Travaux pratique : Jpa n°3 : Les associations Jpa

Introduction

Ce TP consiste à utiliser les annotations des associations offerts par JPA

- One-to-many
- Many-to-one

Éléments à maîtriser

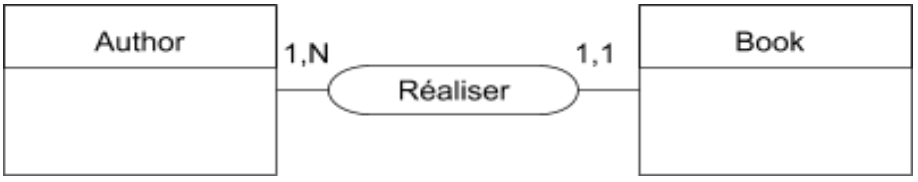
@OneToMany
@ManyToOne
@JoinColumn
2 Fetch types : eager et lazy
6 cascade types : persist, remove, refresh, merge, detach, all

Pré-requis

Il est obligatoire d'avoir realiser les travaux pratiques : Jpa n°1 et Jpa n°2

Les associations un-à-plusieurs ou plusieurs à un @One-to-many et @Many-to-one

Notre exemple prototypique est la mise en place de la relation entre un book et son (unique) auteur.



Un premier constat, très important: en java nous pouvons représenter cette association de trois manières.

- dans la classe book, on place un lien vers la classe auteur unidirectionnel, à gauche
- dans la classe auteur, on place des liens vers les books qu'il a réalisé (unidirectionnel, centre)
- On place les liens dans les deux classes book et auteur (bidirectionnel, droite)

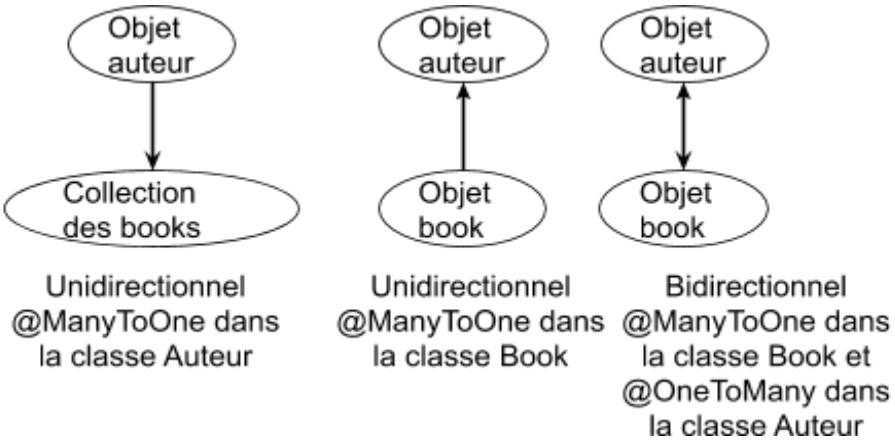


FIGURE 1: TROIS POSSIBILITÉS POUR UNE ASSOCIATION UN À PLUSIEURS

RAPPELONS QUE DANS UNE BASE RELATIONNELLE, LE PROBLÈME NE SE POSE PAS: UNE ASSOCIATION REPRÉSENTÉE PAR UNE CLÉ ÉTRANGÈRE EST PAR NATURE BIDIRECTIONNELLE. CETTE DIFFÉRENCE EST DUE AUX DEUX PARADIGMES OPPOSÉS SUIVANTS:

- LE MODÈLE RELATIONNEL D'UNE PART (QUI CONSIDÈRE DES *ENSEMBLES* SANS LIENS EXPLICITES ENTRE EUX, UNE ASSOCIATION ÉTANT RECONSTITUÉ PAR DES JOINTURES)
- LE MODÈLE OBJET DE JAVA (QUI ÉTABLIT UNE NAVIGATION DANS LES OBJETS INSTANCIÉS GRÂCE AUX RÉFÉRENCES D'OBJETS).

CECI ÉTANT POSÉ, VOYONS COMMENT NOUS POUVONS REPRÉSENTER LES TROIS SITUATIONS EN UTILISANT LES ANNOTATIONS JPA.

