

# Java Persistence API (la suite)

---

**Note** : Quelques liens :

- API JPA 3.0 (JEE 10)<sup>a</sup>
- Tutorial JPA (JEE 9)<sup>b</sup>
- Une documentation sur JPQL<sup>c</sup>
- La page de Wikipedia<sup>d</sup>

a. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/package-summary.html>

b. <https://jakarta.ee/learn/docs/jakartaee-tutorial/9.1/persist/persistence-intro/persistence-intro.html>

c. [http://download.oracle.com/docs/cd/E13189\\_01/kodo/docs40/full/html/ejb3\\_overview\\_query.html](http://download.oracle.com/docs/cd/E13189_01/kodo/docs40/full/html/ejb3_overview_query.html)

d. [http://en.wikipedia.org/wiki/Java\\_Persistence\\_API](http://en.wikipedia.org/wiki/Java_Persistence_API)

## 1 Rendre la DAO plus générique

Afin d'éviter la création de nombreuses méthodes (quatre pour chaque entité), nous pouvons maintenant doter notre classe `Dao` de nouvelles méthodes génériques :

```
public <T> T find(Class<T> clazz, Object id) {
    return em.find(clazz, id);
}

public <T> Collection<T> findAll(String query, Class<T> clazz) {
    TypedQuery<T> q = em.createQuery(query, clazz);
    return q.getResultList();
}

public <T> T add(T entity) {
    em.persist(entity);
    return entity;
}

public <T> T update(T entity) {
    return em.merge(entity);
}

public <T> void remove(Class<T> clazz, Object pk) {
    T entity = em.find(clazz, pk);
    if (entity != null) {
        em.remove(entity);
    }
}
```

## 2 Relation Un-Plusieurs

Pour étudier la gestion des relations, nous allons créer une nouvelle entité Voiture (classe `Car`) et une relation père-fils entre une personne et plusieurs voitures. Le code de la classe `Car` est le suivant :

```

package myapp.jpa.model;

import jakarta.persistence.Basic;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
import lombok.ToString;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = "immatriculation")
public class Car {

    // properties
    @Id
    private String immatriculation;

    @Basic(optional = false)
    private String model;

    @ManyToOne(optional = true)
    @JoinColumn(name = "owner_id") // optionnelle
    @ToString.Exclude // afin d'éviter les boucles
    private Person owner;

    public Car(String immatriculation, String model) {
        this(immatriculation, model, null);
    }
}

```

**i Explications :** L'annotation `ManyToOne`<sup>a</sup> indique que la propriété code une relation fils-vers-le-père et l'annotation `JoinColumn`<sup>b</sup> permet de donner un nom à la colonne de la table qui va jouer ce rôle. **Attention :** pour manipuler cette relation, vous devrez passer obligatoirement par la modification de cette propriété. C'est donc le fils qui abrite (`owning`) cette relation.

a. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/ManyToOne.html>

b. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/JoinColumn.html>

▶▶ **Travail à faire :** Créez un test unitaire afin d'ajouter une voiture et son propriétaire (qui doit déjà exister).

## 2.1 Inverser la relation Un-Plusieurs

Nous allons ajouter à la classe `Person`, la liaison dans l'autre sens (père-vers-les-fils) avec une annotation `OneToMany`<sup>1</sup> appliquée à une collection de voitures. L'attribut `mappedBy` est particulièrement important car il indique le nom de la propriété (côté fils) qui code la relation un-plusieurs.

1. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/OneToMany.html>

```

public class Person {
    ...

    @OneToMany(//
        // chargement retardé (optionnel)
        fetch = FetchType.LAZY,
        // utile si plusieurs liaisons personne -> voiture
        mappedBy = "owner",
        cascade = {//
            // Modifier une personne -> modification des voitures
            CascadeType.MERGE,
            // Création d'une personne -> création des voitures
            CascadeType.PERSIST,
            // Suppression d'une personne -> suppression des voitures
            CascadeType.REMOVE
        }//
    )
    @ToString.Exclude // pour éviter les boucles
    private Set<Car> cars;

    public void addCar(Car c) {
        if (cars == null) {
            cars = new HashSet<>();
        }
        cars.add(c);
        c.setOwner(this);
    }

    ...
}

```

**Remarque :** Vous n'êtes pas obligé de préciser les attributs de `@OneToMany`.

### ► Travail à faire :

- Le lien père-vers-les-fils est géré en mode retardé (`LAZY`). Modifiez la méthode `findPerson` pour être sûr que les voitures soient chargées (utilisez simplement la méthode `size()` sur la collection de voitures pour forcer le chargement).
- modifiez votre classe de test unitaire afin d'ajouter des voitures (en insertion de personne et en mise à jour de personne).
- Maintenant que nous avons une relation, préparez une requête nommée (annotation `NamedQuery`<sup>a)</sup> pour renvoyer les personnes qui possèdent un modèle de voiture. Prévoyez la méthode ci-dessous et un test unitaire :

```

public List<Person> findPersonsByCarModel(String model) {
    ...
}

```

a. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/NamedQuery.html>

## 2.2 Ordonner les relations

Il est souvent utile d'ordonner les collections récupérées par JPA. Vous pouvez le faire simplement en ajoutant l'annotation `OrderBy`<sup>2</sup> comme le montre la nouvelle version de la classe `Person` ci-dessous :

2. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/OrderBy.html>

```
...  
  
@OneToMany(...)  
@OrderBy("immatriculation ASC")  
private Set<Car> cars;  
  
...
```

►► Travail à faire : Vérifiez l'ordre lors des récupérations de voitures.

