

# TP - SPRING DATA/JPA/HIBERNATE

## Contents

INTRODUCTION : .....	2
1-Cas de Patient, Médecin, Rendez-vous, Consultation.....	2
CREATION DU PROJET (INITIALISATION) : .....	2
Conception : .....	4
Création de la couche Dao : .....	4
Création de la couche Web : .....	11
2. Cas de Users et Roles.....	13
Conception : .....	13
Création de la couche Dao : .....	13
Basculer vers une base de données MySQL au lieu de H2 : .....	18
Créer l'application Web qui permet de chercher les users : .....	18
Conclusion : .....	19

## INTRODUCTION :

Dans ce deuxième tp nous pratiquerons toutes les nouveaux acquis qui concernent JPA, Hibernate et Spring Data et pour cela ce TP va être deviser en deux parties :

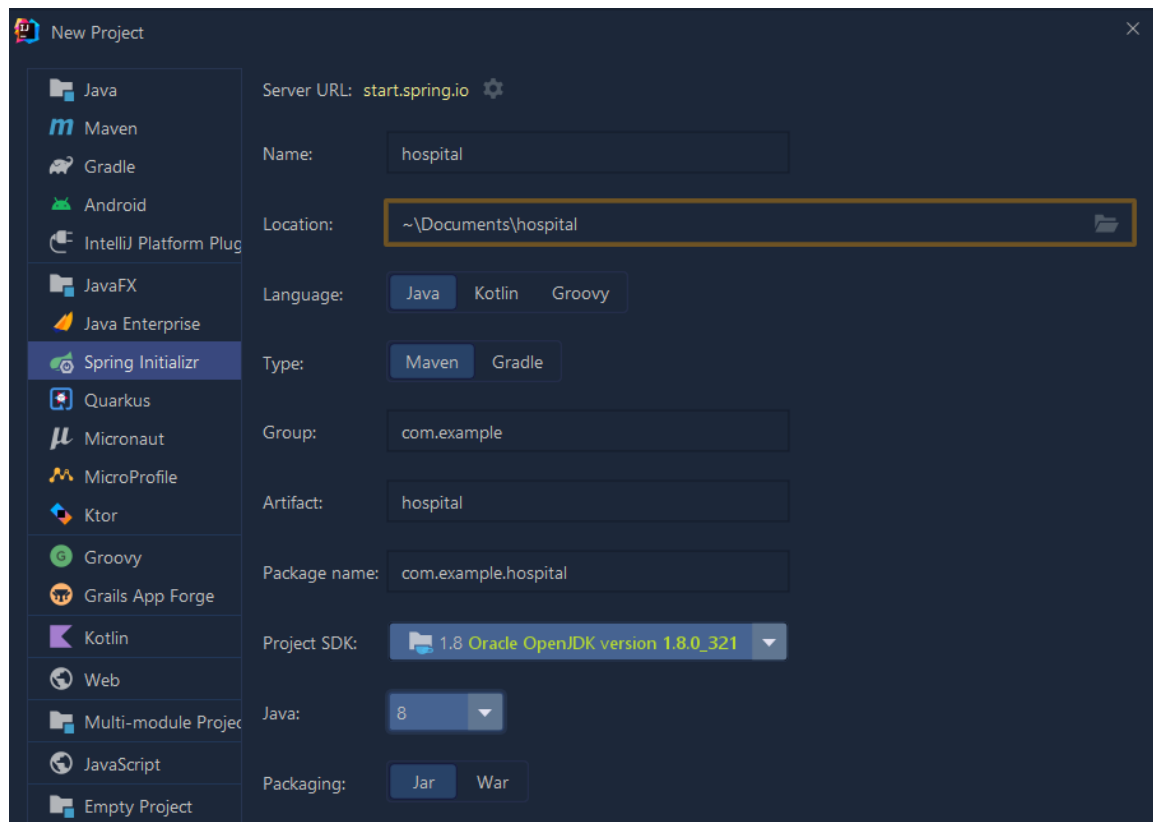
- 1-Cas de Patient, Médecin, Rendez-vous, Consultation
2. Cas de Users et Roles

