

Master Intelligence Artificielle et Analyse des Données

Systèmes Distribués

COMPTE RENDU DU TP2

Réalisé par :

HALIMA DAOUDI

Année universitaire : 2023 – 2024

INTRODUCTION

Pour ce travail pratique, je vais me plonger plus profondément dans le monde du développement avec Spring Boot, une étape essentielle pour affiner mes compétences en programmation et en gestion de bases de données. À travers ce projet, je vais non seulement apprendre à intégrer des dépendances clés comme JPA, H2, Spring Web, et Lombok, mais aussi à manipuler et persister des données de manière efficace.

Ce TP représente une opportunité précieuse de mettre en pratique des concepts théoriques en créant une application de gestion de patients, depuis la définition des modèles de données jusqu'à la migration entre différents systèmes de gestion de bases de données. Voici ce que je vais faire :

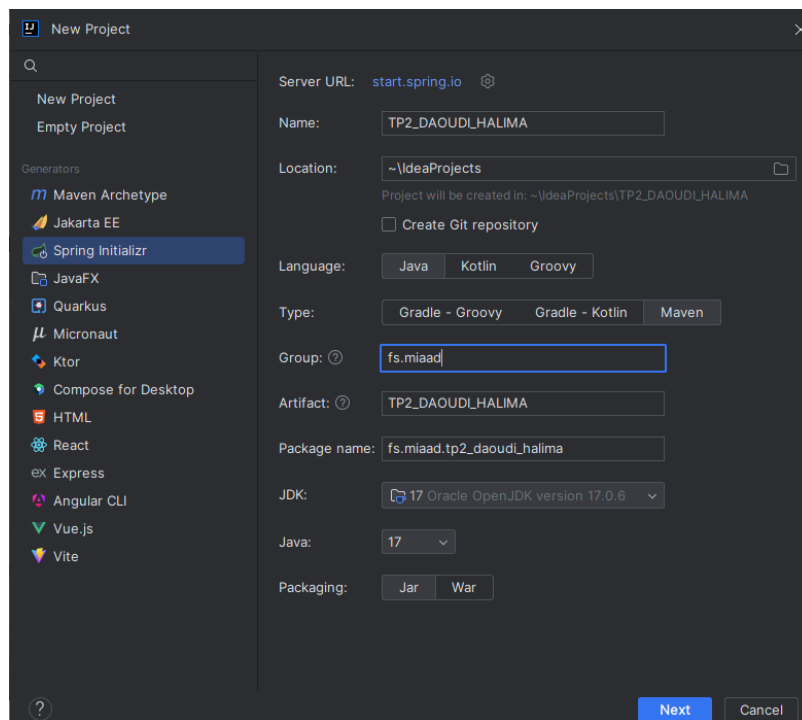
- ✓ Installer IntelliJ Ultimate
- ✓ Créer un projet Spring Initializer avec les dépendances JPA, H2, Spring Web et Lombok
- ✓ Créer l'entité JPA Patient ayant les attributs :
 - id de type Long
 - nom de type String
 - dateNaissance de type Date
 - malade de type boolean
 - score de type int
- ✓ Configurer l'unité de persistance dans le fichier application.properties
- ✓ Créer l'interface JPA Repository basée sur Spring data
- ✓ 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
- ✓ Migrer de H2 Database vers MySQL
- ✓ Reprendre les exemples du Patient, Médecin, rendez-vous, consultation, users et roles

1. Installer IntelliJ Ultimate

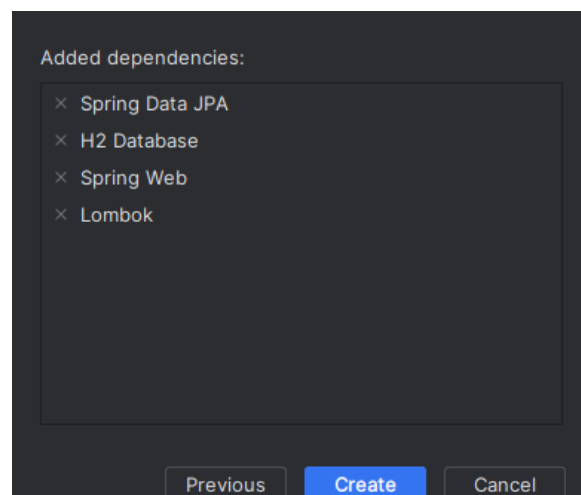
Je commence par télécharger le programme à partir du site officiel. Une fois l'installation terminée, je lance IntelliJ Ultimate

2. Créer un projet Spring Initializer avec les dépendances

Pour créer un projet Spring Initializer dans IntelliJ Ultimate, je démarre IntelliJ et je sélectionne "New Project". Je choisis "Spring Initializr" comme type de projet. Ensuite, je définis le nom et l'emplacement de mon projet, puis je clique sur "Next".



À ce stade, je suis invité à configurer les dépendances de mon projet. Je m'assure de sélectionner les dépendances nécessaires telles que JPA, H2, Spring Web et Lombok. Enfin, je clique sur "Create" pour créer le projet.