L'ASSOCIATION @ONE-TO-MANY

1. DANS LE PACKAGE MODELS DU PROJET PRÉCÉDENT CRÉER UNE CLASSE AUTHOR.

```
package model;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Author {
	@Id
	@GeneratedValue(strategy = GenerationType.TABLE)
	private int id;
	private String name;
	@Override
	public String toString() {
```

```
		return "Author [id=" + id + ", name=" + name + "]";
	}
	public Author(String name) {
		super();
		this.name = name;
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
}
```

2. Ajouter l'attribut books dans la classe auteur. son type est une collection de type Set. Cet attribut doit être précédé par @OneToMany comme montrer ci-après.

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy = "author")
// Nom de la colonne porteuse de la clé étrangère dans la table des books
@JoinColumn(name="fk_author")
private Set<Book> books;
// Générer le Getter et le setter pour l'attribut books
```

Les types des cascades sont illustrés dans le tableau suivant:

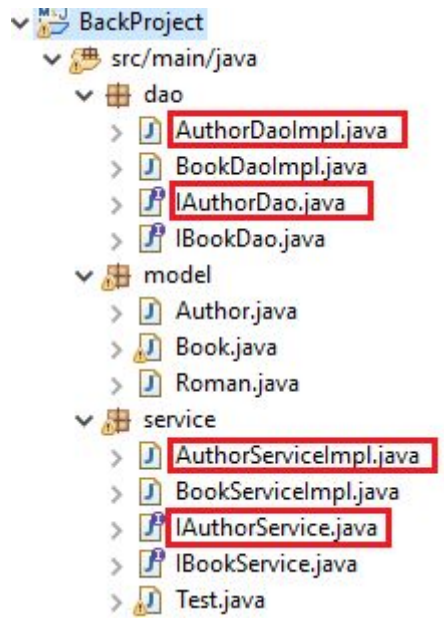
CascadeType.PERSIST	suite à l'appel de save(author a) ou bien persiste(author a), aussi les books de cet auteur seront sauvegardés ou persisté
CascadeType.MERGE	Suite à l'appel de marge(author a) , aussi la méthode merge(book b) sera appelé pour tous les books de l'auteur en question
CascadeType.REFRESH	Suite à l'appel de marge(author a) , aussi la méthode merge(book b) sera appelée pour tous les books de l'auteur en question
CascadeType.REMOVE	Suite à l'appel de remove(author a) , aussi la méthode remove(book b) sera appelée pour tous les books de l'auteur en question
CascadeType.DETACH	Suite à l'appel de detach(author a) , aussi la méthode detach(book b) sera appelée pour tous les books de l'auteur en question

CascadeType.ALL	tous les cascades précédents seront appliqués aux books de l'auteur en question.

Les types des fetchs possibles sont illustrés dans le tableau suivant

FetchType.EAGER	charger le père et les fils au même temps. Dans notre cas charger l'auteur et ses books au meme temps.
FetchType.LAZY	Charger le père uniquement. Les fils seront chargés lors du besoin: l'appel du setter par exemple

3. Testons : Compléter le projet back office en ajoutant les classes Dao et Service pour la classe modèle : Author.



Mêmes classes que celle de la classe book a condition de changer book par author. les classe à créer sont en rouge. Suivre l'ordre suivant:

1. Creer IauthorDao.java
2. Creer AuthorDaoImpl.java
3. Creer IauthorService.java
4. Creer AuthorServiceImpl.java

4. Modifier ensuite la classe de test en ajoutant une instance de author service. Le changement à faire est le suivant:

```
package service;
import java.util.HashSet;
import java.util.Set;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import model.Author;
import model.Book;

public class Test {
	public static void main(String[] args) {
		ApplicationContext ap= new ClassPathXmlApplicationContext("spring.xml");
		IAuthorService author_service=
(IAuthorService)ap.getBean("author-service");
		IBookService book_service= (IBookService)ap.getBean("book-service");

