

# RAPPORT TP2

## SPRING DATA JPA

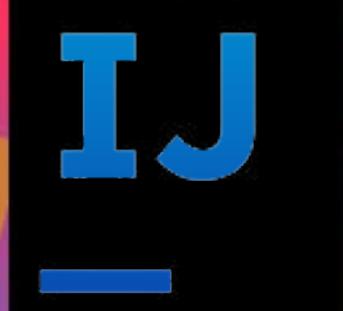
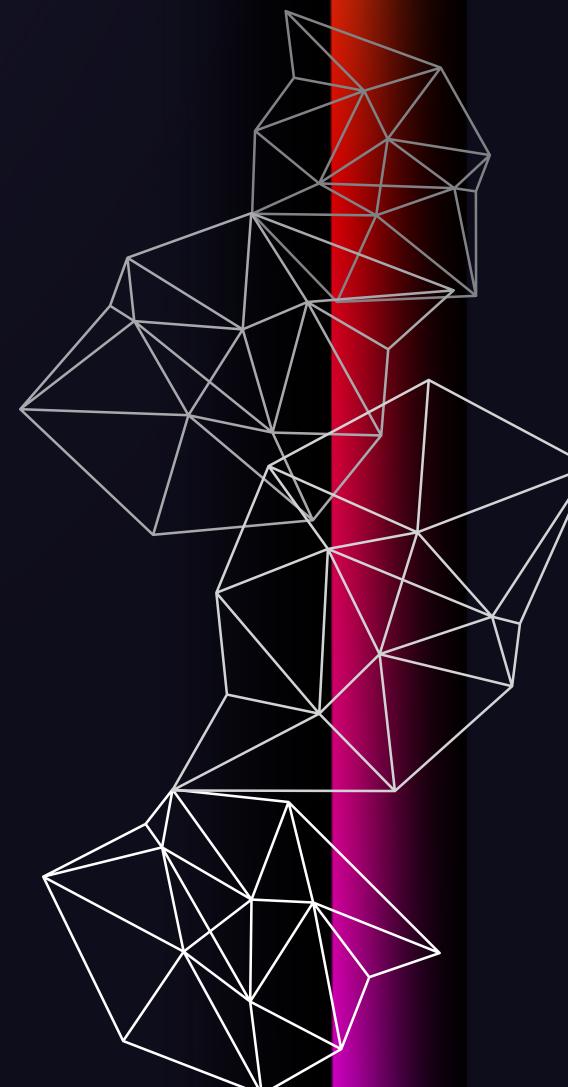
## HIBERNATE

Réalisé par :

Bouchama Hajar

Demandé par :

Mohamed Youssfi



## Introduction

Le présent rapport expose le travail réalisé dans le cadre de l'activité pratique numéro deux du module Systèmes Distribués. L'objectif principal de cette activité était de mettre en pratique les concepts et les technologies liés à Spring Data JPA et Hibernate, en implémentant un système de gestion de patients et en étendant cette application pour inclure d'autres entités telles que les médecins, les rendez-vous, les consultations, ainsi que la gestion des utilisateurs et des rôles.

Le rapport est structuré de manière à présenter d'abord les étapes de mise en place du projet initial, puis les différentes fonctionnalités implémentées. Dans un premier temps, j'ai utilisé IntelliJ IDEA Ultimate pour créer un projet Spring, en y incluant les dépendances nécessaires telles que JPA, H2, Spring Web et Lombok. Ensuite, j'ai défini une entité JPA Patient avec les attributs requis, configuré l'unité de persistance dans le fichier de configuration application.properties, et créé un repository JPA pour la gestion des patients.

Dans la deuxième partie de ce rapport, j'ai étendu l'application pour inclure d'autres entités telles que les médecins, les rendez-vous, les consultations, ainsi que la gestion des utilisateurs et des rôles. Chaque entité a été implémentée avec ses propres attributs et relations, et des opérations de gestion appropriées ont été définies pour chaque entité.

Finalement, j'ai migré la base de données de H2 vers MySQL pour une meilleure gestion des données à grande échelle. Ce rapport met en lumière les défis rencontrés, les solutions apportées et les apprentissages tirés de cette expérience pratique.

## Enoncé !

1. Installer IntelliJ Ultimate
2. Créer un projet Spring Initializer avec les dépendances JPA, H2, Spring Web et Lombok
3. Créer l'entité JPA Patient ayant les attributs :
  - id de type Long
  - nom de type String
  - dateNaissance de type Date
  - malade de type boolean
  - score de type int
4. Configurer l'unité de persistance dans le fichier application.properties
5. Créer l'interface JPA Repository basée sur Spring data
6. Tester quelques opérations de gestion de patients :
  - Ajouter des patients
  - Consulter tous les patients
  - Consulter un patient
  - Chercher des patients
  - Mettre à jour un patient
  - Supprimer un patient
7. Migrer de H2 Database vers MySQL
8. Reprendre les exemples du Patient, Médecin, rendez-vous, consultation, users et roles

## Code et explications :

### 1. Installer IntelliJ Ultimate

L'installation d'IntelliJ Ultimate a été une étape cruciale pour ce projet, fournissant un environnement de développement riche en fonctionnalités et une expérience utilisateur fluide, j'ai envoyé une demande à l'entreprise JetBrains pour me donner l'accès à utiliser une version étudiants en utilisant mon email académique.

