Symfony - Doctrine - API

Doctrine

Fred Hémery

2018/2019

IUT Lens

Département Informatique

IUT LENS

IUT LENS

Introduction

Doctrine propose un mécanisme de **translation de données** entre le monde PHP et un système de gestion de bases de données.

- Une **entité** est une classe PHP classique
- Doctrine contient un Data Mapper (une instance de la classe **EntityManager**) qui prend en charge la classe qui contient l'entité pour la rendre persistante.
- Un cadre de conception **Unit of Work** gère l'état des entités prisent en charge par l'**EntityManager**.

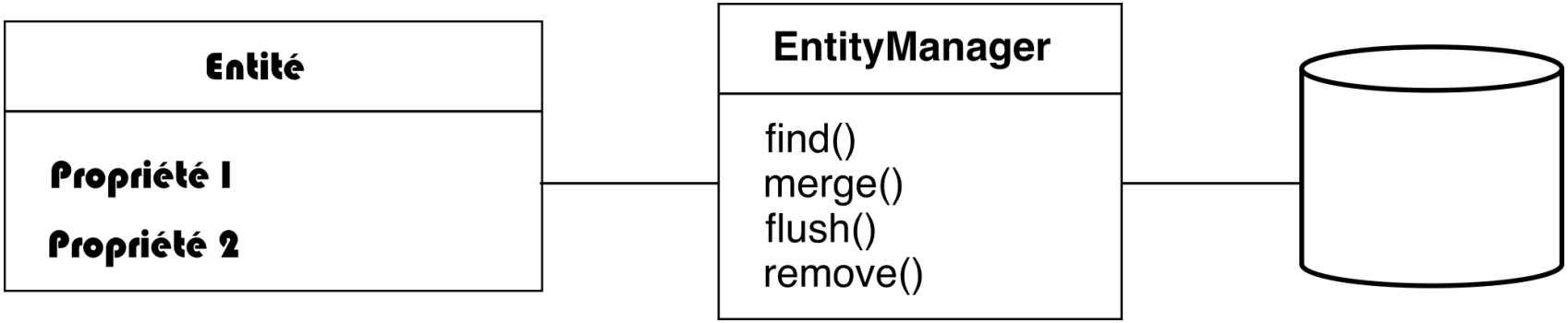
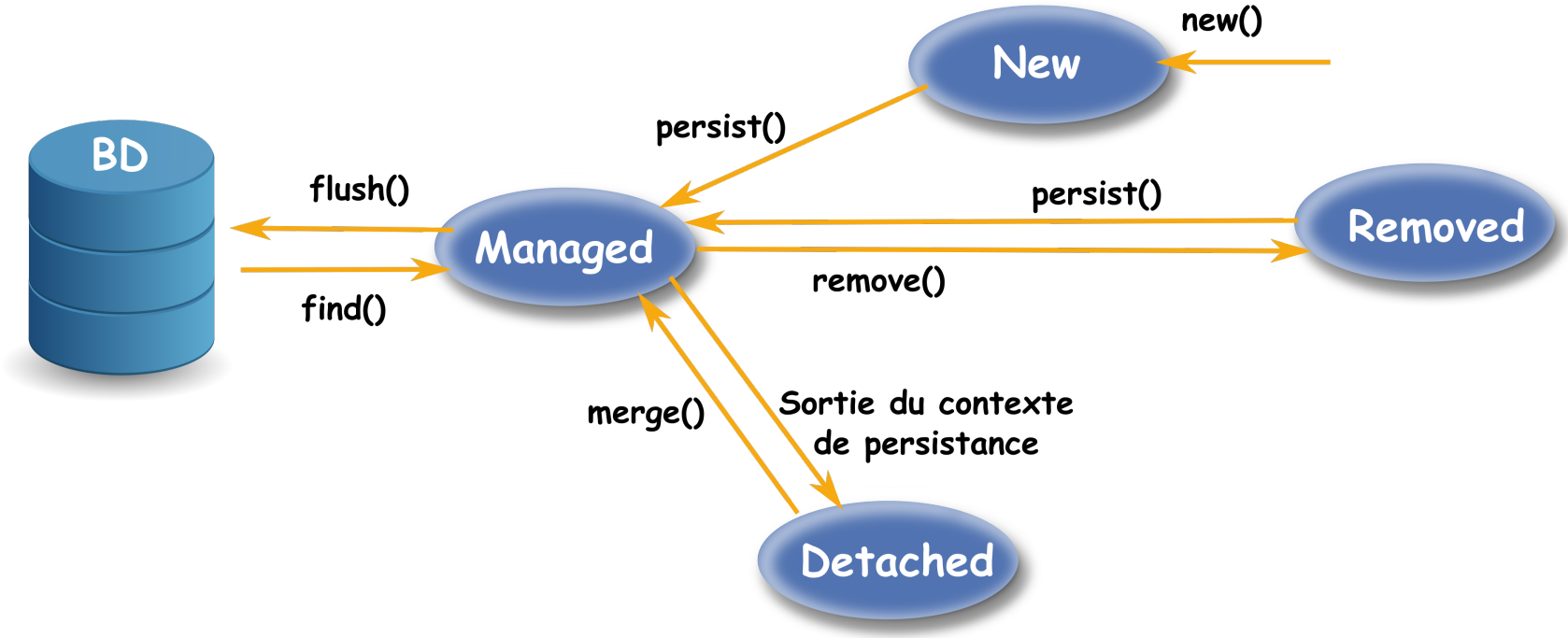
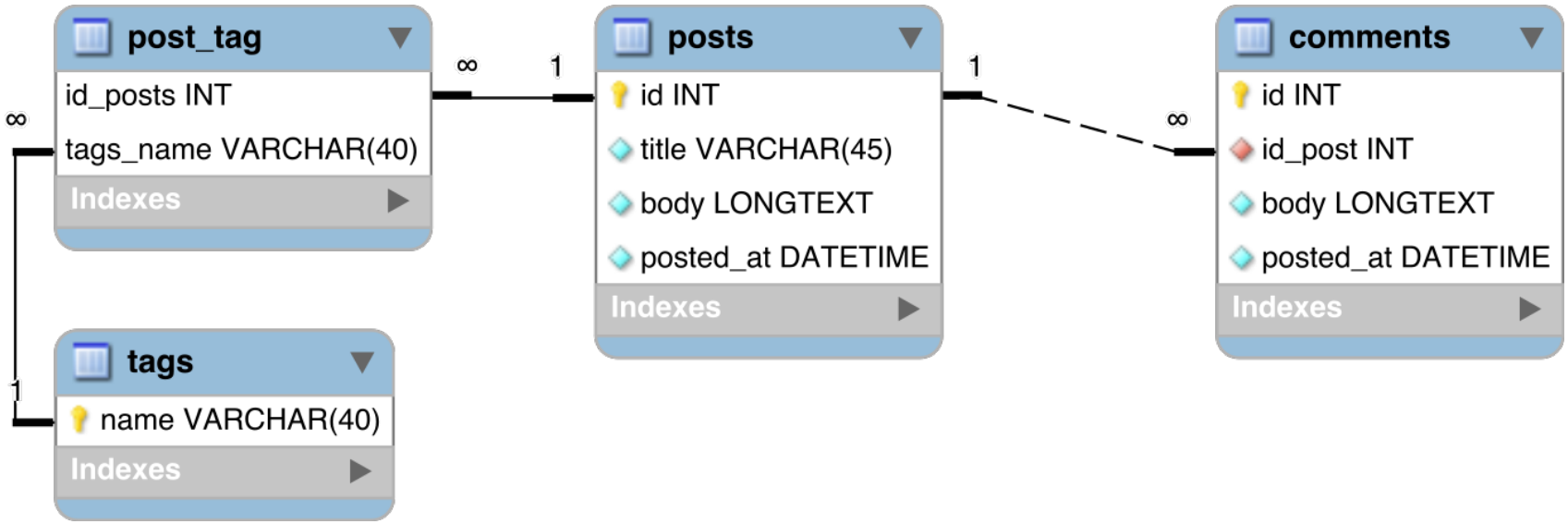


Diagramme d'état des entités



Présentation de l'exemple

Dans la suite du cours nous allons prendre comme exemple le modèle logique de données de la figure ci-dessous



Persistence in PHP with Doctrine ORM.
Kévin Dunglas.
Packt Publishing, 2013.