### 3 Relation Plusieurs-Plusieurs

Nous allons mettre en place une relation plusieurs-plusieurs entre les personnes et des films. Pour ce faire, nous allons ajouter la classe `Movie` définie comme suit :

```
package myapp.jpa.model;  
  
import jakarta.persistence.Basic;  
import jakarta.persistence.Entity;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.Id;  
  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
@Entity  
@Data  
@NoArgsConstructor  
public class Movie {  
  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    @Basic(optional = false)  
    private String name;  
  
    public Movie(String name) {  
        this.name = name;  
    }  
  
}
```

►► **Travail à faire** : Modifiez la classe `Person` en ajoutant le code suivant :

```
...

@ManyToMany(cascade = { CascadeType.MERGE, CascadeType.PERSIST })
// @JoinTable est optionnelle (afin de préciser les tables)
@JoinTable(
    name = "Person_Movie",
    joinColumns = { @JoinColumn(name = "id_person") },
    inverseJoinColumns = { @JoinColumn(name = "id_movie") }
)
@ToString.Exclude
Set<Movie> movies;

public void addMovie(Movie movie) {
    if (movies == null) {
        movies = new HashSet<>();
    }
    movies.add(movie);
}

...
```

Les annotations `ManyToMany`<sup>a</sup> et `JoinTable`<sup>b</sup> vont nous permettre de définir cette relation et de créer la table de correspondance. Dans notre exemple, la relation n'est accessible que via la classe `Person`. Nous aurions pu également la définir dans la classe `Movie` en inversant les attributs `joinColumns` et `inverseJoinColumns`.

a. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/ManyToMany.html>

b. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/JoinTable.html>

**⚠ Attention** : c'est la classe contenant l'annotation `JoinTable`<sup>a</sup> qui abrite (*owning*) la relation. Toute modification de cette relation doit donc passer par cette classe. En d'autres termes, si vous souhaitez ajouter un film à une personne, vous devez charger la personne, puis le film et ajouter le film à la personne.

a. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/JoinTable.html>

►► **Travail à faire** : Ajoutez un test unitaire afin de vérifier l'ajout d'un film à une personne et sa suppression.

## 4 Relation Un-Un

La relation Un-Un (où l'un des côtés peut être optionnel) permet de lier l'existence de deux entités. Cette relation passe par l'annotation `OneToOne`<sup>3</sup>. La JavaDoc présente deux alternatives que nous ne détaillerons pas dans cette séance.

►► **Travail à faire** : Vous pouvez, à titre d'exemple, associer une personne à un CV (et vice-versa) mais en autorisant les personnes sans CV (et non pas les CV sans personne). Pour ce faire, vous devez créer une entité CV et mettre en place les liaisons.

3. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/OneToOne.html>

## 5 Les relations vers des types simples

Si vous souhaitez associer des types simples (chaines, entiers, décimaux) à une entité, vous pouvez le faire simplement avec `ElementCollection` <sup>4</sup>. À titre d'exemple, nous allons ajouter une collection triée de prénoms à notre personne ainsi qu'une collection ordonnée de surnoms :

```
...

@ElementCollection(fetch = FetchType.LAZY)
// annotations optionnelles
@CollectionTable(name = "FIRST_NAMES", //
                 joinColumns = @JoinColumn(name = "person_id"))
@Column(name = "first_name")
Collection<String> firstNames;

@ElementCollection(fetch = FetchType.LAZY)
@OrderColumn(name = "position")
List<String> nickNames;

...
```

▶▶ **Travail à faire** : Vérifiez la structure relationnelle créée.

**Note** : Vous pouvez utiliser des classes embarquées à la place des types simples.

## 6 Traitement de la concurrence

### 6.1 Pose de verrou

Cette approche classique consiste à verrouiller les lignes (donc les objets) de la base après leur récupération pour pouvoir les modifier à sa guise sans craindre qu'une autre transaction ne les altère. Pour ce faire nous allons ajouter un paramètre au chargement `LockModeType` <sup>5</sup> :

```
public void changeFirstName(long idPerson, String firstName) {
    Person p = em.find(Person.class, idPerson, LockModeType.PESSIMISTIC_WRITE);
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
    }
    p.setFirstName(firstName);
}
```

▶▶ **Vérification** : Vous pouvez tester ce code en l'exécutant plusieurs fois ou en essayant une modification via le client du SGBDR (seulement si vous utilisez **MySQL**).

### 6.2 Gestion optimiste

Dans la gestion optimiste, les données sont versionnées et JPA va vérifier que le numéro de version ne change pas entre la lecture et la modification. Ce mécanisme est utile quand l'objet reste longtemps en mémoire entre sa lecture et sa modification (par exemple pour un formulaire de saisie).

4. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/ElementCollection.html>

5. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/LockModeType.html>

Nous avons déjà prévu une propriété *version* dans le JavaBean `Person` (annotation `Version`<sup>6</sup>).

▶▶ **Vérification** : Créez un test unitaire composé de trois parties :

- lecture d'une personne dans *p1*,
- lecture de la même personne dans *p2*,
- changement de *p2* et mise à jour,
- changement de *p1* et mise à jour,

Vous devriez obtenir une erreur d'exécution indiquant un problème dans la gestion optimiste du verrou : le numéro de version a changé lors de la sauvegarde de *p1*.

---

6. <https://jakarta.ee/specifications/platform/10/apidocs//jakarta/persistence/Version.html>