentRepository")
 */
class Etudiant
{
    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;
```

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

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

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

    /**
     * @var string
     *
     * @ORM\Column(name="prenom", type="string", length=255)
     */
```

```
private $prenom;

    /**
     * @var \DateTime
     *
     * @ORM\Column(name="dateNaissance", type="date")
     */
    private $dateNaissance;

    /**
     * Get id
     *
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set numEtudiant
     *
     * @param integer $numEtudiant
     * @return Etudiant
     */
```

```
/**
     * Set numEtudiant
     *
     * @param integer $numEtudiant
     * @return Etudiant
     */
    public function setNumEtudiant($numEtudiant)
    {
        $this->numEtudiant = $numEtudiant;

        return $this;
    }

    /**
     * Get numEtudiant
     *
     * @return integer
     */
    public function getNumEtudiant()
    {
        return $this->numEtudiant;
    }
}
```

9

Configuration des entités

Configuration Externe : YAML, XML, PHP

Configuration Interne : annotations

Choix de la configuration ?

Deux Visions :

Pro-Externe

Séparation complète des fonctionnalités spécialement lorsque l'entité est conséquente

Pro-Interne

Plus facile et agréable de chercher dans un seul fichier l'ensemble des informations, plus de visibilité

10

Mapping : Annotation des entités (1)

Rôle : Faire le lien entre les entités et les tables de la base de données

Lien à travers les **métadonnées**

Remarque : Un seul format par bundle (impossibilité de mélanger)

Syntaxe :

```
/**  
 * les différentes annotations  
 */
```

Remarque : Afin d'utiliser les annotations il faut ajouter :

```
use Doctrine\ORM\Mapping as ORM;
```

11

Mapping : Annotation des entités (2)

Entity

Permet de définir un **objet** comme une **entité**

Applicable sur une **classe**

Placée avant la définition de la classe en PHP

Syntaxe :

```
@ORM\Entity
```

Paramètres :

repositoryClass (facultatif). Permet de préciser le namespace complet du repository qui gère cette entité.

Exemple :

```
@ORM\Entity(repositoryClass="Rt4\AsBundle\Entity\animalRepository")
```

12

Mapping : Annotation des entités (3)

Table

Permet de [spécifier le nom de la table dans la base de données à associer à l'entité](#)

Applicable sur une classe et placée avant la définition de la classe en PHP

[Facultative](#) sans cette annotation le nom de la table sera automatiquement le nom de l'entité

Généralement utilisable pour ajouter des préfixes ou pour forcer la première lettre de la table en minuscule

[Syntaxe](#) : `@ORM\Table()`

Exemple :

```
/**
 *
 * @ORM\Table('animal')
 * @ORM\Entity(repositoryClass="Rt4\AsBundle\Entity\animalRepository")
 */
```

13

Mapping : Annotation des entités (4)

Column

Permet de [définir les caractéristiques](#) de la [colonne](#) concernée

Applicable sur un attribut de classe juste avant la définition PHP de l'attribut concerné.

[Syntaxe](#) : `@ORM\Column()`

Exemple :

```
/**
 *
 * @ORM\Column(param1="valParam1",param2="valParam2")
 */
```

14

Mapping : Annotation des entités (5)

Les paramètres de Column

Paramètre	Valeur par défaut	Utilisation
type	string	Définit le type de colonne comme nous venons de le voir.
name	Nom de l'attribut	Définit le nom de la colonne dans la table. Par défaut, le nom de la colonne est le nom de l'attribut de l'objet
length	255	Définit la longueur de la colonne (pour les strings).
unique	false	Définit la colonne comme unique (Exemple : email).
nullable	false	Permet à la colonne de contenir des NULL.
precision	0	Définit la précision d'un nombre à virgule(decimal)
scale	0	le nombre de chiffres après la virgule (decimal)

15

Mapping : Annotation des entités (6)

Les types de Column

Type Doctrine	Type SQL	Type PHP	Utilisation
string	VARCHAR	string	Toutes les chaînes de caractères jusqu'à 255 caractères.
integer	INT	integer	Tous les nombres jusqu'à 2 147 483 647.
smallint	SMALLINT	integer	Tous les nombres jusqu'à 32 767.
bigint	BIGINT	string	Tous les nombres jusqu'à 9 223 372 036 854 775 807.
boolean	BOOLEAN	boolean	Les valeurs booléennes true et false.
decimal	DECIMAL	double	Les nombres à virgule.

16

Mapping : Annotation des entités (7)

Les types de Column

Type Doctrine	Type SQL	Type PHP	Utilisation
date ou datetime	DATETIME	objet DateTime	Toutes les dates et heures.
time	TIME	objet DateTime-	Toutes les heures.
text	CLOB	string	Les chaînes de caractères de plus de 255 caractères.
object	CLOB	Type de l'objet stocké	Stocke un objet PHP en utilisant serialize/unserialize.
array	CLOB	array	Stocke un tableau PHP en utilisant serialize/unserialize.
float	FLOAT	double	Tous les nombres à virgule. Attention, fonctionne uniquement sur les serveurs dont la locale utilise un point comme séparateur.

17

Mapping : Annotation des entités (8)

Conventions de Nommage

Même s'il reste facultatif, le champs « name » doit être modifié afin de respecter les conventions de nommage qui diffèrent entre ceux de la base de données et ceux de la programmation OO.

- Les noms de classes sont écrites en « Pascal Case » TheEntity.
- Les attributs de classes sont écrites en « camel Case » oneAttribute
- Les noms des tables et des colonnes en SQL sont écrites en minuscules, les mots sont séparés par « _ » one_table, one_column.