3. Créer l'entité JPA Patient :

Dans le package **entities**, je crée une entité JPA nommée **Patient** pour représenter les informations d'un patient. Cette entité a des attributs tels que l'id qui agit comme clé primaire et est généré automatiquement par la base de données et le nom, la date de naissance, l'état de santé (malade), et un score. L'annotation **@Entity** indique que cette classe est une entité JPA associée à une table dans la base de données. Les annotations de Lombok (**@Data**, **@NoArgsConstructor**, **@AllArgsConstructor**) permettent de générer automatiquement des méthodes pour accéder et manipuler les données de l'entité

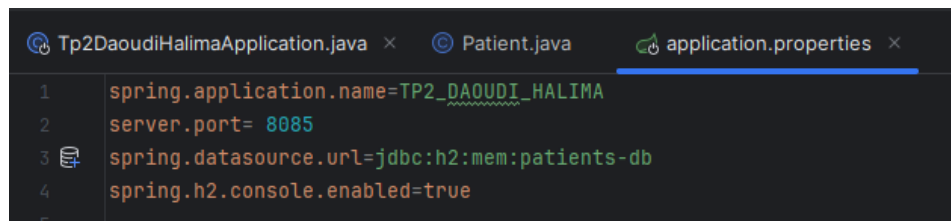
```
package fs.miaad.tp2_daoudi_halima.entities;

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;

@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

Dans le fichier **application.properties**, je configure les paramètres essentiels pour que mon application fonctionne correctement avec la base de données. Cela inclut le nom de l'application, le port sur lequel elle écoute les requêtes, ainsi que l'url de la base de données H2 et l'activation de sa console.

A screenshot of an IDE window showing the 'application.properties' file. The tabs at the top are 'Tp2DaoudiHalimaApplication.java', 'Patient.java', and 'application.properties'. The properties file contains the following configuration:

```
1 spring.application.name=TP2_DAUDI_HALIMA
2 server.port= 8085
3 spring.datasource.url=jdbc:h2:mem:patients-db
4 spring.h2.console.enabled=true
```

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

Dans le package repository, j'ai créé une interface nommée **PatientRepository** qui étend **JpaRepository**. Cette extension permet d'utiliser les méthodes fournies par JpaRepository pour interagir avec la base de données. En plus des opérations CRUD de base, JpaRepository offre également d'autres méthodes pour faciliter la récupération et la manipulation des données. Dans cette interface, j'ai ajouté deux méthodes personnalisées : **findByMalade**(boolean malade) permettant de rechercher des patients en fonction de leur état de santé, et **SupScore**(int sc) qui récupère les patients ayant un score supérieur ou égal à une valeur spécifiée, en utilisant une requête JPQL définie avec l'annotation **@Query**.

```
package fs.miaad.tp2_daoudi_halima.repository;

import fs.miaad.tp2_daoudi_halima.entities.Patient;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

2 usages
public interface PatientRepository extends JpaRepository<Patient, Long> {
    1 usage
    List<Patient> findByMalade(boolean malade);

    // List<Patient> findByScoreGreaterThanOrEqual(int x)
    no usages
    @Query("select p from Patient p where p.score >= :x ")
    List<Patient> SupScore(@Param("x") int sc);
}
```

6. Tester quelques opérations de gestion de patients

J'ai ajouté une classe principale nommée **Tp2DaoudiHalimaApplication** qui est annotée avec **@SpringBootApplication**, ce qui la marque comme une classe de démarrage de l'application Spring Boot.

Implémentant l'interface **CommandLineRunner**, j'ai surchargé la méthode run qui est automatiquement exécutée au démarrage de l'application. À l'intérieur de cette méthode, j'ai utilisé l'injection de dépendances avec **@Autowired** pour obtenir une instance de **PatientRepository**.

```

package fs.miaad.tp2_daoudi_halima;

import ...

@SpringBootApplication
public class Tp2DaoudiHalimaApplication implements CommandLineRunner {

    @Autowired
    private PatientRepository patientRepository;

    public static void main(String[] args) {

        SpringApplication.run(Tp2DaoudiHalimaApplication.class, args);
    }
}

```

J'ai commencé par ajouter quelques patients à la base de données en utilisant la méthode **save()** du **PatientRepository**. Ensuite, j'ai récupéré et affiché la liste de tous les patients en utilisant la méthode **findAll()**. et j'ai consulté un patient spécifique en utilisant son identifiant et affiché ses détails. Aussi, j'ai recherché et affiché la liste des patients malades en utilisant la méthode **findByMalade(true)**.

```

@Override
public void run(String... args) throws Exception {
    patientRepository.save(new Patient(id: null, nom: "HALIMA1", new Date(), malade: true, score: 75));
    patientRepository.save(new Patient(id: null, nom: "HALIMA2", new Date(), malade: false, score: 0));
    patientRepository.save(new Patient(id: null, nom: "HALIMA3", new Date(), malade: true, score: 10));
    // consulter tous les patients
    List<Patient> patients = patientRepository.findAll();
    System.out.println("Liste des patients : ");
    for (Patient patient : patients) {
        System.out.println(patient);
    }
    // Consulter un patient
    Patient p = patientRepository.findById(Long.valueOf(1)).get();
    System.out.println("*****");
    System.out.println(p.getId());
    System.out.println(p.getNom());
    System.out.println(p.getDateNaissance());
    System.out.println(p.isMalade());
    System.out.println(p.getScore());
    System.out.println("*****");
    // Chercher des patients
    List<Patient> patientsByMalades = patientRepository.findByMalade(true);
    System.out.println("Liste des patients malades : ");
    for (Patient patient : patientsByMalades) {
        System.out.println(patient);
    }
}

```

Et j'ai mis à jour les détails d'un patient en modifiant son état de santé et son score, puis j'ai sauvegardé les modifications avec la méthode **save()**. puis j'ai recherché et affiché la liste des patients ayant un score supérieur à 50 en utilisant la méthode **SupScore(50)** personnalisée. Enfin, j'ai supprimé un patient spécifique en utilisant son identifiant et j'ai vérifié le succès de la suppression en vérifiant si le patient n'est plus présent dans la base de données.

