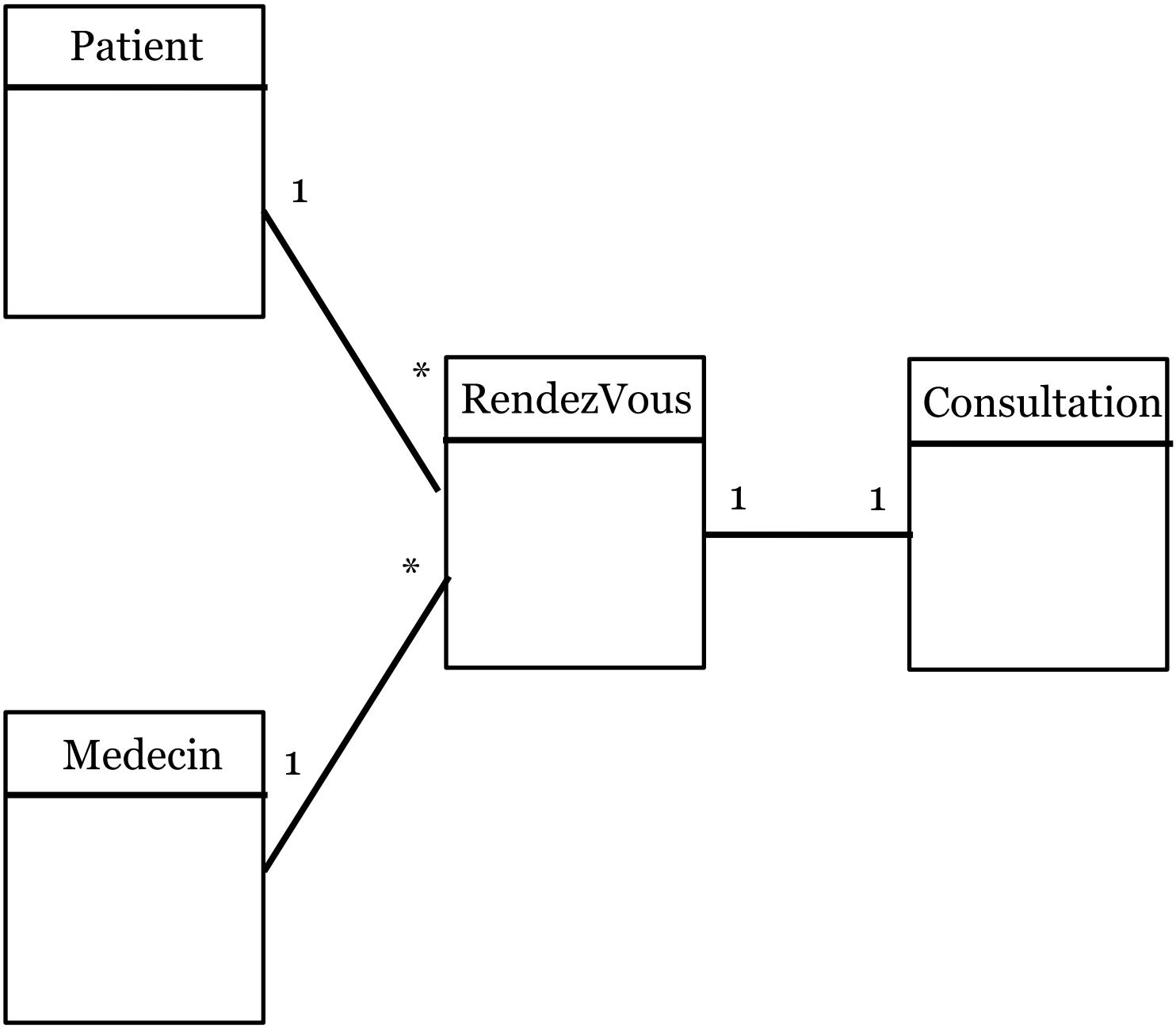
Mapping des associations Cas de l'hospital

AZEHAF Issam

II-BDCC2

Introduction

Ce tp consiste a traiter les différents cas de relation entre les entités, Voici le digramme de classe qu'on va traiter



Démonstration

Tout d'abord on va commencer par la création et le mapping des classes

```java
@Entity
@Data @AllArgsConstructor @NoArgsConstructor
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;
}
```

Figure 1 : Classe Patient

```java
@Entity
@Data @AllArgsConstructor @NoArgsConstructor
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;

}
```

