

# Spring, JPA (seconde partie)

## GLG 203/Architectures Logicielles Java

Serge Rosmorduc  
serge.rosmorduc@lecnam.net  
Conservatoire National des Arts et Métiers

2019–2020

# Démonstrations

`https://gitlab.cnam.fr/gitlab/glg203\_204\_demos/09\_spring\_jpa2.git`

..

# Fetch glouton ou paresseux

## Eager/Lazy par défaut

(suite de ce qui a été dit à propos du n+1 select).

Le mode de récupération est par défaut :

`FetchType.EAGER` pour les champs `@Basic`, `@ManyToOne` et `@OneToOne` ;

`FetchType.LAZY` pour `@OneToMany`, `@ManyToMany` et  
`@ElementCollection`.

- important pour éventuelle utilisation « non managée » des objets ;
- important surtout pour l'efficacité du code (n+1 select !!!) ;
- contrôlable au niveau entités : attribut `fetch` (LAZY ou EAGER) des annotations comme `@ManyToMany` — probablement à éviter ;
- contrôlable au niveau des repositories :
  - ▶ annotation `@Query`, code JPQL avec `LEFT JOIN FETCH` (déjà vu) ;
  - ▶ annotation `@EntityGraph`

## Exemple d'utilisation de l'attribut fetch

```
@Entity
class Employe {
    ...
    @ManyToOne(fetch=FetchType.LAZY)
    Entreprise entreprise;
}
```

- Par défaut, la récupération d'un employé entraînerait celle de son entreprise (pour @ManyToOne, le défaut pour fetch est EAGER ;
- on estime qu'on ne consulte pas souvent ces données, et que ce chargement ralentit l'application ;
- on fixe donc le fetch à LAZY
-

# @EntityGraph

Décrit l'arbre des propriétés de l'objet qu'on veut récupérer directement dans une requête

- peut se définir au niveau de l'entité (`@NamedEntityGraph`);
- annote les méthodes du repository;
- peut s'utiliser aussi comme « hint » sur un objet Query en JPA « pur ».

# @EntityGraph

```
public interface FactureRepository
    extends JpaRepository<Facture, Long> {
    // Force le chargement des détails de la facture.
    @EntityGraph(attributePaths = { "lignesCommandes" },
        type = EntityGraphType.LOAD)
    List<Facture> findAll();
}
```

**attributePaths** chemins des attributs à récupérer ;

**type**        **FETCH** le graphe décrit *exactement* quels attributs  
seront récupérés ;

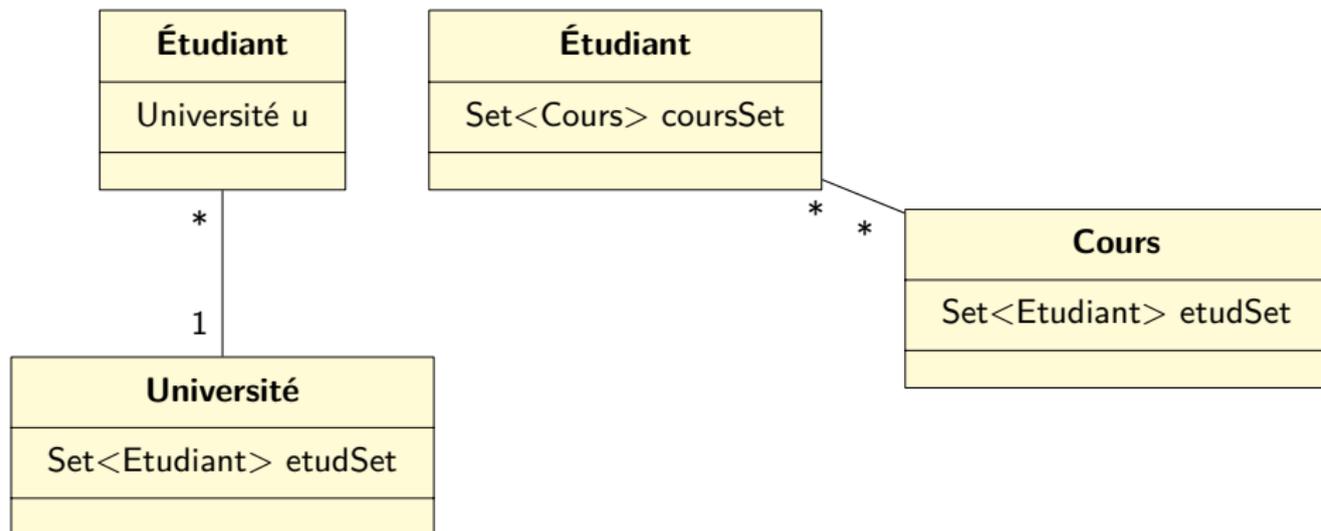
**LOAD** pour les attributs non spécifiés, le mode de  
chargement par défaut est utilisé ;

- Les types primitifs et les String simples sont *a priori* récupérés.
- on peut définir dans l'entité un ou plusieurs @NamedEntityGraphs qui peuvent être rappelés par leur nom.

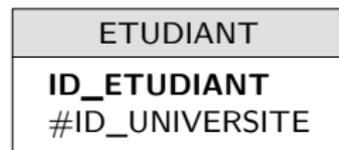
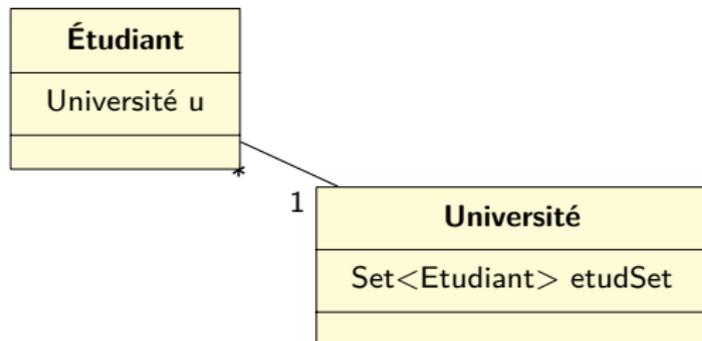
# Annotations avancées

## Liens bidirectionnels

- **Deux liens** à maintenir en java ;
- Toute modification d'un côté doit être répercutée de l'autre ;
- **couplage augmenté** ;
- 1..\* ou n..n ;
- alternative : requêtes sur la base.