```
// Mettre à jour un patient
Patient patientUpdate = patientRepository.findById(Long.valueOf(1: 2)).get();
patientUpdate.setMalade(true);
patientUpdate.setScore(80);
patientRepository.save(patientUpdate);
System.out.println("Patient mis à jour avec succès : " + patientUpdate);

// score sup a 50
List<Patient> patientsScore = patientRepository.SupScore(50);
System.out.println("Liste des patients avec score plus 50 : ");
for (Patient patient : patientsScore) {
    System.out.println(patient);
}

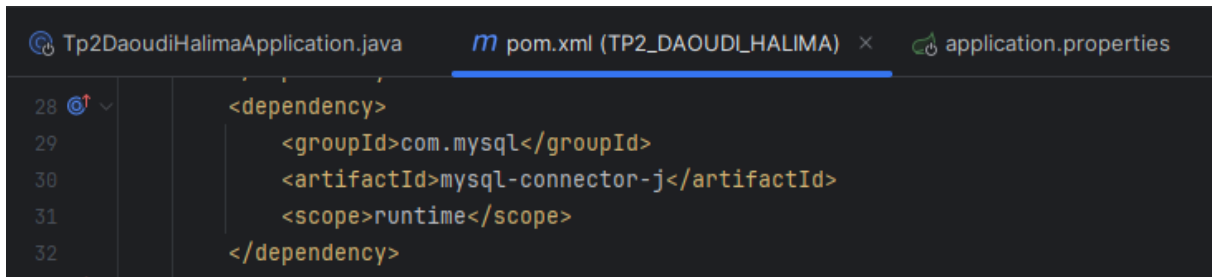
// supprimer un patient
patientRepository.delete(patientRepository.findById(Long.valueOf(1: 3)).get());
if (! patientRepository.findById(Long.valueOf(1: 3)).isPresent()) {
    System.out.println("Patient supprimé avec succès. ");
}
}
```

Voici les **résultats** obtenus lors de l'exécution de cette classe :

```
Tp2DaoudiHalimaApplication in 4.908 seconds (process running for 5.557)
Liste des patients :
Patient(id=1, nom=HALIMA1, dateNaissance=2024-03-23 20:55:39.0, malade=true, score=75)
Patient(id=2, nom=HALIMA2, dateNaissance=2024-03-23 20:55:39.0, malade=false, score=0)
Patient(id=3, nom=HALIMA3, dateNaissance=2024-03-23 20:55:39.0, malade=true, score=10)
*****
1
HALIMA1
2024-03-23 20:55:39.0
true
75
*****
Liste des patients malades :
Patient(id=1, nom=HALIMA1, dateNaissance=2024-03-23 20:55:39.0, malade=true, score=75)
Patient(id=3, nom=HALIMA3, dateNaissance=2024-03-23 20:55:39.0, malade=true, score=10)
Patient mis à jour avec succès : Patient(id=2, nom=HALIMA2, dateNaissance=2024-03-23 20:55:39.0, malade=true, score=80)
Liste des patients avec score plus 50 :
Patient(id=1, nom=HALIMA1, dateNaissance=2024-03-23 20:55:39.0, malade=true, score=75)
Patient(id=2, nom=HALIMA2, dateNaissance=2024-03-23 20:55:39.0, malade=true, score=80)
Patient supprimé avec succès.
```

7. Migrer de H2 Database vers MySQL

Avant de migrer de **H2 Database** vers **MySQL**, je vais d'abord ajouter la dépendance MySQL Connector/J au fichier **pom.xml** de mon projet. Cette configuration spécifie que je veux utiliser le pilote JDBC MySQL dans mon application. La portée "runtime" indique que cette dépendance ne sera nécessaire que lors de l'exécution de mon application, et non lors de la compilation. Une fois que cette dépendance est ajoutée, je serai prêt à migrer mon application de H2 Database vers MySQL.



```
28 <dependency>
29     <groupId>com.mysql</groupId>
30     <artifactId>mysql-connector-j</artifactId>
31     <scope>runtime</scope>
32 </dependency>
```

Je configure les paramètres dans le fichier **application.properties** pour migrer de H2 Database vers MySQL. Cela inclut la définition du nom de l'application et du port, ainsi que les informations de connexion à la base de données MySQL, telles que l'url, le nom d'utilisateur et le mot de passe. Une fois ces configurations ajoutées, l'application est prête à utiliser MySQL comme base de données.



```
spring.application.name=TP2_DAOUDI_HALIMA
server.port= 8085

#Configuration pour MySQL
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

#spring.datasource.url=jdbc:h2:mem:patients-db
#spring.h2.console.enabled=true
```

Après avoir **exécuté** la classe **Tp2DaoudiHalimaApplication**, nous avons vérifié l'état de notre base de données MySQL à l'aide de phpmyadmin. Voici un aperçu de la table patient et des données stockées dans la base de données "**patients_db**". Cela confirme que la migration de H2 vers MySQL s'est déroulée avec **succès**.

✓ Affichage des lignes 0 - 1 (total de 2, traitement en 0,0003 seconde(s).)

```
SELECT * FROM `patient`
```

☐ Profilage [Éditer en ligne] [Éditer] [Expliquer SQL] [Créer le code source PHP] [Actualiser]

☐ Tout afficher | Nombre de lignes : 25 ▼ Filtrer les lignes: Chercher dans cette table Trier par clé :

Options supplémentaires

				id	date_naissance	malade	nom	score
<input type="checkbox"/>	✎ Éditer	📄 Copier	🗑 Supprimer	1	2024-03-23 20:55:39.000000	1	HALIMA1	75
<input type="checkbox"/>	✎ Éditer	📄 Copier	🗑 Supprimer	2	2024-03-23 20:55:39.000000	1	HALIMA2	80

8. Reprendre les exemples

Les entités :

J'ai créé une classe **Patient**, utilisant des annotations JPA et Lombok pour définir ses propriétés, telles que l'ID, le nom, la date de naissance, l'état de santé, et un score. Elle intègre également une relation un-à-plusieurs avec des rendez-vous, gérée de manière paresseuse pour optimiser les performances.

