SPRING BOOT – TP N°06

REALISATION D'UNE API

OBJECTIF

- Ajoutez le module Spring Data JPA
- Ajoutez le driver MySQL
- Relier l'application à une base de données
- Réaliser une API complète pour extraire, ajouter, modifier, supprimer des villes.

TACHES

Etape 1 :

Installation du module Spring Data JPA et du Driver MySQL

Ajoutez les 2 dépendances suivantes à votre pom.xml :

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>

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

Etape 2 :

- Créer une base de données **recensement** en utf8_general_ci
- Compléter le fichier **application.properties** pour accéder à votre base de données
- N'oubliez pas d'ajouter la librairie d'accès à votre base de données dans votre fichier pom.xml

```
spring.datasource.url=jdbc:mysql://localhost:3306/mabase
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Eventuellement, ajoutez la propriété suivante pour droper et créer la base de données à chaque démarrage :

```
spring.jpa.hibernate.ddl-auto=create-drop
```

Etape 3 :

- Transformer votre classe **Ville** en entité JPA. Rappels :
  - Positionnez l'annotation @Entity sur la classe
  - Ajoutez un identifiant avec l'annotation @Id
  - Si vous souhaitez que l'identifiant de la ville soit auto-incrémentée utilisez l'annotation @GeneratedValue avec la stratégie IDENTITY
  - La classe doit posséder le constructeur sans paramètre

Etape 4 :

- Créez une classe VilleService qui possède les méthodes suivantes :
  - List<Ville> extractVilles() : extrait et retourne les villes qui sont en base.
  - Ville extractVille(int idVille) : extrait la ville dont l'id est passé en paramètre.
  - Ville extractVille(String nom) : extrait la ville dont le nom est passé en paramètre.
  - List<Ville> insertVille(Ville ville) : insère une nouvelle ville en base et retourne la liste des villes après insertion
  - List<Ville> modifierVille(int idVille, Ville villeModifiee) : modifie la ville dont l'identifiant est passé en paramètre. Les nouvelles données sont portées par l'instance villeModifiee. La méthode retourne la liste des villes après modification.
  - List<Ville> supprimerVille(int idVille) : supprime la ville dont 'id est passé en paramètre et retourne la liste des villes après suppression.

Etape 5 :

- Modifiez votre classe VilleControleur avec les 6 méthodes suivantes :
  - 1 méthode GET qui retourne la liste des villes
  - 1 méthode GET qui retourne une ville en fonction de son id
  - 1 méthode GET qui retourne une ville en fonction de son nom
  - 1 méthode PUT qui prend une ville en paramètre et la met en base de données
  - 1 méthode POST qui prend une ville en paramètre et permet de modifier les données qui sont en base
  - 1 méthode DELETE qui permet de supprimer une ville en fonction de son id.

Etape 6 :

- Vérification du fonctionnement de votre API avec POSTMAN

FACULTATIFS

Pour ceux qui ont terminé le TP en avance, nous allons complexifier un peu le modèle en ajoutant la notion de département.

Etape 7 :

- Nous allons ajouter maintenant la classe (qui est aussi une entité JPA) Departement.
  - Un département peut posséder plusieurs villes
  - Une ville n'appartient qu'à un seul département.

Etape 8 :

- Ajouter un contrôleur permettant de réaliser les 4 opérations de CRUD sur les départements
- Ajouter également les méthodes suivantes :
    - Lister les n plus grandes villes d'un département
    - Lister les villes ayant une population comprise entre un min et un max et qui appartiennent à un département donné.

Commitez et poussez votre code sur GitHub.