

Relations JPA

Types de Relations

@OneToMany / @ManyToOne

Cas d'usage : Un auteur a plusieurs livres, un livre a un seul auteur.

Configuration Unidirectionnelle (☐ Rarement recommandée)

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToMany
    @JoinColumn(name = "author_id") // Crée une FK dans Book
    private List<Book> books = new ArrayList<>();
}

@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    // Pas de référence à Author
}
```



Problème : Génère des UPDATE supplémentaires !

Lancés automatiquement par Hibernate pour mettre à jour la clé étrangère.

Configuration Bidirectionnelle (☑ Recommandée)

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToMany(
        mappedBy = "author", // Référence au champ dans Book
        cascade = CascadeType.ALL,
        orphanRemoval = true,
```

```
        fetch = FetchType.LAZY           // Par défaut
    )
    private List<Book> books = new ArrayList<>();
    // Méthodes helper pour synchroniser les deux côtés
    public void addBook(Book book) {
        books.add(book);
        book.setAuthor(this);
    }
    public void removeBook(Book book) {
        books.remove(book);
        book.setAuthor(null);
    }
}

@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToOne(
        fetch = FetchType.LAZY,           // Recommandé
        optional = false                  // NOT NULL en base
    )
    @JoinColumn(name = "author_id")      // Nom de la FK
    private Author author;
}
```



Le côté **@ManyToOne** est TOUJOURS le propriétaire (owner) de la relation.

Toutes les modifications portant sur cette relation devront se faire côté **owner**.

@ManyToMany

Cas d'usage : Un étudiant suit plusieurs cours, un cours a plusieurs étudiants.

Configuration Unidirectionnelle

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private Set<Course> courses = new HashSet<>();
}
```

```
}  
  
@Entity  
public class Course {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    // Pas de référence à Student  
}
```

Configuration Bidirectionnelle (☐ Recommandée)

```
@Entity  
public class Student {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    @ManyToMany(  
        cascade = {CascadeType.PERSIST, CascadeType.MERGE},  
        fetch = FetchType.LAZY  
    )  
    @JoinTable(  
        name = "student_course",  
        joinColumns = @JoinColumn(name = "student_id"),  
        inverseJoinColumns = @JoinColumn(name = "course_id")  
    )  
    private Set<Course> courses = new HashSet<>();  
    public void enrollCourse(Course course) {  
        courses.add(course);  
        course.getStudents().add(this);  
    }  
    public void unenrollCourse(Course course) {  
        courses.remove(course);  
        course.getStudents().remove(this);  
    }  
}  
  
@Entity  
public class Course {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    @ManyToMany(mappedBy = "courses")  
    private Set<Student> students = new HashSet<>();  
}
```

ManyToMany avec attributs

Ce n'est pas vraiment une ManyToMany...

Avec @EmbeddedId

```
@Entity
public class Enrollment {
    @EmbeddedId
    private EnrollmentId id;
    @ManyToOne
    @MapsId("studentId")
    private Student student;
    @ManyToOne
    @MapsId("courseId")
    private Course course;
    private LocalDate enrollmentDate;
    private Integer grade;
}

@Embeddable
public class EnrollmentId implements Serializable {
    private Long studentId;
    private Long courseId;
    // equals() et hashCode()
}

@Entity
public class Student {
    @OneToMany(mappedBy = "student", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<Enrollment> enrollments = new HashSet<>();
}

@Entity
public class Course {
    @OneToMany(mappedBy = "course", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<Enrollment> enrollments = new HashSet<>();
}
```

Sans @EmbeddedId, mais avec Id supplémentaire (☐ Recommandée)

```
@Entity
@Table(
    name = "enrollment",
    indexes = {
        @Index(
            name = "idx_enrollment_pk",
            columnList = "student_id, course_id",
            unique = true
        )
    }
)
```