La documentation officielle de Doctrine
`http://docs.doctrine-project.org/en/latest/index.html`

IUT Institut Universitaire Technologie LENS

Entité

Une **entité** est une classe PHP à laquelle on ajoute des annotations qui vont permettre de faire le lien avec la **table** associée dans la base de données.

Classe `Post` sans annotation

```
<?php
class Post {
    private $id;
    private $title;
    private $body;
    private $publicationDate;
    // constructeur & getters & setters ...
}
```

Entité Annotée

Classe `Post` avec annotations

```
<?php
/**
 * @Entity
 * @Table(name="posts")
 */
class Post {
    /**
    * @Id
    * @GeneratedValue
    * @Column(type="integer", )
    */
    private $id;
    /** @Column(type="string", length=140) */
    private $title;
    /** @Column(type="text") */
    private $body;
    /** @Column(type="datetime", name="posted_at") */
    private $publicationDate;
}
```

Les annotations de colonne : `@Column`

`type` : le type de la colonne

`name` : le nom de la colonne dans la table

`length` : La taille de la colonne (pour le type `string`)

`type` : le type de la colonne

`unique` : contrainte de valeur unique pour la colonne

`nullable` : contrainte de valeur nulle pour la colonne

`precision` : nombre total de digits pour le type `decimal`

`scale` : nombre de digit après la virgule pour le type `decimal`(doit être inférieur à `precision`)

`columnDefinition` : permet d'indiquer le type au format DDL

IUT Institut Universitaire Technologique LENS

La valeur de type de colonne

string : SQL VARCHAR ⇔ PHP string

integer, smallint : SQL INT (SMALLINT) ⇔ PHP integer.

bigint : SQL BIGINT ⇔ PHP string

boolean : SQL boolean ou (TINYINT) ⇔ PHP boolean.

decimal : SQL DECIMAL ⇔ PHP string.

date : SQL DATETIME ⇔ PHP DateTime.

time : SQL TIME ⇔ PHP DateTime.

text : SQL CLOB ⇔ PHP string.

Voir la liste complète dans la documentation

http://docs.doctrine-project.org/en/latest/reference/basic-mapping.html

IUT Institut Universitaire Technologie LENS

La clé primaire

`@id`

Chaque **entité** doit avoir une **clé primaire**. La propriété qui représente la clé unique est annotée par `@id`.

Stratégies

Si la valeur unique pour la clé primaire est générée de manière automatique, la propriété est annotée par `@GeneratedValue`. Il existe plusieurs stratégies.

Les différentes stratégies

`AUTO` : (utilisé par défaut) c'est le SGBD cible qui fera le choix le plus adéquat.

`SEQUENCE` : utilise une sequence pour générer le numéro unique.

`IDENTITY` : utilise une colonne particulière dans la base de données.

`UUID` : utilise une fonction qui génère un **Universally Unique Identifier**.

`TABLE` : utilise une table particulière pour la génération de l'ID.

`NONE` : indique que la valeur est assignée par le code (doit être indiquée avant l'insertion). A le même effet que si vous n'indiquez pas `@GeneratedValue` dans les annotations.

`CUSTOM` : permet de spécifier une classe qui génère le numéro unique en association avec l'annotation `@CustomIdGenerator`.

IUT Institut Universitaire Technologie LENS

Gestion des associations entre classes

- Une classe peut être en association avec une autre. Pour implémenter ces associations, les tables dans la base de données utilisent des clés étrangères (références).
- Doctrine va utiliser les références des objets associés plutôt que manipuler les clés étrangères.
- Il existe plusieurs types d'associations

`OneToOne`	Une instance de l'entité courante de l'association est référencée par une instance de l'entité partenaire.
`OneToMany`	Une instance de l'entité courante de l'association est référencée par plusieurs instances de l'entité partenaire.
`ManyToOne`	Plusieurs instances de l'entité courante de l'association référencent une entité partenaire.
`ManyToMany`	Plusieurs instances de l'entité courante de l'association référencent plusieurs instances de l'entité partenaire.

IUT Institut Universitaire Technologique LENS

Nature de l'association

- Une association peut être

  Unidirectionnelle : Seules les instances de l'une des entités de l'association peuvent retrouver les instances de l'entité partenaire.