# Liens Bidirectionnel Many-to-one



Comment annoter ce lien en java ?

# Annotations

```
@Entity
public class Student {
...
    @ManyToOne
    @JoinColumn(name="UNIVERSITY_ID")
    private University university;
```

```
@Entity
public class University {
...
    @OneToMany(mappedBy="university")
    private Set<Student> students;
```

- le côté « many » : annotation `@ManyToOne` (bd : clef étrangère);
- le côté « one » : annotation inverse `@OneToMany`; renvoie avec `mappedBy` à la *propriété* inverse dans l'autre classe.

Et pour les méthodes ?

# Un essai... incorrect

```
@Entity
public class Student implements Serializable {
    ...
    @ManyToOne
    @JoinColumn(name="UNIVERSITY_ID")
    private University university;
    ...
    public void setUniversity(University u) {
        this.university= u;
        this.university.addStudent(this);
    }
}
```

```
@Entity
public class University implements Serializable {
    ...
    @OneToMany(mappedBy="university")
    private Set<Student> students;

    public void addStudent(Student s) {
        students.add(s);
        s.setUniversity(this);
    }
}
```

# Un essai... incorrect

```
@Entity
public class Student implements Serializable {
    ...
    @ManyToOne
    @JoinColumn(name="UNIVERSITY_ID")
    private University university;
    ...
    public void setUniversity(University u) {
        this.university= u;
        this.university.addStudent(this);
    }
}
```

réursion infinie !!

```
@Entity
public class University implements Serializable {
    ...
    @OneToMany(mappedBy="university")
    private Set<Student> students;

    public void addStudent(Student s) {
        students.add(s);
        s.setUniversity(this);
    }
}
```

## Que doivent faire les méthodes ?

- Enlever, si nécessaire le lien précédent ;  
*un étudiant qui change d'université ne doit pas apparaître dans la liste des étudiants de son ancienne université*
- Mettre en place le lien, des deux côtés ;
- Éviter cependant la récursion infinie.

# Solution 1 : casser la symétrie

Un des côtés est en charge de toute la gestion (le « maître »).

```
@Entity
public class Student {
...
    @ManyToOne
    @JoinColumn(name="UNIVERSITY_ID")
    private University university;
...
    public void setUniversity(University u){
        if (this.university != null) {
            this.university.removeStudent(this);
        }
        this.university= u;
        this.university.addStudent(this);
    }
}
```

*Maître*

```
@Entity
public class University {
...
    @OneToMany(mappedBy="university")
    private Set<Student> students;
...
    void addStudent(Student s) {
        students.add(s);
    }
    void removeStudent(Student s) {
        students.remove(s);
    }
}
```

*Esclave*

pas public ! appelé seulement par le côté maître

## Solution 2 : tester

```
@Entity
public class Student {
    ...
    @ManyToOne
    @JoinColumn(name="UNIVERSITY_ID")
    private University university;
    ...
    public
    void setUniversity(University u){
        if (this.university != null) {
            if (this.university.equals(u))
                return;
            this.university.removeStudent(this);
        }
        this.university= u;
        if (u != null)
            this.university.addStudent(this);
    }
}
```

```
@Entity
public class University {
    ...
    @OneToMany(mappedBy="university")
    private Set<Student> students;

    public
    void addStudent(Student s) {
        if (students.contains(s))
            return;
        this.students.add(s);
        s.setUniversite(this);
    }

    public
    void removeStudent(Student s) {
        students.remove(s);
        s.setUniversity(null);
    }
}
```

# LIENS N..N BIDIRECTIONNELS

Un côté « maître » gère la relation...



```
@Entity
```

```
public class Student implements Serializable {
```

```
@ManyToMany
```

```
@JoinTable(name = "STUDENT_COURSE",
    joinColumns = @JoinColumn(name = "SC_STUDENT_ID"),
    inverseJoinColumns = @JoinColumn(name = "SC_COURSE_ID"))
```

```
private Collection<Course> courses;
```

```
    ...
```

```
}
```

```
@Entity
```

```
public class Course implements Serializable {
```

```
@ManyToMany(mappedBy = "courses")
```

```
private Collection<Student> students;
```

```
    ...
```

# LIENS N..N BIDIRECTIONNELS

Un côté « maître » gère la relation...



```
@Entity
public class Student implements Serializable {

    @ManyToMany
    @JoinTable(name = "STUDENT_COURSE",
        joinColumns = @JoinColumn(name = "SC_STUDENT_ID"),
        inverseJoinColumns = @JoinColumn(name = "SC_COURSE_ID"))
    private Collection<Course> courses;
    ...
}

@Entity
public class Course implements Serializable {

    @ManyToMany(mappedBy = "courses")
    private Collection<Student> students;
    ...
}
```

# Gestion des relations : rupture de la symétrie

```
@Entity
public class Student implements Serializable {
    @ManyToMany
    @JoinTable(name = "STUDENT_COURSE",
        joinColumns = @JoinColumn(name = "SC_STUDENT_ID"),
        inverseJoinColumns = @JoinColumn(name = "SC_COURSE_ID"))
    private Collection<Course> courses;

    public void addCourse(Course c) {
        courses.add(c);
        c.addStudent(this); // addStudent PAS public !
    }
    ...
}
```

```
@Entity
public class Course implements Serializable {
    @ManyToMany(mappedBy= "courses")
    private Collection<Student> students;
    ...

    void addStudent(Student s) { // PAS public !
        students.add(s);
    }
}
```