## 1-Cas de Patient, Médecin, Rendez-vous, Consultation

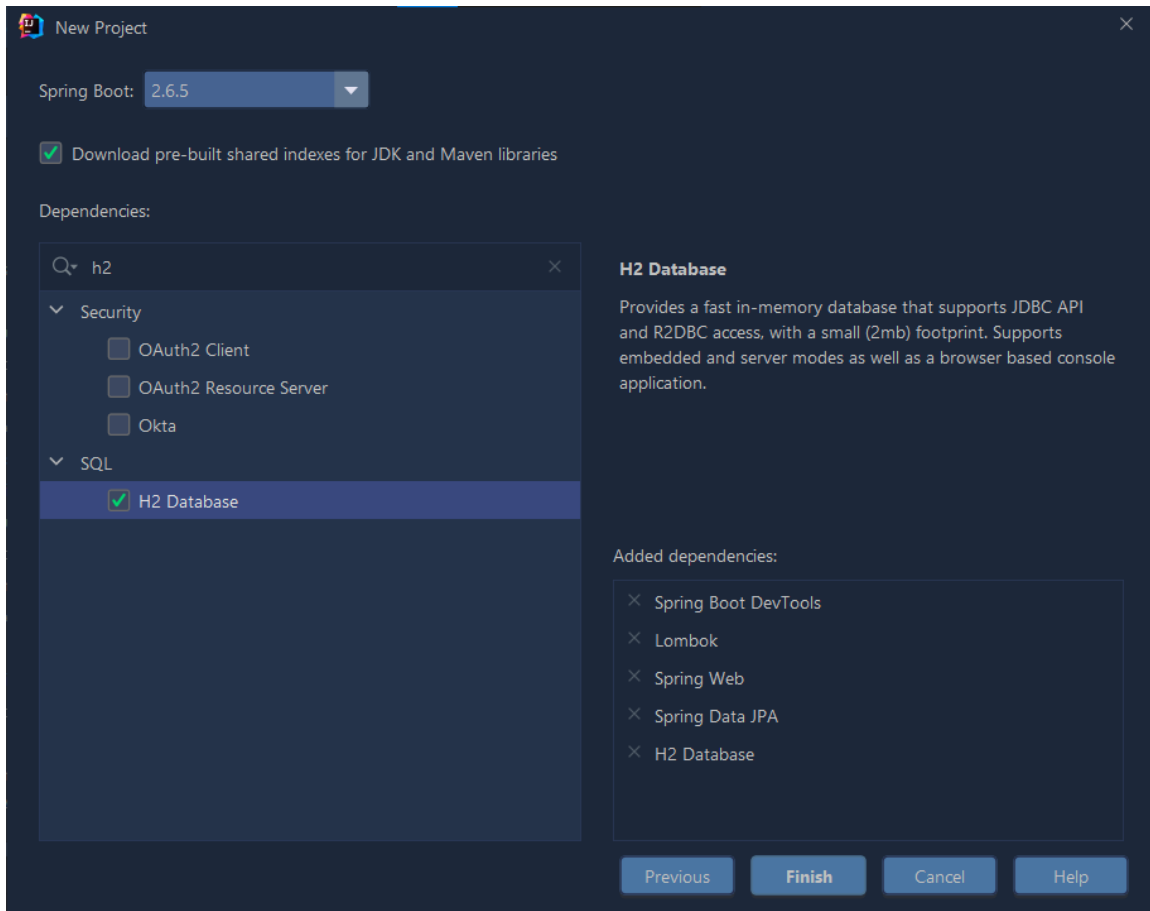
### CREATION DU PROJET (INITIALISATION) :

On choisi un projet Spring initializr ,il nous permet de sélectionner les éléments de bases qui constituent notre application.

On rempli les informations relatives à notre projet et on clique sur **Next**



Voyons comment si facile d'entamer le developpement en utilisant spring boot. Il suffit de cocher les technologies qu'on souhaite ajouté à notre projet et c'est spring boot qui se charge de les ajouter et configurer dans le projet.

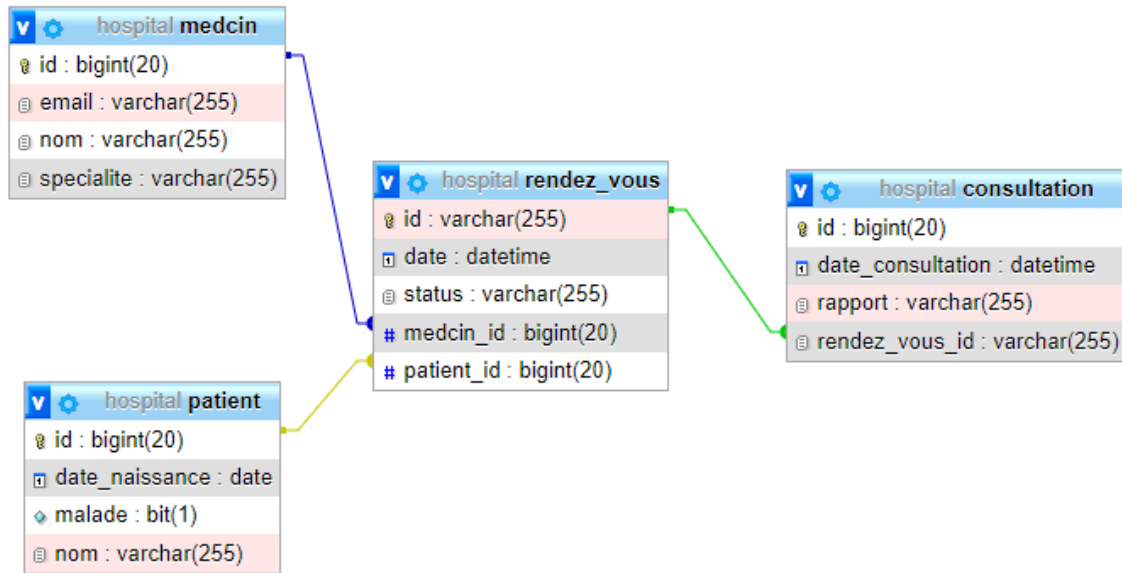


nous avons ajouté :