```
package fs.miaad.tp2_daoudi_halima.entities;

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

import java.util.Collection;
import java.util.Date;
10 usages
@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class Patient {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    @Temporal(TemporalType.DATE) // prend la date sans heure
    private Date dateNaissance;
    private boolean malade;
    private int score;
    @OneToMany(mappedBy = "patient", fetch = FetchType.LAZY)
    private Collection<RendezVous> rendezVous ;
}
```

J'ai créé une classe **Medecin** avec des champs pour l'ID, le nom, l'email, et la spécialité, plus une relation un-à-plusieurs avec les rendez-vous, optimisée pour la base de données et la confidentialité des données JSON.

```
package fs.miaad.tp2_daoudi_halima.entities;
import ...
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)
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Collection<RendezVous> rendezVous ;
}
```

J'ai développé une classe **RendezVous** contenant l'ID, la date, le statut, les liens vers un patient et un médecin, et une relation avec une consultation. La classe utilise des annotations pour la gestion des entités et la sécurité JSON.

```
package fs.miaad.tp2_daoudi_halima.entities;
import ...
14 usages
@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class RendezVous {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date date;
    @Enumerated(EnumType.STRING)
    private StatusRDV status;
    @ManyToOne
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Patient patient;
    @ManyToOne
    private Medecin medecin;
    @OneToOne(mappedBy = "rendezVous")
    private Consultation consultation ;
}
```

J'ai ajouté une classe **Consultation** avec un ID, une date de consultation, un rapport, et une association **un-à-un** avec un RendezVous, utilisant des annotations pour la gestion des entités et la confidentialité des données JSON.

```
package fs.miaad.tp2_daoudi_halima.entities;
import ...

12 usages
@Entity
@Data @AllArgsConstructor @NoArgsConstructor
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 ;
}
```

J'ai créé une énumération **StatusRDV** pour définir les statuts possibles d'un rendez-vous : En Cours, Annulé, et Réalisé.

```
package fs.miaad.tp2_daoudi_halima.entities;

1 usage
public enum StatusRDV {
    no usages
    EnCours, Annule, Realise
}
```

Repository :

J'ai mis en place une interface **ConsultationRepository** qui étend **JpaRepository**, me permettant de gérer les données de la classe Consultation dans la base de données sans avoir à écrire de code supplémentaire pour les opérations CRUD.

```
package fs.miaad.tp2_daoudi_halima.repository;

import fs.miaad.tp2_daoudi_halima.entities.Consultation;
import org.springframework.data.jpa.repository.JpaRepository;

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

J'ai créé l'interface **MedecinRepository** qui hérite de **JpaRepository**, permettant ainsi la manipulation des données des médecins. En plus des opérations CRUD automatiques, j'ai ajouté une méthode personnalisée `findByNom` pour rechercher un médecin par son nom.

```
package fs.miaad.tp2_daoudi_halima.repository;

import ...

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

Dans l'interface **PatientRepository**, que j'ai étendue de **JpaRepository**, j'ai inclus des méthodes pour rechercher un patient par son nom et filtrer les patients selon leur état de santé. De plus, j'ai personnalisé une requête, **SupScore**, en utilisant l'annotation **@Query** pour sélectionner les patients dont le score est supérieur ou égal à un seuil spécifié, ce qui me permet une flexibilité accrue dans la gestion des données des patients.

```
package fs.miaad.tp2_daoudi_halima.repository;

import ...

7 usages
public interface PatientRepository extends JpaRepository<Patient, Long> {
    1 usage
    Patient findByNom(String nom);
    no usages
    List<Patient> findByMalade(boolean malade);
    // List<Patient> findByScoreGreaterThan(int x)
    no usages
    @Query("select p from Patient p where p.score >= :x ")
    List<Patient> SupScore(@Param("x") int sc);
}
```

J'ai établi l'interface **RendezVousRepository**, qui hérite de **JpaRepository**, pour faciliter la gestion des entités RendezVous dans la base de données. Cette interface me permet d'effectuer automatiquement les opérations CRUD standards sur les rendez-vous sans nécessiter de code supplémentaire.

```
package fs.miaad.tp2_daoudi_halima.repository;

import ...

5 usages
public interface RendezVousRepository extends JpaRepository<RendezVous, Long> {

}
```

Service :

L'interface **IHospitalService** offre des méthodes pour enregistrer des entités clés comme Patient, Medecin, RendezVous, et Consultation, assurant une gestion centralisée et efficace des données dans mon application hospitalière.

```
package fs.miaad.tp2_daoudi_halima.service;

import ...

1 usage 1 implementation
public interface IHospitalService {
    no usages 1 implementation
    Patient savePatient(Patient patient);
    no usages 1 implementation
    Medecin saveMedecin(Medecin medecin);
    no usages 1 implementation
    RendezVous saveRDV(RendezVous rendezVous);
    no usages 1 implementation
    Consultation saveConsultation(Consultation consultation);
}
```

Ma classe **IHospitalServiceImpl** réalise l'interface **IHospitalService**, gérant la sauvegarde des entités comme les patients et les médecins via des repositories. Elle assure l'intégrité des données avec le support transactionnel de Spring.

```
package fs.miaad.tp2_daoudi_halima.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 p, MedecinRepository m, RendezVousRepository r, ConsultationRepository c) {
        this.patientRepository = p;
        this.medecinRepository = m;
        this.rendezVousRepository = r;
        this.consultationRepository = c;
    }
}
```

```
no usages
@Override
public Patient savePatient(Patient patient) { return patientRepository.save(patient); }

no usages
@Override
public Medecin saveMedecin(Medecin medecin) {
    return medecinRepository.save(medecin);
}

no usages
@Override
public RendezVous saveRDV(RendezVous rendezVous) {
    return rendezVousRepository.save(rendezVous);
}

no usages
@Override
public Consultation saveConsultation(Consultation consultation) {
    return consultationRepository.save(consultation);
}
}
```

La classe principale :

Mon application **Tp2DaoudiHalimaApplication**, développée avec Spring Boot, Elle utilise des repositories et l'interface service pour interagir avec la base de données, assurant la persistance des données

```

package fs.miaad.tp2_daoudi_halima;

import ...