## Autre solution

```
@Entity
public class Student {
    @ManyToMany
    ....
    private Collection<Course> courses;

    public void addCourse(Course c) {
        if (! courses.contains(c)) {
            courses.add(c);
            c.addStudent(this);
        }
    }
}
```

le test évite la récursion infinie

```
@Entity
public class Course implements Serializable {
    @ManyToMany(mappedBy= "courses")
    private Collection<Student> students;
    ...
    void addStudent(Student s) {
        if (! students.contains(s)) {
            students.add(s);
            s.addCourse(this);
        }
    }
}
```

# Mapping de Listes

Deux possibilités :

- liste indexée : la position dans la liste est conservée ; annotation `@OrderColumn`

- `public class Forum {`

```
    @OneToMany(mappedBy = "forum")
```

```
    @OrderColumn
```

```
    private List<Message> messages = new ArrayList<>();
```

- liste ordonnée : la position dans la liste est donnée par une propriété ; annotation `@OrderBy`

```
public class Forum {
```

```
    @OneToMany(mappedBy = "forum")
```

```
    @OrderBy("date");
```

```
    private List<Message> messages = new ArrayList<>();
```

« date » est une propriété des messages.

## Mapping de Map

- la **valeur** des éléments de la map est une entité ;
- la **clef** dans la map est une propriété tirée de l'entité valeur
- on indique que c'est une Map avec l'annotation `@MapKey` ;
- par défaut, la clef de la Map est l'id de la valeur ;
- sur `@OneToMany` ou `@ManyToMany`.

@Entity

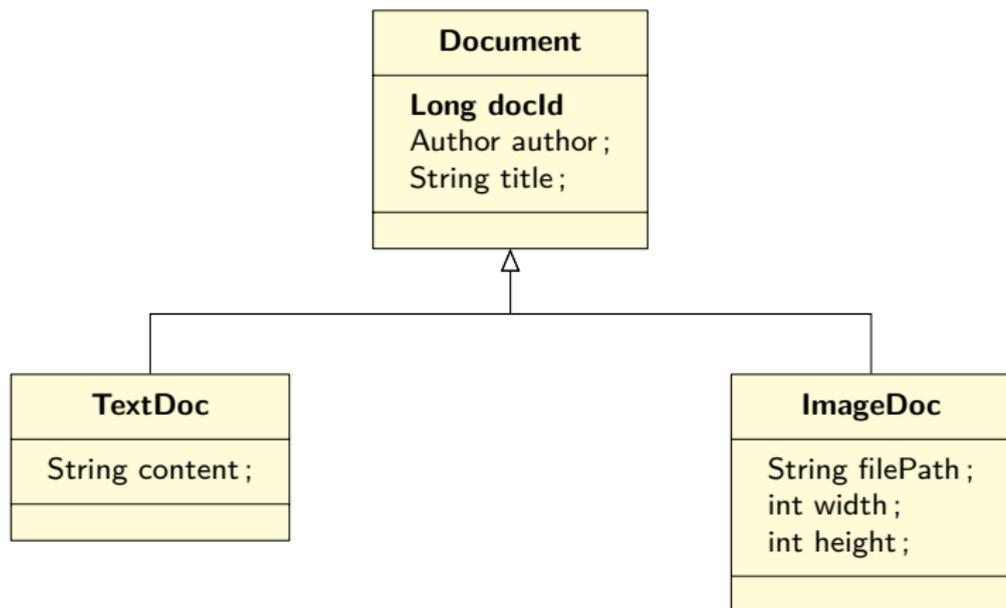
```
public class Livre {  
    @Id Long id;  
    @OneToMany(mappedBy="livre")  
    @MapKey("titreChapitre")  
    Map<String,Chapitre> chapitreMap;  
}
```

@Entity

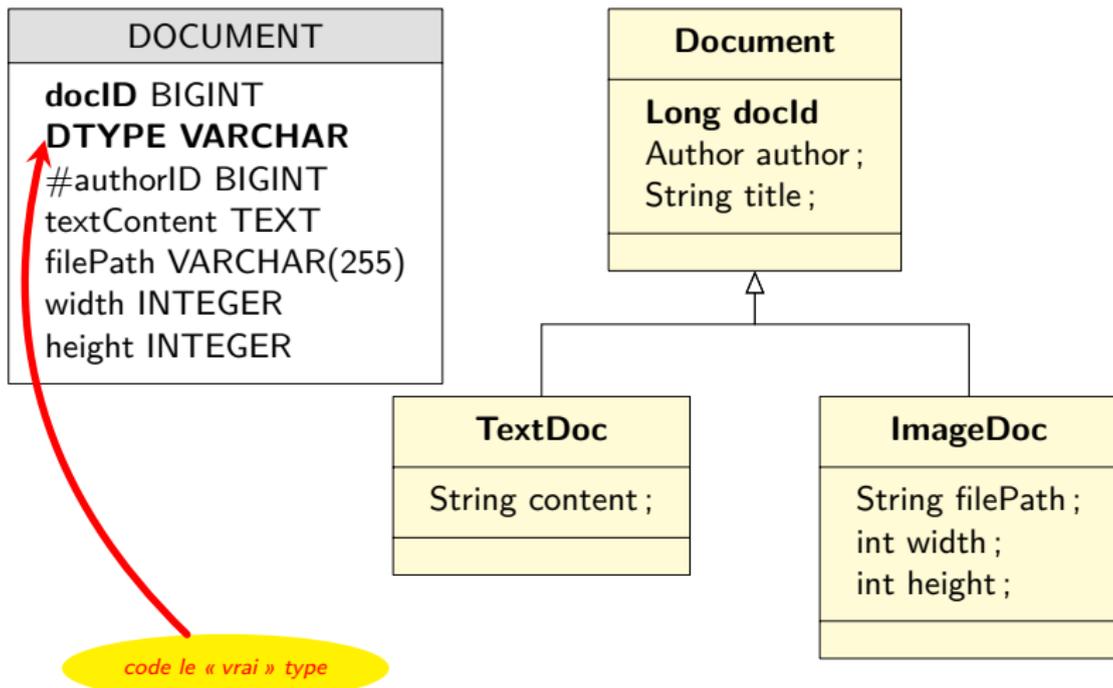
```
public class Chapitre {  
    @Id Long id;  
    String titreChapitre;  
    @ManyToOne Livre livre;  
}
```

