Persistance JPA

Entity

Les Entity JPA permette de décrire un objet correspondant à une table.

Exemple d'une entity simple :

```
@Entity
@Table(name="MyTable")
public class Truc {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    Long id;
    @Column(name="my_name")
        String myName;
    @Column(name="myBidule")
        String bidule;

…
}
```

Le @Entity signale à Spring qu'il s'agit (justement) d'une Entity.
@Table et @Column sont inutile si le nom de la table est le même que celui de la classe et si le nom de la colonne de la Table est le même que celui du champ de la classe.

@Id est évidemment obligatoire (c'est la clé primaire).
@GeneratedValue indique ici que l'id est automatique

Pour MySql (utiliser l'auto-increment) :

```
@Id
@GeneratedValue(
         strategy= GenerationType.AUTO,
         generator="native"
)
@GenericGenerator(
         name = "native",
         strategy = "native"
)
```

Exemple ManyToOne (ou OneToOne)

```
@Entity
@Table(name="Glagla")
public class Chose {
        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        Long id;
        @Column(name="ChoseName")
        String name;
        @Column(name="ChoseColor")
        String color;
        @ManyToOne
        @JoinColumn(name="ID_TRUC")
        Truc truc;

…
```

Ici @JoinColumn indique quelle colonne dans la table contient l'id de Truc permettant de faire la jointure.

Exemple ManyToMany

```
@Entity
@Table(name="machin")
public class Machin {
        @Id
```

```
        @GeneratedValue(strategy=GenerationType.AUTO)
        Long id;
        @Column(name="LOCALNAME")
        String name;
        @ManyToMany(fetch = FetchType.EAGER,cascade= {CascadeType.PERSIST,CascadeType.MERGE})
        @JoinTable(name="machin_truc",joinColumns=@JoinColumn(name="MACHIN_ID"),inverseJoinColumns=@Joi
nColumn(name="TRUC_ID"))
        Set<Image> trucs = new HashSet<>();
```

Remarque : ici on récupère la liste des Truc dans un Set<Truc> et pas dans une List<Truc>. Une bonne pratique en JPA est d'utiliser des Set<T> plutôt que des List<T>.D'ailleurs ne pas le faire peut entraîner des problèmes dans certains cas.

Le @ManyToMany suppose qu'on passe par une table d'indirection. C'est ce qu'indique le @JoinTable, sachant que le joinColums indique quelle colonne correspond à l'id de Machin et inverseJoinColums, l'id de Truc. A noter qu'on peut avoir une relation bi-directionnelle en déclarant un @ManyToMany dans Truc (mais la syntaxe est différente : voir doc sur internet).

Ce qu'il faut bien voir, c'est que l'implémentation JPA (Hibernate) fait en sorte que par défaut, les élements provenant d'une table d'indirection (d'un @ManyToMany, donc) ne sont pas chargé en mémoire lors d'un findAll() par exemple. C'est ce qu'on appelle la lazyness, et c'est le comportement par défaut. Ici, on a indiqué `fetch = FetchType.EAGER,` ce qui signifie qu'on force le chargement de ces données (mais ce n'est pas la bonne politique en général : si on ne fait pas attention, on se retrouve avec toute la base en mémoire).

Attention au CascadeType : Le CascadeType.PERSIST signifie que lorsqu'on sauvegarde un Machin, les Truc (dans le Set) vont être sauvegardés aussi. Donc, si on a récupéré des Truc (via un findById() par ex.) pour les mettre dans le Set, le save() ne marchera PAS (puisque les Truc sont déjà sauvegardés). Pour que ça marche il faut que les Truc soit « neufs » et donc créés en même temps que le Machin. Si on veut utiliser des Truc existants, la solution est d'abord de sauvegarder un Machin sans Truc, puis d'ajouter des Truc dans le Machin et d'updater.
Ou alors on déclare CascadeType.REFRESH à la place de CascadeType.PERSIST (mais alors, on ne peut plus utiliser de Truc « neufs »)
Il faut donc voir quel type de CascadeType on emploie à la conception de la base.

Repository

Le Repository est l'équivalent fonctionnel du DAO. Toute la beauté de Spring, c'est qu'en dérivant son Repository de CrudRepository, il n'y a rien à faire, toutes les fonctions de base sont directement implémentées : Spring va créer un objet en mémoire à partir de l'interface qu'on vient de déclarer.

```
public interface TrucRepository extends CrudRepository<Truc, Long> {
}
```

Par défaut le TrucRepository fourni les fonctions suivantes :

- deleteXX
- findXX
- save : cette dernière fait create ou update selon que l'objet existe ou pas dans la BDD.

Attention, le findById() renvoie un Optional<T> et pas T lui-même. Pour obtenir T, il faut faire findById().get().