```
public class Enrollment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; // ← Clé primaire auto-générée simple
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "student_id", nullable = false)
    private Student student;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "course_id", nullable = false)
    private Course course;
    private Integer grade;
    private LocalDate enrollmentDate;
    // Constructeurs
    // Getters/Setters
    // equals et hashCode
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Enrollment)) return false;
        Enrollment that = (Enrollment) o;
        return id != null && id.equals(that.getId());
    }
    @Override
    public int hashCode() {
        return getClass().hashCode();
    }
    @Override
    public String toString() {
        return "Enrollment{" +
            "id=" + id +
            ", student=" + (student != null ? student.getName() : "null") +
            ", course=" + (course != null ? course.getName() : "null") +
            ", grade=" + grade +
            ", enrollmentDate=" + enrollmentDate +
            '}';
    }
}
```

Avantages de cette approche :



- Code simple et lisible
- Pas de classe @Embeddable à gérer
- Fonctionne parfaitement avec Spring Data
- L'index unique garantit l'unicité (student_id, course_id)
- Performance identique à une PK composite

FetchType : LAZY vs EAGER

LAZY (☐ Recommandé par défaut)

```
@Entity
public class Book {
    @ManyToOne(fetch = FetchType.LAZY)
    private Author author;
}
```

Comportement :

- Les données ne sont chargées **que si on y accède**
- Évite les requêtes inutiles
- Peut lancer une `LazyInitializationException` si la session Hibernate est fermée

Exemple :

```
Book book = bookRepository.findById(1L).get();
// Pas de requête vers Author ici

String authorName = book.getAuthor().getName();
// ← Requête SQL lancée ici (si session ouverte)
```

LazyInitializationException : Se produit quand on accède à une relation LAZY en dehors d'une transaction.



Solutions :

- Utiliser `@Transactional` sur la méthode
- Utiliser un `@EntityGraph`
- Utiliser une query avec `JOIN FETCH`

EAGER (⚠ À utiliser avec précaution)

```
@Entity
public class Book {
    @ManyToOne(fetch = FetchType.EAGER)
    private Author author;
}
```

Comportement :

- Les données sont chargées **immédiatement** avec l'entité principale
- Peut créer des problèmes de performance (N+1)
- Utile seulement si on a **toujours** besoin de la relation

FetchType par défaut

| Annotation | FetchType par défaut |
|------------|--|
| @ManyToOne | LAZY (depuis Hibernate 5.1+) ou EAGER (JPA standard) |
| @OneToOne | EAGER |

| Annotation | FetchType par défaut |
|-------------|----------------------|
| @OneToMany | LAZY |
| @ManyToMany | LAZY |



Bonne pratique : Toujours spécifier explicitement le FetchType pour éviter les surprises.

Optimiser avec JOIN FETCH

```
public interface BookRepository extends JpaRepository<Book, Long> {
    // [] Charge Book + Author en une seule requête
    @Query("SELECT b FROM Book b JOIN FETCH b.author WHERE b.id = :id")
    Optional<Book> findByIdWithAuthor(@Param("id") Long id);
    // [] Charge tous les Books + leurs Authors
    @Query("SELECT DISTINCT b FROM Book b JOIN FETCH b.author")
    List<Book> findAllWithAuthors();
}
```

Optimiser avec @EntityGraph

```
public interface BookRepository extends JpaRepository<Book, Long> {
    @EntityGraph(attributePaths = {"author"})
    Optional<Book> findById(Long id);
    @EntityGraph(attributePaths = {"author", "publisher"})
    List<Book> findAll();
}
```

Cascade

Les CascadeType définissent quelles opérations sont propagées aux entités liées.

Les différents types

```
public enum CascadeType {
    PERSIST, // entityManager.persist()
    MERGE, // entityManager.merge()
    REMOVE, // entityManager.remove()
    REFRESH, // entityManager.refresh()
    DETACH, // entityManager.detach()
    ALL // Tous les types ci-dessus
}
```

Exemples

PERSIST : Propager la création

```
@Entity
public class Author {
    @OneToMany(
        mappedBy = "author",
        cascade = CascadeType.PERSIST
    )
    private List<Book> books = new ArrayList<>();
}

