AUTOCOMPLETION INTELLIJ CTRL+SPACE

Decoupage du projet library en modules maven
Création d'un nouveau projet maven.

Création d'un sous projet maven en faisant new project sur la racine dans l'IDE.
L'ajout des sous modules met à jour le pom parent.
Dans le main java du parent, je vais tenter de mettre application et swagger config
Sur le repertoire java, utiliser clic droit, mark directory... as Sources Root

Clic droit sur le pom, maven, reinport => met à jour les dépendances

3. Opérations sur les entités

3.1. Introduction

Toutes les opérations que l'on peut faire sur des entités JPA passent directement ou indirectement par l' *entity manager* . Cet objet est central dans cette API. Ainsi, toute entité JPA persistante possède une référence sur l' *entity manager* qui l'a créée ou qui a permis de la retrouver, et un *entity manager* connaît toutes les entités JPA qu'il gère.

Ces opérations peuvent être invoquées directement par appel à la méthode correspondante sur l' *entity manager* . Elles peuvent aussi être invoquées automatiquement sur les entités en relation d'une entité père. Par exemple, il est possible de propager une opération `PERSIST` d'une entité père vers les entités qu'elle a en relation. Nous verrons ce mécanisme de propagation en détails dans le chapitre suivant.

Il n'est pas possible de mettre en relation une entité gérée par un *entity manager* avec une autre entité appartenant à un autre *entity manager* . En revanche, on peut lier une entité à un autre *entity manager* que celui avec lequel elle a été créée ou lue par une opération `MERGE`.

Toutes les opérations de création, effacement ou modification doivent nécessairement se faire dans une transaction. Une transaction démarre sur appel à la méthode `begin()` de la transaction dans laquelle on se trouve, et est validée sur appel à sa méthode `commit()`. On peut annuler une transaction par appel à sa méthode `rollback()`.

Les spécifications JPA définissent cinq opérations sur une entité : `PERSIST`, `REMOVE`, `REFRESH`, `DETACH` et `MERGE`. Ces opérations correspondent à autant de méthodes sur l'objet `EntityManager`.

3.2. Opération PERSIST

Cette opération a pour effet de rendre une entité persistante. Si cette entité est déjà persistante, alors cette opération n'a aucun effet. Rendre une entité persistante consiste à l'écrire en base sur le prochain *commit* de la transaction dans laquelle on se trouve.

3.3. Opération REMOVE

L'opération `REMOVE` a pour effet de rendre une entité non persistante. Si cette entité est déjà non persistante, alors cette opération n'a aucun effet. Une entité rendue non persistante sera effacée de la base sur le prochain *commit* de la transaction dans laquelle on se trouve.

3.4. Opération REFRESH

L'opération `REFRESH` ne s'applique qu'aux entités persistantes. Si l'entité passée en paramètre n'est pas persistante, alors une exception de type `IllegalArgumentException` est générée.

L'opération `REFRESH` a pour effet de synchroniser l'état d'une entité avec son état en base. Si les champs d'une entité ont été modifiés dans la transaction courante, ces modifications seront donc effacées par cette opération. Si l'entité a été modifiée en base, alors ces modifications seront prises en compte lors d'un `REFRESH`.

3.5. Opération DETACH

Une opération `DETACH` sur une entité persistante a pour effet de la détacher de l' *entity manager* qui la gère. Cette entité persistante ne sera donc pas prise en compte lors du prochain *commit* de la transaction dans laquelle on se trouve.

3.6. Opération MERGE

Une opération `MERGE` sur une entité persistante attache cette entité à l' *entity manager* courant. On utilise cette opération pour associer une entité à un autre *entity manager* que celui qui a été utilisé pour la créer ou la lire.

Un opération `MERGE` sur une entité qui a été effacée (opération `REMOVE`) génère une exception de type `IllegalArgumentException`.

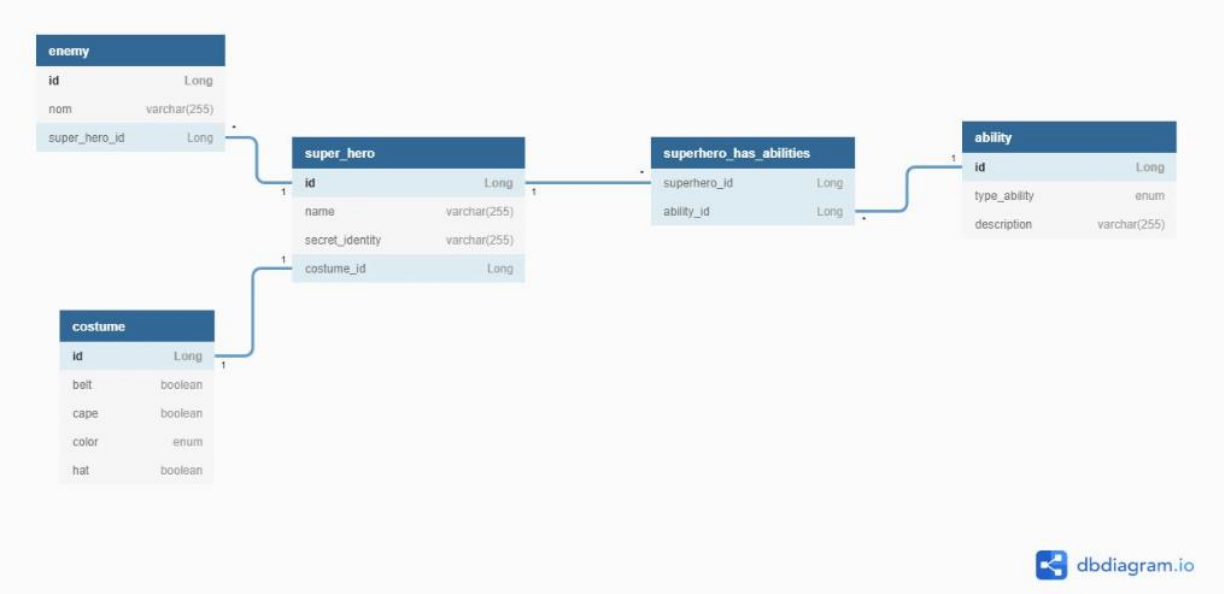
Depuis GitHub, avant de faire un git fork avant de cloner pour pouvoir ensuite faire des push sur MON github et non sur celui du prof.

Ne pas (forcément) mettre de column sur les ID
Ne pas mettre de column sur les attributs qui proviennent d'une autre classe et qui servent à opérer des jointures, ils n'appartiennent pas à la table. (Ils y appartiendrait en cas d'@Embedded / @Embeddable en cas de relation @OneToOne.

Pour les liens entre classes, penser objet. Ne pas mettre en attribut les identifiants (cf clefs étrangères) des autres classes mais la totalité de la classe.

Il faut des annotations dans les deux sens dès lors que l'on référence les classes jointes dans les deux classes sinon, pb hibernate. Spring Boot ne se lance pas. En référençant les informations dans les 2 sens , on permet la naviagtion depuis les deux classes.
Dans library Books du TP, on avait mis un set de books dans library d'où le @OneToMany mais on n'avait pas référencé Library dans Book, donc pas besoin de mettre un @ManyToOne de ce côté.

Utilisation de Set pour gérer les liens many
Ecrire les constructeurs vides requis par SPRING en private.
Générer les constructeurs sans donner accès à la manipulation des identifiants en public.
On n'écrit pas les Getter car on utilise l'annotation @Getter de la library Lombok.




```java
import javax.persistence.*;
import java.util.Set;

@JsonIdentityInfo(generator = ObjectIdGenerators.UUIDGenerator.class,property = "@UUID")
@Getter
@Entity
public class Ability {
    @Id
    @GeneratedValue
    private Long id;
    @Enumerated(EnumType.STRING)
    @Column
    private TypeAbility typeAbility;
    @Column
    private String description;

    @ManyToMany(mappedBy = "abilities")
    private Set<SuperHero> superHeroes;

    private Ability() {}

    public Ability(TypeAbility typeAbility, String description, Set<SuperHero> superHeroes) {
        this.typeAbility = typeAbility;
        this.description = description;
        this.superHeroes = superHeroes;
    }
}
```

```java
package com.bnpparibas.itg.superhero.superhero.mapping;

public enum Color {
    BLUE,
    GREEN,
    ORANGE,
    RED;
}
```

```java
package com.bnpparibas.itg.superhero.superhero.mapping;

import lombok.Getter;

import javax.persistence.*;

@Getter
@Entity
public class Costume {
    @Id
    @GeneratedValue
    private Long id;
    @Column
    private boolean belt;
    @Column
    private boolean cape;
    @Enumerated(EnumType.STRING)
    @Column
    private Color color;
    @Column
    private boolean hat;

    private Costume() {};

    public Costume(boolean belt, boolean cape, Color color, boolean hat) {
        this.belt = belt;
        this.cape = cape;
        this.color = color;
        this.hat = hat;
    }
}
```

```java
package com.bnpparibas.itg.superhero.superhero.mapping;

import com.fasterxml.jackson.annotation.JsonIdentityInfo;
import com.fasterxml.jackson.annotation.ObjectIdGenerator;
import com.fasterxml.jackson.annotation.ObjectIdGenerators;
import lombok.Getter;

import javax.persistence.*;

@JsonIdentityInfo(generator = ObjectIdGenerators.UUIDGenerator.class,property = "@UUID")
@Getter
@Entity
public class Enemy {
    @Id
    @GeneratedValue
    private Long id;
    @Column
    private String name;
    @ManyToOne(cascade = CascadeType.ALL)
    private SuperHero superHero;

    private Enemy() {}

    public Enemy(String name, SuperHero superHero) {
        this.name = name;
        this.superHero = superHero;
    }
}
```

```java
package com.bnpparibas.itg.superhero.superhero.mapping;

import lombok.Getter;

import javax.persistence.*;
import java.util.Set;

@Getter
@Entity
public class SuperHero {

    @Id
    @GeneratedValue
    private Long id;
    @Column
    private String name;
    @Column
    private String secret_identity;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "costume_id")
    private Costume costume;

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(
            name = "superhero_has_abilities",
            joinColumns = @JoinColumn(name = "superhero_id"),
            inverseJoinColumns = @JoinColumn(name = "ability")
    )
    private Set<Ability> abilities;
```

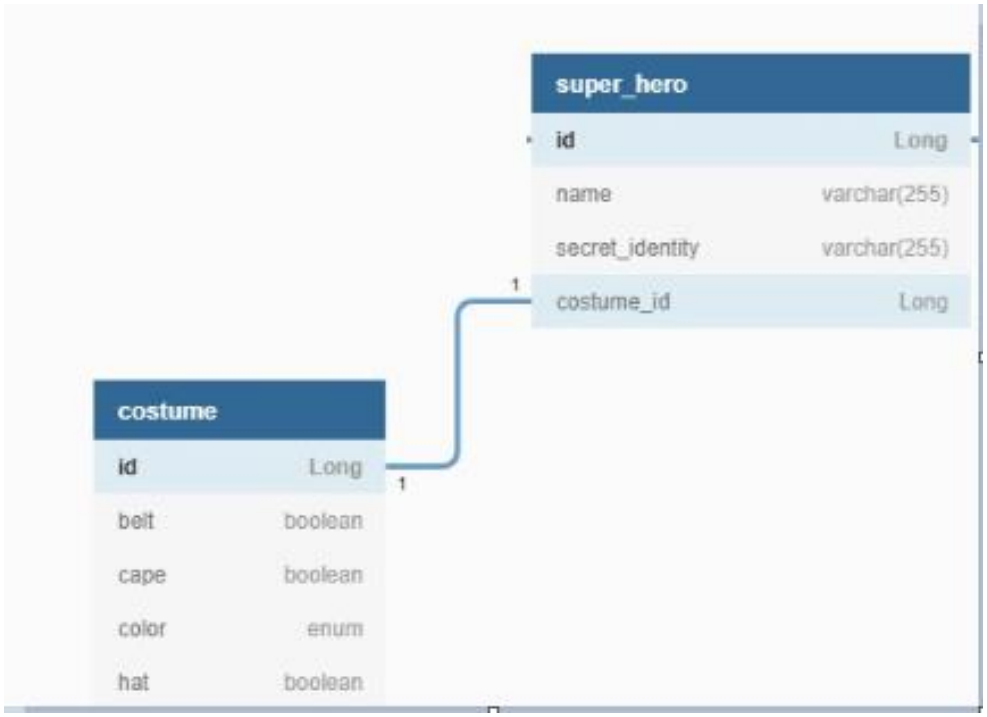
```java
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "super_hero_id")
    private Set<Enemy> enemies;

    private SuperHero() {}

    public SuperHero(String name, String secret_identity, Costume costume,
                     Set<Ability> abilities, Set<Enemy> enemies) {
        this.name = name;
        this.secret_identity = secret_identity;
        this.costume = costume;
        this.abilities = abilities;
        this.enemies = enemies;
    }
}
```

```java
package com.bnpparibas.itg.superhero.superhero.mapping;

public enum TypeAbility {
    FLY,
    RUN_FAST,
    USE_ELEMENT;
}
```

- Utiliser des Set et non des List (Hibernate gère mieux les cycles récursifs)
  - https://docs.jboss.org/hibernate/orm/3.2/api/org/hibernate/collection/PersistentBag.html
- Indiquer les relations avec les propriétés
  - Costume [OneToOne, JoinColumn, CascadeType.ALL]
  - Capacités [ManyToMany, JoinTable, CascadeType.PERSIST, CascadeType.MERGE]
  - Enemies [OneToMany, JoinColum, CascadeType.ALL]
- N'oublier pas de mettre @JsonIdentityInfo(generator = ObjectIdGenerators.UUIDGenerator.class, property="@UUID") sur les classes filles (Set)
  - Cela permet d'éviter les boucles infinies pour la sérialisation vers JSON (Jackson)

Lien Costume SuperHero

Lien 1<->1 – Unidirectionnelle (d'après le corrigé - l'absence de clef étrangère sur SuperHero n'est pas une indication suffisante pour le savoir dans le modèle).
Annotation @OneToOne uniquement dans la classe maitre, cad superhero.



Lien

Lien @ManyToMany avec création d'une table d'association.

http://blog.paumard.org/cours/jpa/chap03-entite-relation.html

4. Mise en relation d'entités

4.1. Introduction

Tout comme en SQL, on peut définir quatre types de relations entre entités JPA :

- relation 1:1 : annotée par `@OneToOne` ;
- relation n:1 : annotée par `@ManyToOne` ;
- relation 1:p : annotée par `@OneToMany` ;
- relation n:p : annotée par `@ManyToMany`.