18

Gestion de la base de données (1)

Configuration de l'application

Afin de configurer la base de données de l'application il faut renseigner les champs dans le fichier `.env` qui est renseigné dans le fichier [config/packages/doctrine.yml](#)

```
doctrine:
    dbal:
        driver: 'pdo_mysql'
        server_version: '5.7'
        charset: utf8mb4
        default_table_options:
            charset: utf8mb4
            collate: utf8mb4_unicode_ci
        url: '%env(resolve:DATABASE_URL)%'
    orm:
        auto_generate_proxy_classes: true
        naming_strategy: doctrine.orm.naming_strategy.underscore
        auto_mapping: true
        mappings:
            App:
                is_bundle: false
                type: annotation
                dir: '%kernel.project_dir%/src/Entity'
                prefix: 'App\Entity'
                alias: App
```

19

```
# In all environments, the following files are loaded if they exist,
# the later taking precedence over the former:
# * .env contains default values for the environment variables needed by the app
# * .env.local uncommitted file with local overrides
# * .env.$APP_ENV committed environment-specific defaults
# * .env.$APP_ENV.local uncommitted environment-specific overrides
# Real environment variables win over .env files.
# DO NOT DEFINE PRODUCTION SECRETS IN THIS FILE NOR IN ANY OTHER COMMITTED FILES.
# Run "composer dump-env prod" to compile .env files for production use (requires symfony/flex >=1.2).
# https://symfony.com/doc/current/best_practices/configuration.html#infrastructure-related-configuration
###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=a52c08f24a2d463ab694946d4f1452ce
APP_DEFAULT_IMAGE=images/default.png
#TRUSTED_PROXIES=127.0.0.1,127.0.0.2
#TRUSTED_HOSTS='^localhost|example\.com$'
###< symfony/framework-bundle ###
###> doctrine/doctrine-bundle ###
# Format described at http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using-a-url
# For an SQLite database, use: "sqlite:///kernel.project_dir%/var/data.db"
# Configure your db driver and server version in config/packages/doctrine.yaml
DATABASE_URL=mysql://root@127.0.0.1:3306/sf4Forma
###< doctrine/doctrine-bundle ###
###> symfony/swiftmailer-bundle ###
# For Gmail as a transport, use: "gmail://username:password@localhost"
# For a generic SMTP server, use: "smtp://localhost:25?encryption=&auth_mode="
# Delivery is disabled by default via "null://localhost"
MAILER_URL=null://localhost
###< symfony/swiftmailer-bundle ###
```

Gestion de la base de données (2)

Création de base de données

Afin de créer la base de données du projet 2 méthodes sont utilisées :

- Manuelle en utilisant le SGBD (le nom de la BD doit être le même que celui mentionné dans le fichier doctrine.yml)
- En utilisant la ligne de command avec la command suivante :
 - `php bin/console doctrine:database:create`
 - Une base de données avec les propriétés mentionnées dans .env sera automatiquement générée

21

Gestion de la base de données (3)

Génération des entités

Deux méthodes pour générer les entités :

- Méthode manuelle (non recommandée)
 - Créer la classe
 - ajouter le mapping
 - ajouter les getters et les setters (manuellement ou en utilisant la commande suivante :
`php bin/console make:entity --regenerate App` (elle crée les getters et les setters de toutes les entités)
- Méthode en utilisant les commandes
 - Il suffit de lancer la commande suivante :
`php bin/console make:entity`
 - Ajouter les attributs ainsi que les paramètres qui vont avec
 - Une fois terminé, Doctrine génère l'entité avec toutes les métadonnées de mapping

22

Gestion de la base de données

Les migrations

Les migrations sont une nouvelle façon utilisée par Symfony 4 afin de gérer les mises à jours et évolutions de votre base de données.

Ayant une images des différentes évolutions de votre base de donnée, vous pouvez annuler des changements ou passer d'une version à une autre.

23

Gestion de la base de données

Les migrations : les commandes

Pour connaître l'état de vos fichiers de migrations, vous pouvez utiliser la commande

`php bin/console doctrine:migration:status`

```
>> Name: Application Migrations
>> Database Driver: pdo_mysql
>> Database Host: 127.0.0.1
>> Database Name: sf4Forma
>> Configuration Source: manually configured
>> Version Table Name: migration_versions
>> Version Column Name: version
>> Migrations Namespace: DoctrineMigrations
>> Migrations Directory: G:\symfony\my-project\src\Migrations
>> Previous Version: 2019-06-03 14:23:41 (20190603142341)
>> Current Version: 2019-06-04 15:20:25 (20190604152025)
>> Next Version: Already at latest version
>> Latest Version: 2019-06-04 15:20:25 (20190604152025)
>> Executed Migrations: 2
>> Executed Unavailable Migrations: 0
>> Available Migrations: 2
>> New Migrations: 0
```

24

Gestion de la base de données

Les migrations : les commandes

Créer un fichier de migration

Pour créer un fichier de migration, utilisez la commande :

`php bin/console doctrine:migration:generate`. Ceci vous créera un fichier de migration vide de tout traitement.

Si vous avez besoin d'écrire votre propre code de migration vous pouvez le faire.

Les migrations

Créer un fichier de migration pour adapter votre base de données à vos entités

Pour créer un fichier de migration, utilisez la commande :

`php bin/console doctrine:migration:diff`.