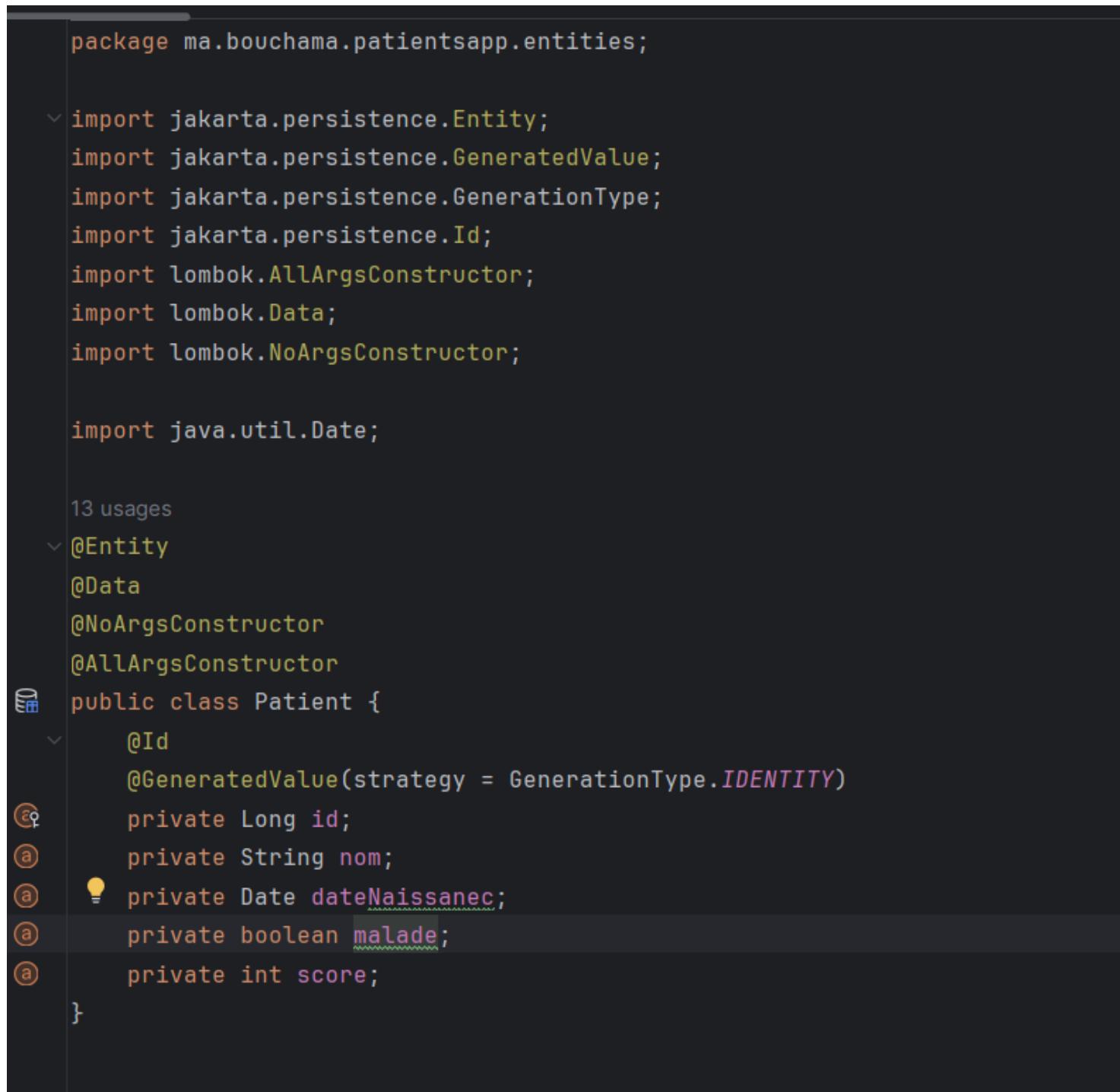
### 2. Créer un projet Spring Initializer avec les dépendances JPA, H2, Spring Web et Lombok

En utilisant Spring Initializer, j'ai pu créer rapidement un projet Spring avec les dépendances essentielles (JPA, H2, Spring Web et Lombok) pour notre application, ce qui m'a permis de me concentrer davantage sur le développement fonctionnel.

### 3. Créer l'entité JPA Patient ayant les attributs :

En utilisant les annotations JPA telles que @Entity, @Id et @GeneratedValue, j'ai pu définir une entité persistante qui sera stockée dans notre base de données. Les attributs de l'entité ont été soigneusement choisis pour capturer toutes les informations essentielles sur les patients.

- l'annotation **@Entity** est utilisée pour marquer la classe Patient comme une entité persistante, ce qui signifie que chaque objet Patient sera persisté dans la base de données.
- l'attribut id est annoté avec **@Id** pour indiquer qu'il est l'identifiant unique de chaque patient
- l'annotation **@GeneratedValue(strategy = GenerationType.IDENTITY)** est utilisée, ce qui signifie que les valeurs de la clé primaire seront générées par la base de données. La stratégie IDENTITY est spécifique à certaines bases de données, où la base de données attribue automatiquement une valeur à la clé primaire lors de l'insertion d'une nouvelle ligne dans la table.



```
package ma.bouchama.patientsapp.entities;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;

13 usages
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Patient {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private Date dateNaissance;
    private boolean malade;
    private int score;
}
```

## 4. Configurer l'unité de persistance dans le fichier application.properties

La configuration de l'unité de persistance dans le fichier application.properties a été nécessaire pour établir la connexion à la base de données. Cela a permis une gestion efficace des entités JPA et des opérations de base de données.

```
spring.application.name=Patients-app
server.port=8085
spring.datasource.url=jdbc:h2:mem:patients-db
spring.h2.console.enabled=true
```

## 5. Créer l'interface JPA Repository basée sur Spring data

Dans notre classe principale PatientsAppApplication, annotée avec @SpringBootApplication, nous avons utilisé l'injection de dépendances avec @Autowired pour injecter le PatientRepository. Cette approche nous a permis d'accéder facilement aux méthodes prédéfinies pour interagir avec la base de données.

- L'annotation **@SpringBootApplication** est une annotation de commodité qui combine trois annotations couramment utilisées dans les applications Spring Boot : **@Configuration**, **@EnableAutoConfiguration** et **@ComponentScan**. Elle facilite le démarrage rapide et la configuration automatique des applications Spring Boot
- **@Autowired** est utilisé pour simplifier le câblage des dépendances et améliore la modularité du code en permettant l'inversion de contrôle (IoC) et l'injection de dépendances (DI) dans Spring..

En implémentant l'interface CommandLineRunner, nous avons pu exécuter du code au démarrage de l'application. Dans la méthode run, nous avons effectué plusieurs opérations de gestion de patients

```
package ma.bouchama.patientsapp;

import ma.bouchama.patientsapp.entities.Patient;
import ma.bouchama.patientsapp.repository.PatientRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.Date;
import java.util.List;

@SpringBootApplication
public class PatientsAppApplication implements CommandLineRunner {
    @Autowired
    private PatientRepository patientRepository;

    public static void main(String[] args) { SpringApplication.run(PatientsAppApplication.class, args); }
}
```

## 6. Tester quelques opérations de gestion de patients :

### - Ajouter des patients

```
public static void main(String[] args) { SpringApplication.run(PatientsAppApplication.class, args);

@Override
public void run(String... args) throws Exception {
    patientRepository.save(new Patient( id: null, nom: "Anis",new Date(), malade: false, score: 10));
    patientRepository.save(new Patient( id: null, nom: "Rachid",new Date(), malade: true, score: 50));
    patientRepository.save(new Patient( id: null, nom: "Khadija",new Date(), malade: true, score: 49));
    patientRepository.save(new Patient( id: null, nom: "Amal",new Date(), malade: false, score: 11));
    patientRepository.save(new Patient( id: null, nom: "Nadir",new Date(), malade: false, score: 13));
```

### - Consulter tous les patients

```
// Consulter tous les patients
List<Patient> patients = patientRepository.findAll();
patients.forEach(p->{
    System.out.println(p.toString());
});

System.out.println("*****");
```

### - Consulter un patient

```
System.out.println("*****");

// Consulter un patient par son ID
Patient p = patientRepository.findById(Long.valueOf( 1)).get();
Patient p2 = patientRepository.findById(Long.valueOf( 2)).get();
System.out.println(p.getId());
System.out.println(p.getNom());
System.out.println(p.getDateNaissance());
System.out.println(p.getDateNaissance());
System.out.println(p.getScore());

System.out.println("*****");
```

### - Mettre à jour un patient

```
// Mettre à jour un patient
Patient patientUpdate = patientRepository.findById(Long.valueOf( 3)).get();
patientUpdate.setMalade(true);
patientUpdate.setScore(80);
patientRepository.save(patientUpdate);
System.out.println("Patient mis à jour avec succès : " + patientUpdate);
```