## Mapping de l'héritage

- plusieurs représentations possibles de l'héritage dans une BD ;
- **Important** : l'espace des ID est le même pour toutes les classes filles ;
- ici, **docId** est hérité par les classes filles.



## Première solution : tout dans une seule table



- on met tout dans la même table
- un champ code la « vraie » sous-classe de l'objet ;
- positif : simple, pas de duplication ; négatif : espace gaspillé ;

## Codage en Java : Single\_TABLE

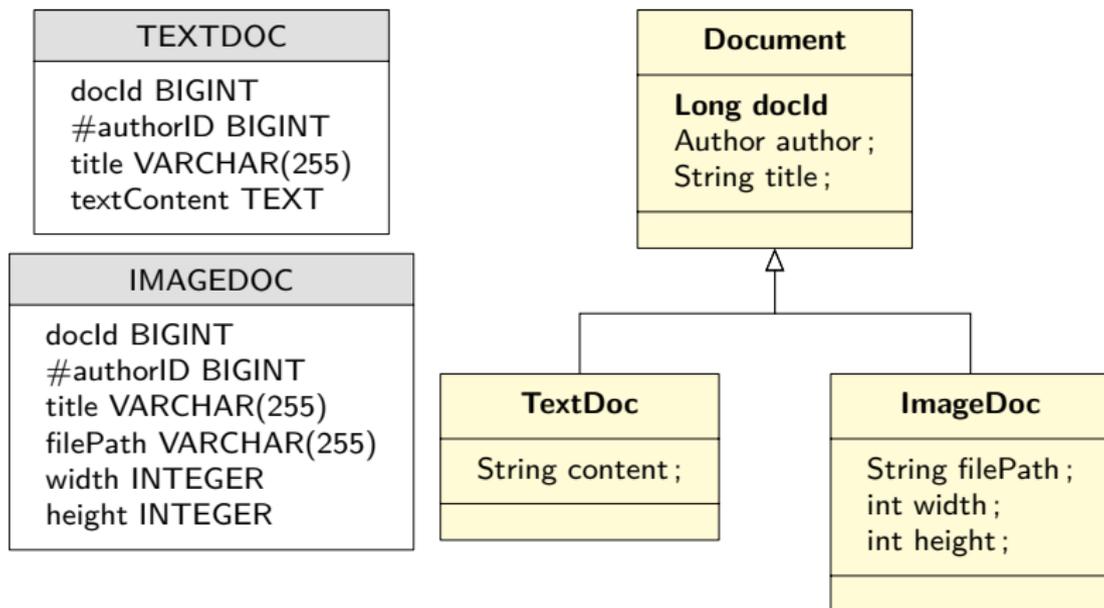
```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DTYPE")
abstract class Document {
    @Id Long docId;
}
```

la valeur de DTYPE n'est pas copiée dans un champ.



```
@Entity
@DiscriminatorValue("txt")
class TextDoc extends Document {
}
```

# Une table par classe



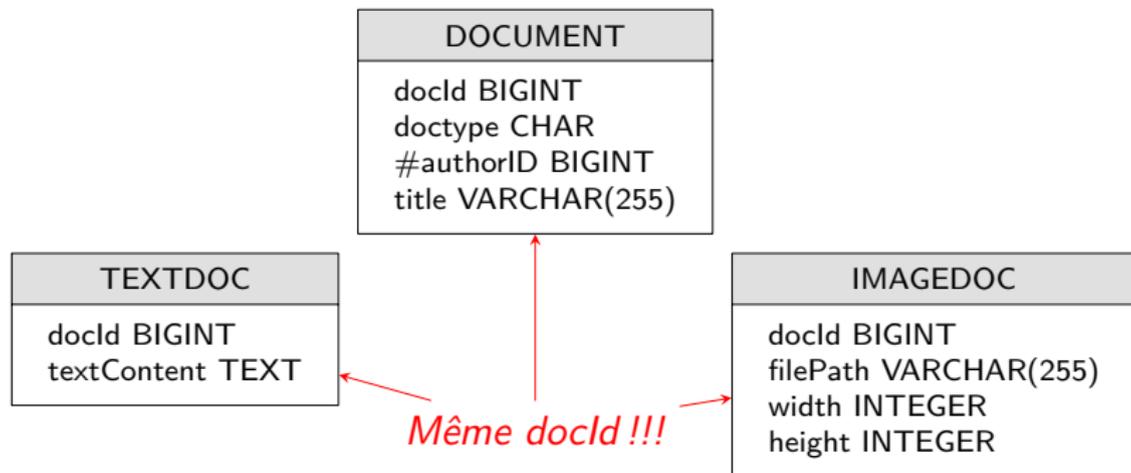
- + simple ; - information dupliquée ;
- bien si la classe parente est très simple, avec très peu d'information commune ;
- **docId** unique : utiliser même séquence.