Vous pouvez aussi utiliser

`php bin/console make:migration` cette commande fait appel à la commande précédente.

```

<?php
declare(strict_types=1);
// use and namespace
/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20190604160541 extends AbstractMigration
{
    public function getDescription() : string
    {
        return '';
    }

    public function up(Schema $schema) : void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can
only be executed safely on \'mysql\'.');

        $this->addSql('CREATE TABLE personne (id INT AUTO_INCREMENT NOT NULL, name VARCHAR(100) NOT NULL,
job VARCHAR(100) NOT NULL, designation VARCHAR(100) NOT NULL, age INT NOT NULL, PRIMARY KEY(id)) DEFAULT
CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci ENGINE = InnoDB');
    }

    public function down(Schema $schema) : void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can
only be executed safely on \'mysql\'.');
        $this->addSql('DROP TABLE personne');
    }
}

```

Migration

Exécuter une migration particulière

Afin d'exécuter une migration particulière que ce soit la méthode up ou la méthode down utiliser la méthode suivante :

`php bin/console doctrine:migration:execute numVersion -fct`

Exemple

`php bin/console doctrine:migration:execute 20190604162216 --down`

Résumé

:diff [diff] Génère une migration en comparant la base de données avec les informations de mapping.

:execute [execute] Exécute une migration manuellement.

:generate [generate] Crée une classe de Migration.

:migrate [migrate] Effectue une migration vers le fichier de migration le plus récent ou celui spécifié.

:status [status] Affiche le status des migrations.

:version [version] Ajoute et supprime manuellement des versions à partir de la version en base.

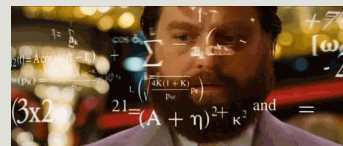
29

Exercice

Créer votre base de données à travers la ligne de commande.

Générer une Entité Personne.

Cette entité contient un attribut id, un attribut nom, prenom, age, cin et path.



Le service Doctrine

Rôle : permet la gestion des données dans la BD : **persistance** des données et **consultation** des données.

Méthode :

- `$this->get('doctrine');`
- `$this->getDoctrine();` //helper (raccourcie de la classe Controller)

Le service Doctrine offre deux services pour la gestion d'une base de données :

- Le Repository qui se charge des requêtes Select
- L'EntityManager qui se charge de persister la base de données donc de gérer les requêtes INSERT, UPDATE et DELETE.

Le service EntityManager

Rôle : L'interface ORM proposée par doctrine offrant des méthodes prédéfini pour persister dans la base ou pour mettre à jour ou supprimer une entité.

Méthode :

- `$EntityManager = $this->get('doctrineorm.entity_manager');`
- `$this->getDoctrine()->getManager();` //helper (raccourcie de la classe Controller)
- L'injecter en tant que service (à voir dans la partie service)

Remarque il y a aussi la méthode `getEntityManager()` mais elle est devenu obsolète.

Le service EntityManager

Insertion des données

Enregistrement des données

Etant un ORM, Doctrine traite les objets PHP

Pour enregistrer des données dans la BD il faut préparer les objets contenant ces données la

La méthode `persist()` de l'EntityManager permet d'associer les objets à persister avec Doctrine

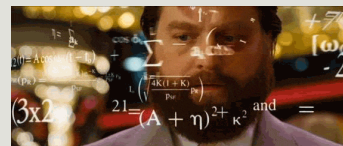
Afin d'exécuter les requêtes sur la BD (enregistrer les données dans la base) il faut utiliser la méthode `flush()`

L'utilisation de `flush` permet de profiter de la force de Doctrine qui utilise les Transactions

La persistance agit de la même façon avec l'ajout (`insert`) ou la mise à jour (`update`)

33

Exercice



Créer un contrôleur `PersonneController`

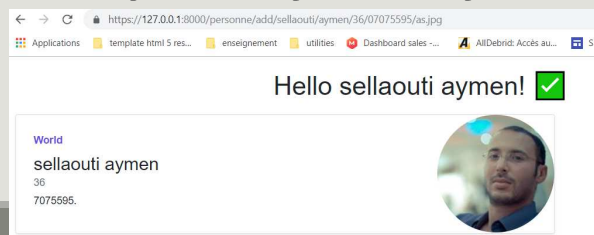
Créer une action qui permet l'ajout d'une Personne au niveau de la base de données.

Les données seront introduites via la route.

Une fois la personne ajoutée, une page contenant ses informations sera affichée.

<https://getbootstrap.com/docs/4.3/examples/blog/>

Faite de même pour la mise à jour, faite en sorte que cette action prenne l'id de la personne à modifier et les nouvelles valeurs.



Le service EntityManager

Suppression d'une entité

Suppression d'une entité

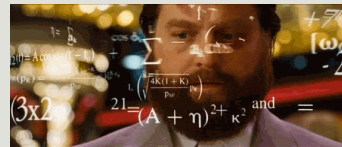
La méthode `remove()` permet de supprimer une entité

35

Exercice

Créer une action qui permet la suppression d'une personne en utilisant son id.

Si la personne n'existe pas un message d'erreur est affiché



Une petite parenthèse : Fixtures

Les fixtures sont utilisées pour charger des données « fake » au sein de votre base de données pour tester les fonctionnalités que vous avez développées.

Installer le bundle responsable des Fixtures.

`composer require orm-fixtures --dev`

Créer une classe `VotreEntitéFixture` à l'aide de la commande :

`php bin/console make:fixtures`

Implémenter la méthode `load` avec le fonctionnement que vous voulez pour charger vos données.