		Book book1 = new Book("Titre1");
```

```
            Book book2 = new Book("Titre2");
            Book book3 = new Book("Titre2");
            // Collection of books
            Set<Book> books= new HashSet<>();
            books.add(book1);
            books.add(book2);
            books.add(book3);
            // Affectation to the author
            Author author=new Author("Hassan");
            author.setBooks(books);
            author_service.add(author);
      }
}
```

5. Faire l'exécution de test pour insérer un auteur. Vérifier que les books de l'auteur sont aussi inserer automatiquement. La création des tables est la suivante. Vérifier la clé étrangère

```
Hibernate:
    create table Author (
        id integer not null,
        name varchar(255),
        primary key (id)
    )

Hibernate:
    create table tbooks (
        id integer not null,
        title varchar(255),
        fk_author integer,
        primary key (id)
    )

Hibernate:
    alter table tbooks
        add constraint FK63rc994gp19xb7mjj07hh41kp
        foreign key (fk_author)
        references Author (id)
```

Les requêtes d'insertion générées sont comme suivant. Analyser ces requêtes

```
Hibernate:
    insert
    into
        Author
        (name, id)
    values
        (?, ?)
Hibernate:
    insert
```

```
    into
        tbooks
        (title, id)
    values
        (?, ?)
Hibernate:
    insert
    into
        tbooks
        (title, id)
    values
        (?, ?)
Hibernate:
    insert
    into
        tbooks
        (title, id)
    values
        (?, ?)
Hibernate:
    update
        tbooks
    set
        fk_author=?
    where
        id=?
Hibernate:
    update
        tbooks
    set
        fk_author=?
    where
        id=?
Hibernate:
    update
        tbooks
    set
        fk_author=?
    where
        id=?
```

LES DONNÉES AU NIVEAU DE LA BASE SONT COMME SUIVANT:

```
1 SELECT * FROM author a;
```

id	name
1	Hassan

```
1 SELECT * FROM `books`.`tbooks`
```

id	title	fk_author
2	Titre2	1
3	Titre2	1
4	Titre1	1

6. Testons maintenant la sélection d'un auteur par son id. Est ce que ces books seront aussi sélectionnés ? c'est le role de l'attribut fetch de l'association.
7. Ajouter la méthode selectById et findById dans la couche dao et la couche service de Auteur. On peut appeler la méthode find de l'interface EntityManager

IAuthorDao.java	`Author selectById(int id);`

AuthorDaoImpl.java	`@Override`
	`public Author selectById(int id) {`
	`  return em.find(Author.class, id);`
	`}`

IAuthorService.java	`Author findById(int id);`

AuthorServiceImpl.java	`@Override`
	`public Author findById(int id) {`
	`  return dao.selectById(id);`
	`}`

8. Modifier la classe de test pour charger un auteur via la méthode findById implémentée dans le question précédente.

```
package service;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import model.Author;

public class Test {
	public static void main(String[] args) {
		ApplicationContext ap= new ClassPathXmlApplicationContext("spring.xml");
		IAuthorService author_service=
(IAuthorService)ap.getBean("author-service");
		Author author = author_service.findById(1);
	}
}
```

Remarquer La requête de sélection générée si l'attribut fetch=eager dans l'annotation @OneToMany

```
Hibernate:
    select
        author0_.id as id1_0_0_,
        author0_.name as name2_0_0_,
        books1_.fk_author as fk_autho3_2_1_,
        books1_.id as id1_2_1_,
        books1_.id as id1_2_2_,
        books1_.title as title2_2_2_,
        books1_.themeLitteraire as themeLit1_1_2_,
```

```
        books1_.clazz_ as clazz_2_
    from
        Author author0_
    left outer join
        (
            select
                id,
                title,
                fk_author,
                null as themeLitteraire,
                0 as clazz_
            from
                tbooks
            union
            select
                id,
                title,
                fk_author,
                themeLitteraire,
                1 as clazz_
            from
                Roman
        ) books1_
            on author0_.id=books1_.fk_author
    where
        author0_.id=?
```

Changer le fetcha lazy . Remarquer La requête de sélection générée si l'attribut fetch=lazy dans l'annotation @OneToMany

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
```

La requête générée est la suivante:

```
Hibernate:
    select
        author0_.id as id1_0_0_,
        author0_.name as name2_0_0_
    from
        Author author0_
    where
        author0_.id=?
```

Quelle est la différence entre le fetch eager et lazy?