- Spring Web : voici l'élément le plus important , c'est celui qui va entraîner la création d'une application spring avec toutes les dépendances y nécessaires.
- Spring DevTools : permettent d'automatiser le redémarrage et le rechargement de votre application lorsque les fichiers sources sont recompilés ou modifiés.
- Lombok : est une API dont le but est de générer à la compilation, du code Java(getters()/setters(), toString()...), à notre place.
- Spring Data JPA : fournit une implémentation de la couche d'accès aux données pour une application **Spring** .
- H2 Database : est un système de gestion de base de données relationnelles écrit en Java. Il peut être intégré à une application Java ou bien fonctionner en mode client-serveur.

Finalement on clique sur Finish et voilà notre application est prête.

## Conception :



## Création de la couche Dao :

a) Créer l'entité JPA Patient :

La première chose à faire est de créer l'entité à persister, c'est-à-dire la classe Patient, Alors on crée une classe nommée Patient et on la modifie comme ci-dessous

```

package ma.enset.hospital.entities;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;
import java.util.Collection;
import java.util.Date;

@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class Patient {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom ;
    @Temporal(TemporalType.DATE)
    private Date dateNaissance ;
    private boolean malade ;
    @OneToMany(mappedBy = "patient", fetch = FetchType.LAZY)
    private Collection<RendezVous> rendezVous ;
}
  
```

- **@Entity** : spécifie que la classe est une entité et qu'elle est mappée à une table de base de données.
- **@Table** : spécifie la table dans la base de données.

- @Id : cette annotations est obligatoire à ajouter dès qu'on ajoute l'annotation @entity, l'annotation @Id spécifie le champs primaire de la table.
- @GeneratedValue : spécifie comment le sgbd affecte la valeur de l'id.
- @Column : spécifié qu'il faut mappé ce attribut à une colonne dans la table et on peut configurer la collonne dans la table comme on veut, on note que toutes les attribut dans une entité sont mappé vers des colonnes dans la table de bdd meme si on utilise pas l'annotation @column

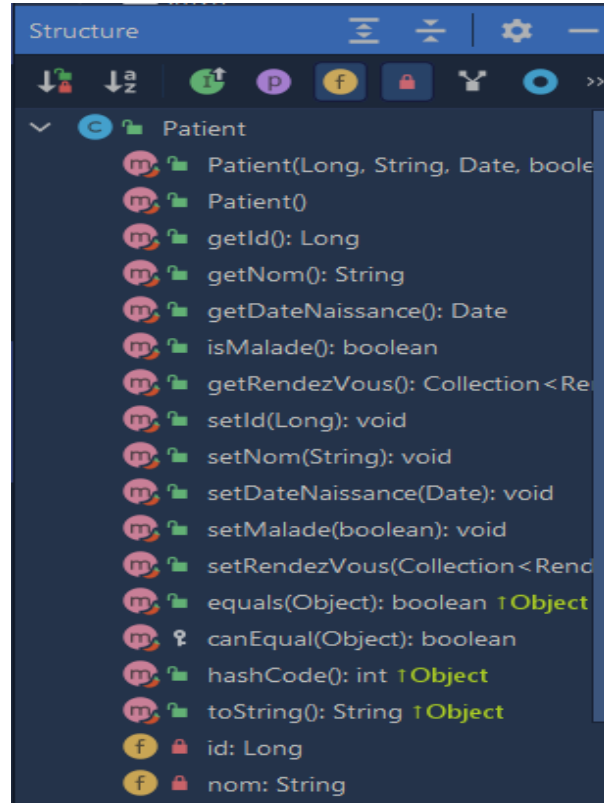
Et voila notre entité est près à etre mappé vers une table dans la bdd.

Maintenant on souhaite ajouter des getters and setters et les constructeur et une toString, or on va pas le faire manuellement, c'est Lombok qui se charge de les ajouter dans le byte code de notre application.

Alors pour ce faire il suffit d'ajouter les annotations :

- @Data : ajoute les getters et setters.
- @NoArgsConstructor : ajoute un constructeur par défaut.
- @AllArgsConstructor : ajoute un constructeur avec toutes les paramètres.
- @ToString : ajoute une méthode toString.

Pour verifier que Lombok fonctionne correctement on affiche le sturcture de notre projet :



On voit bien la présence des methodes et constructeurs souhaité.