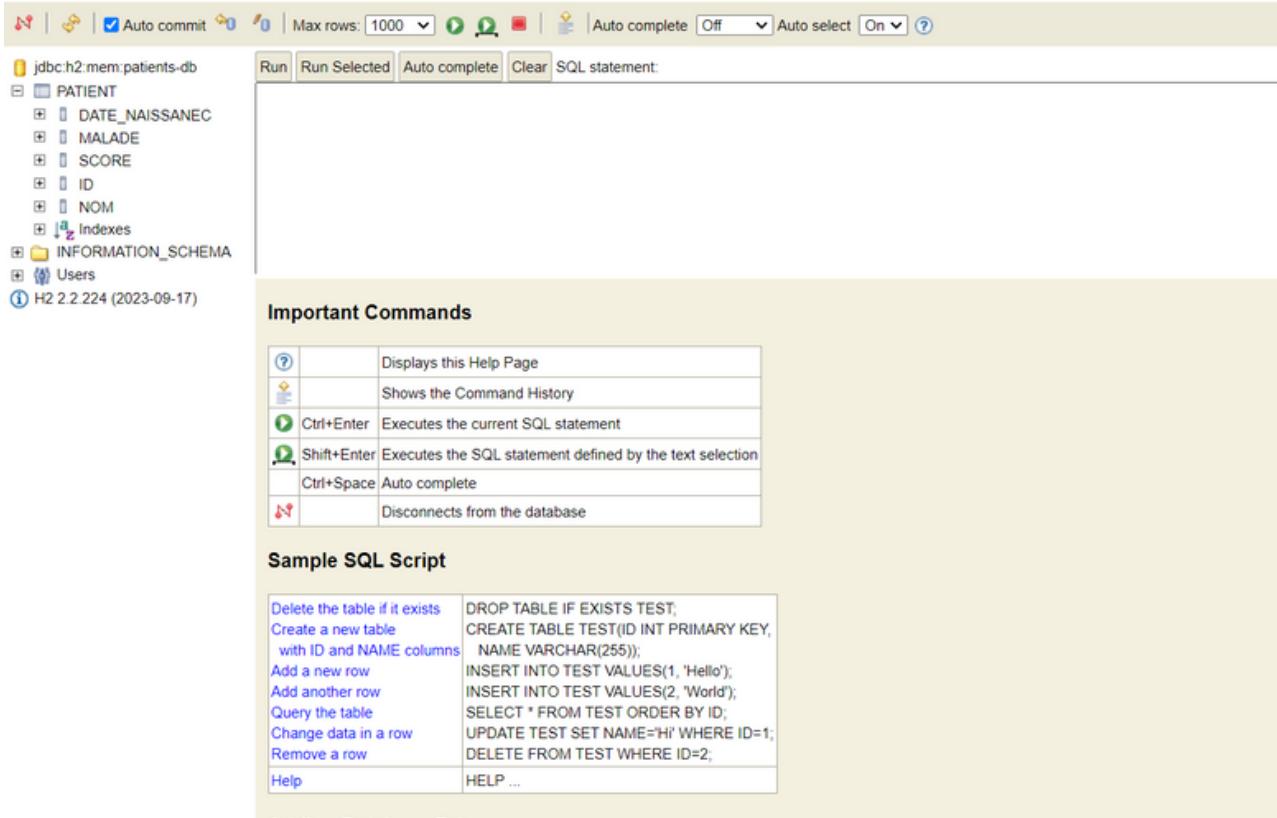
## - supprimer un patient

```
// Supprimer un patient par son ID  
patientRepository.deleteById(2L);  
System.out.println("Patient avec l'ID 2 supprimé.");
```

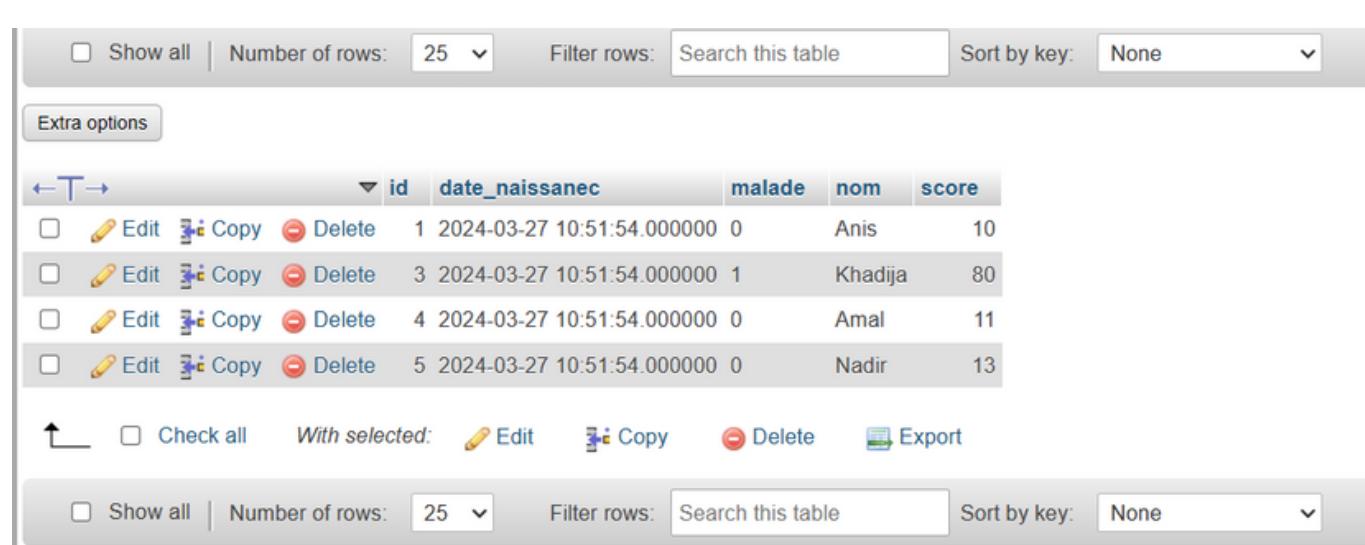
```
// Vérifier les patients après la suppression  
List<Patient> remainingPatients = patientRepository.findAll();  
remainingPatients.forEach(pa -> {  
    System.out.println(pa.toString());  
});
```