  - Par exemple : un utilisateur peut obtenir la liste de ses adresses connues, par contre il n'est pas possible de retrouver un utilisateur à partir d'une adresse.

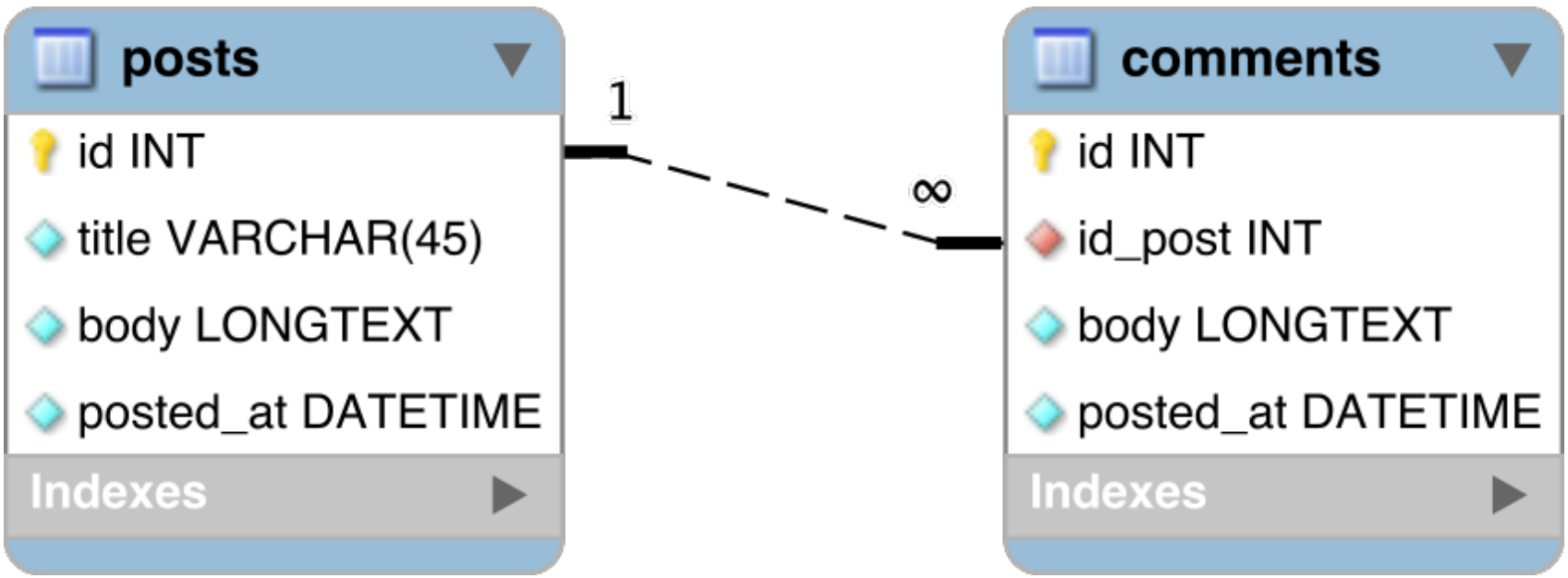
  Bidirectionnelle Les instances de l'une ou de l'autre des entités de l'association peuvent retrouver les instances de l'entité partenaire.

  - Par exemple : un utilisateur peut obtenir la liste des commandes qu'il a effectué et on peut retrouver un utilisateur à partir d'une commande.

IUT Institut Universitaire Technologie LENS

Exemple d'association `OneToMany`

Soit dans l'exemple l'association entre la table `Posts` et la table `Comments`.
C'est une association **bidirectionnelle** de type `OneToMany`. Un `post` peut être commenté par plusieurs `comments`.