Lancer vos fixtures : `php bin/console doctrine:fixtures:load`, cette commande effacera le contenu de votre Base de données.

Si vous voulez garder la base ajouter l'option `--append`

`php bin/console doctrine:fixtures:load --append`

<https://symfony.com/doc/master/bundles/DoctrineFixturesBundle/index.html>

37

Fixture exemple

```
<?php

namespace App\DataFixtures;

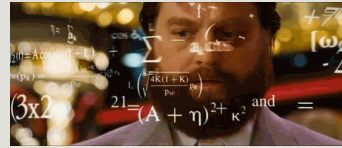
use App\Entity\Personne;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

class PersonneFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        for($i=0; $i<10; $i++) {
            $personne = new Personne();
            $personne->setAge(mt_rand(20, 80));
            $personne->setName("personne $i");
            $personne->setJob("Job $i");
            $personne->setDesignation("Designation $i");
            $manager->persist($personne);
        }

        $manager->flush();
    }
}
```

38

Exercice



Créer un fixture permettant d'ajouter quelques personnes dans la base de données.

Vous pouvez utiliser le composant faker

<https://github.com/fzaninotto/Faker>

Le Repository

Les repositories (dépôts)

Des classes PHP dont le rôle est de permettre à l'utilisateur de récupérer des entités d'une classe donnée.

Syntaxe :

Pour accéder au repository de la classe Maclasse on utilise Doctrine ou on l'injecte (voir dans la partie Service)

```
$repo = $this->getDoctrine->getRepository(ClassName::class);
```

Quelques méthodes offertes par le repository :

```
$repository->findAll(); // récupère tous les entités (enregistrements) relatifs à l'entité associée au repository
```

```
$repository->find($id); // requête sur la clé primaire
```

```
$repository->findBy(); // retourne un ensemble d'entités avec un filtrage sur plusieurs critères (nbre donné)
```

```
$repository->findOneBy(); // même principe que findBy mais une seule entité
```

```
$repository->findByNomPropriété(); $repository->findOneByNomPropriété();
```

Le Repository findAll

`findAll()`

Rôle : retourne l'ensemble des entités qui correspondent à l'entité associée au repository. Le format du retour est un Array

Exemple :

//On récupère le repository de l'entity manager correspondant à l'entité Etudiant

```
$repository = $this
```

```
->getDoctrine()
```

```
->getRepository(ClassName::class) ;
```

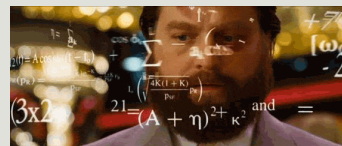
//On récupère la liste des étudiants

```
$listAdverts = $repository->findAll();
```

Généralement le tableau obtenu est passé à la vue (TWIG) et est affiché en utilisant un foreach

41

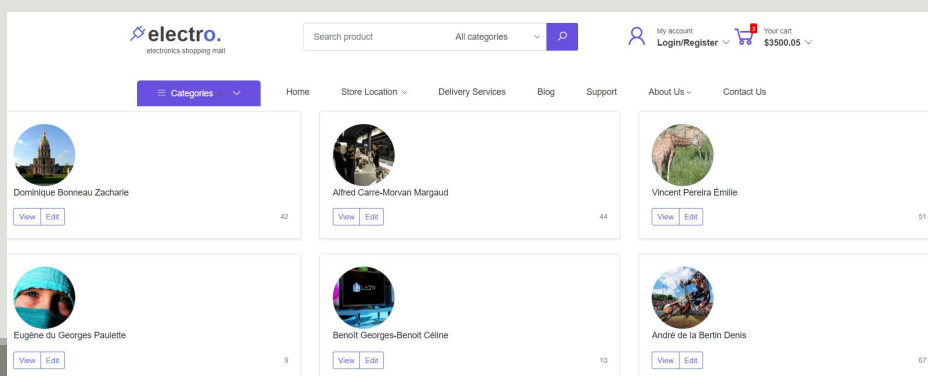
Exercice



Créer une page `list.html.twig`

Faite en sorte que cette page affiche la liste des personnes de votre base de données.

Vous pouvez utiliser l'affichage suivant : <https://getbootstrap.com/docs/4.3/examples/album/>



Le Repository find

`find($id)`

Rôle : retourne l'entité qui correspond à la clé primaire passé en argument. Généralement cette clé est l'id.

Exemple :

//On récupère le repository de l'entity manager correspondant à l'entité Etudiant

```
$repo = $this
```

```
->getDoctrine()
```

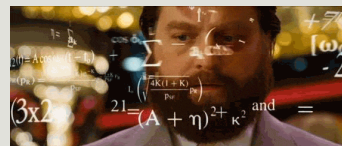
```
->getRepository(Etudiant::class);
```

//on lance la requête sur l'étudiant d'id 2

```
$etudiant = $repository->find(2);
```

43

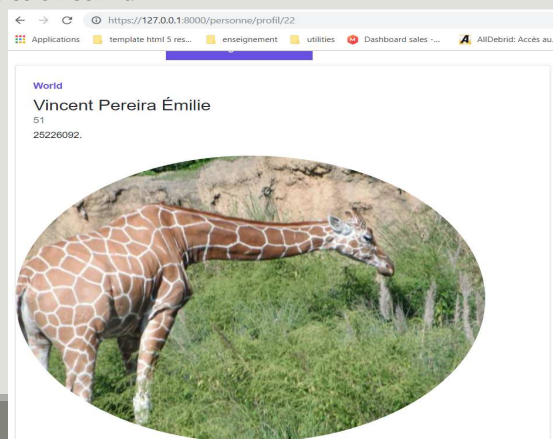
Exercice



Créer une page profil.html.twig

Faire en sorte que cette page affiche le profil d'une personne selon son id.

