Spring Data JPA

Dans une application backend, on a (presque) toujours besoin de persister des données dans un système de gestion de base de donnée. On pourrait écrire des requêtes JDBC pour tout mais c'est du code super long et répétitif. Pour pallier ce problème, il existe JPA (Java Persistance API), une spécification d'ORM pour Java. Un ORM (Object Relationnal Mapper) est un composant logiciel qui va traduire automatiquement des instances d'objets du modèle orienté objet, en ligne d'enregistrement dans le modèle relationnel. JPA étant une spécification, nous avons besoin d'une implémentation, nous allons utiliser la plus connue qui s'appelle Hibernate.

Installer les dépendances

Pour installer JPA et Hibernate, rien de plus simple, il suffit d'ajouter la dépendance suivante à notre fichier `pom.xml` :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Il nous faut aussi un Driver JDBC, car ce dernier sera utilisé par Hibernate sous le capot.

Selon votre SGBD :

MySQL

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.22</version>
</dependency>
```

Postgres

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
</dependency>
```

Oracle

```
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>21.1.0.0</version>
</dependency>
```

Configurer la connexion

Afin de configurer la connexion à la base de donnée, ouvrez le fichier `application.properties` situé dans `src/main/resources`. Et ajoutez les entrées suivantes :

```
spring.datasource.url=ici l'url de ma base
spring.datasource.username=ici le login de ma base
spring.datasource.password=ici le mot de passe de ma base

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.temp.use_jdbc_metadata_defaults=false
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Les Entités

Les entités sont les objets qui sont persistés par l'ORM dans la base de donnée. Pour définir nos classes d'objets qui seront enregistrées dans la base de donnée, il faut les définir en tant qu'entité. Cela se fait à l'aide de l'annotation `@Entity` sur la classe. Une entité doit avoir un champs qui correspondra à se clé primaire dans la base de donnée, il est désigné avec l'annotation `@Id`. l'ORM peut le générer automatiquement, et même aléatoirement dans le cas d'une UUID (voir exemple).

Les autres champs de la classe entités sont persistés automatiquement. Pour qu'un champs ne soit pas persisté, il faut l'annoter avec `@Transient`.

Exemple d'entité pour notre Todo :

```
@Entity
public class Todo {
  @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

  private String title;
    private String description;

  ... getters & setters ...
}
```

Les annotations `@GeneratedValue` et `GenericGenerator` permet de générer automatiquement un UUID aléatoire pour l'entité.

On peut aussi définir des contraintes sur les colonnes en rajoutant l'annotation `@Column` sur le champs et en lui passant des paramètres, par exemple :

```
@Column(unique = true, length = 32)
```

Cela permet de mettre la contrainte "unique" sur le champs et d'imposer une longueur maximum de 32 caractère

Les Repository

Pour interagir avec les entités, il faut créer des Repository. Ce sont des interface que nous allons définir, mais qui seront implémentées non pas nous, mais pas l'ORM Hibernate. Ils peuvent ensuite être injectés dans les services par le conteneur d'injection de dépendances. Une interface Repository gère les interactions pour une entité et doit étendre l'interface `JpaRepository` en fournissant en paramètre de type, le type de l'entité, ainsi que le type de son Id.

Exemple :

```
public interface TodoRepository extends JpaRepository<Todo, String> {


}
```

Méthodes de base

Voilà, juste en étendant cette interface, on peut accèder à tout un tas de méthodes intéressantes. Les principales sont :

- `save` : Sauvegarder (créer ou mettre à jours) une instance d'une entités
- `findById` : récupérer une entité à partir de son Id
- `findAll` : récupérer toutes les entités contenues dans la base (⚠ attention ça peut faire beaucoup de tout charger dans la mémoire)
- `deletebyId` : supprimer une entité à partir de son Id

Et bien d'autre qui peuvent être utile, à retrouver dans la javadoc de l'interface

Méthodes Custom

On peut également définir de nouvelles méthodes dans l'interface pour filtrer sur d'autre champs, en nommant ces méthode de façon particulière :

```
findByTitle(String title);
```

`findBy` veut dire qu'on filtre sur un champs et `Title` est le nom d'un champs de notre entité, cette requête va donc récupérer les Todos qui ont le titre passé en paramètre.

Voici une références des mots utilisables dans les noms de méthode

Mot	Exemple	Logique SQL correspondante
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname,findByFirstnameIs,findByFirstnameEquals	… where x.firstname = ?1
Between	findByStartDateBetween	… where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	… where x.age < ?1

Mot	Exemple	Logique SQL correspondante
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection ages)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)

Relations

Les relations permettent de faire des liens entre les entités. Il existe quatres types de relations, définies par des annotations :

- `@ManyToOne` : plusieurs instances de cette classe entité sont en relation avec une unique instance d'une autre (ex: Pages d'un livre - plusieurs pages sont reliées à un unique livre)
- `OneToMany` : une unique instance de cette classe entité est en relations avec plusieurs instances d'une autre (ex: Livre qui contient des pages - un unique livre est relié à plusieurs pages)

- `@ManyToMany` : plusieurs instance de cette classe entité sont en relations avec plusieurs isntance d'une autre (ex: Classes et Professeurs - Les professeurs ont plusieurs classes et les classes ont plusieurs professeurs.
- `@OneToOne` : une unique instance de cette classe entités est reliée à une unique instance d'une autre (ex: Dircteur et Ecole - une directeur dirige une seule école et une école est dirrigée par un seul directeur).

Exemple :

```
@Entity
public class Todo {
  ... autres propriétés ...

  @ManyToOne
  private ApplicationUser owner;

  ... getters & setters ....
}
```

```
@Entity
public class ApplicationUser {
    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

  private String name;

  @OneToMany
    private Set<Todo> todos;

    ... getters & setters ...
}
```

Les annotations de relation possèdes plusieurs paramètres utiles à connaitres :

- `fetch` :
  - `FetchType.EAGER` : charge les données de la relation directement (défaut pour `@ManyToOne` et `@OneToOne`)
  - `FtechType.LAZY` : charge les données de la relation que quand le getter est appelé (défaut pour `@OneToMany` et `@ManyToMany`
- `orphanRemoval` : `true` ou `false`, détermine dans un `@OneToMany` si les enfant doivent être supprimés quand le parent est supprimé → un enfant ne peut exister sans parent.

Revision #12
Created 2021-01-22 09:47:23 UTC by Arsène Lapostolet
Updated 2021-05-23 12:09:24 UTC by Arsène Lapostolet