Bien qu'exprimées entre des classes Java, les relations sont définies entre des entités JPA. Il n'est donc pas légal d'annoter une relation qui pointerait vers une classe qui ne serait pas une entité.

4.2. Relations unidirectionnelles et bidirectionnelles

Les relations entre entités, telles que définies en JPA peuvent être unidirectionnelles ou bidirectionnelles. Dans ce second cas, l'une des deux entités doit être maître et l'autre esclave.

Dans le cas des relations 1:1 et n:p, on peut choisir le côté maître comme on le souhaite. Dans le cas des relations 1:p et n:1, l'entité du côté 1 est l'entité esclave.

4.3. Relation 1:1

Prenons comme exemple un modèle simple qui comporte des `Commune` et des `Maire`. Chaque commune possède un `maire`, et un `maire` ne peut être `maire` que d'une seule `commune`. Nous avons donc bien une relation 1:1 entre les communes et leurs maires. Supposons que dans notre exemple, ce sont les communes qui tiennent la relation entre les communes et les maires.

4.3.1. Cas unidirectionnel

Une première façon d'écrire notre modèle est la suivante :

Exemple 7. Relation 1:1 unidirectionnelle

```
//
 // Entité Maire
 //

 @Entity
 public  class Maire  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @Column(length=40)
     private String nom ;

     // suite de la classe
}


 //
 // Entité Commune
 //

 @Entity
 public  class Commune  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @Column(length=40)
     private String nom ;

     @OneToOne
     private Maire maire ;

     // suite de la classe
}
```

La relation que nous venons d'écrire est unidirectionnelle, en ce sens que l'on peut connaître le maire d'une commune, mais rien dans la classe `Maire` ne nous permet de connaître la commune dont il est maire. Ce cas est le plus simple à traiter : il suffit d'annoter la relation `maire` avec `@OneToOne`.

En base, les choses se passent assez simplement : une colonne `maire_id` va être créée dans la table `Commune`. Cette colonne sera une clé étrangère référençant la colonne `id` de la table `Maire`. On peut imposer le nom de cette colonne en ajoutant l'annotation `@JoinColumn(name="...")` sur la relation `maire`.

L'annotation `@OneToOne` permet de préciser deux comportements importants dans la gestion des relations en JPA : le comportement *cascade* et le comportement *fetch* , que nous verrons à la fin de cette partie.

4.3.2. Cas bidirectionnel

L'entité esclave doit préciser un champ retour par une annotation `@OneToOne` et un attribut `mappedBy`, qui doit référencer le champ qui porte la relation côté maître. Créons par exemple un champ retour dans notre classe `Maire`. L'attribut `mappedBy` est défini sur l'entité esclave de la relation.

Exemple 8. Relation 1:1 bidirectionnelle

```
//
 // Entité Maire
 //

 @Entity
 public  class Maire  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @Column(length=40)
     private String nom ;

     @OneToOne(mappedBy="maire")  // référence la relation dans la classe Commune
     private Commune commune ;

     // suite de la classe
}
```

Définir une relation bidirectionnelle en JPA permet de créer le jeu de clés étrangères qui permet de garantir l'intégrité référentielle de la base de données. Il appartient au code Java de fixer la valeur du champ retour afin de garantir la cohérence du modèle objet.

Une relation bidirectionnelle ne se comporte pas comme deux relations unidirectionnelles. Effectivement, une relation unidirectionnelle est caractérisée par une colonne de jointure sur la table maître, et une clé étrangère de cette colonne vers la clé primaire de l'entité en relation.

Deux relations unidirectionnelles créeront donc deux colonnes de jointure, dans chacune des deux tables en relation.

Une relation bidirectionnelle ne crée pas cette deuxième colonne de la table de jointure, de la table esclave vers la table maître. Lorsque l'on veut lire la relation retour, à partir de l'esclave, une requête est lancée sur la base, en utilisant le caractère bidirectionnel de la relation.

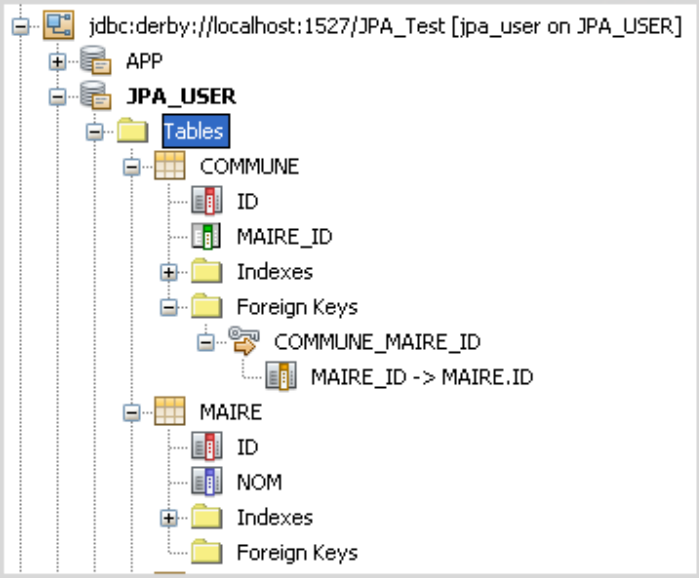


Figure 9. Schéma pour une relation 1:1 unidirectionnelle

Dans notre exemple:
Dans SuperHero:

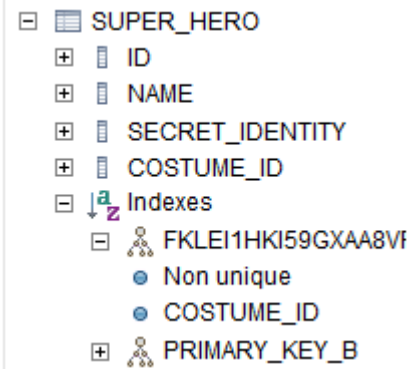
```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "costume_id")
private Costume costume;
```

CascadeType.all => tout ce qui impacte le superHero impacte le costume. La jointure se fait automatiquement en précisant dans la table Super_Hero l'id du costume en tant que clef étrangère. Le name sert uniquement ici à préciser le nom de la colonne dans la table.

```
SELECT * FROM SUPER_HERO;
```

ID	NAME	SECRET_IDENTITY	COSTUME_ID

(no rows, 4 ms)

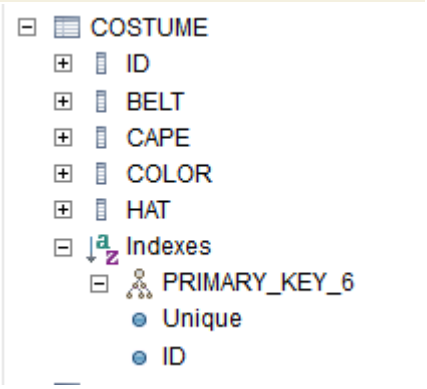


Dans Costume:
Pas de mention de SuperHero. La relation n'est pas bidirectionnelle. Il n'est pas prévu de chercher le SuperHero à partir des caractéristiques de son costume.

```
SELECT * FROM COSTUME;
```

ID	BELT	CAPE	COLOR	HAT

(no rows, 2 ms)



Remarque:
Pour créer une relation bidirectionnelle, procéder ainsi dans la table Costume:

```
@OneToOne(mappedBy = "costume")
private SuperHero superHero;
```

IMPORTANT
Comme expliqué plus haut, l'ajout de ce lien bidirectionnel ne créé par pour autant une clef etrangère dans la table Costume. Pour lire la relation retour à partir de la table esclave, on lit la base en utilisant le caractère bidirectionnel de la relation.
Table costume:

```
SELECT * FROM COSTUME;
```

ID	BELT	CAPE	COLOR	HAT

(no rows, 5 ms)

Si on met des liens bidirectionnels, ils doivent exister dans les deux classes et par conséquent, être annotés dans les deux classes.

```
@OneToOne
private SuperHero superHero;
```

L'instruction ci-dessus semble équivalente à celle qui précise le MappedBy.

https://howtodoinjava.com/hibernate/hibernate-jpa-cascade-types/

JPA Cascade Types

The cascade types supported by the Java Persistence Architecture are as below:

1. **CascadeType.PERSIST** : cascade type `presist` means that save() or persist() operations cascade to related entities.
2. **CascadeType.MERGE** : cascade type `merge` means that related entities are merged when the owning entity is merged.
3. **CascadeType.REFRESH** : cascade type `refresh` does the same thing for the refresh() operation.
4. **CascadeType.REMOVE** : cascade type `remove` removes all related entities association with this setting when the owning entity is deleted.
5. **CascadeType.DETACH** : cascade type `detach` detaches all related entities if a "manual detach" occurs.
6. **CascadeType.ALL** : cascade type `all` is shorthand for all of the above cascade operations.

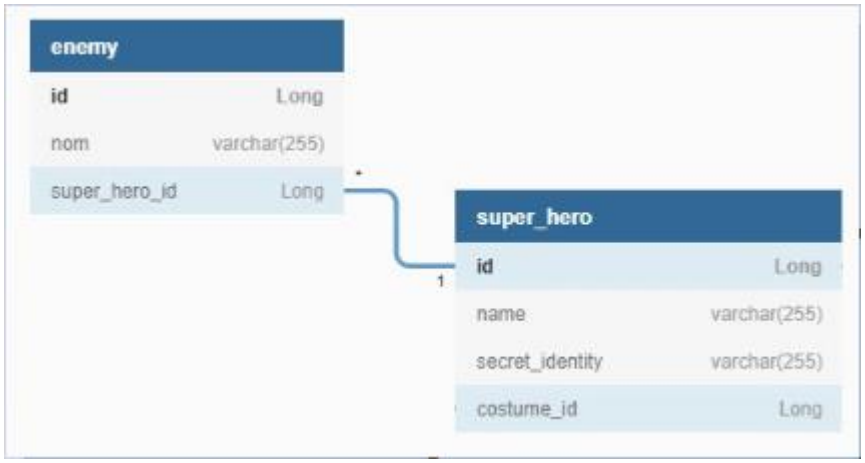
> There is no **default cascade type in JPA**. By default no operations are cascaded.

The cascade configuration option accepts an array of CascadeTypes; thus, to include only refreshes and merges in the cascade operation for a One-to-Many relationship as in our example, you might see the following:

```
@OneToMany(cascade={CascadeType.REFRESH, CascadeType.MERGE}, fetch = FetchType.LAZY)
@JoinColumn(name="EMPLOYEE_ID")
private Set<AccountEntity> accounts;
```

Above cascading will cause accounts collection to be only merged and refreshed.

Lien Enemy SuperHero
Lien 1<->n – Bidirectionnelle Il y a une Annotation @OneToMany dans la classe maitre, cad superhero. Annotation @ManyToOne dans la classe Enemy


enemy
id Long
nom varchar(255)
super_hero_id Long
*
1
super_hero
id Long
name varchar(255)
secret_identity varchar(255)
costume_id Long

4.4. Relation 1:p

Une relation 1:p est caractérisée par un champ de type `Collection` dans la classe maître. La classe esclave ne porte pas de relation retour. Cette relation peut être spécifiée soit par l'annotation `@OneToMany` ou `@ManyToMany`.

Elle peut être unidirectionnelle ou bidirectionnelle. Dans ce second cas, le côté maître est obligatoirement le côté qui tient la relation monovaluée.

4.4.1. Cas unidirectionnel

Dans ces deux cas, JPA crée une table de jointure entre les deux tables associées aux deux entités. Cette table de jointure porte une clé étrangère vers la clé primaire de la première table, et une clé étrangère vers la clé primaire de la deuxième table.

Voyons ceci sur un exemple.

Exemple 9. Relation 1:p unidirectionnelle

```
//
 // Entité Marin
 //

 @Entity
 public  class Marin  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     // reste de la classe
}

 //
 // Entité Bateau
 //
 @Entity
 public  class Bateau  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @OneToMany
     private Collection<Marin> marins ;

     // reste de la classe
}
```

Voici le schéma créé par EclipseLink.

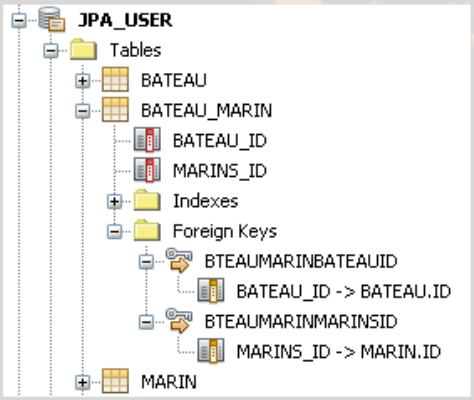


Figure 10. Schéma pour une relation 1:p unidirectionnelle

On peut ici se poser une question : pourquoi la spécification JPA prévoit-elle la création d'une table de jointure alors que la relation est de type 1:p ? La réponse est simple : parce qu'aucune information n'existe sur le champ retour, et que JPA n'a donc aucune information sur la colonne de jointure à utiliser dans la table destination. Dans le cas 1:p bidirectionnel, nous verrons que JPA ne crée pas cette table de jointure.

Nous avons écrit que l'on pouvait utiliser l'annotation `@OneToMany` ou `@ManyToMany` pour spécifier cette relation. Cela dit, il faut noter que les deux annotations ne sont pas équivalente. Dans le premier cas, JPA ajoute une contrainte d'unicité sur la clé étrangère de la table de jointure vers la table `Marin`. C'est bien ce que nous voulons ici : un même marin ne peut pas appartenir à plusieurs équipages à la fois. Il faut toutefois bien avoir présent à l'esprit cette contrainte : une relation de type 1:p est parfois une relation `@ManyToMany`.

4.4.2. Cas bidirectionnel

Une relation 1:p bidirectionnelle doit correspondre à une relation p:1 dans la classe destination de la relation. Comme pour le cas des relations 1:1, le caractère bidirectionnel d'une relation 1:p est marqué en définissant l'attribut `mappedBy` sur la relation.

JPA nous pose une contrainte ici : l'attribut `mappedBy` est défini pour l'annotation `@OneToMany`, mais pas pour l'annotation `@ManyToOne`. Or, comme nous l'avons vu dans le cas de l'annotation `@OneToOne`, `mappedBy` doit être précisé sur le côté esclave d'une relation. Dans le cas d'une relation 1:p bidirectionnelle, JPA ne nous laisse donc pas le choix de l'entité maître et de l'entité esclave.