Utiliser la page de l'ajout.



Le Repository findBy

`findBy()`

Rôle : retourne l'ensemble des entités qui correspondent à l'entité associée au repository comme `findAll` sauf qu'elle permet d'effectuer un filtrage sur un ensemble de critères passés dans un Array. Elle offre la possibilité de trier les entités sélectionnées et facilite la pagination en offrant un nombre de valeur de retour.

Syntaxe:

```
$repository->findBy( array $criteria, array $orderBy = null, $limit = null, $offset = null);
```

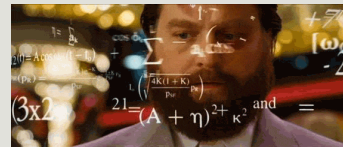
Exemple :

```
$repository = $this->getDoctrine()->getRepository(Etudiant::class);
```

```
$listeEtudiants = $repository->findBy(array('section' => 'RT4','nom' => 'Mohamed'),  
array('date' => 'desc'),10, 0);
```

45

Exercice



Créer une action qui permet d'afficher la liste des personnes d'un nom donnée ordonnée par prénom.

Le nombre maximum de résultat à afficher est de 5.

Le Repository

findOneBy

`findOneBy()`

Rôle : Même principe ue FindBy mais en retournant une seule entité ce qui élimine automatiquement les paramètres d'ordre de limite et d'offset

Exemple :

```
$repository = $this->getDoctrine()->getRepository(Etudiant::class);  
$Etud = $repository->findOneBy(array('section' => 'RT4','nom' => 'Mohamed'));
```

47

Le Repository

findByPropriété

`findByPropriété()`

Rôle : En remplaçant le mot **Propriété** par le **nom d'une des propriété de l'entité**, la fonction va faire le même rôle que **findBy** mais avec un seul critère qui est le nom de la propriété et sans les options.

Exemple :

```
$repository = $this->getDoctrine()->getRepository(Etudiant::class);  
$listeEtudiants = $repository->findByNom('Aymen');
```

48

Le Repository

findOneByPropriété

`findOneByPropriété()`

Rôle : En remplaçant le mot **Propriété** par le **nom d'une des propriété de l'entité**, la fonction va faire le même rôle que `findOneBy` mais avec un seul critère.

Exemple :

```
$repository = $this->getDoctrine()->getRepository(Etudiant::class);  
$listeEtudiants = $repository->findOneByNom('Aymen');
```

49

Le Repository

Création de requêtes

Les requêtes de doctrine sont écrites en utilisant le langage de doctrine le Doctrine Query Language **DQL** ou en utilisant un Objet créateur de requêtes le **CreateQueryBuilder**

`createQuery` : Méthode de l'Entity Manager

➤ `CreateQueryBuilder` : Méthode du repository

DQL	≈	SQL
Classes + Propriétés		Tables + colonnes

Le Repository

CreateQuery et DQL

- Le DQL peut être défini comme une adaptation du SQL adapté à l'orienté objet et donc à DOCTRINE
- La requête est défini sous forme d'une chaîne de caractère
- Afin de créer une requête DQL il faut utiliser la méthode `createQuery()` de l'EntityManager
- La méthode `setParameter('label','valeur')` permet de définir un paramètre de la requête
- Pour définir plusieurs paramètres ou bien utiliser `setParameter` plusieurs fois ou bien la méthode `setParameters(array('label1','valeur1', 'label2','valeur2'),... 'labelN','valeurN'))`
- Une fois la requête créée la méthode `getResult()` permet de récupérer un tableau de résultat
- Le langage DQL est explicité dans le lien suivant :

<http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html>

51

Le Repository : Création de requêtes

createQuery

```
public function findPersonneByIntervalAge2($min, $max)
{
    $query = $this->_em->createQuery(
        'SELECT p
        FROM App\Entity\Personne p
        WHERE p.age >= :ageMin And p.age <= :ageMax
        ORDER BY p.age
        ASC'
    )
    ->setParameter('ageMin', $min)
    ->setParameter('ageMax', $max)
    ;
    return $query->execute();
}
```

<https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/dql-doctrine-query-language.html> 52

Le Repository QueryBuilder

Constructeur de requête DOCTRINE Alternative au DQL

Accessible via le [Repository](#)

Le résultat fourni par la méthode `getQuery` du `QueryBuilder` permet de [générer la requête en DQL](#).

De même que le `createQuery`, une fois la requête créée la méthode `getResult()` permet de récupérer un tableau de résultat.