## 7. Migrer de H2 Database vers MySQL

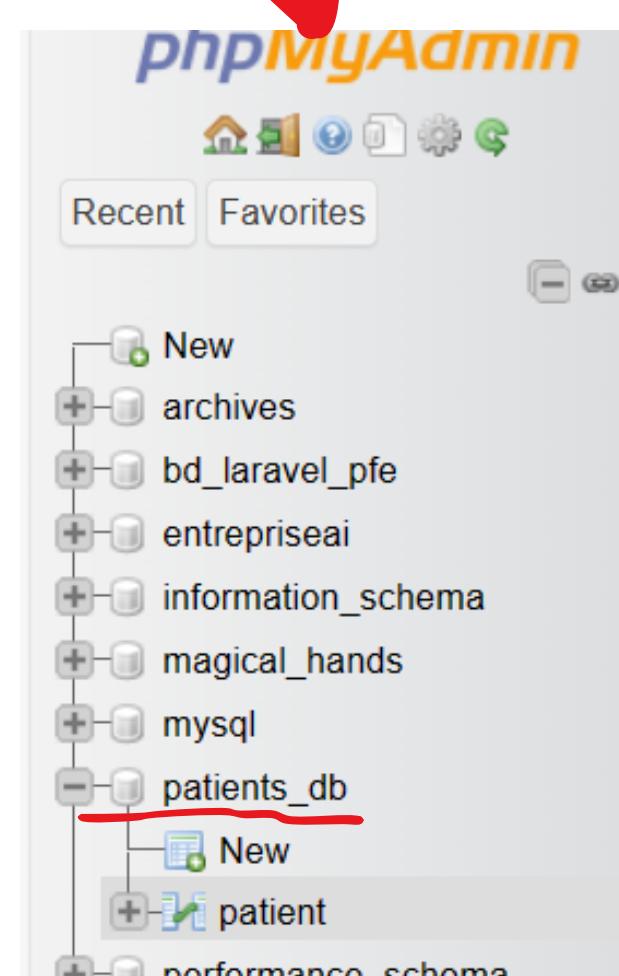
```
spring.application.name=Patients-app  
server.port=8085  
  
spring.datasource.url=jdbc:mysql://localhost:3306/patients_db?createDatabaseIfNotExist=true  
spring.datasource.username=root  
spring.datasource.password=  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
```



The screenshot shows the H2 Database browser interface. On the left, the database structure is displayed with the PATIENT schema expanded, showing columns like DATE\_NAISSANCE, MALADE, SCORE, ID, and NOM. Below the schema tree, there's a section titled "Important Commands" with various keyboard shortcuts for database operations. A large red arrow points from this interface down towards the phpMyAdmin interface.



The screenshot shows the H2 Database browser interface displaying the PATIENT table. The table has columns: id, date\_naissance, malade, nom, and score. There are five rows of data: Anis (id 1), Khadija (id 3), Amal (id 4), and Nadir (id 5). At the bottom, there are buttons for "Check all", "With selected:", "Edit", "Copy", "Delete", and "Export".



The screenshot shows the phpMyAdmin interface. On the left, the database structure is shown with the patients\_db database selected. Inside patients\_db, there are tables named "archives", "bd\_laravel\_pfe", "entrepriseai", "information\_schema", "magical\_hands", "mysql", and "patient". The "patient" table is highlighted with a red underline. A red arrow points from the H2 browser interface up towards this phpMyAdmin interface.

## 8. Reprendre les exemples du Patient, Médecin, rendez-vous, consultation, users et roles

### le premier projet :

#### La classe Patient

La classe **Patient** représente les informations essentielles sur les patients de notre application hospitalière. Elle est annotée avec `@Entity` pour la persistance en base de données et utilise `@Id` et `@GeneratedValue` pour la gestion de l'identifiant. L'utilisation de Lombok pour la génération automatique des getters et setters réduit la redondance du code.

- l'annotation `@Temporal(TemporalType.DATE)`, qui permet de spécifier le type de données temporelles stockées dans la base de données.
- La relation `@OneToMany` avec la classe `RendezVous` dans l'entité `Patient` permet de représenter le fait qu'un patient peut avoir plusieurs rendez-vous associés à lui. En d'autres termes, un patient peut être lié à plusieurs instances de la classe `RendezVous`

```
1 package ma.bouchama.hospital.entities;
2
3 > import ...
4
5
6 17 usages
7 @Entity
8 @Data
9 @NoArgsConstructor
10 @AllArgsConstructor
11
12 public class Patient {
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     private Long id;
16     private String nom;
17     @Temporal(TemporalType.DATE)
18     private Date dateNaissance;
19     private boolean malade;
20     @OneToMany(mappedBy = "patient", fetch = FetchType.LAZY)
21     private Collection<RendezVous> rendezVous;
22
23 }
24
```

#### la classe RendezVous

la classe **RendezVous** modélise un rendez-vous dans notre système hospitalier. Elle est annotée avec `@Entity` pour être persistée en base de données. Les annotations Lombok `@Data`, `@NoArgsConstructor` et `@AllArgsConstructor` sont utilisées pour générer automatiquement les méthodes getters, setters, un constructeur sans paramètre et un constructeur avec tous les paramètres.

la classe contient plusieur attribut parmis lesquels on trouve l'attribut status et l'annotation `@Enumerated(EnumType.STRING)` qui indique que son type est une énumération `StatusRDV` et que sa valeur sera stockée sous forme de chaîne de caractères dans la base de données.

```

1 package ma.bouchama.hospital.entities;
2
3 import jakarta.persistence.*;
4 import lombok.AllArgsConstructor;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7 import java.util.Date;
8
9 15 usages
10 @Entity
11 @Data
12 @NoArgsConstructor
13 @AllArgsConstructor
14
15 public class RendezVous {
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Long id;
19     private Date date;
20     @Enumerated(EnumType.STRING)
21     private StatusRDV status;
22     @ManyToOne
23     private Patient patient;
24     @ManyToOne
25     private Medecin medecin;
26     @OneToOne(mappedBy = "rendezVous")
27     private Consultation consultation;
28 }

```

- Les attributs patient et medecin sont annotés avec **@ManyToOne**, indiquant qu'un rendez-vous est associé à un seul patient et à un seul médecin, respectivement.
- l'attribut consultation est annoté avec **@OneToOne(mappedBy = "rendezVous")**, ce qui signifie qu'un rendez-vous est associé à une seule consultation. Cela permet de représenter le lien entre un rendez-vous et la consultation qui en découle.

## La classe Medecin

classe **Medecin** représente un médecin dans notre système hospitalier.

```

Consultation.java      PatientRepository.java      MedecinRepository.java      Rende
import java.util.Collection;

13 usages
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Medecin {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String email;
    private String specialite;
    @OneToMany(mappedBy = "medecin", fetch = FetchType.LAZY)
    private Collection<RendezVous> rendezVous;
}

```

## La classe Consultation

La classe **Consultation** représente une consultation médicale dans notre système hospitalier.

```
package ma.bouchama.hospital.entities;
import jakarta.persistence.*;
import lombok.AllArgsConstructorConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;

11 usages
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Consultation {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date dateConsultation;
    private String rapport;
    @OneToOne
    private RendezVous rendezVous;
```

enum **StatusRDV** représente les différents états qu'un rendez-vous peut avoir dans notre système hospitalier. Elle est utilisée pour indiquer si un rendez-vous est en attente (PENDING), annulé (CANCELED) ou terminé (DONE).

```
package ma.bouchama.hospital.entities;

2 usages
public enum StatusRDV {
    1 usage
    PENDING,
    no usages
    CANCELED,
   💡 no usages
    DONE
}
```

## les interfaces

### l'interface ConsultationRepository

```
package ma.bouchama.hospital.repositories;

> import ...;

4 usages
public interface ConsultationRepository extends JpaRepository<Consultation, Long> {
}
```

### l'interface ConsultationRepository

```
package ma.bouchama.hospital.repositories;

import ma.bouchama.hospital.entities.Patient;
import org.springframework.data.jpa.repository.JpaRepository;

5 usages
public interface PatientRepository extends JpaRepository<Patient, Long> {
    1 usage
    Patient findByNom(String name);
}
```

### l'interface ConsultationRepository

```
package ma.bouchama.hospital.repositories;

> import ...;

5 usages
public interface MedecinRepository extends JpaRepository<Medecin, Long> {
    1 usage
    Medecin findByNom(String name);
}
```

La méthode **findByNom(String name)** déclare une méthode personnalisée qui permet de rechercher un patient par son nom dans la base de données. Cette méthode sera automatiquement implémentée par Spring Data JPA en fonction du nom de la méthode, ce qui simplifie la recherche de patients par leur nom dans notre application.

