Spring Boot : Spring Data Jpa

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

1. Rappel
2. Configuration
3. Entité
4. Repository

Spring Boot

Rappel

- **JPA** : spécification (comme une interface en POO)
- **Hibernate**, **EclipseLink** ... : implémentation de **JPA**

Spring Boot

Spring Data

- Projet **Spring Framework**
- Simplifiant l'interaction avec différents systèmes de stockage de données
- Composé de plusieurs sous-projets
  - **Spring Data JPA**
  - **Spring Data REST**
  - **Spring Data MongoDB**
  - ...

Spring Boot

Spring Data JPA

- Sous-projet **Spring Data**
- **Spring Data JPA** = **Spring Data** + **JPA**
- Offrant plus d'abstraction d'accès aux données
- Fonctionnant avec un fournisseur implémentant **JPA**
  - **Hibernate**
  - **EclipseLink**
  - ...

Spring Boot

Modèle : accès et traitement de données

- Utilisation de
  - **MySQL** (ou **H2**) pour le stockage de données
  - **Hibernate** pour le mapping
  - **Spring-Data-JPA** pour la génération du **DAO**
- Précision de données de connexion dans `application.properties`

Spring Boot

Organisation du projet

- Créons un premier package `com.example.demo.model` où nous placerons les entités **JPA**
- Créons un deuxième package `com.example.demo.dao` où nous placerons les classes **DAO** (ou ce qu'on appelle `Repository` dans **Spring**)

Pour ajouter les dépendances **MySQL** (ou **H2**) et **Spring-Data-JPA**

- Faire clic droit sur le projet et aller dans `Spring > Edit Starters`
- Cocher les cases respectives de `MySQL Driver` (ou `H2 Database`) et `Spring Data JPA`

Pour ajouter les dépendances **MySQL** (ou **H2**) et **Spring-Data-JPA**

- Faire clic droit sur le projet et aller dans `Spring > Edit Starters`
- Cocher les cases respectives de `MySQL Driver` (ou `H2 Database`) et `Spring Data JPA`

Ou ajouter les dépendances suivantes

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

Pour ajouter les dépendances MySQL (ou H2) et Spring-Data-JPA

- Faire clic droit sur le projet et aller dans `Spring > Edit Starters`
- Cocher les cases respectives de `MySQL Driver` (ou `H2 Database`) et `Spring Data JPA`

Ou ajouter les dépendances suivantes

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

Ou pour le case de H2

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

Spring Boot

Dans `application.properties`, on ajoute les données concernant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url = jdbc:mysql://localhost:3306/spring_data?createDatabaseIfNotExist=true
%spring.datasource.driver-class-name=com.mysql.cj.jdbc.driver
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Spring Boot

Dans `application.properties`, on ajoute les données concernant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url = jdbc:mysql://localhost:3306/spring_data?createDatabaseIfNotExist=true
%spring.datasource.driver-class-name=com.mysql.cj.jdbc.driver
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Explication

- Les propriétés ajoutées remplacent les beans utilisés par **Spring MVC**.
- La chaîne `characterEncoding=UTF-8&useUnicode=yes` permet d'ajouter les caractères accentués dans la base de données.
- La chaîne `createDatabaseIfNotExist=true` permet de créer la base de données si elle n'existe pas.
- La propriété `spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl` permet de forcer **Hibernate** à utiliser les mêmes noms pour les tables et les colonnes que les entités et les attributs.

Spring Boot

Pour le cas d'une base de données `H2`

```
spring.datasource.url=jdbc:h2:mem:spring_data
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.hibernate.ddl-auto = create
spring.jpa.show-sql = true

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Spring Boot

Pour le cas d'une base de données `H2`

```
spring.datasource.url=jdbc:h2:mem:spring_data
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.hibernate.ddl-auto = create
spring.jpa.show-sql = true

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```


Utiliser H2 en mode mémoire ou fichier

- **Mémoire** : La base de données **H2** sera réinitialisée à chaque démarrage de l'application. Exemple d'**URL JDBC** : `jdbc:h2:mem:spring_data`.
- **Fichier** : Pour une base de données persistante, vous pouvez configurer **H2** pour écrire dans un fichier. Exemple d'**URL JDBC** : `jdbc:h2:file:./data/spring_data`.