Exemple 10. Relation 1:p bidirectionnelle

```
//
 // Entité Marin
 //

 @Entity
 public  class Marin  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @ManyToOne
     private Bateau bateau ;

     // reste de la classe
}

 //
 // Entité Bateau
 //
 @Entity
 public  class Bateau  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @OneToMany(mappedBy="bateau")
     private Collection<Marin> marins ;

     // reste de la classe
}
```

Examinons le schéma de base généré par JPA.

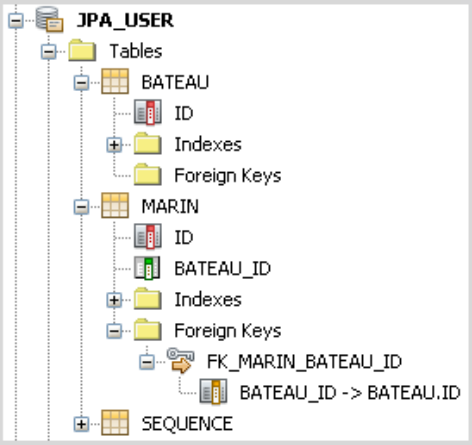


Figure 11. Schéma pour une relation 1:p bidirectionnelle

JPA a créé une colonne de jointure dans la table de l'entité cible (ici `Marin`), de façon à établir la jointure. Il a de plus créé une clé étrangère vers la clé primaire de la table `Bateau`.

Comme on le voit, JPA n'a plus besoin dans ce cas d'une table de jointure pour coder cette relation. On retombe donc dans le cas nominal d'une relation 1:p avec colonne de jointure dans la table cible de la relation.

La classe maître est du côté 1 donc du côté SuperHero.
Dans la table côté many, une colonne pour mettre la clef étrangère (l'info concernant le nom de cette colonne de jointure est néanmoins portée dans la classe maître).

Classe SuperHero

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "super_hero_id")
private Set<Enemy> enemies;
```

Classe Enemy

```
@ManyToOne(cascade = CascadeType.ALL)
private SuperHero superHero;
```

En base:

```
SELECT * FROM SUPER_HERO;
```

ID	NAME	SECRET_IDENTITY	COSTUME_ID

```
SELECT * FROM ENEMY;
```

ID	NAME	SUPER_HERO_ID

(no rows, 4 ms)

Cas @ManyToOne unidirectionnel – non traité par le TP

4.5. Relation p:1

Tout d'abord, notons que le cas bidirectionnel a déjà été traité, puisqu'une relation bidirectionnelle 1:p en JPA est identique à une relation p:1 bidirectionnelle. Voyons donc le cas unidirectionnel sur un exemple.

Exemple 11. Relation p:1 unidirectionnelle

```
//
 // Entité Marin
 //

 @Entity
 public  class Marin  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @ManyToOne
     private Bateau bateau ;

     // reste de la classe
}


 //
 // Entité Bateau
 //
 @Entity
 public  class Bateau  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     // reste de la classe
}
```

Cette fois-ci notre classe `Marin` possède un champ `bateau`, sans que cette classe n'ait accès à la liste de son équipage.

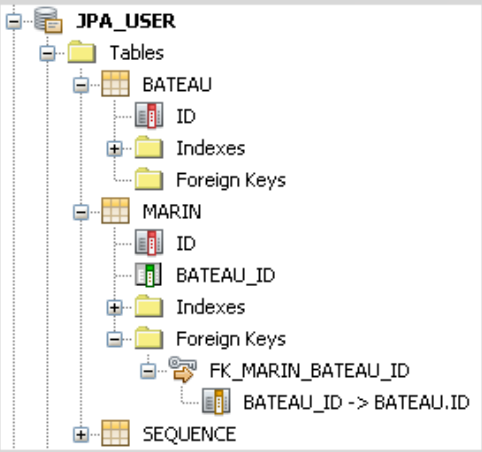


Figure 12. Schéma pour une relation p:1 unidirectionnelle

Comme on le voit, JPA a créé une colonne de jointure dans la table `Marin`, et une clé primaire qui référence la clé primaire de la table `Bateau`.

SuperHero – Ability

@ManyToMany dans les deux sens. Table d'association (dont il faut définir le nom et les noms de colonnes).

4.6. Relation n:p

Une relation n:p est une relation multivaluée des deux côtés de la relation. La façon classique d'enregistrer ce modèle en base consiste à créer une table de jointure, et c'est ce que fait JPA.

4.6.1. Cas unidirectionnel

Voyons ce cas sur un exemple.

Exemple 12. Relation n:p unidirectionnelle

```
//
 // Entité Musicien
 //

 @Entity
 public  class Musicien  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @ManyToMany
     private Collection<Instrument> instruments ;

     // reste de la classe
}

 //
 // Entité Instrument
 //
 @Entity
 public  class Instrument  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     // reste de la classe
}
```

JPA génère le schéma suivant pour ce jeu de classes. Une table de jointure est créé entre les deux tables qui portent les entités. Cette table référence les deux clés primaires des deux entités au travers de clés étrangères. Le schéma généré est donc ici tout à fait classique.

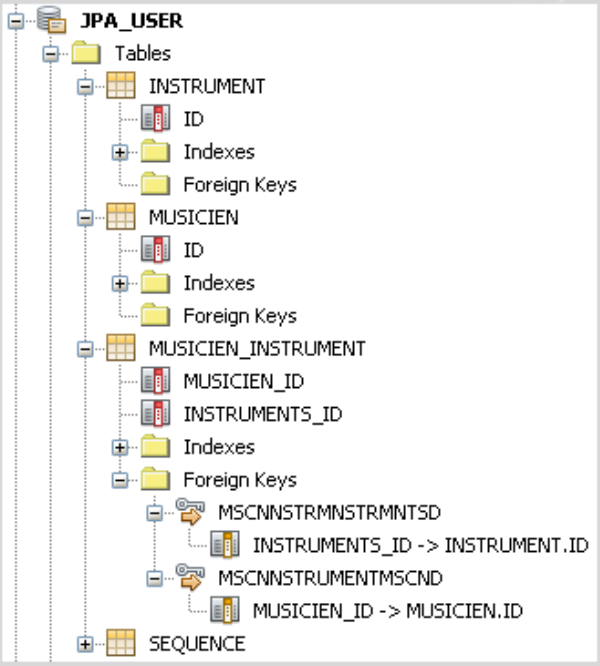


Figure 13. Schéma pour une relation n:p unidirectionnelle

4.6.2. Cas bidirectionnel

Dans le cas bidirectionnel, l'entité cible porte également une relation `@ManyToMany` vers l'entité maître. Cette relation doit comporter un attribut `mappedBy`, qui indique le nom de la relation correspondante dans l'entité maître.

Exemple 13. Relation n:p bidirectionnelle

```
//
 // Entité Musicien
 //

 @Entity
 public  class Musicien  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @ManyToMany
     private Collection<Instrument> instruments ;

     // reste de la classe
}

 //
 // Entité Instrument
 //
 @Entity
 public  class Instrument  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @ManyToMany(mappedBy="instruments")
     private Collection<Musicien> musiciens ;

     // reste de la classe
}
```

Dans ce deuxième cas, la structure de tables générée est la même. Notons encore une fois que c'est la présence de l'attribut `mappedBy` qui crée le caractère bidirectionnel de la relation. Si l'on ne le met pas, alors JPA créera une seconde table de jointure.

Voir plus haut l'importance du mappedBy pour éviter la création d'une seconde table de jointure (cad 2 unidirectionnels au lieu d'un bidirectionnel).

SuperHero

```java
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(
        name = "superhero_has_abilities",
        joinColumns = @JoinColumn(name = "superhero_id"),
        inverseJoinColumns = @JoinColumn(name = "ability")
)
private Set<Ability> abilities;

@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "super_hero_id")
private Set<Enemy> enemies;
```

Description de la table de jointure et de ses colonnes.

Ability

```java
@ManyToMany(mappedBy = "abilities")
private Set<SuperHero> superHeroes;
```

MappedBy pour créer le caractère bidirectionnel de la relation.

En base:

```
SELECT * FROM ABILITY;
```

ID	DESCRIPTION	TYPE_ABILITY

(no rows, 2 ms)

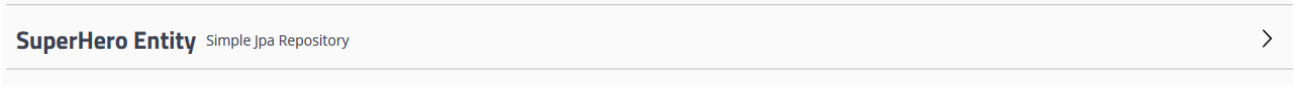
Table de jointure:

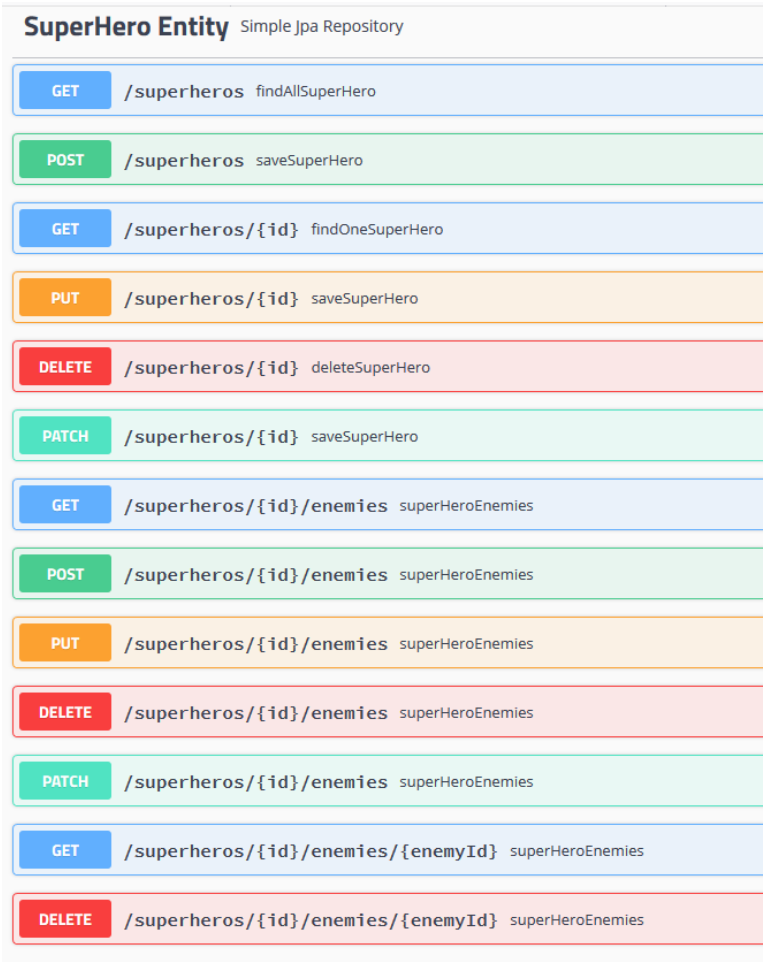
```
select * from SUPERHERO_HAS_ABILITIES;
```

SUPERHERO_ID	ABILITY

(no rows, 1 ms)

Ce qui sous SWAGGER génèree toutes les API sous SWAGGER:





C’est l'interface

```java
package com.bnpparibas.itg.superhero.superhero.restjpa;

import com.bnpparibas.itg.superhero.superhero.mapping.SuperHero;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(path = "superheros")
public interface SuperHeroRepositoryRest extends JpaRepository<SuperHero, Long> {}
```

Annotée @RepositoryRestRessource (path=”superheros”)

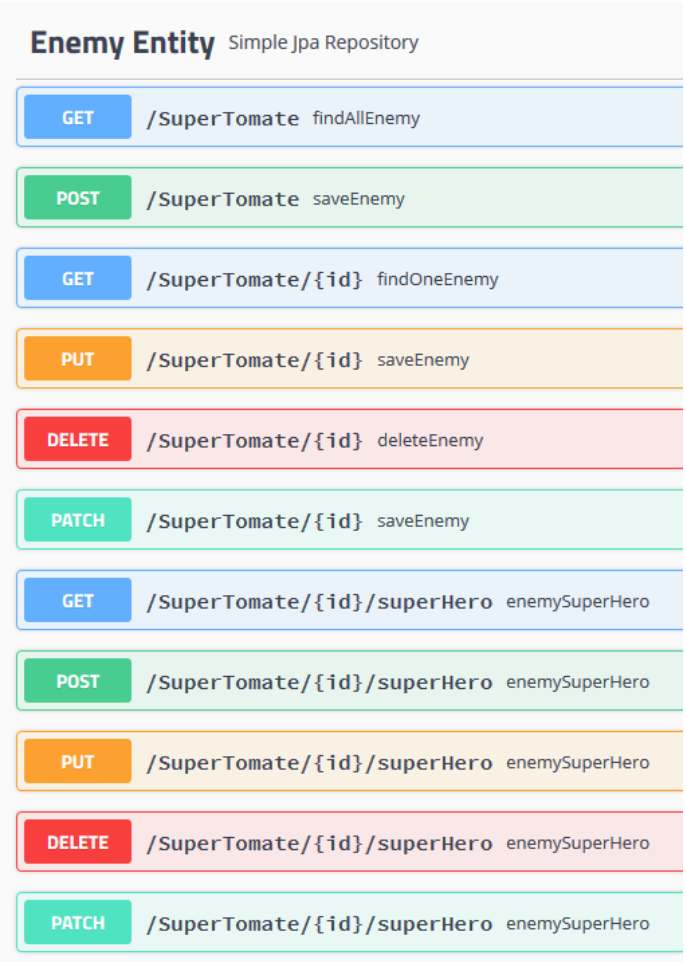
Le path Est-ce qui est dans le premier /path/

Test: création d’une classe analogue basée sur Enemy

```java
package com.bnpparibas.itg.superhero.superhero.restjpa;

import com.bnpparibas.itg.superhero.superhero.mapping.Enemy;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(path = "Enemy")
public interface EnemyRepositoryRest extends JpaRepository<Enemy, Long> {
}
```

Résultat:

Rq au début, le path s'appelait supertomate et le résultat était:




```java
package com.bnpparibas.itg.superhero.superhero.restjpa;

import com.bnpparibas.itg.superhero.superhero.mapping.Enemy;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(path = "SuperTomate")
public interface EnemyRepositoryRest extends JpaRepository<Enemy, Long> {
}
```

A ce stade aucun service rest n'a été explicitement codé. Voir pourquoi ils sont générés de base? Est-ce du à la présence de l'interface?

Remarque, sans l'annotation, les commandes apparaissent dans Swagger et semblent marcher (Post et création directe d'enemy en base).

Retour à la norme.

Api Documentation 1.0

[Base URL: localhost:8080/]
http://localhost:8080/v2/api-docs

Api Documentation

Terms of service

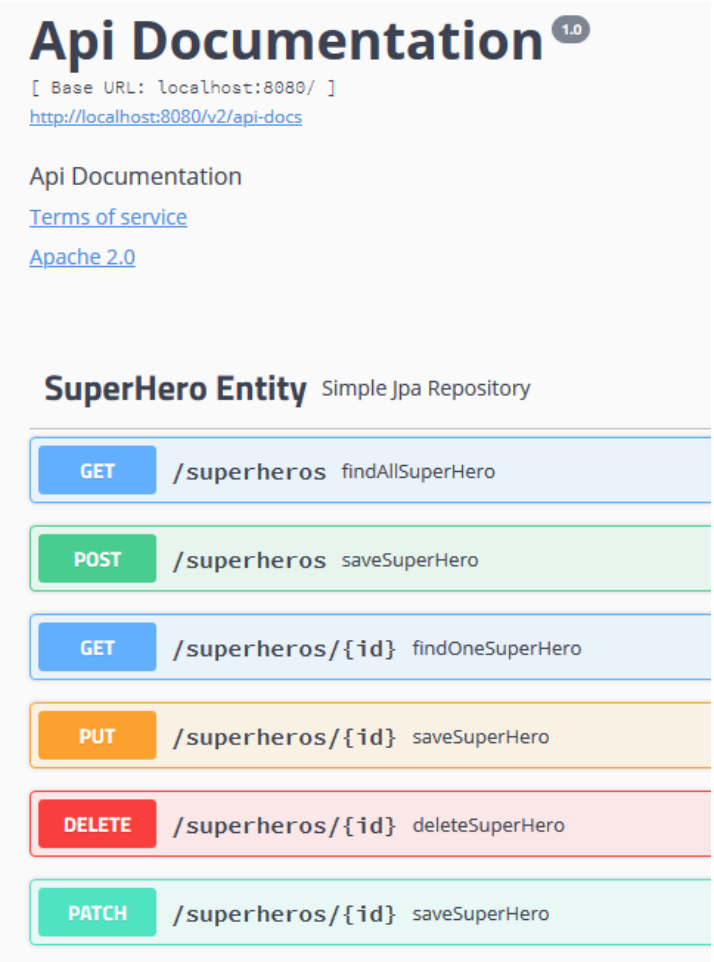
Apache 2.0

SuperHero Entity Simple Jpa Repository

basic-error-controller Basic Error Controller

profile-controller Profile Controller

Models



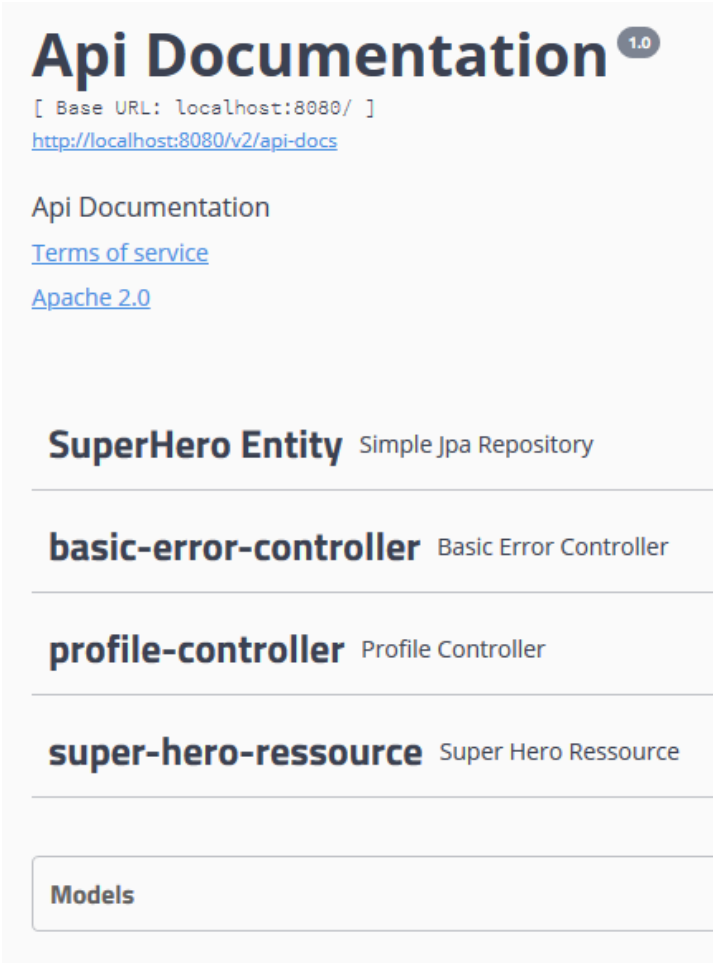
Ajout d'une classe SuperHeroRessource

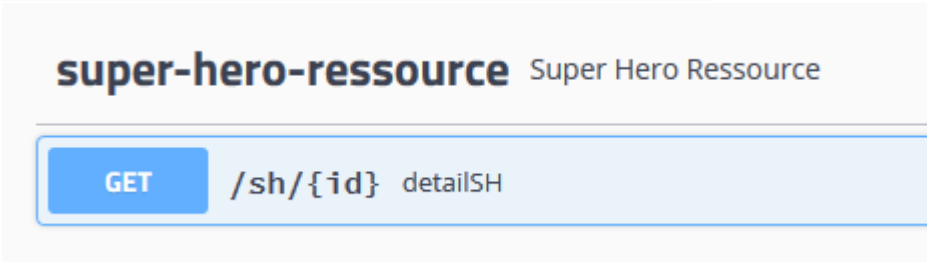
```java
package com.bnpparibas.itg.superhero.superhero.restjpa;

import com.bnpparibas.itg.superhero.superhero.mapping.SuperHero;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

public class SuperHeroRessource {
    @Autowired
    private SuperHeroRepositoryRest superHeroRepositoryRest;

    @RequestMapping(method = RequestMethod.GET, path = {"/sh/{id}"})
    public SuperHero detailSH(@PathVariable("id") Long id){
        SuperHero sh  = superHeroRepositoryRest.findOne(id);
        return sh;
    }
}
```

Aucun changement car pas signalé à Swagger que la classe doit être un restController, donc il ne se passe rien.

Ajouter @RestController sur la déclaration de la classe.

```java
import com.bnpparibas.itg.superhero.superhero.mapping.SuperHero;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SuperHeroRessource {
    @Autowired
    private SuperHeroRepositoryRest superHeroRepositoryRest;

    @RequestMapping(method = RequestMethod.GET, path = {"/sh/{id}"})
    public SuperHero detailSH(@PathVariable("id") Long id){
        SuperHero sh  = superHeroRepositoryRest.findOne(id);
        return sh;
    }
}
```



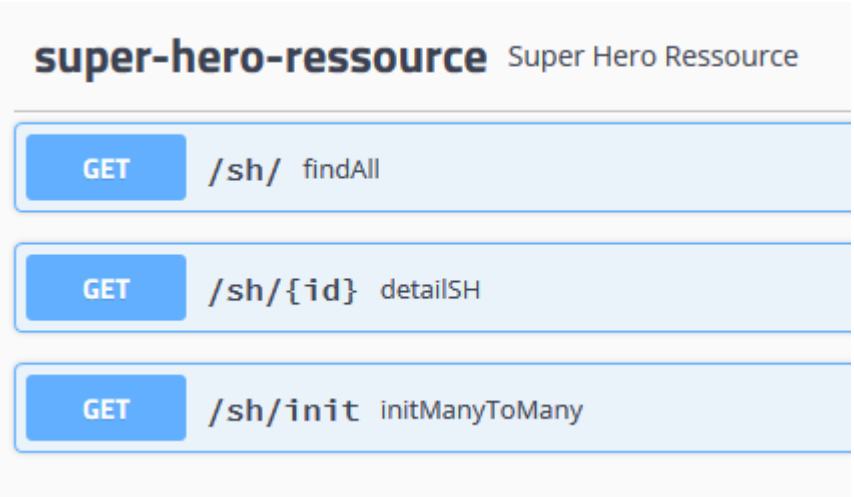


La nouvelle API apparait.

```java
public class SuperHeroRessource {
    @Autowired
    private SuperHeroRepositoryRest superHeroRepositoryRest;

    @RequestMapping(method = RequestMethod.GET, path = {"/sh/{id}"})
    public SuperHero detailSH(@PathVariable("id") Long id){
        SuperHero sh  = superHeroRepositoryRest.findOne(id);
        return sh;
    }

    @RequestMapping(method = RequestMethod.GET, path = {"/sh/"})
    public List<SuperHero> findAll(){
        return superHeroRepositoryRest.findAll();
    }

    @RequestMapping(method = RequestMethod.GET,path = {"/sh/init"})
    public void initManyToMany (){
        Costume c1 = new Costume( belt: true, cape: true, Color.BLUE, hat: true);
        Set<Enemy> e1 = new HashSet<>(Arrays.asList(new Enemy( name: "e1", superHero: null)));
        Set<Ability> a1 = new HashSet<>(Arrays.asList(new Ability(TypeAbility.RUN_FAST, description: "RUNNNNN", superHeroes: null)));
        SuperHero sh1 = new SuperHero( name: "sh1", secret_identity: "secreteSh1",c1,a1,e1);
        superHeroRepositoryRest.save(sh1);

        Set<Enemy> e2 = new HashSet<>(Arrays.asList(new Enemy( name: "e2", superHero: null)));
        SuperHero sh2 = new SuperHero( name: "sh2", secret_identity: "secreteSh2",c1,a1,e2);
        superHeroRepositoryRest.save(sh2);

        Set<Enemy> e3 = new HashSet<>(Arrays.asList(new Enemy( name: "e3", superHero: null)));
        Set<Ability> a3 = new HashSet<>(Arrays.asList(new Ability(TypeAbility.FLY, description: "FLYYYY", superHeroes: null)));
        SuperHero sh3 = new SuperHero( name: "sh3", secret_identity: "secreteSh3",c1,a3,e3);
        superHeroRepositoryRest.save(sh1);
    }
}
```

Dernière ligne de code lire :

```java
superHeroRepositoryRest.save(sh3);
```



Après un init

select * from SUPER_HERO;

ID	NAME	SECRET_IDENTITY	COSTUME_ID
1	sh1	secreteSh1	1
2	sh2	secreteSh2	1
3	sh3	secreteSh3	1

(3 rows, 2 ms)

select * from ABILITY;

ID	DESCRIPTION	TYPE_ABILITY
1	RUNNNNN	RUN_FAST
2	FLYYYY	FLY

(2 rows, 2 ms)

select * from COSTUME;

ID	BELT	CAPE	COLOR	HAT
1	TRUE	TRUE	BLUE	TRUE

(1 row, 1 ms)

select * from ENEMY;

ID	NAME	SUPER_HERO_ID
1	e1	1
2	e2	2
3	e3	3

(3 rows, 2 ms)

select * from SUPERHERO_HAS_ABILITIES;

SUPERHERO_ID	ABILITY
1	1
2	1
3	2

(3 rows, 1 ms)

Remarque – avec 2 inits

select * from SUPER_HERO;

ID	NAME	SECRET_IDENTITY	COSTUME_ID
1	sh1	secreteSh1	1
2	sh2	secreteSh2	1
3	sh3	secreteSh3	1
4	sh1	secreteSh1	2
5	sh2	secreteSh2	2
6	sh3	secreteSh3	2

(6 rows, 2 ms)

select * from ABILITY;

ID	DESCRIPTION	TYPE_ABILITY
1	RUNNNNN	RUN_FAST
2	FLYYYY	FLY
3	RUNNNNN	RUN_FAST
4	FLYYYY	FLY

(4 rows, 2 ms)

select * from COSTUME;

ID	BELT	CAPE	COLOR	HAT
1	TRUE	TRUE	BLUE	TRUE
2	TRUE	TRUE	BLUE	TRUE

(2 rows, 1 ms)

select * from ENEMY;

ID	NAME	SUPER_HERO_ID
1	e1	1
2	e2	2
3	e3	3
4	e1	4
5	e2	5
6	e3	6

(6 rows, 2 ms)

select * from SUPERHERO_HAS_ABILITIES;

SUPERHERO_ID	ABILITY
1	1
2	1
3	2
4	3
5	3
6	4

(6 rows, 1 ms)

Ajout de requêtes particulières dans SuperHeroRepositoryRest

jparepository findby field

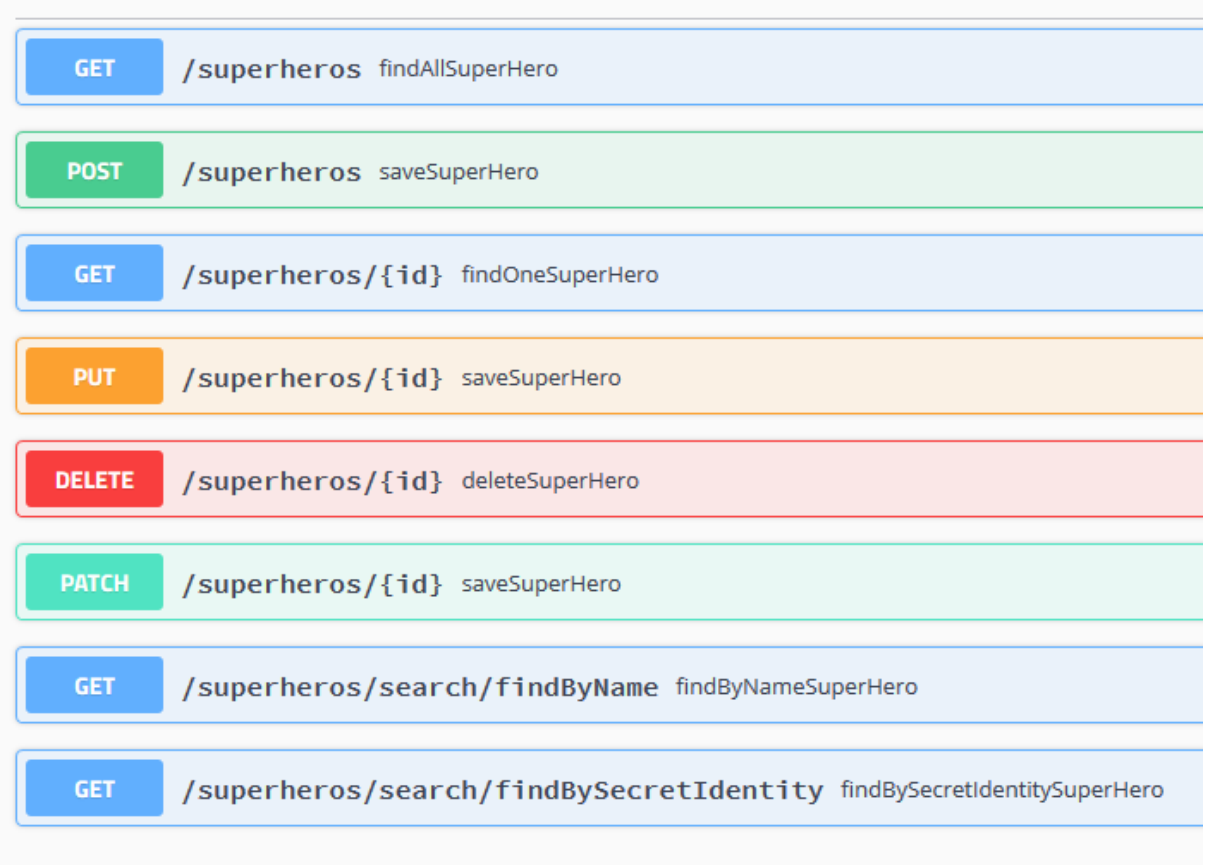
Attention, on peut faire un findByNImporteQuelField mais pas s'il y a un underscore dans le nom de l'attribut.
Avec le nom secret_identity, plantage.
En renommant secret_identity en secretIdentity ça marche!