**b) Créer l'interface PatientRepository :**

Alors puisque Spring Data a déjà créé des interfaces génériques et des implémentations génériques qui permettent de gérer les entités JPA, on n'aura plus besoin de faire appel à l'objet EntityManager pour gérer la persistance. Spring Data le fait à notre place.

Il suffit de créer une interface qui hérite de l'interface JpaRepository pour hériter toutes les méthodes classiques qui permettent de gérer les entités JPA, de plus si on a besoin d'autre methodes nous avons la possibilité d'ajouter d'autres méthodes en les déclarant à l'intérieur de l'interface JpaRepository, sans avoir besoin de les implémenter. Spring Data le fera à notre place.

Le resultat finale semble à la figure ci-dessous :

```
package ma.enset.hospital.repositories;
import ma.enset.hospital.entities.Patient;
import org.springframework.data.jpa.repository.JpaRepository;
public interface PatientRepository extends JpaRepository<Patient,Long>
{
    public Patient findByNom(String name);
}
```

Alors dans cet exemple la méthode findAll() qui est une methode « habituelle » dans les classes DAO est fournit par défaut , et on a pas besoins de la definir ni de l'implémenter. De plus on peut definir autres méthode, en cas de besoins, et c'est Spring qui analyse le nom de la méthode définit et ses paramètre pour l'implémenter.(comme findByNom qui se traduit vers une requete qui retourne toutes l'enregistrement qui ont une valeur dans le champ name qui contient la chaine name fournit en paramètres )

**c) Créer l'entité JPA Medcin:**

```
package ma.enset.hospital.entities;
+import ...
@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class Medcin {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String email;
    private String specialite;
    @OneToMany(mappedBy = "medcin", fetch = FetchType.LAZY)
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Collection<RendezVous> rendezVous ;
}
```

**d) Créer l'interface PatientRepository :**

```
package ma.enset.hospital.repositories;
+import ...
public interface MedcinRepository extends
JpaRepository<Medcin, Long> {
    public Medcin findByNom(String nom);
}
```

**e) Créer l'entité JPA Consultation:**

```
package ma.enset.hospital.entities;
+import ...
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Consultation {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date dateConsultation ;
    private String rapport;
    @OneToOne
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private RendezVous rendezVous;
}
```

**a) Créer l'interface ConsultationRepository :**

```
b) package ma.enset.hospital.repositories;
import ma.enset.hospital.entities.Consultation;
import ma.enset.hospital.entities.Patient;
import org.springframework.data.jpa.repository.JpaRepository;
public interface ConsultationRepository extends
JpaRepository<Consultation, Long> {
}
```

**f) Créer l'entité JPA Rendezvous:**

```
g) package ma.enset.hospital.entities;
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;
import java.util.Date;
@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class RendezVous {
    @Id // @GeneratedValue(strategy = GenerationType.IDENTITY)
    private String id;
    private Date date;
    @Enumerated(EnumType.STRING)
    private StatusRDV status;
    @ManyToOne
```

```

        @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
        private Patient patient;
        @ManyToOne
        private Medcin medcin;
        @OneToOne(mappedBy = "rendezVous")
        private Consultation consultation;
    }

```

c) Créer l'interface **RendezvousRepository** :

```

d) package ma.enset.hospital.repositories;
import ma.enset.hospital.entities.Medcin;
import ma.enset.hospital.entities.RendezVous;
import org.springframework.data.jpa.repository.JpaRepository;
//public interface RendezvousRepository extends
JpaRepository<RendezVous, Long> {
public interface RendezvousRepository extends
JpaRepository<RendezVous, String> {
}

```

nous avons aussi cree un package service qui va contient une interface **IHospitalService** et une implementtation de cette interface **HospitalServiceImpl**

e) Créer l'interface **IHospitalService**:

```

package ma.enset.hospital.service;
+import ...
public interface IHospitalService {
    Patient savePatient(Patient patient);
    Medcin saveMedcin(Medcin medcin);
    RendezVous saveRDV(RendezVous rendezVous);
    Consultation saveConsultation(Consultation consultation);
}

```

f) Créer de la class **HospitalServiceImpl**:

```

package ma.enset.hospital.service;
+import ...
import java.util.UUID;
@Service
@Transactional
public class HospitalServiceImpl implements IHospitalService {
    private PatientRepository patientRepository;
    private MedcinRepository medcinRepository;
    private RendezvousRepository rendezvousRepository;
    private ConsultationRepository consultationRepository;
    public HospitalServiceImpl(PatientRepository patientRepository,
                              MedcinRepository medcinRepository,
                              RendezvousRepository
rendezvousRepository,
                              ConsultationRepository
consultationRepository) {
        this.patientRepository = patientRepository;
        this.medcinRepository = medcinRepository;
        this.rendezvousRepository = rendezvousRepository;
        this.consultationRepository = consultationRepository;
    }
}