@SpringBootApplication
public class Tp2DaoudiHalimaApplication implements CommandLineRunner {

    @Autowired
    private IHospitalService iHospitalService;
    @Autowired
    private PatientRepository patientRepository;
    @Autowired
    private MedecinRepository medecinRepository;
    @Autowired
    private RendezVousRepository rendezVousRepository;
    @Autowired
    private ConsultationRepository consultationRepository;

    public static void main(String[] args) {

        SpringApplication.run(Tp2DaoudiHalimaApplication.class, args);
    }
}

```

J'initialise la base de données avec des patients et des médecins, en utilisant des noms prédéfinis. Les patients sont créés avec des dates de naissance et marqués comme non malades. Les médecins sont assignés aléatoirement à des spécialités entre "Cardio" et "Dentiste". Ensuite, un rendez-vous est programmé entre un patient et un médecin spécifiques, et une consultation est enregistrée pour ce rendez-vous, complétée par un rapport. Cette séquence d'actions démontre l'interaction entre différentes entités et la gestion des données à travers les services de l'application.

```

@Override
public void run(String... args) throws Exception {
    Stream.of(...values: "HALIMA1", "DAOUDI1", "HALIMA2", "DAOUDI2").forEach(name->{
        Patient p = new Patient();
        p.setNom(name);
        p.setDateNaissance(new Date());
        p.setMalade(false);
        iHospitalService.savePatient(p);
    });
    Stream.of(...values: "YASMINE", "AMINE", "HANANE", "Youssef").forEach(name->{
        Medecin m = new Medecin();
        m.setNom(name);
        m.setEmail(name+"@gmail.com");
        m.setSpecialite(Math.random()>0.5 ? "Cardio" : "Dentiste");
        iHospitalService.saveMedecin(m);
    });
    Patient patient = patientRepository.findByNom("HALIMA2");
    Medecin medecin = medecinRepository.findByNom("YASMINE");

    RendezVous rendezVous = new RendezVous();
    rendezVous.setDate(new Date());
    rendezVous.setStatus(StatusRDV.EnCours);
    rendezVous.setMedecin(medecin);
    rendezVous.setPatient(patient);
    iHospitalService.saveRDV(rendezVous);
}

```

```

RendezVous rv = rendezVousRepository.findById(1L).orElse( other: null);
Consultation consultation = new Consultation();
consultation.setDateConsultation(rv.getDate());
consultation.setRendezVous(rv);
consultation.setRapport("Rapport de la consultation...");
iHospitalService.saveConsultation(consultation);

```

RestController :

Dans le package **web**, j'ai créé un contrôleur REST nommé **PatientRestService**. Qui utilise l'annotation **@RestController** pour indiquer qu'il s'agit d'un contrôleur REST. Il contient **deux** méthodes **GET**, La première renvoie la liste de tous les patients stockés dans la base de données, et la deuxième renvoie les détails d'un patient spécifique identifié par son ID.

```

package fs.miaad.tp2_daoudi_halima.web;

import fs.miaad.tp2_daoudi_halima.entities.Patient;
import fs.miaad.tp2_daoudi_halima.repository.PatientRepository;
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;

import java.util.List;

@RestController
public class PatientRestService {

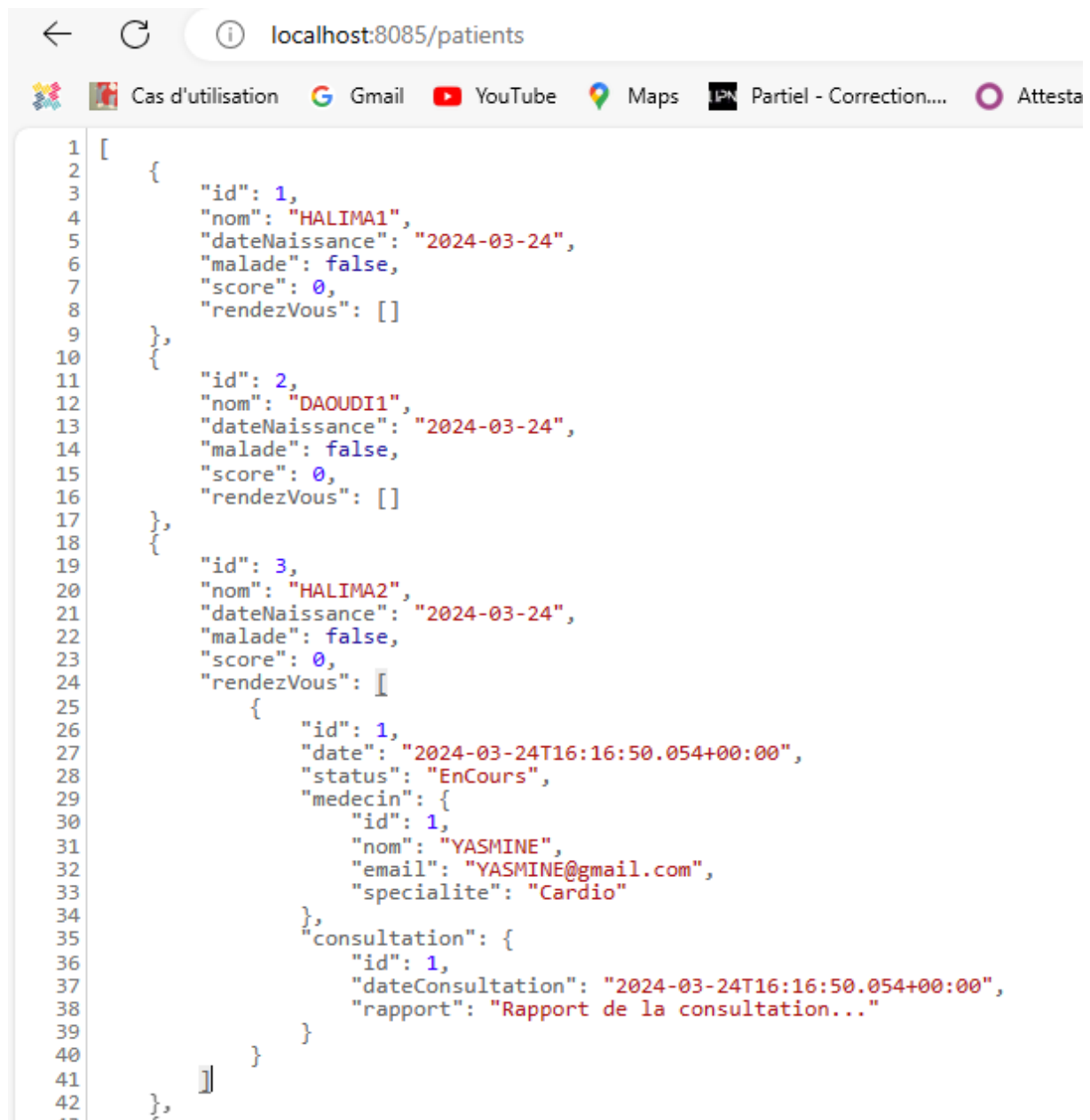
    @Autowired
    private PatientRepository patientRepository;