### l'interface ConsultationRepository

```
1 package ma.bouchama.hospital.repositories;
2
3 > import ...;
4
5 5 usages
6
7 public interface RendezVousRepository extends JpaRepository<RendezVous, Long> {
8 }
9
```

## La classe HospitalApplication

La classe **HospitalApplication** est la classe principale de notre application hospitalière.

Dans cette classe, nous avons défini une méthode main qui démarre l'application Spring Boot.

La méthode **start** est annotée avec `@Bean` et implémente l'interface `CommandLineRunner`. Cette méthode est exécutée au démarrage de l'application. À l'intérieur de cette méthode, nous initialisons quelques données de test pour les patients et les médecins. Nous utilisons également les repositories correspondants (`PatientRepository` et `MedecinRepository`) pour sauvegarder les données dans la base de données.

nous créons un rendez-vous (`RendezVous`) entre un patient et un médecin, en utilisant les données précédemment créées. Nous définissons la date du rendez-vous, son statut (PENDING dans cet exemple), et les associations au patient et au médecin concernés."

```
package ma.bouchama.hospital;

> import ...;

@SpringBootApplication
public class HospitalApplication {

>     public static void main(String[] args) { SpringApplication.run(HospitalApplication.class, args); }

    @Bean
    CommandLineRunner start(PatientRepository patientRepository,
                           MedecinRepository medecinRepository,
                           RendezVousRepository rendezVousRepository,
                           ConsultationRepository consultationRepository){

        return args -> {

            return args -> {
                Stream.of( ...values: "Mohammed", "Farid", "Khalid")
                    .forEach(name->{
                        Patient patient=new Patient();
                        patient.setNom(name);
                        patient.setDateNaissance(new Date());
                        patient.setMalade(false);
                        patientRepository.save(patient);

                    });

                Stream.of( ...values: "Hajar", "Kaotar", "hind")
                    .forEach(name->{
                        Medecin medecin=new Medecin();
                        medecin.setNom(name);
                        medecin.setSpecialite(Math.random()>0.5?"Cardio":"Dentiste");
                        medecin.setEmail(name+"@gmail.com");
                        medecinRepository.save(medecin);

                    });

            };
        };
    }
}
```

## le package service

pour améliorer notre code nous allons créer un package Service qui est généralement la couche métier car dans nos applications on a deux couches : la couche DAO et la couche métier

L'interface **IHospitalService** définit les méthodes qui représentent la couche métier de notre application hospitalière. Cette couche est responsable de la logique métier de l'application, qui traite les données avant de les envoyer à la couche de persistance (DAO) ou de les présenter à l'utilisateur.

Les méthodes déclarées dans cette interface permettent de sauvegarder les entités principales de notre système hospitalier : les patients, les médecins, les rendez-vous et les consultations. Chacune de ces méthodes prend en paramètre l'objet correspondant à sauvegarder et retourne l'objet sauvégarde après l'opération.

```
package ma.bouchama.hospital.service;

import ma.bouchama.hospital.entities.Consultation;
import ma.bouchama.hospital.entities.Medecin;
import ma.bouchama.hospital.entities.Patient;
import ma.bouchama.hospital.entities.RendezVous;

3 usages 1 implementation
public interface IHospitalService {
    1 usage 1 implementation
    Patient savePatient(Patient patient);
    1 usage 1 implementation
    Medecin saveMedecin(Medecin medecin);
    1 usage 1 implementation
    RendezVous saveRendezVous(RendezVous rendezVous);
    1 usage 1 implementation
    Consultation saveConsultation(Consultation consultation);

}
```

## classe IHospitalServiceImpl

Cette classe **IHospitalServiceImpl** est une implémentation de l'interface **IHospitalService**, représentant la couche métier de notre application hospitalière. Elle est annotée avec `@Service` pour être détectée et gérée par Spring en tant que composant de service.

```
package ma.bouchama.hospital.service;

> import ...

@Service
@Transactional
public class IHospitalServiceImpl implements IHospitalService {

    2 usages
    private PatientRepository patientRepository;
    2 usages
    private MedecinRepository medecinRepository;
    2 usages
    private RendezVousRepository rendezVousRepository;
    2 usages
    private ConsultationRepository consultationRepository;

    public IHospitalServiceImpl(PatientRepository patientRepository,
                                MedecinRepository medecinRepository,
                                RendezVousRepository rendezVousRepository,
                                ConsultationRepository consultationRepository) {
        this.patientRepository = patientRepository;
        this.medecinRepository = medecinRepository;
        this.rendezVousRepository = rendezVousRepository;
        this.consultationRepository = consultationRepository;
    }
}
```

Dans cette classe, j'ai défini des méthodes pour sauvegarder les entités principales de notre système hospitalier : les patients, les médecins, les rendez-vous et les consultations. Chaque méthode utilise les repositories correspondants pour effectuer les opérations de sauvegarde dans la base de données.

La méthode saveRendezVous mérite une attention particulière car elle génère un identifiant unique (UUID) pour chaque rendez-vous avant de le sauvegarder

```
1 usage
@Override
public Patient savePatient(Patient patient) { return patientRepository.save(patient); }

1 usage
@Override
public Medecin saveMedecin(Medecin medecin) {

    return medecinRepository.save(medecin);
}

1 usage
@Override
public RendezVous saveRendezVous(RendezVous rendezVous) {
    rendezVous.setId(UUID.randomUUID().toString());

    return rendezVousRepository.save(rendezVous);
}

1 usage
@Override
public Consultation saveConsultation(Consultation consultation) {

    return consultationRepository.save(consultation);
}
}
```

## Maintenant on va utiliser la couche service dans notre couche Application

Méthode start : Cette méthode est annotée avec @Bean et implémente l'interface CommandLineRunner. Elle est exécutée au démarrage de l'application. À l'intérieur de cette méthode, plusieurs opérations sont effectuées :