```
@RepositoryRestResource(path = "superheros")
public interface SuperHeroRepositoryRest extends JpaRepository<SuperHero, Long> {
    List<SuperHero> findByName (@Param("name") String name);
    List<SuperHero> findBySecretIdentity (@Param("secret") String secretIdentity);
}
```

SELECT * FROM SUPER_HERO;

ID	NAME	SECRET_IDENTITY	COSTUME_ID
1	sh1	secreteSh1	1
2	sh2	secreteSh2	1
3	sh3	secreteSh3	1
4	sh1	secreteSh1	2
5	sh2	secreteSh2	2
6	sh3	secreteSh3	2

(6 rows, 2 ms)



COMPLEMENTS

4.7. Comportement *cascade*

Comme nous l'avons vu, l' *entity manager* permet de mener à bien cinq opérations sur une entité : `DETACH`, `MERGE`, `PERSIST`, `REMOVE`, `REFRESH`.

Le comportement *cascade* consiste à spécifier ce qui se passe pour une entité en relation d'une entité père (que cette relation soit monovaluée ou multivaluée), lorsque cette entité père subit une des opérations définies ci-dessus.

Prenons l'exemple suivant.

Exemple 14. Comportement cascade sur une relation `@OneToOne`

```
@Entity
 public  class Commune  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
     private Maire maire ;

     // reste de la classe
}
```

Le comportement *cascade* est précisé par l'attribut `cascade`, disponible sur les annotations : `@OneToOne`, `@OneToMany` et `@ManyToMany`. La valeur de cet attribut est une énumération de type `CascadeType`. En plus des valeurs `DETACH`, `MERGE`, `PERSIST`, `REMOVE`, `REFRESH`, cette énumération définit la valeur `ALL`, qui correspond à toutes les valeurs à la fois.

Remarquons bien que l'annotation `@ManyToOne` ne définit pas cet attribut.

4.8. Effacement des entités orphelines

JPA 2.0 apporte une amélioration très intéressante sur JPA 1.0 : la détection d'entités orphelines. Dans de nombreux cas, l'existence d'une entité n'a de sens que si elle est en relation d'une autre entité (cas des compositions de l'UML). Effacer cette entité père doit alors entraîner l'effacement de cette entité en relation.

Dans la plupart des cas, le comportement *cascade* suffit à traiter ce cas. Mais il arrive que des effacements d'entité soient nécessaires sans qu'il y ait eu d'appel à la méthode `remove()` de l' *entity manager* .

Dans l'exemple de nos communes et de nos maires, effacer une commune entraînera l'effacement du maire. Mais l'appel à `commune.setMaire(null)` doit aussi entraîner l'effacement de ce maire, dans la mesure où cette entité sera orpheline.

On dispose pour cela d'un attribut défini sur `@OneToOne` et `@OneToMany` : `orphanRemoval`. Le fait de mettre cet attribut à `true` activera la détection d'entités orphelines, et leur effacement automatique.

Exemple 15. Effacement des orphelins sur une relation `@OneToOne`

```
@Entity
 public  class Commune  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE}
        orphanRemoval=true)
     private Maire maire ;

     // reste de la classe
}
```

5. Charger des entités et leurs relations

L'utilisation de JPA, ou de tout autre outil automatique d'accès à des données en base au travers d'un modèle objet, masque la complexité de l'écriture des requêtes SQL au développeur. Cela représente un gain de productivité énorme, qui explique largement l'extraordinaire succès de ces outils. Cela dit, la génération automatique de code SQL et son exécution sont toujours présentes, et les éventuels problèmes, notamment de performance, qui se posent le sont aussi. Il est donc important de bien maîtriser le SQL pour pouvoir utiliser efficacement de tels outils.

L'un des points coûteux de la lecture des données en base est précisément la façon dont on explore les relations des objets que l'on manipule. Reprenons l'exemple de notre `Bateau` qui possède un équipage, stocké sous forme d'une collection de `Marin`.

Une manière naïve de lire un bateau en base, serait de lire ce bateau avec une requête `SELECT`, et de laisser la collection de marins vide. Si l'application explore la relation `marins`, par itération ou autre, alors un deuxième `SELECT` est émis, qui peuple la relation de façon à faire fonctionner le système correctement. Au total, deux `SELECT` sont émis, donc deux allers-retours avec la base de données.

Dans le cas où l'on sait que la relation `marins` sera explorée systématiquement après la lecture d'un bateau, il serait plus malin de n'émettre qu'un seul `SELECT`, avec une jointure, de manière à peupler la relation `marins` à l'avance. Cela ne ferait qu'un seul aller-retour avec la base de données, et serait de ce fait beaucoup plus performant.

En revanche, dans le cas d'une relation qui, pour des raisons applicatives, ne serait pas explorée, ou rarement, alors l'exécution de la jointure lors du `SELECT` serait un surcoût inutile.

JPA nous permet de régler, relation par relation, la façon dont il doit se comporter :

- doit-il la charger par défaut ?
- doit-il la laisser vide, et la charger à la demande ?

On utilise pour cela l'attribut `fetch`, défini sur les annotations `@OneToOne`, `@OneToMany`, `@ManyToOne` et `@ManyToMany`. Cet attribut peut prendre deux valeurs :

- `FetchType.LAZY` : indique que la relation doit être chargée à la demande ;
- `FetchType.EAGER` : indique que la relation doit être chargée en même temps que l'entité qui la porte.

Voyons ceci sur un exemple.

Exemple 16. Utilisation de `fecth` sur une relation

```
@Entity
 public  class Bateau  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @OneToMany(fetch=FetchType.EAGER)  // la relation est chargée par défaut
     private Collection<Marin> marins ;

     // reste de la classe
}
```

Encore une fois, il n'y a pas de règle générale pour l'utilisation de cet attribut. Chaque cas est un cas particulier, qu'il faut traiter en tant que tel.

6. Objets inclus

6.1. Introduction

Dans de nombreux cas, mettre des objets en relation est très coûteux et inutile. Coûteux pour deux raisons :

- Surcharge en lecture et écriture : la lecture entraîne une jointure, ou une requête supplémentaire, l'insertion une requête supplémentaire.
- Les objets en relations sont stockées dans une table à part, qui peut contenir des quantités très importantes d'objets.

Elle peut être inutile, notamment dans le cas où l'objet en relation partage complètement le cycle de vie de l'objet maître. Dans ce cas, créer une relation `@OneToOne` est un luxe.

JPA propose pour cela une fonctionnalité, qui permet d'inclure les champs de l'objet en relation avec ceux de l'objet maître. Cette technique résout tous les problèmes.

6.2. Déclaration d'un objet inclus

On déclare une classe d'objets inclus en annotant simplement la classe de ces objets avec `@Embeddable` plutôt qu' `@Entity`. Les contraintes sur ces classes sont les mêmes : elles doivent être sérializables, et comporter un constructeur vide, par défaut ou explicite.

Une entité incluse n'a pas besoin de déclarer de clé primaire : elle n'a pas de table en propre, et partagera donc la même clé primaire que les objets qui la déclarent en relation.

Voyons un exemple d'objet inclus.

Exemple 17. Déclaration d'objets inclus

```
@Embeddable
 public  class Adresse  implements Serializable {

     @Column(length=40)
     private String rue ;

     @ManyToOne
     private Commune commune ;

     // reste de la classe
}
```

On remarque tout d'abord que cette classe n'est pas annotée avec `@Entity`, il ne s'agit donc pas d'une entité JPA. Elle est annotée avec `@Embeddable`, et ne pourra être utilisée que dans des relations annotées avec `@Embedded`.

Cette classe va nous permettre d'ajouter une adresse à nos marins. On remarque que l'objet inclus est lui-même en relation avec les communes, ce qui est autorisé.

6.3. Utilisation d'objets inclus

Ajoutons une adresse à nos marins sur l'exemple suivant.

Exemple 18. Utilisation d'objets inclus

```
@Entity
 public  class Marin  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @Column(length=40)
     private String name ;

     @Embedded
     private Adresse adresse ;

     // reste de la classe
}
```

Plutôt que d'annoter la relation `adresse` avec `@OneToOne`, comme on l'aurait fait pour une relation classique, nous l'avons annotée avec `@Embedded`, ce qui signifie que les champs de notre adresse seront créés directement dans la table `Marin`. Un objet référencé dans une relation annotée par `@Embedded` doit nécessairement être annoté par `@Embeddable`.

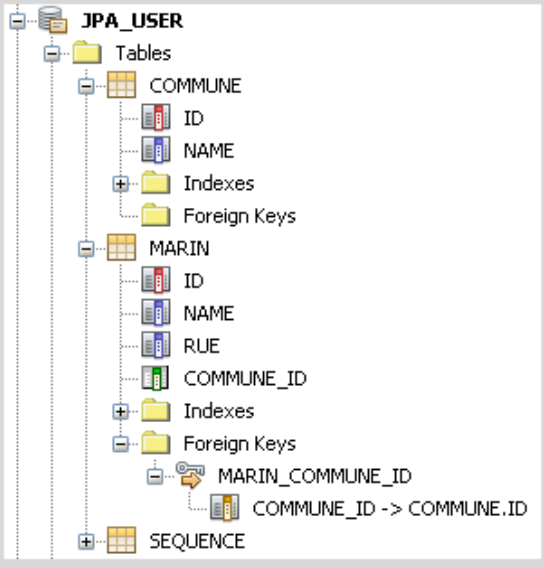


Figure 14. Schéma pour un objet inclus – 1

On constate que le schéma généré comporte bien la colonne `rue` et la colonne de jointure vers la table `Commune`.

6.4. Cas où l'objet inclus est nul

Du point de vue du graphe d'objets rien n'empêche un objet inclus d'être nul : après tout, ce n'est qu'une relation comme les autres, qui peut très bien ne pointer vers rien. Du point de vue de la base de données, le problème est différent : rien ne différencie un marin dont on ne connaît pas l'adresse, d'un marin qui en a une.

Si l'on tente d'écrire en base un marin qui n'a pas d'adresse, on risque d'obtenir une exception, dans la mesure où les objets inclus nuls ne sont pas autorisés en JPA.

Si l'on lit un marin en base qui n'a pas d'adresse, on obtiendra un objet Java `marin`, qui possède une adresse dont tous les champs sont vides.

On peut mettre en place plusieurs solutions pour régler ce problème :

- Créer un objet `adresse` particulier, dont les champs indiquent que cet objet correspond à un objet nul. Cette approche n'est valide que dans certains cas, dans d'autres, une telle configuration de champs n'existe pas.
- Ajouter à l'objet inclus une colonne technique, booléenne, qui indique que cet objet est en fait nul. Cette approche fonctionne dans tous les cas.

Dans ces deux cas il faut modifier le *getter* du champ inclus de façon à ce qu'il retourne `null` quand le champ inclus est en fait nul. Voyons ceci sur un exemple.

On notera deux choses sur cet exemple :

- Le booléen `isNull` de la classe `Adresse` est positionné à `true` par défaut. Donc, toute relation vers une adresse initialisée avec une adresse vide génèrera un retour `null` sur appel du *getter* de cette relation.
- Dans la classe `Marin`, le champ `adresse` est initialisé sur une valeur particulière : `Adresse.ADRESSE_NULL`, de façon à ne jamais être nul. Cela permet d'éviter les éventuelles erreurs si cette relation n'a pas été initialisée. Notons que l'objet `Adresse.ADRESSE_NULL` génère bien un retour nul du *getter* associé.

Exemple 19. Champ inclus nul

```
//
 // Entité Adresse
 //
 @Embeddable
 public  class Adresse  implements Serializable {

     private  boolean isNull = true ;

     private String rue ;

     @ManyToOne
     private Commune commune ;

     // objet adresse utilisé pour l'initialisation
     // le champ isNull de cet objet est true
     public  static ADRESSE_NULL =  new Adresse() ;

     // méthode appelée par les getters
     public  static Adresse getAdresse(Adresse adresse) {
         if (adresse == null || adresse.isNull) {
             return null ;
         }  else {
             return adresse ;
         }
    }

     // reste de la classe
}

 //
```

```java
//
// Entité Marin
//
@Entity
public  class Marin  implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(length=40)
    private String name ;

    // le champ est initialisé non nul, afin d'éviter
    // les erreurs à l'écriture
    @Embedded
    private Adresse adresse = Adresse.ADRESSE_NULL ;

    // Getter pour l'adresse
    public Adresse getAdresse() {
        return Adresse.getAdresse(this.adresse) ;
   }