# Une table par classe

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
abstract class Document {
@Id Long docId;
}
```

```
@Entity
@DiscriminatorValue("txt")
class TextDoc extends Document {
}
```

# Tables jointes



- solution la plus logique ;
- + pas de duplication ni d'espace gaspillé ;
- - plus difficile à utiliser, surtout si on doit faire du SQL derrière ;
- optimal quand classes parentes et filles sont complexes.

# Tables jointes

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="DTYPE")
abstract class Document {
@Id Long docId;
}
```

```
@Entity
@DiscriminatorValue("txt")
class TextDoc extends Document {
}
```

## Données « non entités »

Toutes les tables ne contiennent pas de « vraies » entités ;

Exemples :

- Exemple : lignes d'une facture (discutable) ;
- « labels » d'un message dans un forum ;
- numéros de téléphone d'un contact.

C'est en partie une question de point de vue...

### Critère pratique

Quand un objet représente une « valeur » pure (identité par contenu, immuable), c'est un bon candidat pour l'enchâssement.

## @Embeddable

- Marque qu'une classe n'a pas d'identité propre dans la base ;
- les données de cette classe seront copiées telles quelle dans les tables des entités qui l'utilisent ...
- on a probablement intérêt à redéfinir `equals` et `hashCode`...

# @Embeddable et @Embedded

Dans l'entité...

```
@Entity
public class Client {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;
    private String nom;
    @Embedded
    private Adresse adresse;
```

Dans la classe à enchâsser...

```
@Embeddable
public class Adresse {

    private String rue;
    private String ville;
    private String codePostal;

    ...
}
```

## Dans la base de données...

CLIENT
<b>ID</b>
NOM
RUE
VILLE
CODE_POSTAL

## Avec des collections

Quand on les éléments/non entités sont dans une collection, on utilise `@ElementCollection` ;

On peut placer dans une collection annotée par `@ElementCollection` :

- des objets dont la classe est `@Embeddable` ;
- des objets de types « simples » : les types primitifs, mais aussi ceux qui comme `String` ou `Date` peuvent être stockés directement dans un champ de base de données.

Les `ElementCollection` sont récupérés de manière paresseuse par défaut !  
Envisagez éventuellement un `EntityGraph`.

# ElementCollection et Set

Fonctionne assez simplement :

```
@Entity
public class Message {
    @Id
    private Long id;

    @ElementCollection
    Set<String> labels;
}
```

## ElementCollection et Liste

Pour forcer la représentation de l'indice de l'élément en BD, il faut utiliser `@OrderColumn` :

```
@Entity
public class TableDesMatieres {
    @Id
    private Long id;
    private String titreOuvrage;

    @ElementCollection
    @OrderColumn
    private List<String> entrees;
```

# Représentation en base de données

TABLE_DES_MATIERES
<b>ID</b>
TITRE_OUVRAGE

TABLE_DES_MATIERES_ENTREES
<b>#TABLE_DES_MATIERES_ID</b>
<b>ENTREES_ORDER</b>
ENTREES

- @OrderColumn ajoute la colonne ENTREES\_ORDER ;
- la clef primaire de cette table est (**#TABLE\_DES\_MATIERES\_ID, ENTREES\_ORDER**).

## @ElementCollection et Map (valeur non entité)

Version minimaliste, quand la **clef** est une entité :

```
@Entity
```

```
public class Facture {
```

```
    @Id
```

```
    private Long id;
```

```
    private String nom;
```

```
    @ElementCollection
```

```
    private Map<Produit, Integer> lignesFacture = new HashMap<>();
```

Dans les tables :

FACTURE
ID
NOM

FACTURE_LIGNES_FACTURE
#FACTURE_ID
#LIGNES_FACTURE_KEY
LIGNES_FACTURE

**#LIGNE\_FACTURE\_KEY** clef du *Produit* (= clef de la map)

**LIGNES\_FACTURE** valeur stockée ;

# Map, version annotée

```
public class Facture {  
    @Id  
    private Long id;  
    private String nom;  
  
    @ElementCollection  
    @CollectionTable(name = "FACTURE_LIGNE",  
                    joinColumns = @JoinColumn(name = "FACTURE_ID"))  
    @MapKeyJoinColumn(name = "PRODUIT_ID")  
    @Column(name = "QUANTITE")  
    private Map<Produit, Integer> lignesFacture = new HashMap<>();  
}
```

FACTURE_LIGNE
#FACTURE_ID #PRODUIT_ID QUANTITE

FACTURE
ID NOM

## @ElementCollection et Map : clefs et valeurs

Pour les Map annotées par @ElementCollection, on a les possibilités suivantes :

- la clef et la valeur sont des types simples (primitifs, String, Date...) ou @Embeddable
- la clef est une entité, la valeur est un type simple ou @Embeddable.

# Cascade

- contrôle si les opérations comme persist, merge, etc... seront appliquées récursivement aux **entité liées** ;
- concerne des liens **parents-enfants** ;
- ne concerne que les **entités**, pas les @Embedded ou les @ElementCollection
- se place sur les annotations @OneToMany, etc.
- valeurs possibles : CascadeType.ALL, CascadeType.DETACH, CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH, CascadeType.REMOVE
- si on veut utiliser une cascade :
  - ▶ vérifier qu'elle est justifiée ;
  - ▶ se demander si l'entité liée ne serait pas mieux représentée par une @ElementCollection ou un @Embedded.

## Cascade : exemple

```
@Entity
public class Forum implements Serializable {
    .....
    @OneToMany(mappedBy="forum",
                cascade= CascadeType.ALL)
    private List<Message> messages= new HashSet<Message>();
}
```

- si on détruit le forum, ses messages seront aussi détruits ;
- si on persiste un nouveau forum, contenant des messages, ceux-ci seront aussi persistés ;
- de même pour les opérations `merge()`, `refresh()` et `detach()` ;
- les vraies cascades sont rares (**composition vs. agrégation**) ;
- pour le forum : voir démo.