```



```

    }
    @Override
    public Patient savePatient(Patient patient) {
        return patientRepository.save(patient);
    }
    @Override
    public Medcin saveMedcin(Medcin medcin) {
        return medcinRepository.save(medcin);
    }
    @Override
    public RendezVous saveRDV(RendezVous rendezVous) {
        rendezVous.setId(UUID.randomUUID().toString());
        return rendezVousRepository.save(rendezVous);
    }
    @Override
    public Consultation saveConsultation(Consultation consultation) {
        return consultationRepository.save(consultation);
    }
}

```

**g) Configurer l'unité de persistance : application.properties :**

Pour le moment nous travaillons avec une base de donnée mémoire qui s'appelle h2, alors il suffit d'ajouter une ligne qui définit l'url de cette bdd dans le fichier application.properties comme ci-dessous :

```

spring.datasource.url=jdbc:h2:mem:hospital
spring.h2.console.enabled=true
server.port=8086

```

**h) Tester la couche DAO :**

Nous modifions la classe **HospitalApplication** pour avoir le resultat suivant:

Nous avons implémenter l'interface CommandLineRunner et réimplémenter la methode Run, alors maintenant dès que notre Spring Container est prés il va executer le code dedans la méthode run donc c'est dedans qu'on va tester notre couche dao.

On injecte une PatientRepositories dans la classe et on commence à enregistrer et afficher les patients :

```

package ma.enset.hospital;
+import ...
@SpringBootApplication
public class HospitalApplication {
    public static void main(String[] args) {
        SpringApplication.run(HospitalApplication.class, args);
    }
    @Bean
    //CommandLineRunner start(ConsultationRepository
    consultationRepository, PatientRepository patientRepository,
    MedcinRepository medcinRepository, RendezVousRepository
    rendezVousRepository){
        CommandLineRunner start(IHospitalService hospitalService,
        PatientRepository patientRepository,
        MedcinRepository medcinRepository,
        RendezVousRepository rendezVousRepository){

```

```

return args -> {
    Stream.of("Mohammed", "Ouassime", "Khadija")
        .forEach(name->{
            Patient patient = new Patient();
            patient.setNom(name);
            patient.setDateNaissance(new Date());
            patient.setMalade(false);
            //patientRepository.save(patient);
            hospitalService.savePatient(patient);
        });
    Stream.of("Aymane", "Hanane", "Bilal")
        .forEach(name->{
            Medcin medcin = new Medcin();
            medcin.setNom(name);
            medcin.setSpecialite(Math.random()>0.5?"Cardio":"Dentiste");
            medcin.setEmail(name+"@gmail.com");
            //medcinRepository.save(medcin);
            hospitalService.saveMedcin(medcin);
        });
    Patient patient = patientRepository.findByNom("Ouassime");
    Medcin medcin = medcinRepository.findByNom("Hanane");
    RendezVous rendezVous = new RendezVous();
    rendezVous.setDate(new Date());
    rendezVous.setStatus(StatusRDV.PENDING);
    rendezVous.setPatient(patient);
    rendezVous.setMedcin(medcin);
    //rendezVousRepository.save(rendezVous);
    hospitalService.saveRDV(rendezVous);
    //RendezVous rendezVous1 =
    rendezVousRepository.findById(1L).orElse(null);
    RendezVous rendezVous1 =
    rendezVousRepository.findAll().get(0);
    Consultation consultation = new Consultation();
    consultation.setDateConsultation(new Date());
    consultation.setRendezVous(rendezVous1);
    consultation.setRapport("le rapport de la consultation ...");
    //consultationRepository.save(consultation);
    hospitalService.saveConsultation(consultation);
    };
}
}

```

resultat dans la base de donnees :

The screenshot shows the H2 Console interface in a web browser. The address bar indicates the URL: `localhost:8086/h2-console/login.do?jsessionId=183c83c062666285e4585d629473d245`. The interface includes a toolbar with options like 'Auto commit', 'Max rows: 1000', 'Auto complete', and 'Auto select'. On the left, a tree view shows the database structure: `jdbc:h2:mem:hospital`, `CONSULTATION`, `MEDCIN`, `PATIENT`, `RENDEZ_VOUS`, `INFORMATION_SCHEMA`, `Sequences`, `Users`, and `H2 1.4.200 (2019-10-14)`. The main area displays the results of three SQL queries.

**Query 1: SELECT \* FROM PATIENT;**

ID	DATE_NAISSANCE	MALADE	NOM
1	2022-03-26	FALSE	Mohammed
2	2022-03-26	FALSE	Ouassime
3	2022-03-26	FALSE	Khadija

(3 rows, 8 ms)

**Query 2: SELECT \* FROM MEDCIN;**

ID	EMAIL	NOM	SPECIALITE
1	Aymane@gmail.com	Aymane	Dentiste
2	Hanane@gmail.com	Hanane	Dentiste
3	Bilal@gmail.com	Bilal	Cardio

(3 rows, 4 ms)

**Query 3: SELECT \* FROM RENDEZ\_VOUS;**

ID	DATE	STATUS	MEDCIN_ID	PATIENT_ID
08f013be-b3bc-4671-a418-9a62fc03bed4	2022-03-26 17:16:39.441	PENDING	2	2

(1 row, 4 ms)

### Création de la couche Web :

La création de la couche dao va être facilitée avec l'utilisation de du module SPRING MVC, c'est spring qui gère la création du servlet, et nous fournit toutes ses fonctionnalités sans complications et sans code technique, il suffit qu'on crée une classe et l'y ajouter l'annotation `@Controller`.