// Utilisation
Author author = new Author("John Doe");
Book book1 = new Book("Book 1");
Book book2 = new Book("Book 2");

author.addBook(book1);
author.addBook(book2);

entityManager.persist(author);
// ☐ author, book1 et book2 sont tous persistés automatiquement
```

REMOVE : Propager la suppression (⚠ Dangereux)

```
@Entity
public class Author {
    @OneToMany(
        mappedBy = "author",
        cascade = CascadeType.REMOVE // ⚠ Attention !
    )
    private List<Book> books = new ArrayList<>();
}

// Utilisation
Author author = authorRepository.findById(1L).get();
authorRepository.delete(author);
// ⚠ Tous les livres de cet auteur sont aussi supprimés !
```

CascadeType.REMOVE vs orphanRemoval :



- REMOVE : Supprime les entités liées quand on supprime le parent
- orphanRemoval : Supprime les entités liées quand on les retire de la collection

Exemple :

```
// Avec CascadeType.REMOVE uniquement
```



```
author.getBooks().remove(book); // ☐ book reste en base avec author_id =
NULL
// Avec orphanRemoval = true
author.getBooks().remove(book); // ☐ book est supprimé de la base
```

ALL : Propager toutes les opérations

```
@Entity
public class Author {
    @OneToMany(
        mappedBy = "author",
        cascade = CascadeType.ALL,
        orphanRemoval = true
    )
    private List<Book> books = new ArrayList<>();
}
```

Bonnes pratiques Cascade

Recommandations :



- **@OneToMany** : Utiliser `CascadeType.ALL + orphanRemoval = true` pour les relations de composition (ex: `Order` → `OrderItems`)
- **@ManyToOne** : NE PAS utiliser de cascade (sauf cas très particuliers)
- **@ManyToMany** : Utiliser `CascadeType.PERSIST` et `CascadeType.MERGE` uniquement
- **CascadeType.REMOVE** : Attention aux suppressions en cascade non voulues !

orphanRemoval

`orphanRemoval = true` supprime automatiquement les entités "orphelines" (qui ne sont plus dans la collection parente).

Exemple

```
@Entity
public class Author {
    @OneToMany(
        mappedBy = "author",
        cascade = CascadeType.ALL,
        orphanRemoval = true // ← Suppression automatique des orphelins
    )
```

```

private List<Book> books = new ArrayList<>();
public void removeBook(Book book) {
    books.remove(book);
    book.setAuthor(null);
}
}

// Utilisation
Author author = authorRepository.findById(1L).get();
Book book = author.getBooks().get(0);

author.removeBook(book);
authorRepository.save(author);
// ☐ book est automatiquement supprimé de la base

```

Différence avec CascadeType.REMOVE

| Opération | orphanRemoval | CascadeType.REMOVE |
|------------------------------------|------------------------|------------------------------|
| Supprimer le parent | ☐ Supprime les enfants | ☐ Supprime les enfants |
| Retirer un enfant de la collection | ☐ Supprime l'enfant | ☐ L'enfant reste (FK = NULL) |



orphanRemoval = true est idéal pour les relations de **composition** (parent-enfant fort) où l'enfant n'a pas de sens sans le parent.

Exemples : Order → OrderItem, Invoice → InvoiceLine, Blog → Comment

Incidence du Owner (propriétaire) de la Relation

Définition

Le **owner** (propriétaire) de la relation est le côté qui :

- Possède la colonne de clé étrangère (FK) en base de données
- Est responsable de la synchronisation avec la base de données
- **N'a PAS** l'attribut mappedBy



Règle absolue : Pour qu'une modification de relation soit persistée, elle **DOIT** être faite côté owner.

Schéma de base de données

```

@Entity
public class Author {
    @OneToMany(mappedBy = "author") // ☐ Non owner
    private List<Book> books;
}

```

```
}  
  
@Entity  
public class Book {  
    @ManyToOne  
    @JoinColumn(name = "author_id") // ☐ Owner  
    private Author author;  
}
```

En base de données :

```
CREATE TABLE author (  
    id BIGINT PRIMARY KEY,  
    name VARCHAR(255)  
);  
  