- JPQL est pratique, mais ça n'est pas du Java :
  - ▶ pas de vérification à la compilation de la structure des requêtes ;
  - ▶ pas de réutilisation possible d'un bout de requête ;
  - ▶ pour chercher avec un nombre *variable* de critère,
    - ★ en JPQL, il faut *construire* la chaîne de caractère qui correspond à la requête ;
    - ★ risque d'erreurs, voire d'injections ;
    - ★ une API orientée objet serait plus intéressante.

## Criteria - utilisation

*Exemples dans project 02\_criteria, dans CriteriaApplicationTests*

```
CriteriaBuilder builder = em.getCriteriaBuilder();

CriteriaQuery<Student> cq = builder.createQuery(Student.class);

Root<Student> root = cq.from(Student.class);

cq.where(builder.equal(root.get("name"), "Turing"));

TypedQuery typedQuery = em.createQuery(cq);
List<Student> l = typedQuery.getResultList();
```

## Criteria - utilisation

Création d'un builder



*Exemples dans project 02\_criteria, dans CriteriaApplicationTests*

```
CriteriaBuilder builder = em.getCriteriaBuilder();
```

```
CriteriaQuery<Student> cq = builder.createQuery(Student.class);
```

```
Root<Student> root = cq.from(Student.class);
```

```
cq.where(builder.equal(root.get("name"), "Turing"));
```

```
TypedQuery typedQuery = em.createQuery(cq);
```

```
List<Student> l = typedQuery.getResultList();
```

## Criteria - utilisation

retournera un Student

*Exemples dans project 02\_criteria, dans CriteriaApplicationTests*

```
CriteriaBuilder builder = em.getCriteriaBuilder();
```

```
CriteriaQuery<Student> cq = builder.createQuery(Student.class);
```

```
Root<Student> root = cq.from(Student.class);
```

```
cq.where(builder.equal(root.get("name"), "Turing"));
```

```
TypedQuery typedQuery = em.createQuery(cq);
```

```
List<Student> l = typedQuery.getResultList();
```

## Criteria - utilisation

### Cherche dans la classe Student

*Exemples dans project 02\_criteria, dans CriteriaApplicationTests*

```
CriteriaBuilder builder = em.getCriteriaBuilder();
```

```
CriteriaQuery<Student> cq = builder.createQuery(Student.class);
```

```
Root<Student> root = cq.from(Student.class);
```

```
cq.where(builder.equal(root.get("name"), "Turing"));
```

```
TypedQuery typedQuery = em.createQuery(cq);
```

```
List<Student> l = typedQuery.getResultList();
```

## Criteria - utilisation

*Exemples dans project 02\_criteria, dans CriteriaApplicationTests*

```
CriteriaBuilder builder = em.getCriteriaBuilder();
```

```
CriteriaQuery<Student> cq = builder.createQuery(Student.class);
```

```
Root<Student> root = cq.from(Student.class);
```

```
cq.where(builder.equal(root.get("name"), "Turing"));
```

```
TypedQuery typedQuery = em.createQuery(cq);
```

```
List<Student> l = typedQuery.getResultList();
```

**Equivaut à `s.name= "Turing"`**

## Criteria - utilisation

*Exemples dans project 02\_criteria, dans CriteriaApplicationTests*

```
CriteriaBuilder builder = em.getCriteriaBuilder();

CriteriaQuery<Student> cq = builder.createQuery(Student.class);

Root<Student> root = cq.from(Student.class);

cq.where(builder.equal(root.get("name"), "Turing"));

TypedQuery typedQuery = em.createQuery(cq);
List<Student> l = typedQuery.getResultList();
```

On crée la requête exécutable

## Criteria - utilisation

*Exemples dans project 02\_criteria, dans CriteriaApplicationTests*

```
CriteriaBuilder builder = em.getCriteriaBuilder();

CriteriaQuery<Student> cq = builder.createQuery(Student.class);

Root<Student> root = cq.from(Student.class);

cq.where(builder.equal(root.get("name"), "Turing"));

TypedQuery typedQuery = em.createQuery(cq);
List<Student> l = typedQuery.getResultList();
```

On exécute la requête

# CriteriaBuilder

- crée la requête ;
- crée les paramètres (équivalents de `:nom` en jpql)
- crée les comparaisons dans le where et leurs combinaisons ;

# CriteriaQuery

la structure de la requête qu'on construit ;

- crée les différentes parties de la requête : `select`, `from`, `where` ;
- passée à `TypedQuery` pour exécution ;
- équivalent du code JPQL : structure passive ;

# TypedQuery

La requête *exécutable*.

- permet de fixer les valeurs des paramètres ;
- peut être exécutée ;

# Root

Représente les classes « sources » de la requête (après le `from` en JPQL).

- typés : `Root<Student>` ;
- on peut en définir plusieurs (produits cartésiens) ;
- permet de faire des jointures ;
- source des valeurs utilisées lors des comparaisons du `where`.