    // reste de la classe
}
```

JPA nous crée alors le schéma suivant, nous allons pouvoir enregistrer des valeurs nulles dans notre modèle d'objets Java.

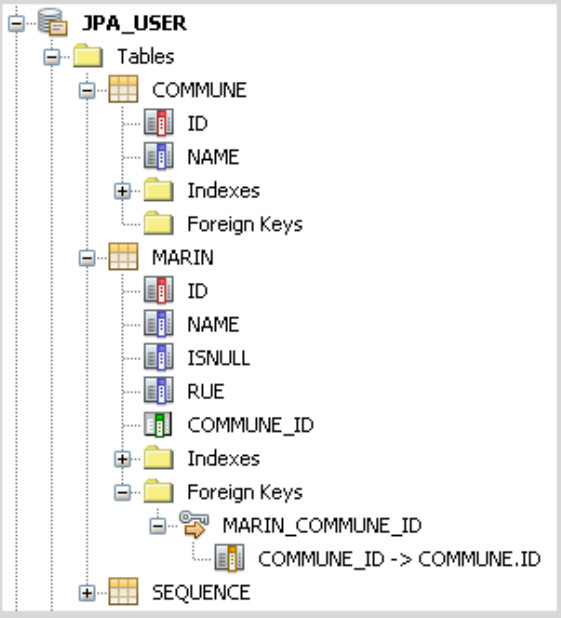


Figure 15. Schéma pour un objet inclus – 2

6.5. Renommer les colonnes incluses

Il nous reste un dernier problème à traiter : que se passe-t-il si nous devons enregistrer plusieurs instances d'une même classe d'objets inclus dans une entité ?

La réponse dépend de l'implémentation, et dans chaque implémentation, peut dépendre de la version que l'on utilise. Si l'implémentation est maligne, elle va se rendre compte qu'il y a une collision de noms, et le gérer en ajoutant des numéros à ces noms. C'est ce que fait Hibernate. Si elle est moins maligne, elle ne va pas voir la collision de nom, et ne pas créer le bon nombre de colonnes. C'est ce que fait EclipseLink 2.0.2.

Comme d'habitude dans pareil cas, il faut appliquer le vieux proverbe : on n'est jamais aussi bien servi que par soi-même, et utiliser la possibilité que JPA nous donne de surcharger le nommage des colonnes des objets inclus.

Exemple 20. Renommage des colonnes des objets inclus

```
@Entity
 public  class Marin  implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @Column(length=40)
     private String name ;

     @Embedded
     @AttributeOverrides({
    @AttributeOverride(
      name="isNull",
      column=@Column(name="adr_courante_isnull")),
    @AttributeOverride(
      name="rue",
      column=@Column(name="adr_courante_rue"))
  })
     @AssociationOverrides({
    @AssociationOverride(
      name="commune",
      joinColumns=@JoinColumn(name="adr_courante_commune_id"))
  })
```

```
    private Adresse adresseCourante ;

    @Embedded
    @AttributeOverrides({
    @AttributeOverride(
      name="isNull",
      column=@Column(name="adr_naissance_courante_isnull")),
    @AttributeOverride(
      name="rue",
      column=@Column(name="adr_naissance_courante_rue"))
})
    @AssociationOverrides({
    @AssociationOverride(
      name="commune",
      joinColumns=@JoinColumn(name="adr_naissance_commune_id"))
})
    private Adresse adresseDeNaissance ;

    // reste de la classe
}
```

Du point de vue de JPA, la classe `Adresse` porte deux types de champs : des types de base et des relations. Les types de base sont stockés dans des colonnes, et les relations dans des colonnes de jointure, qui peuvent être rangées dans la même table qu'une autre entité, ou dans une table propre (table de jointure).

On renomme une colonne par l'annotation `@AttributOverride`. Cette annotation prend deux attributs :

- `name`, de type `String` : désigne le nom du champ dans la classe ;
- `column`, de type `@Column` : définit la colonne dans laquelle ce champ sera enregistré. Le type `@Column` définit lui-même plusieurs attributs, dont `name` que nous utilisons ici.

Comme la classe `Adresse` porte deux champs : `isNull` et `rue`, nous devons utiliser deux annotations `@AttributeOverride`. Ces deux annotations sont regroupées dans une annotation `@AttributeOverrides` (notons le "s" à la fin), placée sur le champ inclus.

Reste la relation que `Adresse` porte, vers `Commune`. Cette relation est ici enregistrée dans une colonne de jointure, clé étrangère qui référence la clé primaire de la table `Commune`.

On redéfinit le nom de cette colonne en utilisant une annotation `@AssociationOverride`, qui fonctionne de la même façon que `@AttributeOverride`. Elle définit trois attributs :

- `name`, de type `String` : désigne le nom du champ dans la classe. Cet attribut est le même que dans `@AttributeOverride`.
- `joinColumns`, de type tableau de `@JoinColumn`. Effectivement, JPA supporte les clés primaires composites (ce point n'est pas traité ici), et dans ce cas une jointure peut être enregistrée dans plusieurs colonnes. Ici ce tableau ne porte qu'une unique valeur, de type `@JoinColumn`.
- `joinTable` : comme on l'a vu, la jointure peut être enregistrée dans une table de jointure. Dans ce cas, cet attribut nous donne le nom de cette table.

Techniquement, l'annotation `@JoinColumn` fonctionne comme l'annotation `@Column`, et définit des attributs équivalents.

On obtient alors le schéma suivant.

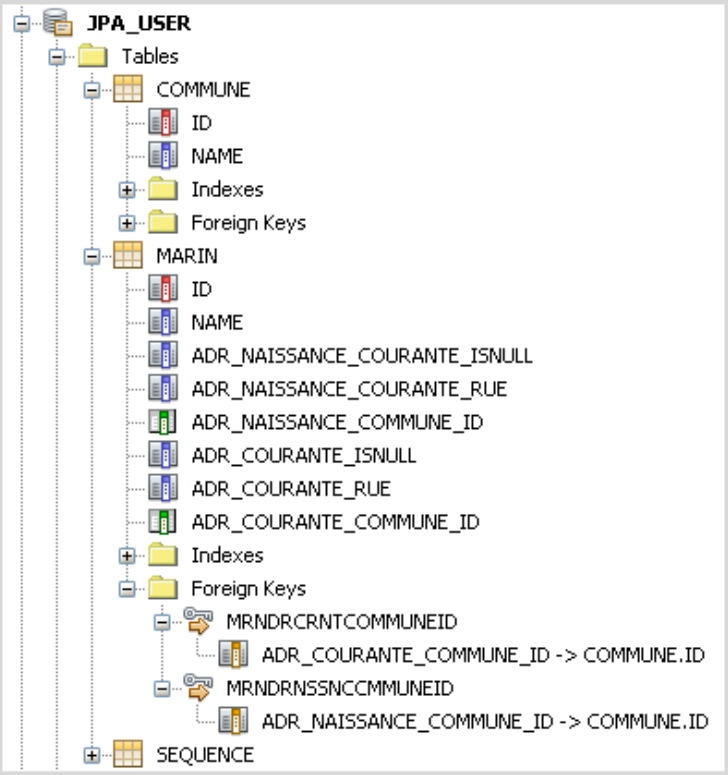


Figure 16. Schéma pour deux objets inclus

6.6. Collections d'objets inclus

Il est possible pour une entité JPA de posséder une collection d'objets `@Embeddable`. Une telle collection doit être annotée `@ElementCollection`, comme une collection dont les éléments sont des types de base. Les objets inclus seront alors enregistrés dans une table à part, de la même manière que les types de base.

Requêtes

1. Introduction

En plus de pouvoir associer un modèle de classes Java et une structure de table dans une base de données, la spécification JPA décrit un langage de requête, appelé JPQL (Java Persistence Query Language). Ce langage, reprend les fonctionnalités du SQL, et permet d'interroger une base de données en utilisant les entités JPA définies dans une unité de persistance.

Même si la syntaxe de JPQL ressemble fort à celle du SQL, ce n'est pas la même, et les notions manipulées sont légèrement différentes. Là où chaque base de données a sa propre version de SQL, il n'existe qu'un seul JPQL. Une requête JPQL est donc écrite indépendamment de la base de données à laquelle elle s'adresse, ce qui est un gain important de productivité.

Le langage JPQL définit trois types de requêtes :

- les requêtes de sélections ;
- les requêtes de mise à jour ;
- les requêtes d'effacement.

Notons qu'une requête opère toujours sur les entités d'une unique unité de persistance. Il n'est pas possible de faire des requêtes portant sur des données gérées par plusieurs unités de persistance.

2. Un premier exemple

2.1. Écriture d'une première requête

Commençons par écrire une première requête JPQL, qui cherche les marins dont le nom est Surcouf.

Exemple 29. Une première requête JPQL

```
select marin from Marin marin where marin.nom =  'Surcouf'
```

Cette requête ressemble grandement à du SQL, notons tout de même les différences suivantes :

- En SQL on sélectionne une liste de colonnes appartenant à une ou plusieurs tables. En JPQL on sélectionne une entité. L'entité que l'on sélectionne, ici `Marin`, est désignée par son nom. Rappelons que le nom d'une entité est donné par l'attribut `name` de l'annotation `@Entity`. S'il n'est pas précisé, alors le nom par défaut de l'entité est le nom complet de la classe.
- En SQL, la clause `where` exprime des contraintes sur les colonnes d'une des tables qui est interrogée par la requête. En JPQL elle s'exprime sur les champs de l'entité sur laquelle la sélection a lieu. L'entité `Marin` doit donc ici posséder un champ `nom`. Il est possible de naviguer au travers du graphe d'objets, par exemple `marin.adresse.rue` est une écriture légale en JPQL.
- Notons tout de suite qu'il est possible de sélectionner une entité en relation d'une autre. Par exemple `select marin.adresse from Marin marin ...` retournera une liste d'objets `adresse`.

2.2. Exécution d'une première requête

L'exécution d'une requête passe par la création d'un objet de type `Query` à partir de l' *entity manager* .

Exemple 30. Exécution d'une première requête

```
// Construction d'un entity manager
EntityManager em = ... ;

 // construction d'un objet Query
Query query = em.createQuery(
    "select marin from Marin marin where marin.nom = 'Surcouf'") ;

 // exécution et récupération de la liste résultat
List<Marin> marins = query.getResultList() ;

 // analyse du résultat (classique)
 for (Marin marin : marins) {
    System.out.println(marin.getNom()) ;
}
```

Exécuter une requête JPQL se fait donc en deux temps :

- Définition de l'objet requête proprement dit, de type `Query`.
- Puis exécution de la requête, et lecture du résultat.

Notons que `Query` possède également deux autres méthodes :

- `getSingleResult()` : retourne l'unique objet résultat de cette requête. Si la requête retourne plusieurs objets, alors une exception de type `NonUniqueResultException` est générée.
- `getFirstResult()` : retourne le premier objet résultat de cette requête. Si la requête ne retourne aucun objet, alors cette méthode renvoie `null`.

2.3. Exécution d'une première requête d'agrégation

De même que le SQL, le JPQL définit la notion d'agrégation. La syntaxe est analogue au SQL. Voyons un exemple qui permet d'obtenir le nom de marins dont le nom commence par un 's'.

Exemple 31. Une première requête d'agrégation

```
// construction d'un objet Query
Query query = em.createQuery(
    "select count(marin) from Marin marin where marin.nom like 'S%'") ;

 // exécution et récupération du résultat
 int nombre = (Integer)query.getSingleResult() ;
```

Dans ce cas on peut utiliser sans crainte la méthode `getSingleResult()`.

3. Définition de requêtes

3.1. Requêtes dynamiques

La première façon de créer une requête JPQL consiste à créer une chaîne de caractères qui porte cette requête, et à la passer à la méthode `createQuery()` de l' *entity manager* . Cette approche est parfaitement légale, mais elle pose un problème, et comporte un piège.

Le problème est que la traduction de la requête JPQL en SQL se fait au moment de la création de l'objet `Query`, et à chaque instanciation de cet objet. Si la requête est créée dans une servlet, ou dans un EJB, le coût de traitement devra être payé à chaque fois que l'utilisateur fait appel à cette servlet ou à cet EJB.

Le piège se trouve dans la façon dont on veut prendre en compte les paramètres de la requête. Supposons que l'on ait la méthode suivante.

Exemple 32. Une (mauvaise) requête dynamique paramétrée

```
public List<Marin> getMarinByName(String name) {

    Query query = em.createQuery("select marin from Marin marin " +
                                 "where marin.nom = '" + name +  "'") ;

    List<Marin> marins = query.getResultList() ;

     return marins ;
}
```

Cette requête est censée retourner tous les marins dont le nom est passé en paramètre, et c'est ce qu'elle fait. Cela dit, on peut aussi lui passer en paramètre la chaîne de caractères suivante :

```
Surcouf' and prenom = 'Robert
```

Et dans ce cas, elle effectue une tout autre requête. Cette attaque, connue sous le nom d'injection de code est très classique, et doit être absolument évitée. Pour cela, JPQL expose une fonctionnalité analogue aux `preparedStatement` de JDBC.

3.2. Requêtes paramétrées

Il existe deux moyens de paramétrer une requête. Les deux moyens consistent à placer des éléments particuliers dans la chaîne JPQL, puis de donner une valeur à ces éléments. Dans le premier cas ces éléments sont juste numérotés : `?1`, `?2`, etc... Dans le second cas, ils sont nommés : `:nom`, `:adresse`, etc...

3.2.1. Paramètrage numéroté

Voyons tout d'abord le paramétrage anonyme.

Exemple 33. Paramètrage d'une requête, première méthode

```
Query query = em.createQuery("select marin from Marin marin " +
                             "where marin.nom = ?1 and marin.prenom = ?2") ;
query.setParameter(1,  "Surcouf") ;
query.setParameter(2,  "Robert") ;

List<Marin> marins = query.getResultList() ;
```

3.2.2. Paramétrage nommé

Voici la même requête, avec des paramètres nommés.

Exemple 34. Paramètrage d'une requête, deuxième méthode

```
Query query = em.createQuery("select marin from Marin marin " +
                             "where marin.nom = :nom and marin.prenom = :prenom") ;
query.setParameter("nom",  "Surcouf") ;
query.setParameter("prenom",  "Robert") ;

List<Marin> marins = query.getResultList() ;
```

On prendra garde dans cette deuxième méthode : les noms des paramètres sont préfixés par ":" dans la chaîne JPQL, mais ce caractère n'est pas repris lors des appels à `setParameter()`.

3.2.3. Cas particulier des dates

Comme à l'habitude, les objets `java.util.Date` et `java.util.Calendar` posent problème en tant que paramètre. De sorte qu'ils soient correctement convertis, il faut ajouter un paramètre à l'appel à `setParameter()`, qui porte le type temporel dans lequel la conversion doit se faire.

Exemple 35. Paramètrage d'une requête, cas des dates