- Création et sauvegarde de patients et de médecins : Des patients et des médecins sont créés à l'aide de flux Java 8 et sauvegardés dans la base de données en utilisant la méthode correspondante de l'interface IHospitalService.
- Création et sauvegarde d'un rendez-vous : Un rendez-vous est créé entre un patient et un médecin, et sauvegardé dans la base de données en utilisant la méthode saveRendezVous de l'interface IHospitalService.
- Création et sauvegarde d'une consultation : Une consultation est créée pour un rendez-vous existant, et sauvegardée dans la base de données en utilisant la méthode saveConsultation de l'interface IHospitalService.

```
package ma.bouchama.hospital;

> import ...

@SpringBootApplication
public class HospitalApplication {

    public static void main(String[] args) { SpringApplication.run(HospitalApplication.class, args); }

    @Bean
    CommandLineRunner start(IHospitalService hospitalService,
                           PatientRepository patientRepository,
                           MedecinRepository medecinRepository,
                           RendezVousRepository rendezVousRepository){

        return args -> {
            Stream.of(...values: "Mohammed", "Farid", "Khalid")
                .forEach(name->{
                    Patient patient=new Patient();
                    patient.setNom(name);
                    patient.setDateNaissance(new Date());
                    patient.setMalade(false);
                    hospitalService.savePatient(patient);
                });
        };
    }
}
```

```
Stream.of( ...values: "Hajar", "Kaotar", "hind")
    .forEach(name->{
        Medecin medecin=new Medecin();
        medecin.setNom(name);
        medecin.setSpecialite(Math.random()>0.5?"Cardio":"Dentiste");
        medecin.setEmail(name+"@gmail.com");
        hospitalService.saveMedecin(medecin);

    });

Patient patient=patientRepository.findAll().get(0);
Patient patient1=patientRepository.findByNom( name: "Farid");

Medecin medecin= medecinRepository.findByNom( name: "Hajar");

RendezVous rendezVous=new RendezVous();
rendezVous.setDate(new Date());
rendezVous.setStatus(StatusRDV.PENDING);
rendezVous.setMedecin(medecin);
rendezVous.setPatient(patient);
RendezVous saveDRV = hospitalService.saveRendezVous(rendezVous);
System.out.println(saveDRV.getId());

RendezVous rendezVous1= rendezVousRepository.findById(1L).orElse( other: null);
Consultation consultation=new Consultation();
consultation.setDateConsultation(new Date());
consultation.setRendezVous(rendezVous1);
consultation.setRapport("Rapport de la consultation ....");
hospitalService.saveConsultation(consultation);
```

```
};

}
```

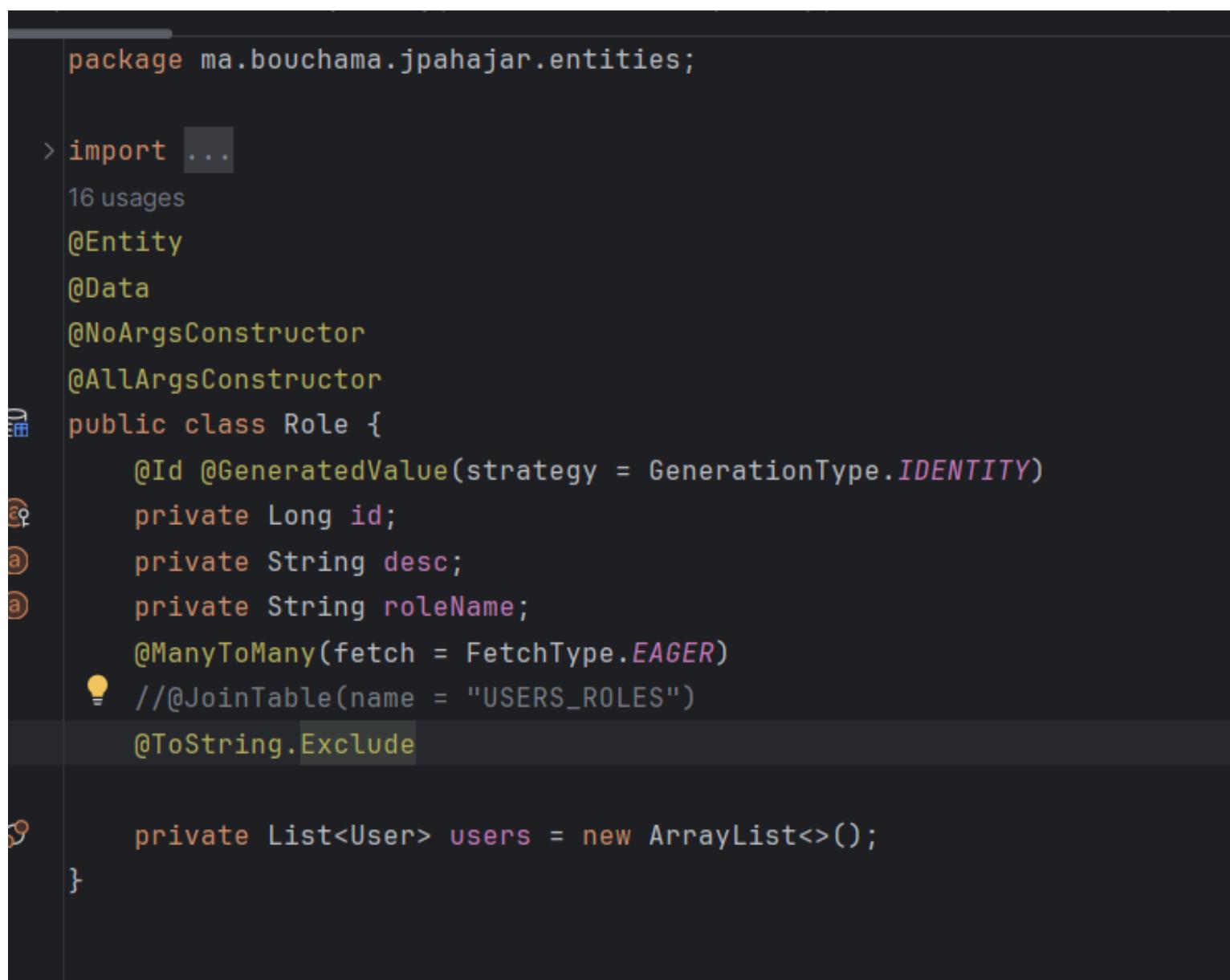
## le deuxième projet :

### La classe Role

la classe **Role** définit la structure et le comportement d'un rôle dans le système, y compris ses attributs, ses constructeurs, ses méthodes générées automatiquement, et sa relation avec les utilisateurs.

- **@Entity:** Cette annotation indique à JPA (Java Persistence API) que cette classe est une entité et qu'elle doit être persistée dans la base de données.
- **@Data:** Cette annotation de Lombok génère automatiquement les méthodes `toString()`, `equals()`, `hashCode()`, les accesseurs et mutateurs pour tous les champs de la classe.
- **@NoArgsConstructor et @AllArgsConstructor:** Ces annotations de Lombok génèrent automatiquement un constructeur sans argument et un constructeur avec tous les arguments, respectivement.