<http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/query-builder.html>

53

Le Repository QueryBuilder

- Afin de récupérer le QueryBuilder dans notre Repository, on utilise la méthode `createQueryBuilder`.
- Cette méthode récupère en paramètre l'alias de l'entité cible et offre la requête « select from » de l'entité en question.
- Généralement l'alias est la première lettre en minuscule du nom de l'entité
- Si aucun paramètre n'est passé à `createQueryBuilder` alors on aura une requête vide et il faudra tout construire.

```
$qb=$this->_em->createQueryBuilder()  
    ->select('t')  
    ->from($this->_entityName, 'alias');
```

```
$qb=$this->createQueryBuilder('t')
```

54

Le Repository

QueryBuilder : Méthodes

- **from('entityName','entityAlias')**
 - `from($this->_entityName,'t')`
- **where('condition')** permet d'ajouter le where dans la requête
 - `where('t.destination= :dest')`
- **setParameter('nomParam',param)** permet d'ajouter la définition d'un des paramètres définis dans le where
 - `setParameter('dest',$dest)`
- **andWhere('condition')** permet d'ajouter d'autres conditions
 - `andWhere('t.statut = :status')`
- **orderBy('nomChamp','ordre')** permet d'ajouter un orderBy et prend en paramètre le champ à ordonner et l'ordre DESC ou ASC.
 - `orderBy('t.dateTransfert','DESC')`
- **setParameters(array(1=>'param1',2=>'param2'))**

Le Repository

QueryBuilder : Méthodes

- **orWhere**
- **groupBy**
- **having**
- **andHaving**
- **orHaving**
- **leftJoin**
- **rightJoin**
- **Join**
- **innerJoin**
- ...

Le Repository QueryBuilder : Méthodes

- **getQuery()** : retourne la requête dql
- **getResult()** : retourne un tableau d'objets contenant le résultat
- **getOneOrNullResult()** : retourne le premier résultat ou Null
- **getSingleScalarResult()** : retourne un résultat sous format scalaire. Imaginer le use case où vous voulez récupérer le COUNT ou la SUM d'un de vos objets.

Requêter des attributs spécifiques

Afin de sélectionner des **propriétés particulières** utiliser la méthode select du **queryBuilder**

Syntaxe :

```
->select('SUM(u.age) as sumAge, AVG(u.age) as avgUserAge')
```

Réutilisation de la logique de vos requêtes

Imaginer les uses cases suivants :

Vous voulez sélectionner les formations d'un topic donné.

Vous voulez sélectionner les formations d'un topic donné dont le nombre d'inscrits est inférieur à un nombre données.

Vous voulez sélectionner les formations d'un topic donné dont le nombre d'inscrits est supérieur à un nombre données.

Vous voulez sélectionner les formations d'un topic donné entre deux dates.

....

On remarque ici une redondance dans ces différentes requêtes.

L'idée est donc d'isoler ce traitements redondant dans une méthode et de la réutiliser.

59

```
/**
 * @param $min
 * @param $max
 * @return mixed
 */
public function getPersonneByAge($min, $max) {

    $qb = $this->createQueryBuilder('p');
    $qb = $this->findByAge($qb, $min, $max);
    return $qb->getQuery()->getResult();
}

/**
 * @param QueryBuilder $qb
 * @param $min
 * @param $max
 * @return QueryBuilder
 */
private function findByAge(QueryBuilder $qb, $min, $max) {
    if($min) {
        $qb->andWhere('p.age > :minAge')
            ->setParameter('minAge', $min);
    }
    if($max) {
        $qb->andWhere('p.age < :maxAge')
            ->setParameter('maxAge', $max);
    }
    return $qb;
}
```

60

```

public function getCommunesByZipCode($zipCode = null, $like = null,
$type = null) {
    $qb = $this->createQueryBuilder('l')
        ->select('l.id, l.name, l.zipCode')
        ->where('1=1');

    if ($zipCode) {
        $qb = $qb->andWhere('l.zipCode = :zipCode ')
            ->setParameter('zipCode', $zipCode);
    }

    if ($like) {
        $qb = $qb->andWhere('l.name like :like ')
            ->setParameter('like', "%$like%");
    }

    return $qb->getQuery()->getArrayResult();
}

```

Gestion des relations entre les entités (1)

Les types de relation

Les entités de la BD présentent des relations d'association :

- A **OneToOne** B : à une entité A on associe une entité de B et inversement
- A **ManyToOne** B : à une entité B on associe plusieurs entité de A et à une entité de A on associe une entité de B
- A **ManyToMany** B : à une entité de A on associe plusieurs entité de B et inversement

Gestion des relations entre les entités (2)

Relation unidirectionnelle et bidirectionnelle

La notion de navigabilité de UML est la source de la notion de relation unidirectionnelle ou bidirectionnelle

Une relation est dite navigable dans les deux sens si les deux entités doivent avoir une trace de la relation.

Exemple : Supposons que nous avons les deux classes `CandidatPresidentielle` et `Electeur`.

L'électeur doit savoir à qui il a voté donc il doit sauvegarder cette information par contre le candidat pour cause d'anonymat de vote ne doit pas connaître les personnes qui ont voté pour lui.

On aura donc un attribut `Candidat` dans la table `Electeur` mais pas de collection ou tableau nommé `electeur` dans la table `CandidatPresidentielle`. Ici on a une relation **unidirectionnelle**.

63

Création de la relation

Vous pouvez créer votre relation via la console.

Créer un attribut et mettez y comme type « **relation** ».