## Exemple : une requête simple

```
// JPQL : "select e from Etudiant e";
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Etudiant> query =
    builder.createQuery(Etudiant.class);
query.select(query.from(Etudiant.class));
TypedQuery<Etudiant> typedQuery = em.createQuery(query);
List<Etudiant> l2 = typedQuery.getResultList();
```

## Requête avec clause dans le where

```
// "select c from Cours c where c.titre = :titre";
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Cours> query = builder.createQuery(Cours.class);
Root<Cours> root = query.from(Cours.class);
query.select(root);
ParameterExpression<String> titre =
    builder.parameter(String.class);
query.where(builder.equal(root.get("titre"), titre));
TypedQuery<Cours> typedQuery = em.createQuery(query);
typedQuery.setParameter(titre, "nfa031");
List<Cours> l2 = typedQuery.getResultList();
```

## Requête avec produit cartésien

```
//select distinct c1.formation.universite from Cours c1, Cours c2
// where c1.titre = 'nfa016' and c2.titre = 'nfa017'
// and c1.formation.universite = c2.formation.universite
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Universite> query = cb.createQuery(Universite.class)
                                .distinct(true);

Root<Cours> cours1 = query.from(Cours.class);
Root<Cours> cours2 = query.from(Cours.class);
query.where(cb.and(
    cb.equal(cours1.get("titre"), "nfa016"),
    cb.equal(cours2.get("titre"), "nfa017"),
    cb.equal(cours1.get("formation").get("universite"),
            cours2.get("formation").get("universite"))));
query.select(cours1.get("formation").get("universite"));
TypedQuery<Universite> typedQuery = em.createQuery(query);
List<Universite> result1 = typedQuery.getResultList();
```

## Requête avec jointure

```
// select c from Cours c join c.themes t1 join c.themes t2
// where t1.label = :l1 and t2.label = :l2"
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Cours> query = cb.createQuery(Cours.class);
Root<Cours> root = query.from(Cours.class);
Join<Object, Object> theme1 = root.join("themes");
Join<Object, Object> theme2 = root.join("themes");
ParameterExpression<String> l1 = cb.parameter(String.class);
ParameterExpression<String> l2 = cb.parameter(String.class);
query.where(
    cb.and(
        cb.equal(theme1.get("label"), l1),
        cb.equal(theme2.get("label"), l2)
    ));
TypedQuery<Cours> typedQuery = em.createQuery(query);
typedQuery.setParameter(l1, "algorithmique");
typedQuery.setParameter(l2, "programmation");
List<Cours> cours01 = typedQuery.getResultList();
```

## Recherche multicritère

Exemple : on a un ensemble de thèmes, et on cherche les messages qui ont *tous* ces thèmes.

En jpql : il faut construire le code JPQL par programme ;

En criteria : aussi, mais la structure nous aide.

```
List<Theme> themes = Arrays.asList(new Theme("algorithmique"),
                                   new Theme("programmation"));
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Cours> query = cb.createQuery(Cours.class);
Root<Cours> root = query.from(Cours.class);
List<Predicate> conditions = new ArrayList<>();
for (Theme theme : themes) {
    Predicate themeIn = cb.isMember(theme, root.get("themes"));
    conditions.add(themeIn);
}
query.where(
    conditions.toArray(new Predicate[conditions.size()]));
TypedQuery<Cours> typedQuery = em.createQuery(query);
List<Cours> cours = typedQuery.getResultList();
```

## Criteria « fortement typé »

- Le code `cb.equal(cours1.get("titre"), "nfa016")` n'est pas « type-safe » ; échec possible à l'exécution.
- on peut décrire la structure de chaque entité par une classe auxiliaire ; classe entité `Personne` → classe auxiliaire `Personne_` ;
- génération automatique de ces classes possible.

```
@StaticMetamodel(Formation.class)
public abstract class Formation_ {

    public static volatile
        SingularAttribute<Formation, Universite> universite;
    public static volatile
        SingularAttribute<Formation, String> titre;
    public static volatile
        SingularAttribute<Formation, Long> id;

    public static final String UNIVERSITE = "universite";
    public static final String TITRE = "titre";
    public static final String ID = "id";
}
```

# Utilisation

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Universite> query= cb.createQuery(Universite.class)
                                .distinct(true);
Root<Cours> cours1 = query.from(Cours.class);
Root<Cours> cours2 = query.from(Cours.class);
query.where(cb.and(
    cb.equal(cours1.get(Cours_.titre), "nfa016"),
    cb.equal(cours2.get(Cours_.titre), "nfa017"),
    cb.equal(
        cours1.get(Cours_.formation).get(Formation_.universite),
        cours2.get(Cours_.formation).get(Formation_.universite)
    )));
query.select(cours1.get(Cours_.formation)
            .get(Formation_.universite));
TypedQuery<Universite> typedQuery = em.createQuery(query);
List<Universite> result1 = typedQuery.getResultList();
```

Idée : les String sont juste des noms. Les attributs portent en plus leur **type**.

## Génération des classes auxiliaires

Dans le build.gradle :

```
// ...
tasks.withType(JavaCompile) {
    options.annotationProcessorGeneratedSourcesDirectory =
        file("src/generated/java")
}
sourceSets {
    generated {
        java {
            srcDirs = ['src/generated/java']
        }
    }
}
...
dependencies {
    // Pour l'API Criteria : générateur (optionnel) de métamodèle.
    annotationProcessor(
        'org.hibernate:hibernate-jpamodelgen:5.4.9.Final')
    ...
}
```

## Utilisation dans un Repository

Si un repository implémente `JpaSpecificationExecutor`, il peut utiliser des requêtes `Criteria`.

```
public interface JpaSpecificationExecutor<T> {  
    Optional<T> findOne(@Nullable Specification<T> spec);  
  
    List<T> findAll(@Nullable Specification<T> spec);  
  
    Page<T> findAll(@Nullable Specification<T> spec, Pageable pageable);  
  
    List<T> findAll(@Nullable Specification<T> spec, Sort sort);  
  
    long count(@Nullable Specification<T> spec);  
}
```

# Specification pour le JpaSpecificationExecutor

Interface fonctionnelle dotée d'une méthode :

```
Predicate toPredicate(Root<T> root,  
                    CriteriaQuery<?> query,  
                    CriteriaBuilder criteriaBuilder);
```

- peut être définie par une lambda ;
- une bibliothèque de critères est dans ce cas généralement fournie par des méthodes statiques ;
- JpaSpecificationExecutor comporte des méthodes auxiliaires prédéfinies : not, where, and, or qui permettent de combiner plusieurs spécifications.

## Exemple