CREATE TABLE book (  
    id BIGINT PRIMARY KEY,  
    title VARCHAR(255),  
    author_id BIGINT,  
    CONSTRAINT fk_author FOREIGN KEY (author_id) REFERENCES author(id)  
);
```

Conséquence : Seul le Owner peut persister la relation

☐ Ce qui NE fonctionne PAS

```
Author author = new Author("John Doe");  
Book book = new Book("My Book");  
  
// ☐ Modifier uniquement le côté inverse (non-owner)  
author.getBooks().add(book);  
entityManager.persist(author);  
entityManager.flush();  
  
// Résultat en base :  
// book.author_id = NULL ☐  
// La FK n'est PAS remplie !
```

Pourquoi ? Hibernate ignore le côté @OneToMany sans mappedBy pour la persistance.

☐ La bonne méthode

```
Author author = new Author("John Doe");  
Book book = new Book("My Book");  
  
// ☐ Modifier le côté owner (@ManyToOne)
```

```
book.setAuthor(author);

entityManager.persist(author);
entityManager.persist(book);
entityManager.flush();

// Résultat en base :
// book.author_id = 1
```

▣ La méthode recommandée : Méthodes helper

```
@Entity
public class Author {
    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Book> books = new ArrayList<>();
    // ▣ Méthode qui synchronise les deux côtés
    public void addBook(Book book) {
        books.add(book); // Côté inverse (pour cohérence en mémoire)
        book.setAuthor(this); // Côté owner (pour persistance)
    }
    public void removeBook(Book book) {
        books.remove(book);
        book.setAuthor(null);
    }
}

// Utilisation
Author author = new Author("John Doe");
Book book = new Book("My Book");

author.addBook(book); // ▣ Les deux côtés sont synchronisés

authorRepository.save(author); // ▣ Tout est persisté correctement
```

Performance : Unidirectionnel vs Bidirectionnel

Configuration Unidirectionnelle @OneToMany (▣ Mauvaise perf)

```
@Entity
public class Author {
    @OneToMany
    @JoinColumn(name = "author_id")
    private List<Book> books = new ArrayList<>();
}
```

Problème : Génère des requêtes supplémentaires !

```

INSERT INTO author (name) VALUES ('John Doe');
INSERT INTO book (title) VALUES ('Book 1');
INSERT INTO book (title) VALUES ('Book 2');

-- Δ UPDATE supplémentaires pour chaque livre !
UPDATE book SET author_id = 1 WHERE id = 1;
UPDATE book SET author_id = 1 WHERE id = 2;

```

Configuration Bidirectionnelle (Meilleure perf)

```

@Entity
public class Author {
    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
    private List<Book> books = new ArrayList<>();
}

@Entity
public class Book {
    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;
}

```

Requêtes optimisées :

```

INSERT INTO author (name) VALUES ('John Doe');
INSERT INTO book (title, author_id) VALUES ('Book 1', 1); -- FK directement
INSERT INTO book (title, author_id) VALUES ('Book 2', 1); -- Pas d'UPDATE

```

ManyToMany : Qui est le owner ?

```

@Entity
public class Student {
    @ManyToMany // Le côté SANS mappedBy est le owner
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private Set<Course> courses = new HashSet<>();
}

@Entity
public class Course {
    @ManyToMany(mappedBy = "courses") // Côté inverse
    private Set<Student> students = new HashSet<>();
}

```

Conséquence : Pour ajouter une relation, il faut modifier `Student . courses` (le owner) :

```
// □ Modifier le owner
student.getCourses().add(course);
studentRepository.save(student);

// □ Modifier l'inverse (ignoré par Hibernate)
course.getStudents().add(student);
courseRepository.save(course); // Ne persiste PAS la relation !
```

Toujours synchroniser les deux côtés avec des méthodes helper :



```
public void enrollCourse(Course course) {
    courses.add(course);           // Owner
    course.getStudents().add(this); // Inverse
}
```

Queries et Problèmes Courants

Problème N+1

Le problème N+1 se produit quand on charge une collection d'entités, puis qu'on accède à une relation, déclenchant **une requête SQL par entité**.

Exemple du problème