Spring Boot

Pour accéder à la console **H2**

```
http://localhost:8080/h2-console
```

Créons une entité `Personne`

```
package com.example.demo.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Personne {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long num;
  private String nom;
  private String prenom;

  public Personne() {  }
  public Personne(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }
  // + getters, setters et toString
}
```

Spring Boot

Pour obtenir le **DAO**, il faut créer une interface qui étend

- soit `CrudRepository` : fournit les méthodes principales pour le **CRUD** (**Spring Data**).
- soit `PagingAndSortingRepository` : hérite de `CrudRepository` et fournit en plus des méthodes de pagination et de tri sur les enregistrements (**Spring Data**).
- soit `JpaRepository` : hérite de `PagingAndSortingRepository` en plus de certaines autres méthodes **JPA** (**Spring Data JPA**).

Spring Boot

Le repository

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository <
   Personne, Long> {
}
```

`Long` est le type de la clé primaire (`Id`) de la table (entité) `Personne`.

Spring Boot

Gardons la vue `addPersonne.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ajouter une nouvelle personne</title>
  </head>
<body>
  <h2>Ajouter une nouvelle personne</h2>
  <form method="POST" action="addPersonne">
        <div>
                Nom : <input type="text" name="nom">
        </div>
        <div>
                Prénom : <input type="text" name="prenom">
        </div>
        <button>Ajouter</button>
  </form>
</body>
</html>
```

Mettons à jour le contrôleur `PersonneController`

```
@Controller
public class PersonneController {

        @Autowired
        private PersonneRepository personneRepository;

        @GetMapping("addPersonne")
        public String addPersonne() {
                return "addPersonne";
        }

        @PostMapping("addPersonne")
        public String addPersonne(@RequestParam String nom,
                                  @RequestParam String prenom,
                                  Model model) {
                Personne personne = new Personne(nom, prenom);
                System.out.println(personneRepository.save(personne));
                model.addAttribute("nom", nom);
                model.addAttribute("prenom", prenom);
                return "confirm";
        }
}
```

Spring Boot

Pour récupérer la liste de toutes les personnes

```
@GetMapping("showPersonnes")
public String showPersonnes(Model model) {
    model.addAttribute("personnes", personneRepository.findAll());
    return "showPersonnes";
}
```

Spring Boot

La vue `showPersonnes.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="jakarta.tags.core"%>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Liste de personnes</title>
  </head>
  <body>
    <h1>Liste de personnes</h1>
    <c:forEach items="${ personnes }" var="personne">
      <div>
        <c:out value="${ personne.prenom } ${ personne.nom }"/>
      </div>
    </c:forEach>
  </body>
</html>
```

Spring Boot

Exercice

- Dans `showPersonnes.jsp`, ajouter deux liens `deletePersonne/id` et `editPersonne/id` devant chaque personne affichée
- En cliquant sur `deletePersonne/id`, la personne avec l'identifiant `id` sera supprimée et `showPersonnes.jsp` sera rafraichie.
- En cliquant sur `editPersonne/id`, les valeurs (nom et prénom) de la personne avec l'identifiant `id` seront affichées dans des `input` dans la vue `editPersonne.jsp` avec un bouton `Enregistrer`.
- En cliquant sur le bouton `Enregistrer`, les nouvelles valeurs seront enregistrées et la page `showPersonnes.jsp` sera affichée.
- Dans `showPersonnes.jsp`, ajouter un lien qui permet d'afficher la vue `addPersonne.jsp`

Spring Boot

On peut aussi récupérer la liste de personnes par page

```
@GetMapping(value = "/showAllByPage/{page}/{size}")
public  ModelAndView showAllByPage(@PathVariable int page,
   @PathVariable int size) {
  Page<Personne> personnes = personneRepository.findAll(PageRequest.of(
     page, size));
  ModelAndView mv = new ModelAndView();
  mv.setViewName("showPersonnes");
  mv.addObject("personnes", personnes.getContent());
  return mv;
}
```


Les variables de chemin `page` et `size`

- `page` : numéro de la page (première page d'indice 0)
- `size` : nombre de personnes par page

Spring Boot

Considérons le contenu suivant de la table `Personne`

Personne		
num	nom	prenom
1	Durand	Philippe
2	Leberre	Bernard
3	Benammar	Pierre
4	Hadad	Karim
5	Wick	John

Spring Boot

Considérons le contenu suivant de la table `Personne`

Personne		
num	nom	prenom
1	Durand	Philippe
2	Leberre	Bernard
3	Benammar	Pierre
4	Hadad	Karim
5	Wick	John

En allant sur l'URL `localhost:8080/showAllByPage/1/2`**, le résultat est**

Personne		
num	nom	prenom
3	Benammar	Pierre
4	Hadad	Karim

Spring Boot

On peut aussi récupérer une liste triée de personnes