Mais sous le capot Toutes les requêtes HTTP sont traitées par un contrôleur frontal fourni par Spring. C'est une servlet nommée `DispatcherServlet`, c'est cette servlet qui devrait exécuter une opération associée à chaque action. Ces opérations sont implémentées dans une classe appelée `PatientController` qui représente un sous contrôleur ou un contrôleur secondaire.

Dans cette classe on crée des méthodes qui définissent les opérations et on les relie à des URL avec @GetMapping()

**a) Créer l'application Web qui permet de chercher les patients**

▪ **Afficher tous les patients :**

Pour afficher tous les patients, tout d'abord il faut créer le contrôleur secondaire et dedans on crée une méthode qui interroge la base de données, récupère la liste des patients, la stocke dans un modèle et finalement doit appeler la page HTML convenable.

```
package ma.enset.hospital.web;
+import ...

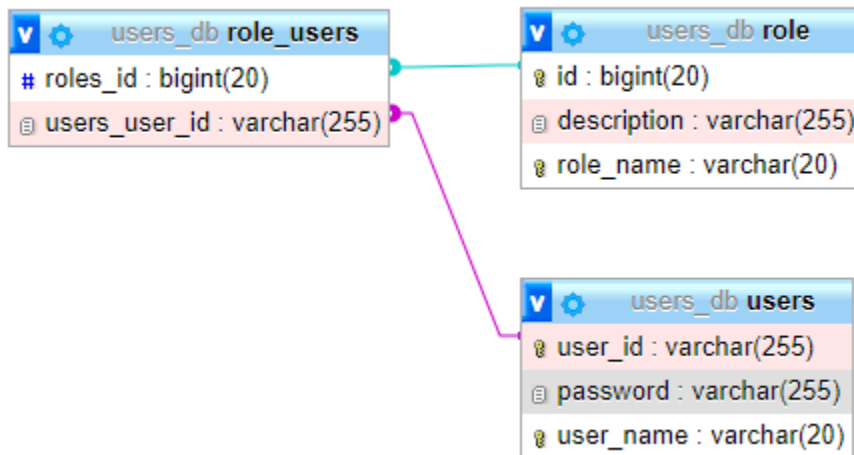
import java.util.List;
@RestController
public class PatientRestController {
    @Autowired
    private PatientRepository patientRepository;
    @GetMapping("/patients")
    public List<Patient> patientList(){
        return patientRepository.findAll();
    }
}
```

resultat en format json:

```
▼ {
  "id": 1,
  "nom": "Mohammed",
  "dateNaissance": "2022-03-26",
  "malade": false,
  "rendezVous": [],
},
▼ {
  "id": 2,
  "nom": "Ouassime",
  "dateNaissance": "2022-03-26",
  "malade": false,
  "rendezVous": [
    ▼ {
      "id": "08f013be-b3bc-4671-a418-9a62fc03bed4",
      "date": "2022-03-26T16:16:39.441+00:00",
      "status": "PENDING",
      "medecin": {
        "id": 2,
        "nom": "Hanane",
        "email": "Hanane@gmail.com",
        "specialite": "Dentiste"
      }
    },
  ],
},
```

## 2. Cas de Users et Roles

Conception :



Création de la couche Dao :

### a) Créer l'entité JPA User :

```

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

### b) Créer l'interface UserRepository:

```

package ma.enset.jpaenset.repositories;
+import ...
@Repository
public interface UserRepository extends JpaRepository<User,String> {
    User findByUsername(String username);
}
  
```

c) Créer l'entité JPA Role :

```
d) package ma.enset.jpaenset.entities;
+import ...
@Entity
@Data @AllArgsConstructor @NoArgsConstructor
public class Role {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(unique = true,length = 20)
    private String roleName;
    @Column(name = "DESCRIPTION")
    private String desc;
    @ManyToMany(fetch = FetchType.EAGER)
    // @JoinTable(name = "USERS_ROLES")
    @ToString.Exclude
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private List<User> users = new ArrayList<>();
}
```

e) Créer l'interface RoleRepository:

```
package ma.enset.jpaenset.repositories;
+import ...
@Repository
public interface RoleRepository extends JpaRepository<Role,Long> {
    Role findByRoleName(String roleName);
}
```

nous avons aussi cree un package service qui va contient une interface **UserService** et une implementation de cette interface **UserServiceImpl**

i) Créer l'interface UserService:

```
package ma.enset.jpaenset.service;
+import ...
public interface UserService {
    User addNewUser(User user);
    Role addNewRole(Role role);
    User findUserByUserName(String userName);
    Role findRoleByRoleName(String roleName);
    void addRoleToUser(String username,String rolename);
    User authenticate(String username,String password);
}
```

j) Créer la classe UserServiceImpl:

```
package ma.enset.jpaenset.service;
+import ...
@Service
@Transactional @AllArgsConstructor
public class UserServiceImpl implements UserService {
    private UserRepository userRepository;
    private RoleRepository roleRepository;
    @Override
    public User addNewUser(User user) {
        user.setUserId(UUID.randomUUID().toString());
        return userRepository.save(user);
    }
}
```

```

    }
    @Override
    public Role addNewRole(Role role) {
        return roleRepository.save(role);
    }
    @Override
    public User findUserByUserName(String userName) {
        return userRepository.findByUsername(userName);
    }
    @Override
    public Role findRoleByRoleName(String roleName) {
        return roleRepository.findByName(roleName);
    }
    @Override
    public void addRoleToUser(String username, String rolename) {
        User user = findUserByUserName(username);
        Role role = findRoleByRoleName(rolename);
        if (user.getRoles() != null) {
            user.getRoles().add(role);
            role.getUsers().add(user);
        }
        //userRepository.save(user);
    }
    @Override
    public User authenticate(String username, String password) {
        User user = findUserByUserName(username);
        if (user == null) throw new RuntimeException("Bad credentials");
        if (user.getPassword().equals(password)) {
            return user;
        }
        throw new RuntimeException("Bad credentials");
    }
}

```

**a) Configurer l'unité de persistance : application.properties :**

Pour le moment nous travaillons avec une base de donnée mémoire qui s'appelle h2, alors il suffit d'ajouter une ligne qui définit l'url de cette bdd dans le fichier application.properties comme ci-dessous :

```

spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:users_db
server.port=8083

```

**a) Tester la couche DAO :**

Nous modifions la classe **JpaEnsetApplication** pour avoir le resultat suivant:

Nous avons implémenter l'interface **CommandLineRunner** et réimplémenter la methode **Run**, alors maintenant dès que notre Spring Container est prés il va executer le code dedans la méthode run donc c'est dedans qu'on va tester notre couche dao.

On injecte une **UserService** dans la classe et on commence à enregistrer et afficher les users

```

package ma.enset.jpaenset;
+import ...
@SpringBootApplication
public class JpaEnsetApplication {
    public static void main(String[] args) {
        SpringApplication.run(JpaEnsetApplication.class, args);
    }
    @Bean
    CommandLineRunner start(UserService userService){
        return args -> {
            User u1=new User();
            u1.setUsername("user1");
            u1.setPassword("123456");
            userService.addNewUser(u1);
            User u2=new User();
            u2.setUsername("admin");
            u2.setPassword("654321");
            userService.addNewUser(u2);
            Stream.of("USER","ADMIN","STUDENT").forEach(r->{
                Role role = new Role();
                role.setRoleName(r);
                userService.addNewRole(role);
            });
            userService.addRoleToUser("user1","USER");
            userService.addRoleToUser("user1","STUDENT");
            userService.addRoleToUser("admin","ADMIN");
            userService.addRoleToUser("admin","USER");
            try {
                User user = userService.authenticate("user1","123456");
                System.out.println("Id : "+user.getUserId());
                System.out.println("UserName : "+user.getUsername());
                System.out.println("Roles : ");
                user.getRoles().forEach(r->{
                    System.out.println("    Role : "+r);
                });
            }catch (Exception e){
                e.printStackTrace();
            }
        };
    }
}

```

resultat dans la console :


```





















Id : 6415c117-1113-4e6c-b172-81d68d61b98a
UserName : user1
Roles :
    Role : Role(id=1, roleName=USER, desc=null)
    Role : Role(id=3, roleName=STUDENT, desc=null)


```

Resultat dans la base de donnes h2 :



 jdbc:h2:mem:users\_db

- [-]  **ROLE**
  - [+]  ID
  - [+]  DESCRIPTION
  - [+]  ROLE\_NAME
  - [+]   Indexes
- [-]  **ROLE\_USERS**
  - [+]  ROLES\_ID
  - [+]  USERS\_USER\_ID
  - [+]   Indexes
- [-]  **USERS**
  - [+]  USER\_ID
  - [+]  PASSWORD
  - [+]  USER\_NAME
  - [+]   Indexes
- [+]  INFORMATION\_SCHEMA
- [+]  Sequences
- [+]  Users

 H2 1.4.200 (2019-10-14)

SELECT \* FROM USERS;

USER_ID	PASSWORD	USER_NAME
18121373-b3f2-43f7-b860-0ccda1411c0e	123456	user1
85c561a1-cfd7-48f0-9d07-47b0593f8e10	654321	admin

(2 rows, 9 ms)

Edit

SELECT \* FROM ROLE;

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

(3 rows, 3 ms)

SELECT \* FROM ROLE\_USERS;

ROLES_ID	USERS_USER_ID
3	18121373-b3f2-43f7-b860-0ccda1411c0e
2	85c561a1-cfd7-48f0-9d07-47b0593f8e10
1	18121373-b3f2-43f7-b860-0ccda1411c0e
1	85c561a1-cfd7-48f0-9d07-47b0593f8e10

(4 rows, 7 ms)

### Basculer vers une base de données MySQL au lieu de H2 :

Le bascule depuis une base de donnée de test comme H2 vers une base de donnée Mysql est très simple grace au Spring, il suffit de changer le fichier application.properties ainsi que créer la base de données dans le SGBDR(il suffi de créer la bd sans créer les tables).