L'association @Many-to-one

1. Modifier la classe book en ajoutant un attribut de type author et l'annotation @ManyToOne [Many books to one author]

```
@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
```

```
	@JoinColumn(name = "fk_author")
	private Author author;
	public Author getAuthor() {
		return author;
	}
	public void setAuthor(Author author) {
		this.author = author;
	}
```

2. Ajouter un book contenant un auteur et vérifier que l'auteur est ajoute aussi automatiquement dans la base. La classe de test est la suivante:

```
package service;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import model.Author;
import model.Book;

public class Test {
	public static void main(String[] args) {
		ApplicationContext ap= new ClassPathXmlApplicationContext("spring.xml");
		IBookService book_service= (IBookService)ap.getBean("book-service");
		Author author= new Author("author1");
		Book book = new Book("Java");
		book.setAuthor(author);
		book_service.add(book);
	}
}
```

Suite à l'exécution de la classe test, la requête suivante sera générée

```
Hibernate:
    insert
    into
        Author
        (name, id)
    values
        (?, ?)
Hibernate:
    insert
    into
        tbooks
        (fk_author, title, id)
    values
        (?, ?, ?)
```

3. Testons maintenant la sélection d'un book par son id. Est ce que son auteur sera aussi sélectionné ? c'est le role de l'attribut fetch de l'association.
4. ajouter la méthode selectById et findById dans la couche dao et la couche service de la classe book. On peut appeler la méthode find de l'interface EntityManager

IBookDao.java	`Book selectById(int id);`

BookDaoImpl.java	`@Override`
	`public Book selectById(int id) {`
	`  return em.find(Book.class, id);`
	`}`

IBookService.java	`Book findById(int id);`

BookServiceImpl.java	`@Override`
	`public Book findById(int id) {`
	`  return dao.selectById(id);`
	`}`

5. Modifier la classe de test pour charger un Book via sa méthode findById implémentée dans le question précédente.

```
package service;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import model.Book;

public class Test {
	public static void main(String[] args) {
		ApplicationContext ap= new ClassPathXmlApplicationContext("spring.xml");
		IBookService book_service= (IBookService)ap.getBean("book-service");

		Book book = book_service.findById(1);
		//System.out.println(book);
	}
}
```

Suite à l'exécution de la classe test si fetch=eager. vous aurez la requête suivante.

@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)

```
Hibernate:
    select
        book0_.id as id1_2_0_,
        book0_.fk_author as fk_autho3_2_0_,
        book0_.title as title2_2_0_,
        book0_.themeLitteraire as themeLit1_1_0_,
        book0_.clazz_ as clazz_0_,
        author1_.id as id1_0_1_,
        author1_.name as name2_0_1_
    from
        ( select
            id,
            title,
            fk_author,
            null as themeLitteraire,
            0 as clazz_
        from
            tbooks
```

```
        union
        select
            id,
            title,
            fk_author,
            themeLitteraire,
            1 as clazz_
        from
            Roman
    ) book0_
left outer join
    Author author1_
        on book0_.fk_author=author1_.id
where
    book0_.id=?
```

Suite à l'exécution de la classe test si fetch=lazy. vous aurez la requête suivante.

@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)

```
Hibernate:
    select
        book0_.id as id1_2_0_,
        book0_.fk_author as fk_autho3_2_0_,
        book0_.title as title2_2_0_,
        book0_.themeLitteraire as themeLit1_1_0_,
        book0_.clazz_ as clazz_0_
    from
        ( select
            id,
            title,
            fk_author,
            null as themeLitteraire,
            0 as clazz_
        from
            tbooks
        union
        select
            id,
            title,
            fk_author,
            themeLitteraire,
            1 as clazz_
        from
            Roman
    ) book0_
where
    book0_.id=?
```