    @GetMapping("/patients")
    public List<Patient> patients(){
        return patientRepository.findAll();
    }

    @GetMapping("/patient/{id}")
    public Patient getPatientById(@PathVariable Long id){
        return patientRepository.findById(id).orElse( other: null);
    }
}

```


Après avoir écrit en URL "**http://localhost:8085/patients**" et envoyé la requête **GET**, le contrôleur REST **PatientRestService** répond en affichant la liste de tous les patients stockés dans la base de données au format **JSON**.



```

1  [
2    {
3      "id": 1,
4      "nom": "HALIMA1",
5      "dateNaissance": "2024-03-24",
6      "malade": false,
7      "score": 0,
8      "rendezVous": []
9    },
10   {
11     "id": 2,
12     "nom": "DAOUDI1",
13     "dateNaissance": "2024-03-24",
14     "malade": false,
15     "score": 0,
16     "rendezVous": []
17   },
18   {
19     "id": 3,
20     "nom": "HALIMA2",
21     "dateNaissance": "2024-03-24",
22     "malade": false,
23     "score": 0,
24     "rendezVous": [
25       {
26         "id": 1,
27         "date": "2024-03-24T16:16:50.054+00:00",
28         "status": "EnCours",
29         "medecin": {
30           "id": 1,
31           "nom": "YASMINE",
32           "email": "YASMINE@gmail.com",
33           "specialite": "Cardio"
34         },
35         "consultation": {
36           "id": 1,
37           "dateConsultation": "2024-03-24T16:16:50.054+00:00",
38           "rapport": "Rapport de la consultation..."
39         }
40       }
41     ]
42   },
43 ]

```

User et Role Project :

Ma classe **User** représente les utilisateurs avec des identifiants uniques, des noms d'utilisateurs, et des mots de passe cachés lors de la sérialisation JSON. Elle établit une relation **ManyToMany** avec des rôles pour gérer les permissions, stockant ces informations dans une table "users".

```
package fs.miaad.application.entities;
import ...
24 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<>();
}
```

Ma classe **Role** définit les rôles avec un ID, un nom unique, et une description. Elle lie les rôles aux utilisateurs via une relation **ManyToMany**, excluant les utilisateurs de la sérialisation JSON pour la sécurité.

```
package fs.miaad.application.entities;
import ...
16 usages
@Entity
@Table(name="roles")
@Data @NoArgsConstructor @AllArgsConstructor
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)
    @ToString.Exclude
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private List<User> users = new ArrayList<>();
}
```

J'ai créé l'interface **UserRepository** dans le package **Repositories**. Elle hérite de **JpaRepository** et ajoute une méthode personnalisée, **findByUsername**, pour rechercher un utilisateur par son nom d'utilisateur.

```
package fs.miaad.application.Repositories;
import ...
2 usages
public interface UserRepository extends JpaRepository<User, String> {
    2 usages
    User findByUsername(String username);
}
```

J'ai créé l'interface **RoleRepository** dans le package **Repositories**. Elle hérite de **JpaRepository** et ajoute une méthode personnalisée, **findByName**, pour rechercher un rôle par son nom.

```
package fs.miaad.application.Repositories;
import ...
2 usages
public interface RoleRepository extends JpaRepository<Role, Long> {
    1 usage
    Role findByName(String roleName);
}
```

J'ai créé l'interface **UserService** dans le package **service**. Cette interface définit des méthodes pour ajouter de nouveaux utilisateurs et rôles, trouver un utilisateur par son nom d'utilisateur ou un rôle par son nom, attribuer un rôle à un utilisateur et authentifier un utilisateur avec un nom d'utilisateur et un mot de passe.