La classe Role est en relation **Many-to-Many** avec la classe User. Cette relation est configurée par la liste users.



```
package ma.bouchama.jpahajar.entities;

import ...;
16 usages
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Role {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String desc;
    private String roleName;
    @ManyToMany(fetch = FetchType.EAGER)
    // @JoinTable(name = "USERS_ROLES")
    @ToString.Exclude
    private List<User> users = new ArrayList<>();
}
```

### La classe User :

la classe User définit la structure et le comportement d'un utilisateur dans le système, y compris ses attributs, ses constructeurs, ses méthodes générées automatiquement, et sa relation avec les rôles

- **@Table(name = "USERS") :** Cette annotation permet de spécifier le nom de la table dans la base de données à laquelle cette entité est mappée.
- **@Column(name = "USER\_NAME", unique = true, length = 20)** pour spécifier le nom de la colonne dans la base de données ainsi que ses contraintes (unicité et longueur maximale).

La classe User est en relation Many-to-Many avec la classe Role. Cette relation est configurée par la liste roles

```
package ma.bouchama.jpahajar.entities;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.ArrayList;
import java.util.List;
23 usages
@Entity
@Table(name = "USERS")
@Data @NoArgsConstructor@AllArgsConstructor
public class User {
    @Id
    private String userId;
    @Column(name = "USER_NAME", unique = true, length = 20)
    private String username;
    private String password;
    @ManyToMany(mappedBy = "users", fetch = FetchType.EAGER)
    private List<Role> roles=new ArrayList<>();
}
```

## les interfaces :

### **I'interface RoleRepository**

l'interface **RoleRepository** qui agit comme un référentiel (repository) pour la classe Role. Cette interface étend JpaRepository<Role, Long>, ce qui signifie qu'elle hérite de méthodes prédéfinies pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur les entités de type Role dans la base de données.

La méthode **findByRoleName(String roleName)** déclare une méthode personnalisée pour rechercher un rôle par son nom. Spring Data JPA infère automatiquement la requête SQL à exécuter en fonction du nom de la méthode, ce qui simplifie considérablement le développement.

```
package ma.bouchama.jpahajar.repositories;

import ma.bouchama.jpahajar.entities.User;
import org.springframework.data.jpa.repository.JpaRepository;
import ma.bouchama.jpahajar.entities.Role;
import org.springframework.stereotype.Repository;

2 usages
@Repository
public interface RoleRepository extends JpaRepository<Role, Long> {
    1 usage
    Role findByRoleName(String roleName);
}
```

## I'interface UserRepository :

interface **UserRepository** qui agit comme un référentiel (repository) pour la classe User. Comme dans le cas précédent, cette interface étend également JpaRepository<User, String>, héritant ainsi des méthodes prédéfinies pour effectuer des opérations CRUD sur les entités de type User dans la base de données.

La méthode **findByusername(String userName)** déclare une méthode personnalisée pour rechercher un utilisateur par son nom d'utilisateur. Spring Data JPA infère automatiquement la requête SQL à exécuter en fonction du nom de la méthode.

```
package ma.bouchama.jpahajar.repositories;

import org.springframework.data.jpa.repository.JpaRepository;
import ma.bouchama.jpahajar.entities.User;
import org.springframework.stereotype.Repository;

2 usages
@public interface UserRepository extends JpaRepository<User, String> {
    2 usages
    User findByusername(String userName);

}
```

## la classe principale

le code ci-dessous représente la classe principale de l'application **JpaHajarApplication**.

**main():** Cette méthode est le point d'entrée de l'application. Elle lance l'application Spring Boot en appelant `SpringApplication.run(JpaHajarApplication.class, args)`.

Méthode **start()** annotée avec `@Bean`: Cette méthode est un bean Spring qui implémente l'interface `CommandLineRunner`. Elle est exécutée au démarrage de l'application. À l'intérieur de cette méthode, des utilisateurs et des rôles sont créés et des opérations sont effectuées à l'aide du service `UserService`. Ces opérations comprennent l'ajout de nouveaux utilisateurs, l'ajout de nouveaux rôles, l'attribution de rôles à des utilisateurs, et l'authentification d'un utilisateur.

**Injection de dépendance UserService:** La méthode `start()` prend en argument une instance de `UserService`, qui est injectée par Spring.

**Appels aux méthodes du service UserService:** À l'intérieur de la méthode `start()`, différentes méthodes du service `UserService` sont appelées pour effectuer des opérations sur les utilisateurs et les rôles.

**Gestion des exceptions:** Un bloc `try-catch` est utilisé pour gérer les exceptions qui peuvent survenir lors de l'authentification d'un utilisateur à l'aide du service `UserService`.

```

package ma.bouchama.jpahajar;

> import ...;

@SpringBootApplication
public class JpaHajarApplication {

    public static void main(String[] args) {
        SpringApplication.run(JpaHajarApplication.class, args);
    }

    @Bean
    CommandLineRunner start(UserService userService){
        return args -> {
            User u = new User();
            u.setUsername("user1");
            u.setPassword("123456");
            userService.addNewUser(u);

            User u2 = new User();
            u2.setUsername("admin");
            u2.setPassword("123456");
            userService.addNewUser(u2);

            Stream.of( ...values: "STUDENT", "USER", "ADMIN").forEach(r->{
                Role role1 = new Role();
                role1.setRoleName(r);
                userService.addNewRole(role1);
            });

            userService.addRoleToUser( username: "user1", roleName: "STUDENT");
            userService.addRoleToUser( username: "user1", roleName: "USER");
            userService.addRoleToUser( username: "admin", roleName: "USER");
            userService.addRoleToUser( username: "admin", roleName: "ADMIN");

            try{
                User user = userService.authenticate( userName: "user1", password: "123456");
                System.out.println(user.getUserId());
                System.out.println(user.getUsername());
                System.out.println("Roles : ");
                user.getRoles().forEach(r->{
                    System.out.println("Role => "+r.toString());
                });
            };
        }
    }
}

```

## I'interface UserService :

Cette interface définit les méthodes disponibles pour la gestion des utilisateurs et des rôles.

- **addNewUser(User user):** Cette méthode permet d'ajouter un nouvel utilisateur.
- **addNewRole(Role role):** Cette méthode permet d'ajouter un nouveau rôle.
- **findUserByUserName(String userName):** Cette méthode permet de trouver un utilisateur par son nom d'utilisateur.
- **findRoleByName(String roleName):** Cette méthode permet de trouver un rôle par son nom.
- **addRoleToUser(String username, String roleName):** Cette méthode permet d'ajouter un rôle à un utilisateur existant.
- **authenticate(String userName, String password):** Cette méthode permet d'authentifier un utilisateur en vérifiant son nom d'utilisateur et son mot de passe.

```
package ma.bouchama.jpahajar.service;

import ma.bouchama.jpahajar.entities.Role;
import ma.bouchama.jpahajar.entities.User;

5 usages 1 implementation
① public interface UserService {
    2 usages 1 implementation
    User addNewUser(User user);
    1 usage 1 implementation
    Role addNewRole(Role role);
    2 usages 1 implementation
    User findUserByUserName(String userName);
    1 usage 1 implementation
    Role findRoleByRoleName(String roleName);
    4 usages 1 implementation
    void addRoleToUser(String username, String roleName);
    1 usage 1 implementation
    User authenticate(String userName, String password);

}

}
```

### UserServiceImpl :

Cette classe implémente l'interface UserService et fournit une implémentation concrète de ses méthodes.