Mise en place de l'association : classe Post

Classe Post

```
<?php
class Post {
    //  ...
    /**
    * @OneToMany(targetEntity="Comment", mappedBy="post")
    */
    private $comments

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

- `targetEntity` : indique le nom de la classe partenaire dans l'association
- `mappedBy` : indique la propriété dans la classe partenaire qui est propriétaire de l'association
- La propriété `comments` est de type `ArrayCollection`.

Mise en place de l'association : classe Comment

Classe Comment

```
<?php
class Comment {
    //  ...
   /**
     * @var Post
     *
     * @ManyToOne(targetEntity="Post", inversedBy="
    ➥comments")
     */
     protected $post;
}
```

- `inversedBy` : indique la propriété dans la classe partenaire qui porte l'association.

Gestion du coté propriétaire et du coté inverse d'une association

- Dans le cas d'une association bidirectionnelle, il y a un coté **propriétaire** et un coté **inverse**.
- Dans le cas d'une association unidirectionnelle, il n'y a qu'un coté **propriétaire**.
- Doctrine ne gère que les changements effectués du coté propriétaire de l'association.
- Lors de modifications dans l'association, il est donc important que le code effectue une modification du coté propriétaire de l'association.

Classe `Post`

```
<?php
class Post {
    //  ...
public function addComment(Comment $comments)  {
  $this->comments[] = $comments;
  $comments->setPost($this);  // IMPORTANT
   return $this;
}
```

IUT Institut Universitaire Technologique LENS

Many-To-One, Unidirectionelle

```php
<?php
/** @Entity */
class User {
    // ...
    /**
     * Many Users have One Address.
     * @ManyToOne(targetEntity="Address")
     * @JoinColumn(name="address_id",
    ➥referencedColumnName="id")
     */
    private $address;
}

/** @Entity */
class Address {
    // ...
}
```

- `@JoinColumn` : indique le nom de la propriété dans la table ainsi que le nom de la propriété dans la table partenaire de l'association.

IUT Institut Universitaire Technologie LENS

One-To-One, bidirectionelle

```php
<?php
/** @Entity */
class Customer {
    // ...

    /**
     * One Customer has One Cart.
     * @OneToOne(targetEntity="Cart", mappedBy="customer")
     */
    private $cart;

    // ...
}

/** @Entity */
class Cart {
    // ...

    /**
     * One Cart has One Customer.
     * @OneToOne(targetEntity="Customer", inversedBy="cart")
     * @JoinColumn(name="customer_id", referencedColumnName="id")
     */
    private $customer;

    // ...
}
```

IUT Institut Universitaire Technologique LENS

Many-To-Many, Unidirectionnelle

```php
<?php
/** @Entity */
class User {
    // ...

    /**
     * Many Users have Many Groups.
     * @ManyToMany(targetEntity="Group")
     * @JoinTable(name="users_groups",
     *      joinColumns={@JoinColumn(name="user_id",
     *                  referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="group_id",
     *                  referencedColumnName="id")}
     *      )
     */
    private $groups;

    // ...

    public function __construct() {
        $this->groups = new \Doctrine\Common\Collections\ArrayCollection();
    }
}

/** @Entity */
class Group {
    // ...
}
```

Many-To-Many, Bidirectionnelle

```php
<?php
/** @Entity */
class User {
    // ...
    /**
     * Many Users have Many Groups.
     * @ManyToMany(targetEntity="Group", inversedBy="users")
     * @JoinTable(name="users_groups")
     */
    private $groups;

    public function __construct() {
        $this->groups = new ArrayCollection();
    }
    // ...
}
/** @Entity */
class Group {
    // ...
    /**
     * Many Groups have Many Users.
     * @ManyToMany(targetEntity="User", mappedBy="groups")
     */
    private $users;

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

Exemple encore : Many-To-Many Bidirectionnelle

Si nous reprenons notre exemple :

La classe Tag

- La classe `Tag` joue le rôle de face inverse de l'association,
- Doctrine n'utilise pas cette classe pour valider les modifications de l'association

Face inverse de l'association (`mappedBy="tags"`)

```
<?php
/**
 * @Entity
 * @Table(name="tags")
 */
class Tag {
      /**
        * @Id
        * @Column(type="string")
        */
       protected $name;
       /** @ManyToMany(targetEntity="Post", mappedBy="tags") */
       protected $posts;

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

La classe Post

- La classe `Post` joue le rôle de face propriétaire de l'association,
- Doctrine utilisera cette classe pour valider les modifications de l'association

Face propriétaire de l'association (`inversedBy="posts"`)