Ceci va générer un attribut annoté avec la relation et les informations minimalistes nécessaires pour sa création.

```
/**
 * @ORM\ManyToMany(targetEntity="App\Entity\Hobbies", inversedBy="personnes")
 */
private $hobbies;
```

64

Gestion des relations entre les entités (3)

OneToOne unidirectionnelle

- Relation **unidirectionnelle** puisque Media ne référence pas Etudiant

```
/**Entity **//
Class Etudiant
{
// ...
/**
 * @ORM\OneToOne(targetEntity=
targetEntity="App\Entity\Media"
)
 */
private $media;
// ...
}
/**Entity **//
Class Etudiant
{
// ...
}
```

65

Gestion des relations entre les entités (4)

OneToOne Bidirectionnelle

- Si nous voulons qu'à partir du media on peut directement savoir à quel étudiant il appartient nous devons faire une relation bidirectionnelle
- Media aussi doit référencer Etudiant

```
/**Entity **//
Class Etudiant
{
// ...
/**
 * @ORM\OneToOne(targetEntity= "App\Entity\Media")
 */
private $media;
// ...
}
/**Entity **//
Class Media
{
/**
 * @ORM\OneToOne(targetEntity=
"App\Entity\Etudiant", mappedBy="media")
 */
private $etudiant;
// ...
}
```

66

Gestion des relations entre les entités (5)

ManyToOne Unidirectionnelle

- Relation **unidirectionnelle** puisque Section ne référence pas Etudiant

```
/**Entity **/  
Class Etudiant  
{  
  // ...  
  /**  
   * @ORM\ManyToOne(targetEntity="App\Entity\Section")  
   */  
  private $section;  
  // ...  
}  
/**Entity **/  
Class Section  
{  
  // ...  
}
```

67

Gestion des relations entre les entités (6)

OneToMany Bidirectionnelle

- Si nous voulons connaître dans l'objet section l'ensemble des étudiants qui lui sont affectés alors on doit avoir une relation bidirectionnelle
- Section aussi doit référencer Etudiant
- On aura une relation **OneToMany** coté Section puisqu'à « One » Section on a « Many » Etudiants.
- On doit ajouter l'attribut **mappedBy** côté **OneToMany** et **inversedBy** côté **ManyToOne**
- On doit spécifier dans le **constructeur** du **OneToMany** que l'attribut **mappé** est de type **ArrayCollection** en l'instanciant

```
/**Entity **/  
Class Section  
{  
  // ...  
  /**  
   * @ORM\OneToMany(targetEntity="App\Entity\Etudiant",  
   * mappedBy="section")  
   */  
  private $etudiants;  
  // ...  
  public function __construct() {  
    $this->etudiants = new ArrayCollection ();  
  }  
}  
/**Entity **/  
Class Etudiant  
{  
  // ...  
  /**  
   * @ORM\ManyToOne(targetEntity="App\Entity\Section",  
   * inversedBy="etudiants")  
   */  
  private $section;  
  // ...  
}
```

Gestion des relations entre les entités (7)

ManyToMany

- Relation **unidirectionnelle** puisque Cours ne référence pas Prof
- Ici on peut savoir quels sont les cours de chaque étudiant mais pas l'inverse (on peut l'extraire via une requête)

```
/**Entity **/  
Class Matiere  
{  
    // ...  
}  
/**Prof**/  
Class Etudiant  
{  
    //...  
    /**  
     * @ORM\ManyToOne(targetEntity= "App\Entity\Matiere")  
     */  
    private $cours ;  
  
    / ...  
    /**  
     * Constructor  
     */  
    public function __construct()  
    {  
        $this->matieres = new  
            \Doctrine\Common\Collections\ArrayCollection();  
    }  
}
```

69

Gestion des relations entre les entités (8)

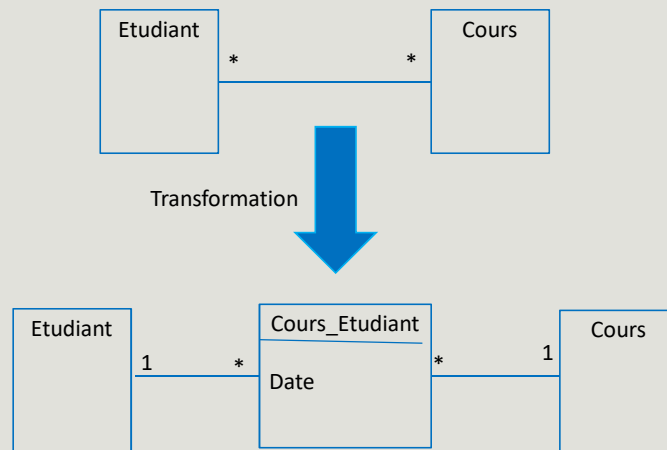
ManyToMany Bidirectionnelle

- On doit spécifier dans les deux constructeurs que l'attribut mappé est de type ArrayCollection en l'instanciant
- Même chose que la bidirectionnelle OneToMany en ajoutant les mappedBy et invertedBy
- Cette relation peut être traduite à deux relation OneToMany/ManyToOne entre les 3 classes participantes.