- **@Service** : Cette annotation indique à Spring que cette classe est un composant de service.
- **@Transactional** : Cette annotation indique que chaque méthode de ce service sera exécutée dans une transaction.
- **addNewUser(User user)**: Génère un UUID pour l'utilisateur, puis l'ajoute à la base de données en utilisant le référentiel UserRepository.
- **addNewRole(Role role)**: Ajoute le rôle à la base de données en utilisant le référentiel RoleRepository.
- **findUserByUserName(String userName)**: Recherche un utilisateur par son nom d'utilisateur en utilisant le référentiel UserRepository.
- **findRoleByRoleName(String roleName)**: Recherche un rôle par son nom en utilisant le référentiel RoleRepository.
- **addRoleToUser(String username, String roleName)**: Ajoute un rôle à un utilisateur existant en récupérant l'utilisateur et le rôle correspondants, puis en les associant et en les enregistrant dans la base de données.
- **authenticate(String userName, String password)**: Authentifie un utilisateur en vérifiant son nom d'utilisateur et son mot de passe. Si l'utilisateur est trouvé et que le mot de passe correspond, l'utilisateur est retourné ; sinon, une exception est levée.

```
package ma.bouchama.jpahajar.service;

> import ...

@Service
@Transactional
@AllArgsConstructor
public class UserServiceImpl implements UserService {
    private UserRepository userRepository;
    private RoleRepository roleRepository;
    2 usages
    @Override
    public User addNewUser(User user) {
        user.setUserId(UUID.randomUUID().toString());
        return userRepository.save(user);
    }

    1 usage
    @Override
    public Role addNewRole(Role role) {
        return roleRepository.save(role);
    }
}
```

```
2 usages
@Override
public User findUserByUserName(String userName) {
    return userRepository.findByusername(userName);
}

1 usage
@Override
public Role findRoleByRoleName(String roleName) {
    return roleRepository.findByName(roleName);
}

4 usages
@Override
public void addRoleToUser(String username, String roleName) {
    User user = findUserByUserName(username);
    Role role = findRoleByRoleName(roleName);
    if(user.getRoles()!=null){
        user.getRoles().add(role);
    }
    //userRepository.save(user)
}
```

```
54    }
55
56    1 usage
57    @Override
58    public User authenticate(String userName, String password) {
59        User user=userRepository.findByusername(userName);
60        if(user==null) throw new RuntimeException("Bad credentials");
61        if(user.getPassword().equals(password)){
62            return user;
63        }
64        throw new RuntimeException("Bad credentials");
65    }
66}
```

Ce fichier de configuration Spring Boot définit les paramètres essentiels pour le déploiement et la connexion à la base de données de l'application. Les options spécifiées incluent le nom de l'application, le port du serveur, les détails de connexion à la base de données MySQL, et la configuration Hibernate pour la gestion du schéma et l'affichage des requêtes SQL.

```

spring.application.name=jpa-hajar
server.port=8083
spring.datasource.url=jdbc:mysql://localhost:3306/users_db?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
#spring.h2.console.enabled=true

spring.jpa.show-sql=true

```

The screenshot shows the MySQL Workbench interface. On the left, the schema browser displays tables such as ROLE, ROLE\_USERS, USERS, INFORMATION\_SCHEMA, and Users. In the center, a query editor window contains the SQL statement "SELECT \* FROM ROLE;". Below the statement, the results are shown in a table:

ID	DESC	ROLE_NAME
1	null	STUDENT
2	null	USER
3	null	ADMIN

(3 rows, 0 ms)

An arrow points from this screenshot down to the table list in the MySQL Workbench interface.

The screenshot shows the MySQL Workbench interface with a focus on the table list in the left sidebar. The list includes:

- role (InnoDB, utf8mb4\_general\_ci)
- role\_users (InnoDB, utf8mb4\_general\_ci)
- users (InnoDB, utf8mb4\_general\_ci)
- 3 tables Sum

## Le Contrôleur UserController :

```

import ma.bouchama.jpahajar.entities.User;
import ma.bouchama.jpahajar.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {
    @Autowired
    private UserService userService;
    @GetMapping("/users/{username}")
    public User user(@PathVariable String username){
        User user = userService.findUserByUserName(username);
        return user;
    }
}

```

Ce contrôleur REST expose un point de terminaison permettant de récupérer les détails d'un utilisateur en fonction de son nom d'utilisateur. Il utilise un service utilisateur pour effectuer la recherche dans la base de données et renvoyer l'utilisateur trouvé sous forme de réponse JSON.

L'utilisation de `@JsonProperty(access = JsonProperty.Access.WRITE_ONLY)` permet de contrôler quelles propriétés sont incluses dans la représentation JSON d'un objet, offrant ainsi plus de flexibilité et de contrôle sur la structure des données exposées via une API.

```
import java.util.ArrayList;
import java.util.List;
16 usages
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Role {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "DESCRIPTION")
    private String desc;
    private String roleName;
    @ManyToMany(fetch = FetchType.EAGER)
    // @JoinTable(name = "USERS_ROLES")
    @ToString.Exclude
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private List<User> users = new ArrayList<>();
}
```

```
package ma.bouchama.jpahajar.entities;

import com.fasterxml.jackson.annotation.JsonProperty;
import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.ArrayList;
import java.util.List;
26 usages
@Entity
@Table(name = "USERS")
@Data @NoArgsConstructor @AllArgsConstructor
public class User {
    @Id
    private String userId;
    @Column(name = "USER_NAME", unique = true, length = 20)
    private String username;
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private String password;
    @ManyToMany(mappedBy = "users", fetch = FetchType.EAGER)
    private List<Role> roles=new ArrayList<>();
}
```

'annotation `@JsonProperty` nous permet de définir comment une propriété d'objet Java doit être sérialisée ou déserialisée vers ou depuis JSON

`@JsonProperty(access = JsonProperty.Access.WRITE_ONLY)` est utilisé pour spécifier que la propriété annotée ne doit être prise en compte que lors de la sérialisation JSON (c'est-à-dire lors de la conversion d'un objet Java en JSON), et elle doit être ignorée lors de la déserialisation (c'est-à-dire lors de la conversion JSON en objet Java). Cela signifie que la propriété ne sera pas mappée depuis JSON vers l'objet Java correspondant lors de la déserialisation, la rendant ainsi uniquement accessible en écriture du point de vue de l'interchange de données JSON.

## Conclusion :

Ce TP a été une exploration fructueuse des principes fondamentaux du développement d'applications web avec Spring Boot et JPA. En mettant en œuvre des entités JPA, des repositories, des services métier et des contrôleurs REST, nous avons pu créer une application fonctionnelle de gestion des utilisateurs et des rôles. La modélisation des données, la persistance, l'exposition des fonctionnalités via une API REST et la configuration de l'application ont été des aspects essentiels de ce processus. En consolidant ces concepts, ce TP a fourni une base solide pour la compréhension et le développement ultérieur d'applications web robustes et évolutives avec Spring Boot.