```
// 1 requête pour charger tous les livres
List<Book> books = bookRepository.findAll();

// N requêtes supplémentaires (une par livre) pour charger les auteurs !
for (Book book : books) {
    System.out.println(book.getAuthor().getName()); // ▲□ 1 requête SQL
}

// Total : 1 + N requêtes !
```

```
SELECT * FROM book;           -- Requête 1
SELECT * FROM author WHERE id = 1; -- Requête 2
SELECT * FROM author WHERE id = 2; -- Requête 3
```

```
SELECT * FROM author WHERE id = 3;    -- Requête 4
...
```

Solution 1 : JOIN FETCH

```
public interface BookRepository extends JpaRepository<Book, Long> {
    @Query("SELECT b FROM Book b JOIN FETCH b.author")
    List<Book> findAllWithAuthors();
}

// Utilisation
List<Book> books = bookRepository.findAllWithAuthors();
// ☐ 1 seule requête SQL avec JOIN !

for (Book book : books) {
    System.out.println(book.getAuthor().getName()); // ☐ Pas de requête
}
```

```
-- ☐ Une seule requête !
SELECT b.*, a.*
FROM book b
INNER JOIN author a ON b.author_id = a.id;
```

Solution 2 : @EntityGraph

```
public interface BookRepository extends JpaRepository<Book, Long> {
    @EntityGraph(attributePaths = {"author"})
    List<Book> findAll();
    @EntityGraph(attributePaths = {"author", "publisher"})
    Optional<Book> findById(Long id);
}
```

Solution 3 : @Fetch(FetchMode.SUBSELECT)

```
@Entity
public class Author {
    @OneToMany(mappedBy = "author")
    @Fetch(FetchMode.SUBSELECT) // ☐ 2 requêtes au lieu de N+1
    private List<Book> books = new ArrayList<>();
}
```

```
SELECT * FROM author; -- Requête 1

-- Requête 2 : charge tous les livres en une fois
SELECT * FROM book WHERE author_id IN (
    SELECT id FROM author
);
```

LazyInitializationException

Erreur : org.hibernate.LazyInitializationException: could not initialize proxy - no Session

Cause : Accès à une relation LAZY en dehors d'une transaction/session Hibernate.

Exemple du problème

```
@Service
public class BookService {
    public Book getBook(Long id) {
        return bookRepository.findById(id).get();
    } // ← La transaction se termine ici
}

// Controller
Book book = bookService.getBook(1L);
String authorName = book.getAuthor().getName();
// ☐ LazyInitializationException !
```

Solution 1 : @Transactional

```
@Service
public class BookService {
    @Transactional(readOnly = true)
    public Book getBook(Long id) {
        Book book = bookRepository.findById(id).get();
        // Accéder aux relations LAZY ici
        book.getAuthor().getName(); // ☐ OK, on est dans la transaction
        return book;
    }
}
```

Solution 2 : JOIN FETCH

```
@Query("SELECT b FROM Book b JOIN FETCH b.author WHERE b.id = :id")
Optional<Book> findByIdWithAuthor(@Param("id") Long id);
```

Solution 3 : @EntityGraph

```
@EntityGraph(attributePaths = {"author"})
Optional<Book> findById(Long id);
```

Solution 4 : DTO Projection

```
public interface BookDTO {
    Long getId();
    String getTitle();
    String getAuthorName(); // author.name
}

@Query("SELECT b.id as id, b.title as title, b.author.name as authorName FROM Book
b WHERE b.id = :id")
Optional<BookDTO> findBookDTOById(@Param("id") Long id);
```

Bonnes Pratiques

1. Toujours initialiser les collections

```
@Entity
public class Author {
    @OneToMany(mappedBy = "author")
    private List<Book> books = new ArrayList<>(); // ☐ Initialiser directement
}
```

Évite : NullPointerException et facilite l'ajout d'éléments.

2. Utiliser des méthodes helper