```
Query query = em.createQuery("select marin from Marin marin " +
                             "where marin.dateDeNaissance = :ddn") ;
query.setParameter("ddn", dateDeNaissance, TemporalType.DATE) ;

List<Marin> marins = query.getResultList() ;
```

3.3. Requêtes nommées

Dans de nombreux cas, les requêtes utilisées sont connues dès la compilation de notre application. JPQL nous offre alors la possibilité de déclarer ces requêtes en tant que *requêtes nommées* . Ces requêtes nommées sont déclarées dans des annotations, et donc analysées par l'implémentation JPA au chargement de la classe. Elles peuvent donc être converties en code SQL au moment de ce chargement, et ne pas surcharger l'exécution de notre application.

Une requête nommée est déclarée dans une annotation `@NamedQuery`. Cette annotation prend deux attributs : `name`, qui porte le nom de la requête, et `query`, qui porte la requête JPQL. Le nom de la requête doit être unique pour une même unité de persistance. Sans que cela soit une obligation, il est donc pratique de le préfixer par le nom d'une entité.

Plusieurs annotations `@NamedQuery` peuvent être regroupées dans une annotation `@NamedQueries`, qui prend un tableau de `@NamedQuery` en attribut. Ces deux annotations doivent être posées sur la classe d'une entité JPA.

Exemple 36. Déclaration de requêtes nommées

```
@NamedQueries({
   @NamedQuery(
      name="Marin.findAll",
      query="select marin from Marin marin"),
   @NamedQuery(
      name="Marin.findByName",
      query="select marin from Marin marin where marin.name = :name"),
   @NamedQuery(
      name="Marin.countAll",
      query="select count(marin) from Marin marin"),
})
 @Entity(name="Marin")
 public  class Marin  implements Serializable {

         // reste de la classe
}
```

Remarquons plusieurs choses sur cet exemple.

Tout d'abord, nous avons donné un nom explicite à notre entité : `Marin`. Sans cela, il aurait fallu utiliser le nom complet de la classe `Marin` dans nos requêtes JPQL.

Les annotations `@NamedQueries` doivent être posées sur la classe d'une entité. Rien ne dit que ces requêtes doivent nécessairement retourner des instances ou collection d'instances de ces entités. Il est simplement pratique, lisible et habituel de mettre sur une classe les requêtes qui s'y rapportent. On notera d'ailleurs que la dernière de nos requête retournent un entier.

Enfin, et afin de garantir l'unicité du nommage de nos requêtes au niveau de l'unité de persistance, nous avons préfixé le nom de ces requêtes par le nom de l'entité sur laquelle elles sont définies. Il ne s'agit pas d'une obligation, mais d'une bonne façon de faire.

3.4. Requêtes natives

Le JPQL ne permet pas de faire tout ce que l'on peut faire en SQL, pour cela il est possible de définir également des requêtes en SQL natif en JPA. On dispose pour cela de deux méthodes.

- On peut utiliser la méthode `createNativeQuery()` de l' *entity manager* . Cette méthode prend une chaîne de caractères en paramètre : le code SQL natif que l'on veut exécuter.
- On peut également définir des requêtes nommées natives. La syntaxe de déclaration est exactement la même que pour les requêtes JPQL nommées que l'on vient de voir, sauf que les annotations à utiliser sont `@NamedNativeQueries` et `@NamedNativeQuery`.

Cette possibilité offerte présente l'inconvénient d'attacher le code JPA à une base de données particulière. Cela dit, elle permet aussi de migrer des applications *legacy* plus simplement.

4. Exécution, analyse du résultat

4.1. Exécution d'une requête dynamique

Nous avons déjà vu comment exécuter une requête dynamique de sélection sur un exemple. Il suffit d'invoquer l'une des trois méthodes `getResultList()`, `getSingleResult()` ou `getFirstResult()` sur l'objet `query` que l'on a créé sur cette requête.

Dans le cas d'une requête de mise à jour (`update` ou `delete`), c'est la méthode `executeUpdate()`, dont le fonctionnement est le même qu'en JDBC. Cette méthode retourne un entier, qui correspond au nombre d'enregistrements modifiés ou effacés par cette requête.

4.2. Exécution d'une requête nommée

Voyons l'exécution d'une requête nommée. Le principe est légèrement différent, puisque l'objet `Query` n'est pas construit de la même façon.

Exemple 37. Exécution d'une requête nommée

```
// construction d'un entity manager
EntityManager em = ... ;

 // récupération d'une requête nommée par interrogation
 // de l'entity manager
Query marinByName = em.createNamedQuery("Marin.findByName") ;

 // paramétrage de la requête
marinByName.setParameter("name",  "Surcouf") ;

 // exécution de la requête
List<Marin> marins = marinByName.getResultList() ;
```

Le paramètre passé à la méthode `createNamedQuery` est le nom de cette requête, défini par l'attribut `name` de l'annotation `@NamedQuery`. JPA nous impose l'unicité de ce nom au sein d'une unité de persistance donnée, ce qui nous permet de le récupérer sans doublon. En revanche, si l'on passe un nom qui n'existe pas, la méthode nous jettera une exception de type `IllegalArgumentException`.

L'exécution d'une requête nommée suit le même processus, que cette requête soit exprimée en JPQL ou en SQL natif. Dans notre exemple, la requête demandée pourrait parfaitement être une requête native.

4.3. Analyse du résultat

4.3.1. Le résultat est une liste d'entités

Le cas où la requête retourne une liste d'entités JPA a déjà été vu en exemple. Cette liste est stockée dans une liste Java classique, sur laquelle on peut itérer. Tout ce qui a été vu, notamment sur le chargement `LAZY` ou `EAGER` en relation de ces entités s'applique bien sûr aux entités résultat.

4.3.2. Le résultat est une liste d'objets

Considérons l'exemple suivant.

Exemple 38. Écriture d'une requête ne retournant pas des entités

```
Query query = em.createQuery("select marin.nom, marin.prenom from Marin marin") ;
List result = em.getResultList() ;
```

Nous avons volontairement laissé la déclaration `List` sans la typer, car son contenu n'est plus le même que dans le cas où notre requête portait sur une entité JPA.

Chaque élément de cette liste est un tableau. Il y a autant d'éléments tableau qu'il y a de lignes dans le résultat de notre requête SQL.

Examinons à présent le contenu de chacun de ces tableaux.

Pour cette requête, chaque ligne du résultat est composée de deux champs : `nom` et `prenom`, qui sont deux chaînes de caractères. On peut spécifier tous les champs que l'on veut, quel que soit leur type : type de base ou entité.

JPA construit un tableau pour chaque ligne du résultat. Chacun de ces tableaux est composé d'autant d'éléments qu'il y a de champs spécifiés. Chaque élément correspond à un des champs spécifiés, dans le bon ordre.

Nous pouvons donc écrire le code qui va nous permettre d'analyser ce résultat.

Exemple 39. Analyse d'une requête ne retournant pas des entités

```
// on reprend la requête précédente
Query query = em.createQuery("select marin.nom, marin.prenom from Marin marin") ;
List result = em.getResultList() ;

 // le résultat est composée d'autant de lignes qu'il y a
 // de ligne dans le résultat :
 for (Object ligneAsObject : result) {

     // ligne correspond à une des lignes du résultat
    Object[] ligne = (Object[])ligneAsObject ;

     // cette liste est composée de deux éléments : nom et prenom
    String nom = (String)ligne[0] ;
    String prenom = (String)ligne[1] ;

    System.out.println(nom +  " " + prenom) ;
}
```

Cette analyse est un peu complexe techniquement, et surtout très fragile. L'utilisation de cast sur des éléments lus en aveugle sera effectivement pénible à mettre au point. De plus, le moindre changement de modèle risque de faire échouer tantôt la requête, tantôt son analyse, puisque l'on ne manipule que des `Object`, convertis à la volée.

Notons que ce système fonctionne également si l'un des champs spécifiés dans la requête est une entité JPA. Dans ce cas, cette entité se retrouvera dans la liste des éléments de la requête.

4.3.3. Le résultat est un objet construit

Il est enfin possible, en JPQL, de construire des objets résultat directement à partir des requêtes. Continuons avec l'exemple de nos objets marin, et de notre requête qui retourne leurs noms et prénoms. Construisons une classe simple, un bean, capable de stocker chaque ligne du résultat de notre requête.

Exemple 40. Objet résultat : `NameBean`

```
package org.paumard.cours.model;

 public  class NameBean {

     private String nom ;
     private String prenom ;

     public NameBean(String nom, String prenom) {
         this.nom = nom ;
         this.prenom = prenom ;
    }

     // suivent les getters et setters
}
```

Exemple 41. Requête retournant un objet résultat

```
Query query = em.createQuery(
     "select new org.paumard.cours.model.NameBean(p.nom, p.prenom) " +
     "from Personne p") ;
List result = query.getResultList() ;
```

La nouveauté de cette requête est l'appel à la construction de l'objet `NameBean`, en passant au constructeur les paramètres du résultat de la requête. Cet objet résultat doit être désigné par son nom complet (donc avec le nom du package), et, bien sûr, ce constructeur doit exister. Notons que de plus, il ne doit pas y avoir ambiguïté sur ce construteur.

Écrivons le code qui permet d'analyser cette requête.

Exemple 42. Analyse d'une requête retournant un objet résultat

```
// exécution de la requête de l'exemple précédent
List result = query.getResultList() ;

 // itération sur les éléments de cette liste
 for (Object element : result) {

     // chaque élément est de type NameBean
    NameBean nameBean = (NameBean)element ;

    System.out.println(nameBean.getNom() +  " " + nameBean.getPrenom()) ;
}
```

Cette approche fonctionne également si l'un des champs sélectionnés est une entité JPA.

4.4. Cas des résultats de grande taille

Que l'on soit en JDBC ou en JPA, il est toujours coûteux de convertir de grandes quantités de données du domaine SQL vers le domaine Java. La solution généralement adoptée pour analyser des quantités importantes de lignes de résultats consiste à les prendre page par page, de façon à minimiser la charge que représenterait la conversion de tout un paquet en une seule fois.

Pour cela, on dispose de deux méthodes sur l'objet `Query`, qui permettent de sélectionner une page de lignes dans un résultat de grande taille :

- `setFirstResult(int)` : permet de fixer le numéro d'index de la première ligne résultat retournée.
- `setMaxResult(int)` : permet de fixer le nombre de lignes retournées par le résultat.

Une fois ces deux paramètres fixés, on peut appeler la méthode `getResultList()`. Si cette méthode retourne un nombre d'enregistrements moins important que la quantité demandée, c'est que la requête a été épuisée.

4.5. Remarques

L'interaction entre les requêtes et les transactions est un point qui mérite que l'on s'y attarde.

Si un résultat de requête va être exploité en vue de modifier les objets retournés par cette requête, alors cette requête doit être exécutée dans une transaction. Ces objets seront créés dans le contexte de cette transaction, et pourront donc être modifiés ou effacés sans problème.

En revanche, si un résultat de requête ne sera exploité qu'en lecture, par exemple à des fins d'affichage, alors cet attachement des objets résultat à la transaction est une surcharge inutile. Exécuter la requête à l'extérieur de la transaction sera moins coûteux en temps de calcul.

Enfin, que se passe-t-il lorsque l'on modifie des objets, et que dans la même transaction on exécute une requête JPQL dont le résultat comporte ces objets modifiés ?

Pour répondre à cette question, il faut bien distinguer les appels à la méthode `find()` de l' *entity manager* , et les requêtes telles que nous les avons vues dans cette partie.

La méthode `em.find()` demande un objet persistant par sa clé primaire. Si l' *entity manager* possède cet objet dans son cache, alors il le retourne sans faire de requête sur la base. Comme on ne peut pas changer la clé primaire d'un objet, tout se passera bien.

Les requêtes JPQL fonctionnent différemment : elles sont converties en SQL, puis exécutées sur la base de données. On peut être dans une configuration où un objet se trouve toujours en base dans l'état dans lequel il était avant la transaction, et dans le cache de l' *entity manager* , dans un état modifié. La modification de cet état peut faire que sa version en base vérifie les contraintes de la requête, mais pas sa version dans le cache. Pour éviter ce problème, une façon de faire consiste à recopier le cache de l' *entity manager* en base, ce sont cette fois les bons objets qui seront retournés par la requête. Notons que certaines implémentations peuvent aussi avoir des moyens pour exécuter des requêtes SQL à la fois sur le cache de chaque *entity manager* et la base de données. Dans ce cas il n'y a pas à recopier le cache en base avant l'exécution de la requête.

5. Clause From

5.1. Définition des entités

Dans un premier temps, la clause `From` doit donner la liste des entités sur lesquelles la requête doit être exécutée. Dans tous les cas, on doit donner au moins une entité. Une entité peut être désignée par le nom complet de sa classe, ou son nom, qui joue le rôle d'alias. Ce nom est la valeur de l'attribut `name` de l'annotation `@Entity`.

Rappelons enfin deux choses :

- une classe annotée `@MappedSuperClass` n'est pas une entité, on ne peut donc pas faire de requête dessus ;
- il est parfaitement possible de faire une requête sur une classe abstraite.

5.2. Jointures dans la clause `From`

Prenons un modèle composé de trois classes : `Personne`, étendue par `Marin` et `Capitaine`. Ces trois classes sont concrètes, et sont des entités. Une quatrième classe, `Bateau`, porte une relation `passager`, de type `Personne`, annotée `@OneToOne`, et une deuxième, `equipage`, également de type `Passager`, annotée `@OneToMany`

On remarquera que cet exemple porte une relation polymorphique (`passager` peut recevoir des instances de nos trois classes `Passager`, `Marin` et `Capitaine`. On choisira une strategie `JOINED` pour la mise en base.

5.2.1. Jointures naturelles

On peut exprimer une jointure de plusieurs façons dans une requête JPQL, tout comme en SQL. La première façon peut être implicite ou explicite. Supposons que l'on cherche le passager d'un bateau dont le nom est passé en paramètre.

Exemple 43. Une première jointure JPQL implicite

```
select bateau.passager
from Bateau bateau
where bateau.nom = :nom
```

Cette requête JPQL se traduit en EclipseLink par une commande SQL qui comporte une jointure (`left outer join`) entre la table `Bateau` et la table `Personne`. Si l'on n'avait pas ajouter la clause `where`, on aurait obtenu en résultat la liste de toutes les personnes qui sont passagers d'un bateau.

On aurait pu écrire une jointure explicite de la façon suivante :

Exemple 44. Une première jointure JPQL explicite