```
public class CoursCriteria {  
  
    public static Specification<Cours>  
        hasTheme(String themeLabel) {  
        Theme theme = new Theme(themeLabel);  
        return (cours, cquery, cbuilder)  
            -> cbuilder.isMember(theme, cours.get("themes"));  
    }  
}
```

# Utilisation

```
@Test
public void testQueryOnCriteria() {
    List<Cours> cours =
        repository.findAll(CoursCriteria.hasTheme("algorithmique"));
    ...
}
```

```
@Test
public void testCombinaisonCriteria() {
    Specification<Cours> criteria =
        CoursCriteria.hasTheme("programmation").and(
            CoursCriteria.hasTheme("analyse"));
    List<Cours> cours = repository.findAll(criteria);
    ...
}
```

# Cache

# Les caches pour hibernate

- Deux niveaux de cache ;
- premier niveau, cache automatique, pour la session de travail (au sens JPA, pas Web) ;
- second niveau : cache global optionnel, fourni par une implémentation extérieure (typiquement ehcache).
- le cache stocke les données provenant et à destination de la base ;
- minimisation des communications ;
- mise à jour par paquets et non donnée par donnée ;
- problème éventuel en cas de plantage.

# Cache en Spring

Importer :

```
implementation 'org.springframework.boot:spring-boot-starter-cache'
```

Annotation d'une configuration avec `@EnableCaching`

Ensuite, annotation sur les *méthodes* :

`@Cacheable("nomDuCache")` :

- à l'appel de la méthode, on cherche dans le cache ;
- si on ne trouve pas, on calcule et on met dans le cache.
- On peut préciser des conditions, avec les attributs `condition` (données d'entrée), `unless` (valeur de retour) et le SpEl.

`@CachePut` comme `@Cacheable`, mais la méthode est *toujours* exécutée ;

`@CacheEvict` : l'appel d'une méthode annotée avec `@CacheEvict` permet de vider tout ou partie d'un cache dont on donne le nom ;

## Utilisation du cache

- attention, le cache comporte a priori des objets détachés ;
- On peut être amené à le mettre à jour avec `@CacheEvict` quand on fait des modifications ;
- globalement, cache à utiliser pour des données essentiellement en *lecture*.

```
@Cacheable("facture")
```

```
public List<Facture> factures() {  
    return factureRepository.findAll();  
}
```

```
// L'appel de cette méthode invalide le cache.
```

```
@CacheEvict(allEntries = true, value = "facture")
```

```
public void retirerUnProduitDeChaqueFacture() {  
    for (Facture f : factureRepository.findAll()) {  
        Set<Produit> produitsCommandes = f.getLignesCommandes().keySet()  
        for (Produit p : produitsCommandes) {  
            f.retirer(p, 1);  
        }  
    }  
}
```

## Clef dans le cache

- Le cache utilise une clef calculée à partir des paramètres de la méthode cachée — en particulier, de leur hashCode ;
- ça peut poser des problèmes selon l'implémentation choisie ;
- on peut préciser la clef explicitement.

```
@Cacheable("facture", key = "#f.id")  
public List<Produit> produitsDansFacture(Facture f) {  
    ...  
}
```

# Configuration de JPA dans Spring sans Spring Boot

Définir des beans :

- `EntityManagerFactory` pour créer les `EntityManager` ;
- `JpaTransactionManager` pour gérer les transactions ;
- `DataSource` pour se connecter à la base de données.

# Configuration Spring seul

```
@ComponentScan(basePackages = {"glg203.jpa"}) )
@EnableJpaRepositories(basePackages = {"glg203.jpa.repositories"})
@Configuration
public class JPAConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .build();
    }
}
```

# Configuration Spring seul

```
@Bean
public Properties properties() {
    Properties properties = new Properties();
    properties.put("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
    properties.put("hibernate.hbm2ddl.auto", "create");
    properties.put("hibernate.show_sql", true);
    return properties;
}
```

```
@Bean
public EntityManagerFactory entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean emff =
        new LocalContainerEntityManagerFactoryBean();
    emff.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
    emff.setJpaProperties(properties());
    emff.setJpaVendorAdapter(jpaVendorAdapter());
    emff.setDataSource(dataSource());
    emff.setPackagesToScan("glg203.jpa.model");
    emff.afterPropertiesSet();
    return emff.getNativeEntityManagerFactory();
}
```

# Configuration Spring seul

```
@Bean
public JpaVendorAdapter jpaVendorAdapter() {
    return new HibernateJpaVendorAdapter();
}
```

```
@Bean
public PlatformTransactionManager transactionManager() {
    return new JpaTransactionManager(entityManagerFactory());
}
}
```

## Autres possibilités

- pagination ;
- extension d'un @Repository par une classe (démon) ;
- événements : Entities managed by repositories are aggregate roots. In a Domain-Driven Design application, these aggregate roots usually publish domain events. Spring Data provides an annotation @DomainEvents
- p. 34 : repository populators / JSON ; <https://www.baeldung.com/spring-data-jpa-repository-populators> ;
- projection : initialisation automatique de DTO ;
- procédures stockées ;

# Dependency Injection, concurrence et EntityManager

- si on injecte un EntityManager, il vaut mieux utiliser `@PersistenceContext` que `@Autowired`;
- `@EntityManager` non thread-safe;
- d'où risque avec `@Autowired`;
- l'objet injecté par `@PersistenceContext` est, quant à lui, *thread-safe*.
- (à vérifier).
- problème assez rare dans Spring : on utilise les Repositories sans se poser de question.

# Bibliographie

- *The Java EE Tutorial Release 7*,  
<https://docs.oracle.com/javaee/7/index.html>
- sur l'optimisation : <https://blog.ippon.fr/2017/07/19/boostez-performances-de-application-spring-data-jpa/>