```
@Entity
public class Author {
    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Book> books = new ArrayList<>();
    // ☐ Encapsule la logique de synchronisation
    public void addBook(Book book) {
        books.add(book);
        book.setAuthor(this);
    }
}
```

```
    }  
    public void removeBook(Book book) {  
        books.remove(book);  
        book.setAuthor(null);  
    }  
}
```

3. Implémenter equals() et hashCode() correctement

```
@Entity  
public class Book {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof Book)) return false;  
        Book book = (Book) o;  
        return id != null && id.equals(book.getId());  
    }  
    @Override  
    public int hashCode() {  
        return getClass().hashCode(); // ☐ Stable, ne change pas  
    }  
}
```



NE PAS utiliser tous les champs dans hashCode() : Le hash doit rester constant, même si l'entité change.

NE PAS utiliser l'ID dans hashCode() : L'ID peut être null avant persistance.

4. Utiliser Set au lieu de List pour @ManyToMany

```
// ☐ Recommandé  
@ManyToMany  
private Set<Course> courses = new HashSet<>();  
  
// ☐ Éviter (problèmes avec equals/hashCode et doublons)  
@ManyToMany  
private List<Course> courses = new ArrayList<>();
```

5. Spécifier fetch = FetchType.LAZY partout

```
@ManyToOne(fetch = FetchType.LAZY) // ☐ Toujours explicite
private Author author;
```

6. Éviter CascadeType.REMOVE sur @ManyToOne

```
@Entity
public class Book {
    @ManyToOne(cascade = CascadeType.REMOVE) // ☐ DANGEREUX !
    private Author author;
}

// Si on supprime un livre, l'auteur est aussi supprimé !
bookRepository.delete(book); // ⚠ Supprime aussi l'auteur
```

7. Utiliser orphanRemoval pour les compositions

```
@Entity
public class Order {
    @OneToMany(
        mappedBy = "order",
        cascade = CascadeType.ALL,
        orphanRemoval = true // ☐ Supprimer les items orphelins
    )
    private List<OrderItem> items = new ArrayList<>();
}
```

8. Éviter les relations bidirectionnelles inutiles

Si vous n'avez **jamais** besoin de naviguer de Course vers Student, restez unidirectionnel :

```
@Entity
public class Student {
    @ManyToMany
    private Set<Course> courses = new HashSet<>();
}

@Entity
public class Course {
    // Pas de référence à Student
}
```

9. Utiliser @Transactional(readOnly = true) pour les lectures

```
@Service
```

```
@Transactional(readOnly = true) // ☐ Optimisation
public class BookService {
    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }
}
```

10. Logger les requêtes SQL en développement

application.properties :


```
# Afficher les requêtes SQL
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# Logger les paramètres
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE


# Détecter le problème N+1
spring.jpa.properties.hibernate.session.events.log.LOG_QUERIES_SLOWER_THAN_MS=10
```

Mémo Final

Template de relation bidirectionnelle OneToMany/ManyToOne :



```
@Entity
public class Parent {
    @OneToMany(
        mappedBy = "parent",
        cascade = CascadeType.ALL,
        orphanRemoval = true,
        fetch = FetchType.LAZY
    )
    private List<Child> children = new ArrayList<>();
    public void addChild(Child child) {
        children.add(child);
        child.setParent(this);
    }
    public void removeChild(Child child) {
        children.remove(child);
        child.setParent(null);
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Parent)) return false;
        Parent parent = (Parent) o;
        return id != null && id.equals(parent.getId());
    }
    @Override
```



```
public int hashCode() {
    return getClass().hashCode();
}

@Entity
public class Child {
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "parent_id")
    private Parent parent;
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Child)) return false;
        Child child = (Child) o;
        return id != null && id.equals(child.getId());
    }
    @Override
    public int hashCode() {
        return getClass().hashCode();
    }
}
```

From:
<http://slamwiki2.kobject.net/> - **SlamWiki 2.1**

Permanent link:
<http://slamwiki2.kobject.net/framework-web/spring/relations>

Last update: **2025/10/08 01:24**