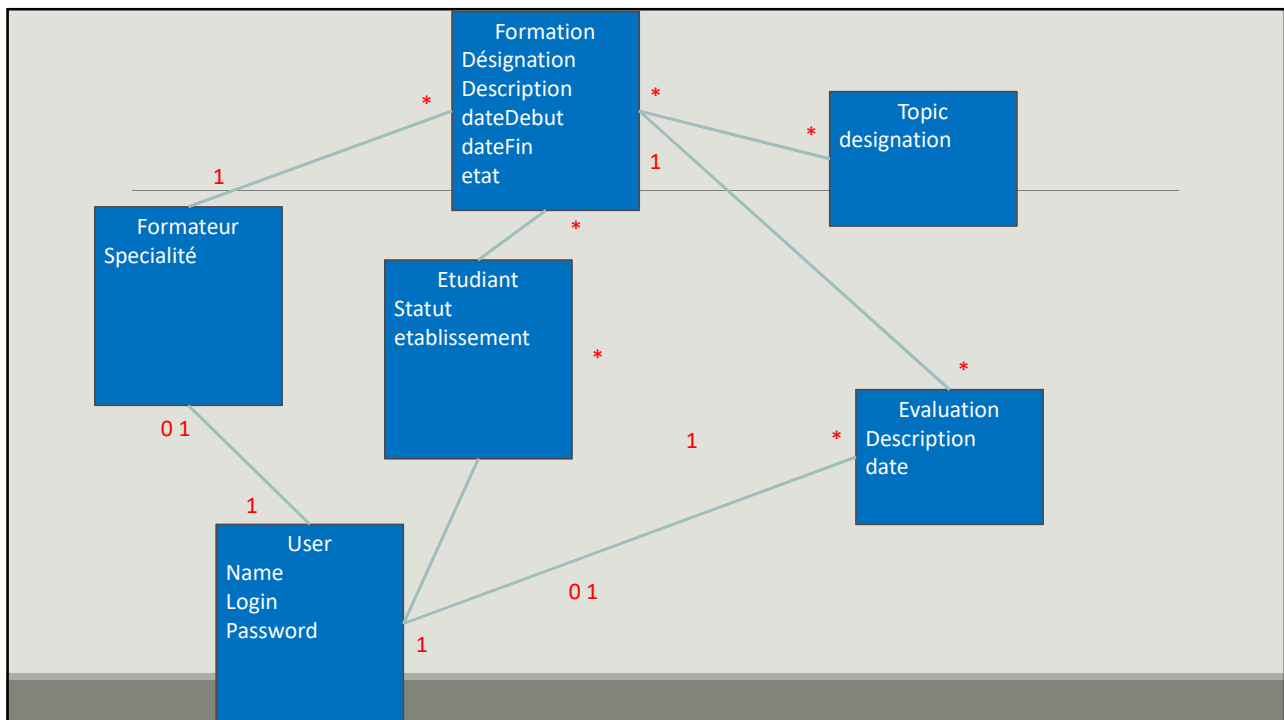
70

Gestion des relations entre les entités (9)

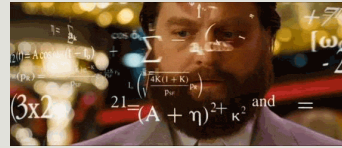
Cas particulier ManyToMany



71



Exercice



Créer les entités suivantes ainsi que les relations qui les relient :

Formateur

Formation

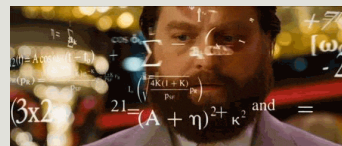
Topic

Evaluation

Etudiant

Créer les fixtures pour les entités Formateur et Topic

Exercice



Ajouter une fonction permettant d'afficher la liste des formations actives de la plus récente vers la plus anciennes et dont la date n'est pas encore dépassée. Elle vous retournera vers la page d'accueil du site.

Ajouter une fonction permettant d'afficher la liste des formations non actives de la plus récente vers la plus anciennes.

Ajouter une fonction permettant d'afficher la liste des formations de la plus récente vers la plus anciennes et dont la date n'est pas encore dépassée.

Détour : les Traits

- Servent à externaliser du code redondant dans plusieurs classes différentes.
- Pourquoi les traits et non l'héritage ? Parce que PHP ne supporte pas l'héritage multiple.
- Non instanciable
- Un trait peut contenir des méthodes et des attributs
- Syntaxe :

```
Trait nomTrait{  
    public $x;  
    Function fct1(){  
    }  
    Function fct2(){  
    }  
}
```

75

Les événements de Doctrine (1)

- Appelé aussi les callbacks du cycle de vie, ce sont des méthodes à exécuter par doctrine et dépendant d'événement précis :
 - PrePersist
 - PostPersist (s'exécute après le \$em->flush() non après \$em->persist())
 - PreUpdate
 - PostUpdate
 - PreRemove
 - PostRemove

Afin d'informer doctrine qu'une entité contient des **callbacks** nous devons utiliser l'annotation **@ORM\HasLifecycleCallbacks()**

Ceci ne s'applique que lors de l'utilisation des annotations

```
/**  
 * Transfer  
 *  
 * @ORM\Table(name="transfer")  
 *  
 * @ORM\Entity(repositoryClass="AppBundle\  
Repository\TransferRepository")  
 * @ORM\HasLifecycleCallbacks()  
 */  
class Transfer
```

76

Les événements de Doctrine (2)

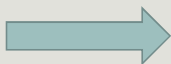
- Pour informer Doctrine de l'existence d'un événement on utilise maintenant l'annotation sur l'action à réaliser.

```
/**
 * @PrePersist()
 */
public function onPersist() {
    $this->createdAt = new \DateTime('NOW');
}
```

77

Les traits et Les événements de Doctrine

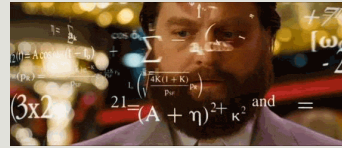
- Imaginons maintenant que nous voulons « sécuriser plusieurs de nos entités » et d'avoir un peu d'historique. L'idée est d'avoir deux attributs qui sont createdAt et modifiedAt pour avoir toujours une idée sur la création de notre entité et de sa dernière modification.
- L'idée est de créer pour chacune des entités à suivre des lifecycle callback qui vont mettre à jour ces deux attributs lors de la création (prePersist) et la modification (preUpdate).
- Est-ce normal de le refaire pour toutes les entités ?
- Si la réponse est non, que faire alors ?



Les traits

78

Exercice



Cr  er le Trait qui permet la gestion de la date d'ajout et de modification d'une entit  .

Associer le avec l'entit   formation

Tester le