```
select personne
from Bateau bateau
        join bateau.passager personne
where bateau.nom = :nom
```

Ces deux écritures sont équivalentes du point de vue JPQL, et génèrent le même code SQL dans EclipseLink.

5.2.2. Jointures externes

Comme en SQL, les jointures externes permettent de préserver l'existence des entités du côté gauche de la jointure, quand bien même ces entités n'auraient pas d'entité jointe. Une jointure externe s'exprime de la façon suivante.

Comme exemple, écrivons une requête qui permet d'afficher la liste des bateaux et de leurs passagers.

Exemple 45. Une jointure externe JPQL

```
select bateau
from Bateau bateau
        left join bateau.passager personne
where bateau.nom like :nom
```

Le SQL généré par EclipseLink est le suivant :

Exemple 46. SQL généré par une jointure externe JPQL

```
SELECT t1.ID, t1.NOM, t1.PASSAGER_ID
FROM BATEAU t1
     LEFT OUTER JOIN PERSONNE t0 ON (t0.ID = t1.PASSAGER_ID)
WHERE (t1.NOM LIKE ?)
```

On constate que l'on a bien une jointure externe générée, qui nous permettra d'obtenir toute la liste des bateaux, quand bien même ils n'ont pas de passagers.

Si l'on retire le mot-clé `left` de notre requête JPQL, on obtient le code généré suivant :

Exemple 47. SQL généré sans jointure externe JPQL

```
SELECT t1.ID, t1.NOM, t1.PASSAGER_ID
FROM PERSONNE t0, BATEAU t1
WHERE ((t1.NOM LIKE ?) AND (t0.ID = t1.PASSAGER_ID))
```

Dans ce cas EclipseLink nous génère une jointure interne, et ne sortiront dans notre requête que les bateaux qui ont un passager.

5.2.3. Jointures de type `fetch`

Ces jointures sont propres au JPQL, cette notion n'existe pas en SQL. Elles permettent de charger explicitement la relation d'une entité, même si les objets de cette relation ne font pas partie du résultat de la sélection. Cela permettra de faire l'économie de la lecture de ces objets en relation lors de l'exploitation du résultat.

Si l'on reprend notre exemple JPQL précédent, on constate que seule la clé primaire de notre passager est présente dans la requête. Lors de l'exploitation de cette requête, si l'on en vient à lire ce passager, une demande devra partir sur le cache de l' *entity manager* . Si le cache ne possède pas l'objet, alors il faudra aller le lire en base, ce qui impliquera un aller et retour supplémentaire avec la base.

La jointure `fetch` permet de lire les objets en relation directement, et donc de faire l'économie de cet aller et retour. Modifions notre requête afin de mettre en œuvre cette technique.

Exemple 48. Jointure `fetch`

```
select bateau
from Bateau bateau left join fetch bateau.passager
where bateau.nom like :nom
```

Remarquons tout d'abord deux choses :

- le mot-clé `fetch` se place à droite du mot-clé `join` ;
- on ne peut plus poser d'alias sur le champ à droite du `join`.

Enfin, notons que ces jointures ne sont pas autorisées dans des requêtes imbriquées.

Voici le SQL généré par EclipseLink pour cette requête :

Exemple 49. SQL généré pour une jointure `fetch`

```
SELECT t1.ID, t1.NOM, t1.PASSAGER_ID,
       t0.ID, t0.DTYPE, t0.PRENOM, t0.NOM
FROM BATEAU t1 LEFT OUTER JOIN PERSONNE t0 ON (t0.ID = t1.PASSAGER_ID)
WHERE (t1.NOM LIKE ?)
```

On constate que les champs de `Personne` (`nom` et `prenom`) ont été ajoutés à la requête, ce qui va autoriser la construction de la relation `passager` de chaque instance de `Bateau` dès l'exécution de cette requête.

5.3. Remarque finale sur les jointures en JPQL

On prendra garde que le fait d'écrire une jointure dans une requête JPQL contraint le résultat a posséder un résultat pour cette jointure, quand bien même on n'utilise pas l'entité en relation dans la clause `where`, même dans le cas où cette clause `where` n'existe pas.

Considérons par exemple la requête JPQL suivante.

Exemple 50. Requête jointe sans clause `where`

```
select bateau
from Bateau bateau
     join bateau.passager personne
```

Bien que cette requête ne pose aucune condition sur la relation `passager`, la jointure sera établie dans le SQL généré. En particulier, si aucune instance de `Personne` ne se trouve en base, aucun `bateau` ne sera sélectionné par cette requête.

6. Clause Where

Nous avons déjà écrit plusieurs clauses `where` dans nos exemples, formalisons tout ceci ici.

6.1. Variables et chemins dans une clause `where`

Une clause `where` utilise le plus souvent des variables. Toutes ces variables doivent être définies dans la clause `from`.

On peut définir un chemin à partir d'une variable, qui permet de naviguer dans le modèle objet d'une entité. Ainsi, si une requête définit une variable `bateau`, alors `bateau.equipage` désigne le champ `equipage` de la classe `bateau`.

On ne peut utiliser une relation multivaluée dans une clause `where` que dans deux cas :

- pour tester si la relation associée est vide (`is empty`) ;
- pour tester le cardinal de la relation associée (`size()`).

Tous les autres cas d'utilisation d'une relation multivaluée dans la clause `select` sont illégaux en JPQL.

6.2. Expressions conditionnelles et opérateurs

JPQL supporte les expressions booléennes suivantes : `and`, `or`, `not`.

Les opérateurs suivants sont supportés par JPQL.

- L'opérateur de navigation ' `.`'.
- Les opérateurs arithmétiques : le + et le - unaires, les opérateurs de multiplication (*), division (/), addition (+) et soustraction (-).
- Les opérateurs de comparaison : =, >, >=, <, <=, <>, `[NOT] BETWEEN`, `[NOT] LIKE`, `[NOT] IN`, `IS [NOT] NULL`, `IS [NOT] EMPTY`, `[NOT] MEMBER [OF]`, `[NOT] EXISTS`.
- Les opérateurs logiques : `NOT`, `AND` et `OR`.

Examinons précisément les opérateurs de comparaison, dont le fonctionnement et la syntaxe peuvent être parfois un peu subtils.

6.2.1. **Opérateur** `Between`

Cet opérateur permet de tester si l'argument se trouve entre deux valeurs. Les types supportés sont les nombres, les chaînes de caractères et les dates. On peut toujours se passer de cet opérateur, et utiliser deux expressions reliées par un `AND` à la place.

6.2.2. Opérateur `In`

Cet opérateur permet de tester si la valeur en paramètre se trouve dans une collection. Cette collection peut être une relation multivaluée, le résultat d'une requête imbriquée, ou une collection explicitement écrite.

Exemple 51. Utilisation de l'opérateur `In`

```
select marin
from Marin marin
where marin.nom in ('Surcouf',  'Tabarly')

select marin
from Marin marin
where marin.commune in (select commune
                        from Commune commune join commune.maire maire
                        where maire.nom = :nom)
```

La première requête se comprend assez simplement. La deuxième sélectionne tous les marins dont la commune de naissance a le maire que l'on passe en paramètre. Notons ici que l'on aurait pu écrire cette requête à l'aide d'une jointure, ce qui est souvent le cas avec les requêtes SQL `IN`. Notons que l'utilisation de ces requêtes est souvent moins performante que l'écriture de jointures.

6.2.3. Opérateur `Like`

Cet opérateur permet de comparer des chaînes de caractères à l'aide d'expressions régulières très simples. Ces expressions régulières supportent deux caractères spéciaux : l' *underscore* (_) qui représente un unique caractère quelconque, et le pourcent (%) qui représente toute séquence de caractères quelconques, y compris la séquence vide.

6.2.4. Opérateur `Is Null`

Cet opérateur permet de tester si une valeur est nulle ou pas.

6.2.5. Opérateur `Is Empty`

Cet opérateur prend en paramètre une collection, qui peut être une relation multivaluée, ou une requêtre imbriquée. Il retourne `true` si cette collection est vide.

6.2.6. Opérateur `Member Of`

Cet opérateur permet de tester si un élément fait partie d'une collection ou pas. Cette collection ne peut être qu'une relation multivaluée.

6.2.7. Opérateur `Exists`

Cet opérateur permet de tester si une requête imbriquée retourne au moins une valeur. Si cette requête ne retourne aucune valeur, alors l'évaluation est `false`.

6.3. Requêtes imbriquées

JPQL supporte les requêtes imbriquées dans les clauses `where` et `having`. Voyons un exemple de sous-requête.

Exemple 52. Requête imbriquée

```
select bateau
from Bateau bateau
where (select count(*)
       from bateau.equipage) >  10
```

Dans ce cas on ne sélectionne que les bateaux qui ont plus de 10 membres d'équipage. Notons que l'on aurait pu écrire la même requête sans faire de requête imbriquée, ce qui est souvent le cas.

6.4. Opérateurs any, all et some

Ces trois opérateurs s'utilisent conjointement avec les requêtes imbriquées. Ils permettent de comparer un élément donné avec l'ensemble des éléments de la collection retournée par la requête devant laquelle ils se placent. Voyons ceci sur un exemple d'utilisation de l'opérateur `some`. Les deux autres opérateurs s'utilisent exactement suivant la même syntaxe.

L'opérateur `all` signifie que la comparaison doit être vraie pour tous les éléments de la liste. L'opérateur `some` signifie qu'elle est vraie pour une partie de ces éléments. Elle est fausse si le résultat de la sous-requête est vide. L'opérateur `any` est synonyme de l'opérateur `some`.

Exemple 53. Opérateur `some`

```
select capitaine
from Bateau bateau join bateau.capitaine capitaine
where bateau.capitaine = capitaine and
      bateau.equipage is not empty and
      capitaine.salaire < some (select marin.salaire from bateau.equipage marin)
```

La requête précédente sélectionne les capitaines dont le salaire est inférieur à au moins un des salaires des marins de l'équipage du bateau qu'ils commandent. On remarquera plusieurs choses sur cette requête :

- L'entité `bateau` tient la relation unidirectionnelle avec `capitaine`. Donc la jointure doit s'exprimer sous la forme `Bateau bateau join bateau.capitaine`.
- On ne sélectionne que les bateaux dont l'équipage n'est pas vide par `bateau.equipage is not empty`.
- On utilise dans la requête imbriquée, le `bateau` courant défini dans la requête principale.
- L'écriture `bateau.equipage marin` permet de désigner un élément générique de la collection `bateau.equipage`. Il n'est pas utilisé ici dans la clause `where`, mais cela serait parfaitement possible.

6.5. Expressions fonctionnelles

6.5.1. Fonctions arithmétiques

JPQL offre toutes les fonctions arithmétiques classiques. Il supporte en plus les fonctions suivantes.

- `ABS(number)` : retourne la valeur absolue du nombre passé en paramètre.
- `SQRT(number)` : retourne la racine carrée du nombre passé en paramètre.
- `MOD(integer, integer)` : retourne le reste de la division entière du premier entier passé en paramètre par le second.
- `SIZE(collection)` : retourne le cardinal de la collection passée en paramètre. La collection doit être une relation multivaluée.
- `INDEX(collection)` : retourne la position de l'objet courant dans la collection passée en paramètre. Cette collection doit être un alias dans une requête JPQL.

6.5.2. Fonctions opérant sur les chaînes de caractères

Il offre également les fonctions suivantes, qui opèrent sur les chaînes de caractères :

- `CONCAT(string, string, ...` : réalise la concaténation des chaînes de caractères passées en paramètres.
- `SUBSTRING(string, begin, length` : extrait une sous-chaîne d'une chaîne donnée. Le paramètre `begin` désigne le premier caractère à sélectionner, et `length` le nombre de ces caractères. S'il n'est pas présent, alors la sous-chaîne s'arrête à la fin de `string`.
- `TRIM(string)` : supprime les blancs en fin de chaîne de caractères. Cette fonction peut prendre deux arguments supplémentaires. Le premier peut prendre la valeur `leading`, `trailing` ou `both`. Il indique quel côté de la chaîne doit voir ses espaces supprimés, éventuellement les deux. Le second indique le caractère qui doit être supprimé, plutôt que les espaces. Exemple : `TRIM(LEADING 'a' FROM nom)` retourne une chaîne dont on a supprimé les a en début.
- `LOWER(string)` et `UPPER(string)` : retournent une chaîne de caractères résultat de la mise en minuscule ou en majuscule (respectivement) de la chaîne passée en paramètres.
- `LENGTH(string)` : retourne le nombre de caractères dans la chaîne passée en paramètres.
- `LOCATE(toLocate, inString, index)` : retourne la position de la chaîne `toLocate` dans la chaîne `inString`. La recherche commence à l'index `index`, qui est un argument optionnel.

6.5.3. Fonctions opérant sur les dates

Trois fonctions sont disponibles sur les dates : `CURRENT_DATE`, `CURRENT_TIME` et `CURRENT_TIME_STAMP`. Ces trois fonctions retournent la date courante du serveur, dans chacun des trois formats.

6.5.4. Fonction opérant sur une entité

Il existe enfin une fonction qui permet de tester le type réel d'une entité : `TYPE`. Cette fonction s'utilise de la façon suivante.

Exemple 54. Fonction TYPE

```
select personne
from Personne personne
where TYPE(personne) IN (Marin, Capitaine)
```

Dans notre exemple, on sélectionne toutes les personnes qui sont marin ou capitaine, à l'exclusion des cuisiniers et des mousses.

7. Clauses Group By et Having

La clause group by a le même rôle qu'en SQL : elle permet de regrouper les résultats d'une sélection en paquets, sur certaines propriétés. Sur chacun de ces paquets, on peut appliquer une ou plusieurs fonctions d'agrégations sur les propriétés restantes.

La clause having permet ensuite de sélectionner les résultats des agrégations, en fonction de leurs valeurs.

Écrivons un exemple d'une telle requête.

Exemple 55. Agrégation JPQL

```
select bateau.nom, AVG(marin.salaire), COUNT(marin)
from Bateau bateau join bateau.equipage marin
having count(marin) >  10
```

La requête précédente sélectionne les noms de bateaux, le nombre de marins qui ont embarqué dessus, et la moyenne de leurs salaires. On ne s'intéresse qu'aux bateaux qui ont plus de 10 membres d'équipage.

Les fonctions d'agrégation supportées par JPQL sont les suivantes :

- AVG : calcule la valeur moyenne ;
- COUNT : retourne le nombre de valeurs ;
- MAX et MIN : retourne la plus grande et la plus petite des valeurs ;
- SUM : retourne la somme des valeurs.