```
<?php
/**
 * @Entity
 * @Table(name="posts")
*/
<?php
class Post {
    //  ...
   /**
     * @ManyToMany(targetEntity="Tag", inversedBy="posts",
     *           fetch="EAGER", cascade={"persist"}, orphanRemoval=true)
     * @JoinTable( inverseJoinColumns={@JoinColumn(name="tag_name",
     *               referencedColumnName="name")})
     */
     protected $tags;

     // ...
}
```

IUT Institut Universitaire Technologie LENS

Lecture des données

Info

Dans la suite, on supposera que la variable `$em` référence une instance de la classe `EntityManager` qui nous permet d'accéder aux données.

- Pour récupérer un élément de la table à l'aide de sa clé

```
$post = $em->find('Blog\Entity\Post',14);
```

- Pour récupérer les données d'une table. La variable `$posts` est de type `Doctrine\Common\Collections\ArrayCollection`

```
$posts = $em->getRepository('Blog\Entity\Post')->
    ➥findAll();
```

- La lecture se fait, par défaut, en mode **LAZY**. Cela signifie que les objets liés par association ne seront pas récupérés. Ceux-ci seront récupérés dès l'appel d'une méthode qui les utilisent explicitement.

```
$tags = $post->getTags();
```

IUT Institut Universitaire Technologie LENS

Persistance des données

```
// ...
$post = new Post();
$post->setTitle('Mon arbre préféré')
        ->setBody('Une démonstration qui tient la route
  ➥...')
        ->setPublicationDate('2017-09-10');
 $post->persist($post);
 $em->flush();
 // ...
```

- La persistance est effective uniquement à la suite du `flush()`

Modification des données

```
// ...
$post = $em->find('Blog\Entity\Post',14);
$post->setBody('Une démonstration qui prend le vert ...')
$post->persist($post);
$em->flush();
// ...
```

- La modification est effective uniquement à la suite du `flush()`

Modification et suppression des données

```
// ...
$post = $em->find('Blog\Entity\Post',14);
$post->setBody('Une démonstration qui prend le vert ...')
$post->persist($post);
$em->flush();
// ...
```

```
// ...
$em->remove($post);
$em->flush();
// ...
```

- La modification et/ou la suppression sont effectives uniquement à la suite du `flush()`

Fonctions et associations

- Explications de la ligne suivante dans la déclaration de l'association entre `posts` et `tags`

```
  // ...
     * fetch="EAGER", cascade={"persist"}, orphanRemoval=true)
  // ...
```

- `fetch="EAGER"` indique que, par défaut, à chaque lecture d'un '`post`', les tags associés seront aussi récupérés.
- `cascade={"persist"}` indique que, lors de la sauvegarde d'un '`post`', les tags associés seront automatiquement sauvegardés.

```
$post = $em->find('Blog\Entity\Post',14);
$labelTags = ["Arbre", "Végétal", "Environnement"];
$tags=[];
foreach ($labelTags as $label) {
  $tag = new Tag();
  $tag->setName($label);
  $tags[] = $tag;
  $post ->addTag($tag );
}
  $post->persist($post);
  $em->flush();
```

- `orphanRemoval=true` indique que, lors de la suppression d'un '`post`', si les tags associés ne sont plus associés avec d'autres '`post`', ils seront automatiquement supprimés.

Introduction

- Le langage DQL (**Doctrine Query Language**) est équivalent du langage SQL mais il manipule des objets. Dans une requête on donnera des noms de classes PHP, des noms de propriété et pas des noms de table ni des noms de colonne.
- Avec le langage DQL, il sera possible de faire des opérations de type `SELECT`, `UPDATE`, `DELETE`.
- Les opérations de type `UPDATE` et `DELETE` seront utilisées pour faire des traitements de masse.
- L'opération `INSERT` n'est pas permise en DQL.
- Un exemple de requête

```
<?php
$query = $em->createQuery('SELECT u FROM App\Mod\User u WHERE u.age > 20')
    ➥;
$users = $query->getResult();
```

  - `u` est une instance de la classe `App\Mod\User`
  - La clause `FROM` est suivi de nom de classe
  - `u.age` est une propriété de la classe `App\Mod\User`

IUT Institut Universitaire Technologie LENS

Exemples

Voir la documentation

De nombreux exemples sont donnés dans la documentation de doctrine :

La documentation officielle de Doctrine

`http://docs.doctrine-project.org/en/latest/reference/dql-doctrine-query-language.html`