```
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.MariaDBDialect
spring.jpa.show-sql=true
```

il faut aussi ajouter la dependance de Mysql dans le fichier pom.xml :

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

ainsi qu'enlever la dependance de H2 du fichier pom.xml

Et on peut voir le table populée dans notre base de donnée

Table	Action	Lignes	Type	Interclassement	Taille	Perte
role	Parcourir Structure Rechercher Insérer Vider Supprimer	3	InnoDB	utf8mb4_general_ci	32,0 kio	-
role_users	Parcourir Structure Rechercher Insérer Vider Supprimer	4	InnoDB	utf8mb4_general_ci	48,0 kio	-
users	Parcourir Structure Rechercher Insérer Vider Supprimer	2	InnoDB	utf8mb4_general_ci	32,0 kio	-
<b>3 tables</b>	<b>Somme</b>	<b>9</b>	<b>InnoDB</b>	<b>utf8mb4_general_ci</b>	<b>112,0 kio</b>	<b>0 0</b>

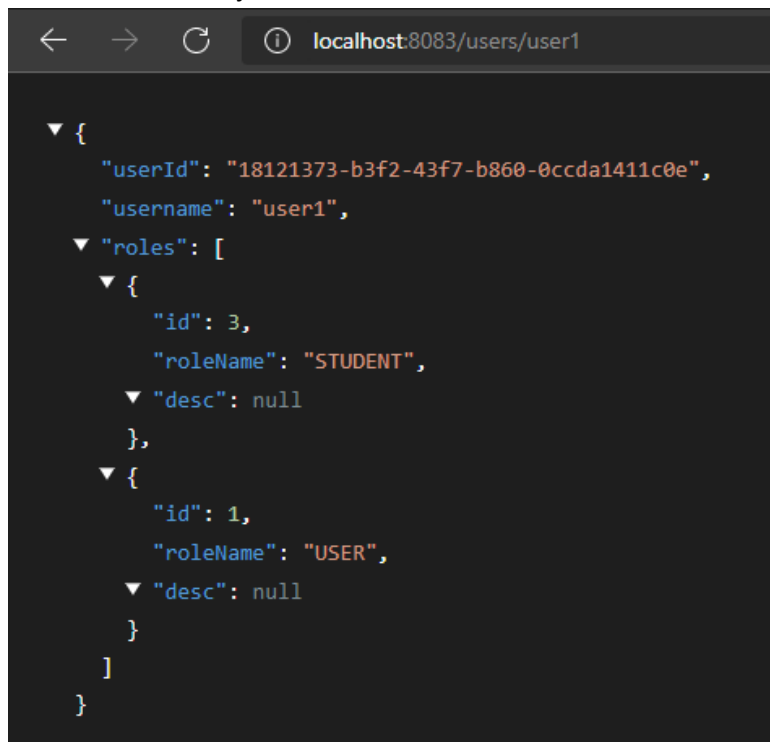
### Créer l'application Web qui permet de chercher les users :

- **Afficher tous les users :**

Pour afficher toutes les patients, tout d'abord il faut créer le controlleur secondaire et dedans on créer une methode qui interroge la base de donnée, recupère la liste des patients, la stocke dans un model et finalement dait appele à la page html convenable.

```
package ma.enset.jpaenset.web;
+import ...
@RestController
public class UserController {
    @Autowired
    private UserService userService;
    @GetMapping("/users/{username}")
    public User user(@PathVariable String username) {
        User user = userService.findUserByUsername(username);
        return user;
    }
}
```

Resultat en format json :



## Conclusion :

Hibernate qui est un Framework open source gérant la persistance des objets en base de données relationnelle. Et aussi pour faciliter notre travail

On a pu utiliser jpa est une interface de programmation Java permettant aux développeurs d'organiser des données relationnelles dans des applications utilisant la plateforme Java. Qui va utiliser Hibernate comme implémentation de cette interface

Mais il est mieux d'utiliser spring data qui a pour objectif de simplifier l'interaction avec différents systèmes de stockage de données et spring data qui va utiliser jpa et hibernate pour faire le travail