```
@GetMapping(value = "/showAllSorted")
public  ModelAndView showAllSorted() {
  List<Personne> personnes = personneRepository.findAll(
     Sort.by("nom").descending());
  ModelAndView mv = new ModelAndView();
  mv.setViewName("showPersonnes");
  mv.addObject("personnes", personnes);
  return mv;
}
```

Explication

Le résultat a été trié selon la colonne `nom` dans l'ordre décroissant

Spring Boot

Les méthodes personnalisées

On peut aussi définir nos propres méthodes personnalisées dans le repository et sans les implémenter.

Spring Boot

Le repository

```
package com.example.demo.dao;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository <Personne,
   Long> {
  List<Personne> findByNom(String nom);
}
```

Spring Boot

Le repository

```
package com.example.demo.dao;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository <Personne,
   Long> {
  List<Personne> findByNom(String nom);
}
```


La méthode de recherche doit

- commencer par `findBy`
- respecter le **Camel Case**

Spring Boot

Code à ajouter dans `PersonneController`

```
@GetMapping("/findByNom")
public String findByNom() {
        return "findByNom";
}

@PostMapping("/findByNom")
public String findByNom(@RequestParam String nom, Model model)
   {
        var personnes = personneRepository.findByNom(nom);
        model.addAttribute("personnes", personnes);
        return "showPersonnes";
}
```

Spring Boot

La vue `findByNom.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Chercher une personne selon le nom</title>
</head>
<body>
        <h1>Chercher une personne selon le nom</h1>
        <form method="POST" action="findByNom">
                Nom : <input type="text" name=nom>
                <button>Chercher</button>
        </form>
</body>
</html>
```

Spring Boot

Ajoutons une méthode qui vérifie si la colonne `nom` contient le motif recherché

```
package com.example.demo.dao;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository <
   Personne, Long> {
     List<Personne> findByNom(String nom);
     List<Personne> findByNomContaining(String nom);
}
```

Spring Boot

Code à ajouter dans `PersonneController`

```
@GetMapping("/findByNom")
public String findByNom() {
        return "findByNom";
}

@PostMapping("/findByNom")
public String findByNom(@RequestParam String nom, Model model) {
        var personnes = personneRepository.findByNomContaining(nom);
        model.addAttribute("personnes", personnes);
        return "showPersonnes";
}
```

Spring Boot

Code à ajouter dans `PersonneController`

```
@GetMapping("/findByNom")
public String findByNom() {
        return "findByNom";
}

@PostMapping("/findByNom")
public String findByNom(@RequestParam String nom, Model model) {
        var personnes = personneRepository.findByNomContaining(nom);
        model.addAttribute("personnes", personnes);
        return "showPersonnes";
}
```

Aucune modification pour la vue.

Spring Boot

Dans la méthode précédente on a utilisé le mot-clé `Containing`

Mais, on peut aussi utiliser

- `Or`, `Between`, `Like`, `IsNull`...
- `StartingWith`, `EndingWith`, `Containing`, `IgnoreCase`
- `After`, `Before` pour les dates
- `OrderBy`, `Not`, `In`, `NotIn`
- Liste complète : `https://docs.spring.io/spring-data/jpa/docs/current/reference/html/`

Spring Boot

Exemple 2

```
package com.example.demo.dao;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository <Personne,
   Long> {
     List<Personne> findByNom(String nom);
     List<Personne> findByNomContaining(String nom);
     List<Personne> findByNomContainingAndPrenomContaining(String nom,
        String prenom);
}
```

Spring Boot

Code à ajouter dans `PersonneController`

```
@GetMapping("/findByNomAndPrenom")
public String findByNomAndPrenom() {
        return "findByNomAndPrenom";
}

@PostMapping("/findByNomAndPrenom")
public String findByNomAndPrenom(@RequestParam String nom,
   @RequestParam String prenom, Model model) {
        var personnes = personneRepository.
           findByNomContainingAndPrenomContaining(nom, prenom);
        model.addAttribute("personnes", personnes);
        return "showPersonnes";
}
```