Le CrudRepository doit être paramétrisé : CrudRepository<T,K> avec :

- T : Class de l'Entity associée

- K : Type de la clé primaire de l'entity

On peut aussi déclarer des findByXXX automatique sur les champs de l'Entity.
On peut aussi ajouter du Distinct et/ou du And/Or

```
public interface TrucRepository extends CrudRepository<Truc, Long> {
	List<Truc> findByMyName(String name);
	List<Truc> findDictinctByBidule(String bidule);
	List<Truc> findByBiduleOrMyName(String bidule,String name) ;
}
```

Ou implémenter directement des requêtes en pseudoSQL (ici, on recherche tous les Truc dont le myName commence par ce qu'on passe en paramètre).

```
public interface TrucRepository extends CrudRepository<Truc, Long> {
	List<Truc> findByMyName(String name);

	@Query("SELECT t FROM Truc t WHERE t.myName LIKE :toto%")
	List<Truc> findByBegin (@Param("toto") String name);
}
```

Si l'on emploie un dérivé de CrudManager, inutile de déclarer @Repository, c'est fait automatiquement. Par contre, si on fait son Repository à la main, il faut mettre le @Repository.

A noter que pour faire des trucs plus fins, on peut passer par l'EntityManager, mais c'est nettement plus complexe.

Exemple :

```
	@PersistenceContext
	private EntityManager entityManager;

	public List<Specie> list() {
		CriteriaBuilder builder = entityManager.getCriteriaBuilder();
		CriteriaQuery<Specie> query = builder.createQuery(Specie.class);
		Root<Specie> r = query.from(Specie.class);
		Predicate p = builder.or(builder.like(r.get("latinName"),
		  "L%"),builder.like(r.get("commonName"), "L%"));
		query.where(p);
		return entityManager.createQuery(query).getResultList();
	}
```

Ici, on recherche dans Specie toutes les valeurs qui commencent par « L », que ce soit sur le commonName ou le latinName.

Configuration

Il faut décrire la connexion à la BDD dans un fichier de configuration, par défaut, *application.properties*. Ce fichier est à mettre dans *src/main/resources*.

Voici un exemple :

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/bestioles
spring.datasource.username=root
spring.datasource.password=root
```

Le *spring.jpa.hibernate.ddl-auto=update* indique que si des Entity sont modifiées, ces modifications seront reportées sur la base (si on rajoute une colonne par exemple). Cela vient de ce que la BDD peut être créée/modifiée à partir des schémas des Entity. Ce qui n'est pas toujours ce que l'on veut.

Donc, il est préférable de mettre ce paramètre à *none*. Par contre il est préférable de toujours positionner ce paramètre (à *none*) sans quoi, dans certains cas, Spring n'est pas prévenu que hibernate/JPA est utilisé.

Transaction

Supposons un Service qui utilise 3 Repository. Soit la fonction :

```
public void process(Entity1 e1, Entity2 e2, Entity3 e3) {
        repo1.save(e1) ;
        repo2.save(e2) ;
        repo3.save(e3) ;
}
```

On voit que si la deuxième ligne rate, on se retrouve avec des données parasites dans la table associée à e1, et il faudrait enlever ces données pour que la table soit dans un état normal.

Pour que cette fonction soit secure, il faudrait donc catcher les exceptions et s'il y a exception, remettre tout dans l'état initial (dans l'exemple : faire un repo1.delete(e1)).
Comme ce serait très laborieux et en plus différent suivant les cas, il existe une notion qui est celle de *Transaction*. Une *Transaction* peut être vue comme une opération globale qui contient des opérations élémentaires (ici les save()). S'il y a une erreur, on appelle la fonction rollback() de la *Transaction* qui remet la BDD dans son état initial. Si tout va bien, on appelle la fonction commit() de la *Transaction* et l'opération est validée (la base est effectivement modifiée).

On voit que c'est tout de suite beaucoup plus simple. Mais JPA permet de faire les choses encore plus simples. Grâce à l'annotation @Transactional, on peut associer directement une *Transaction* à une méthode.

```
@Transactional
public void process(Entity1 e1, Entity2 e2, Entity3 e3) {
```

Dans ce cas, une *Transaction* est associée à la fonction et automatiquement un rollback() sera effectué en cas d'erreur et un commit() si tout va bien.
Il n'y a rien de plus à faire.

Ceci étant dit, ainsi qu'on l'a vu, normalement, @Transactional est plutôt associé dans sa logique aux service. Quel rapport avec les Repository ?

Eh bien, il y a un rapport, mais qui sur le fond n'a rien à voir avec les *Transaction*.
On a vu que les Entity qui possèdent des Set<T> peuvent avoir un fetch LAZY ou EAGER sur ces Set<T>.
Et on a vu que EAGER est plutôt à proscrire. L'ennui, c'est qu'en LAZY, les Set<T> ne pointent sur rien du tout et provoquent des erreurs quand on essaye de les utiliser. Par exemple, comment faire des tests unitaires qui testeraient des Set<T> en LAZY ?

La seule manière de charger les Set<T> en LAZY est de créer ce qu'on appelle une *Session*, c.a.d en gros, un espace mémoire où l'on va stocker ces Set<T>. Et c'est justement ce que fait le @Transactional : il crée une *Session* temporaire pour la méthode annotée.

Si par exemple on utilise un *CommandLineRunner* pour créer/gérer des Entity, il suffit d'annoter le run(), soit :

```
@Transactional
public void run(String... args) throws Exception {
```

Ce qu'il faut retenir, c'est qu'ici le @Transactional ne sert pas vraiment à faire une *Transaction*, mais à créer une *Session* temporaire quand les Entity sont en mode LAZY.