```
package fs.miaad.application.service;
import ...
3 usages 1 implementation
public interface UserService {

    2 usages 1 implementation
    User addNewUser(User user);
    1 usage 1 implementation
    Role addNewRole(Role role);
    1 usage 1 implementation
    User findUserByUsername(String username);

    1 usage 1 implementation
    Role findRoleByName(String roleName);
    4 usages 1 implementation
    void addRoleToUser(String username , String roleName);
    1 usage 1 implementation
    User authenticate(String username , String password);
}
```

Dans le package **service**, j'ai créé **UserServiceImpl** pour gérer les utilisateurs et les rôles. Cette classe utilise les repositories **UserRepository** et **RoleRepository** pour accéder à la base de données. Elle inclut des méthodes pour ajouter, rechercher et attribuer des utilisateurs et des rôles. Elle est annotée avec **@Service** et **@Transactional** pour la gestion des transactions. A l'aide des injections de **UserRepository** et **RoleRepository**, je peux effectuer des opérations CRUD. Parmi les méthodes clés, j'ajoute de nouveaux utilisateurs et rôles à la base

```
package fs.miaad.application.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);
    }
}
```

Et je trouve des utilisateurs et des rôles par leurs noms, j'attribue des rôles aux utilisateurs

```
@Override
public User findUserByUsername(String username) {
    return userRepository.findByUsername(username);
}
1 usage
@Override
public Role findRoleByRoleName(String roleName) {
    return roleRepository.findByRoleName(roleName);
}
4 usages
@Override
public void addRoleToUser(String username, String roleName) {
    User user = findUserByUsername(username);
    Role role = findRoleByRoleName(roleName);
    if (user != null && role != null) {
        user.getRoles().add(role);
        role.getUsers().add(user);
    }
}
}
```

J'authentifie les utilisateurs en vérifiant leur nom et mot de passe, renvoyant l'utilisateur si les informations sont correctes, ou lançant une exception en cas d'identifiants incorrects.

```
@Override
public User authenticate(String username, String password) {
    User user=userRepository.findByUsername(username);
    if(user==null) throw new RuntimeException("Bad credentials");
    if(user.getPassword().equals(password)){
        return user;
    }
    throw new RuntimeException("Bad credentials");
}
}
```

La classe principale

Dans ma classe principale de l'application Spring Boot, je configure le démarrage initial en utilisant **CommandLineRunner** pour exécuter du code au lancement. J'injecte le **UserService** avec **@Autowired** pour accéder aux fonctionnalités de gestion des utilisateurs. Au démarrage, je crée et ajoute deux **utilisateurs**, "halima" et "admin", en définissant leurs noms d'utilisateur et mots de passe,

```
package fs.miaad.application;
import ...
@SpringBootApplication
public class Tp2SuiteDaoudiHalimaApplication implements CommandLineRunner {

    public static void main(String[] args) { SpringApplication.run(Tp2SuiteDaoudiHalimaApplication.class, args); }

    @Autowired
    private UserService userService ;
    @Override
    public void run(String... args) throws Exception {
        User u = new User();
        u.setUsername("halima");
        u.setPassword("123456");
        userService.addNewUser(u);

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

Ensuite, je crée et ajoute les **rôles** "STUDENT", "USER", et "ADMIN" à la base via **userService**, en les parcourant avec un stream.

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

Après, j'attribue les rôles aux utilisateurs et teste l'authentification de "halima", affichant ensuite son ID, nom d'utilisateur, et rôles.

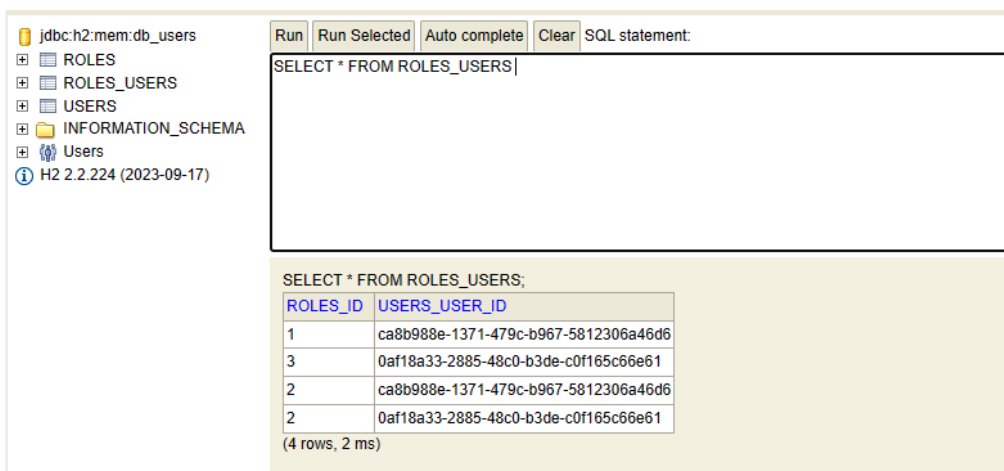
```
userService.addRoleToUser( username: "halima" , roleName: "STUDENT");
userService.addRoleToUser( username: "halima" , roleName: "USER");
userService.addRoleToUser( username: "admin" , roleName: "ADMIN");
userService.addRoleToUser( username: "admin" , roleName: "USER");

try {
    User user = userService.authenticate( username: "halima", password: "123456" );
    System.out.println(user.getUserId());
    System.out.println(user.getUsername());
    user.getRoles().forEach(r ->{
        System.out.println("Role => " +r.toString());
    });
}catch (Exception exception){
    exception.printStackTrace();
}
}
```

J'ai configuré le fichier applictaions.properties pour mon application Spring. J'ai défini le nom de l'application comme "TP2Suite_DAOUDI_HALIMA", le port d'écoute des requêtes HTTP sur 8085, activé la console H2 pour la gestion de la base de données, et spécifié l'URL de la base de données pour utiliser H2 en mémoire avec le nom "db_users"

```
spring.application.name=TP2Suite_DAOUDI_HALIMA
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:db_users
server.port=8085
```

Voici le résultat de l'exécution de mon application, montrant les sorties dans le terminal ainsi que les données stockées dans la base de données H2.



The screenshot shows the H2 database console interface. On the left, a tree view displays the database structure: jdbc:h2:mem:db_users, ROLES, ROLES_USERS, USERS, INFORMATION_SCHEMA, and Users. The main area shows the SQL statement 'SELECT * FROM ROLES_USERS;' and its results. The results are displayed in a table with two columns: ROLES_ID and USERS_USER_ID. There are four rows of data, with a total execution time of 2 ms.