Spring Boot

La vue `findByNomAndPrenom.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Chercher selon nom et prénom</title>
</head>
<body>
        <h1>Chercher une personne selon nom et prénom</h1>
        <form method="POST" action="findByNomAndPrenom">
                <div>
                        Nom : <input type="text" name="nom">
                </div>
                <div>
                        Prénom : <input type="text" name="prenom">
                </div>
                <button>Chercher</button>
        </form>
</body>
</html>
```

Spring Boot

Exercice

- Créer une vue `findPersonne.jsp` contenant une seule zone de saisie (motif) et un bouton de recherche.
- La vue `findPersonne.jsp` permettra à l'utilisateur de saisir une chaîne qu'on doit vérifier sa présence dans les colonnes nom ou prénom dans la base de données.
- Les personnes correspondantes seront affichées dans la vue `showPersonnes.jsp`.
- N'oublions pas de définir une nouvelle méthode de recherche dans `PersonneRepository`.

Spring Boot

On peut utiliser l'annotation `Query` pour exécuter des requêtes JPQL (Java Persistence Query Language)

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import java.util.List;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {

    @Query("select p from Personne p where p.nom = ?1")
    List<Personne> chercherSelonLeNom(String nom);
    List<Personne> findByNom(String nom);
    List<Personne> findByNomContaining(String nom);
    List<Personne> findByNomContainingAndPrenomContaining(String nom,
       String prenom);
}
```

Spring Boot

On peut utiliser l'annotation `Query` pour exécuter des requêtes JPQL (Java Persistence Query Language)

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import java.util.List;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {

    @Query("select p from Personne p where p.nom = ?1")
    List<Personne> chercherSelonLeNom(String nom);
    List<Personne> findByNom(String nom);
    List<Personne> findByNomContaining(String nom);
    List<Personne> findByNomContainingAndPrenomContaining(String nom,
       String prenom);
}
```

On peut utiliser l'annotation `Param` pour nommer les paramètres

```
package com.example.demo.dao;

import java.util.List;

import com.example.demo.model.Personne;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {
     @Query("select p from Personne p where p.nom = ?1")
     List<Personne> chercherSelonNom(String nom);
     @Query("select p from Personne p where p.prenom = :prenom and p.nom
         = :nom")
     List<Personne> chercherSelonNomEtPrenom(@Param("nom") String nom,
        @Param("prenom") String prenom);
     List<Personne> findByNom(String nom);
     List<Personne> findByNomContaining(String nom);
     List<Personne> findByNomContainingAndPrenomContaining(String nom,
        String prenom);
}
```

On peut utiliser l'annotation `Query` aussi pour exécuter des expressions SpEL (Spring Expression Language)

```
package com.example.demo.dao;

import java.util.List;

import com.example.demo.model.Personne;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {
    @Query("SELECT p FROM #{#entityName} p WHERE p.nom = ?1")
    List<Personne> chercherSelonNom(String nom);
    @Query("select p from Personne p where p.prenom = :prenom and p.nom
        = :nom")
    List<Personne> chercherSelonNomEtPrenom(@Param("nom") String nom,
      @Param("prenom") String prenom);
    List<Personne> findByNom(String nom);
    List<Personne> findByNomContaining(String nom);
    List<Personne> findByNomContainingAndPrenomContaining(String nom,
      String prenom);
}
```

Spring Boot

Pour activer les transactions sur les autres méthodes, on peut annoter la méthode par `@Transactional`

```
package com.example.demo.dao;

import java.util.List;

import com.example.demo.model.Personne;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.transaction.annotation.Transactional;

public interface PersonneRepository extends JpaRepository<Personne, Long> {
    @Query("SELECT p FROM #{#entityName} p WHERE p.nom = ?1")
    List<Personne> chercherSelonNom(String nom);
    @Query("select p from Personne p where p.prenom = :prenom and p.nom = :nom")
    List<Personne> chercherSelonNomEtPrenom(@Param("nom") String nom,@Param("prenom") String
      prenom);
    List<Personne> findByNom(String nom);
    List<Personne> findByNomContaining(String nom);
    @Transactional
    List<Personne> findByNomContainingAndPrenomContaining(String nom, String prenom);
}
```

Spring Boot

On peut également annoter le `Repository` **par** `@Transactional`

```
package com.example.demo.dao;

import java.util.List;

import com.example.demo.model.Personne;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.transaction.annotation.Transactional;

@Transactional
public interface PersonneRepository extends JpaRepository<Personne, Long> {
    @Query("SELECT p FROM #{#entityName} p WHERE p.nom = ?1")
    List<Personne> chercherSelonNom(String nom);
    @Query("select p from Personne p where p.prenom = :prenom and p.nom = :nom")
    List<Personne> chercherSelonNomEtPrenom(@Param("nom") String nom,@Param("prenom") String
       prenom);
    List<Personne> findByNom(String nom);
    List<Personne> findByNomContaining(String nom);
    List<Personne> findByNomContainingAndPrenomContaining(String nom, String prenom);
}
```