ROLES_ID	USERS_USER_ID
1	ca8b988e-1371-479c-b967-5812306a46d6
3	0af18a33-2885-48c0-b3de-c0f165c66e61
2	ca8b988e-1371-479c-b967-5812306a46d6
2	0af18a33-2885-48c0-b3de-c0f165c66e61

(4 rows, 2 ms)

```

Tp2SuiteDaoudiHalimaApplication in 4.622 seconds (process running for 5.307)
ca8b988e-1371-479c-b967-5812306a46d6
halima
Role => Role(id=1, roleName=STUDENT, desc=null)
Role => Role(id=2, roleName=USER, desc=null)

```

Pour **migrer vers XAMPP**, j'ajoute cette dépendance MySQL à mon fichier pom.xml.

```

<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>

```

Et je configure ma connexion à **MySQL** dans **application.properties**, en spécifiant le nom de l'application, le port du serveur, l'URL de la base de données, le nom d'utilisateur, le mot de passe, la stratégie de création du schéma et le dialecte Hibernate à utiliser.

```

spring.application.name=TP2Suite_DAOUDI_HALIMA
server.port=8085
#Configuration pour MySQL
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

```

Voici le résultat après migration vers **MySQL**, avec un aperçu de la table **roles_users**.

← Serveur : 127.0.0.1 » Base de données : users_db » Table : roles_users

Parcourir Structure SQL Rechercher Insérer Exporter In

⚠ La sélection courante ne contient pas de colonne unique. Les grilles d'édition, les cases à cocher ainsi qu

✓ Affichage des lignes 0 - 3 (total de 4, traitement en 0,0002 seconde(s).)

SELECT * FROM `roles_users`

☐ Profilage [Éditer en ligne] [Éditer] [Expliquer SQL] [Créer le code source PHP] [Actualiser]

☐ Tout afficher | Nombre de lignes : 25 | Filtrer les lignes: Chercher dans cette table

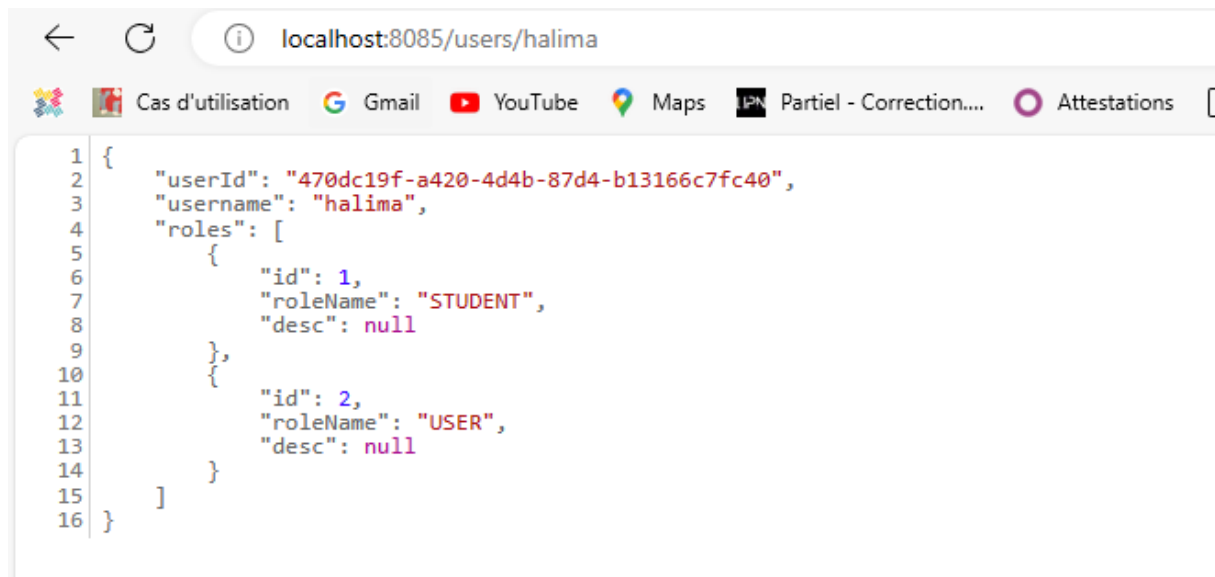
Options supplémentaires

roles_id	users_user_id
1	058ba0fc-d49c-4657-ba4d-24e5175fc60c
3	1750c37b-2658-463f-aebe-9acaf4f17fac
2	058ba0fc-d49c-4657-ba4d-24e5175fc60c
2	1750c37b-2658-463f-aebe-9acaf4f17fac

J'ai créé une classe **UserController**, marquée avec **@RestController**. En utilisant **@Autowired**, j'injecte le **UserService** pour accéder aux opérations sur les utilisateurs. J'ai ensuite défini une méthode mappée à une requête **GET** sur le chemin **/users/{username}**, qui prend le nom d'utilisateur en paramètre et utilise le service pour trouver et retourner les informations de cet utilisateur.

```
package fs.miaad.application.web;
import ...
@RestController
public class UserController {
    @Autowired
    private UserService userService;
    @GetMapping("/users/{username}")
    public User user(@PathVariable String username){
        return userService.findUserByUserName(username);
    }
}
```

Voici le résultat de la recherche d'un utilisateur nommé "halima" via notre API REST, accessible à l'adresse **localhost:8085/users/halima**



CONCLUSION

En concluant ce TP, j'ai significativement élargi ma compréhension et ma maîtrise de Spring Boot ainsi que des bases de données H2 et MySQL. J'ai appris à créer et configurer une entité JPA, à manipuler des données avec Spring Data JPA, et à effectuer une migration de base de données.

Ce projet m'a permis de consolider mes compétences en développement logiciel en abordant des aspects pratiques et théoriques du travail avec des applications Spring Boot.

Je me sens désormais équipé d'une base solide pour développer des applications robustes et performantes, marquant une progression significative dans mon parcours de développement logiciel.