Figure 2 : Classe Medecin

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

}
```

Figure 3 : Classe Consultation

```java
@Entity
@Data @AllArgsConstructor @NoArgsConstructor
public class RendezVous {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Temporal(TemporalType.DATE)
    private Date date;
    private StatusRDV annule;
    @ManyToOne
    private Patient patient;
    @ManyToOne
    private Medecin medecin;
    @OneToOne(mappedBy = "rendezVous")
    private Consultation consultation;
}
```

Figure 4 : Classe RendezVous

Relation entre les classes :

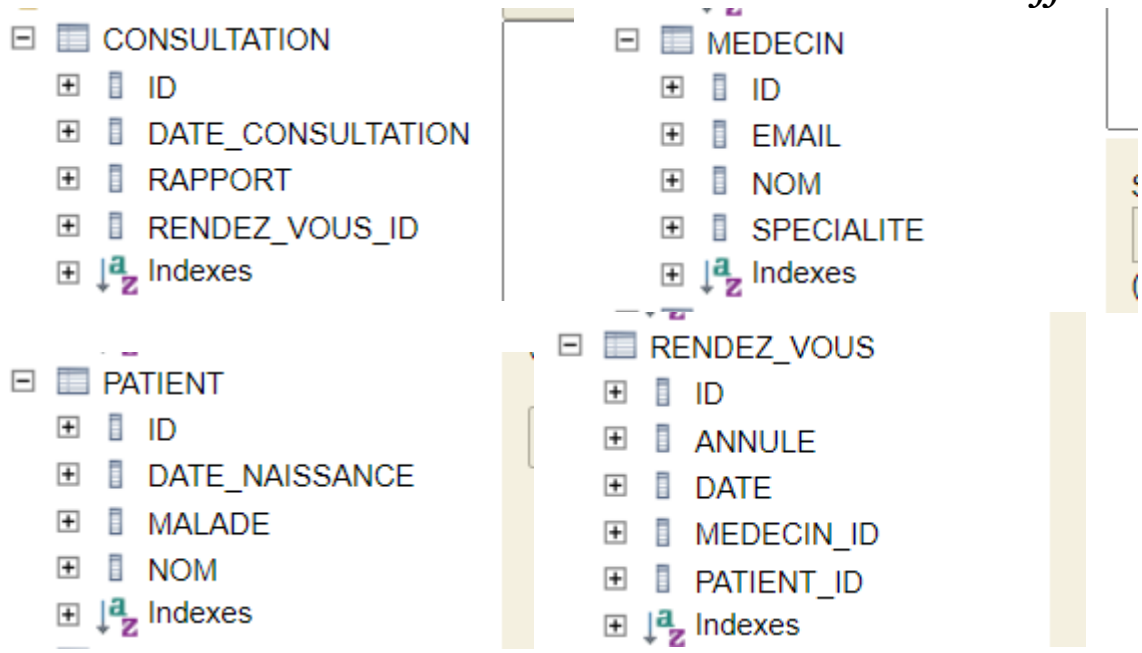
- *Entre la classe Patient et RendezVous on a une relation OneToMany, donc un patient peut avoir plusieurs RendezVous, et un rendezVous concerne un seul patient.*
- *Entre la classe Medecin et RendezVous on a une relation OneToMany, donc un Medecin peut traiter plusieurs RendezVous, et un rendezVous concerne un seul Medecin.*
- *Entre la classe RendezVous et Consultation on a une relation OneToOne, une consultation concerne un seul rendezVous et la meme chose pour rendezVous*

Passons Verifiant si notre mapping est bien effectué, pour cela on utilise un H2 database

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

Figure 5 : H2 Database

Donc les classes et les relations entre eux one été bien effectué



Passant faire quelque exemples d'application
Tout d'abord on va créer nos repositories qui vont extends de l'interface JPARepository qui offre des methode bien définis

```java
public interface PatientRepository extends JpaRepository<Patient, Long> {
    Patient findByNom(String nom);
}
```

Figure 6 : PatientRepository

```java
public interface MedecinRepository extends JpaRepository<Medecin, Long> {
    Medecin findByNom(String nom);
}
```

Figure 7 : MedecinRepository

```java
public interface RendezVousRepository extends JpaRepository<RendezVous, String> {
}
```

Figure 8 : RendezVousRepository

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

Figure 9 : ConsultationRepository

Après on va créer la couche métier qui contient une interface IHospitalService et son implémentation

```java
public interface IHospitalService {
    Patient savePatient(Patient patient);
    Medecin saveMedecin(Medecin medecin);
    RendezVous saveRDV(RendezVous rendezVous);
    Consultation saveConsultation(Consultation consultation);
}
```

Figure 10 : IHospitalService

Donc on va implementer ces methodes dans la classe HospitalServiceImpl

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

```java
@Override
public Medecin saveMedecin(Medecin medecin) {
    return medecinRepository.save(medecin);
}
```

```java
@Override
public RendezVous saveRDV(RendezVous rendezVous) {
    rendezVous.setId(UUID.randomUUID().toString());
    return rendezVousRepository.save(rendezVous);
}
```

```java
@Override
public Consultation saveConsultation(Consultation consultation) {
    return consultationRepository.save(consultation);
}
```

Pour tester ces méthode dans notre application on va remplir notre base de donnée
Tout d'abord on va injecter les interface

```java
@Bean
CommandLineRunner start(IHospitalService iHospitalService,
                        PatientRepository patientRepository,
                        RendezVousRepository rendezVousRepository,
                        MedecinRepository medecinRepository){
    return args -> {
```

Figure 11 : Injection des interfaces

Après on va créer des patients

```
return args -> {
    Stream.of("issam","younes","wiam").forEach(name->{
        Patient patient = new Patient();
        patient.setNom(name);
        patient.setDateNaissance(new Date());
        patient.setMalade(false);
        iHospitalService.savePatient(patient);
    });
```

Figure 12 : création des patients

Ensuite la création des medecins

```
Stream.of("ayman","hanane","yasmine").forEach(name->{
    Medecin medecin = new Medecin();
    medecin.setNom(name);
    medecin.setSpecialite(Math.random()>0.5?"Cardio":"Dentiste");
    medecin.setEmail(name+"@gmail.com");
    iHospitalService.saveMedecin(medecin);
});
```

Figure 13 : Création des medecin

On va créer un rendezVous et l'affecter a un patient et un medecin

```
Patient patient = patientRepository.findById(1L).orElse( other: null);
Patient patient1 = patientRepository.findByNom("issam");

Medecin medecin = medecinRepository.findByNom("ayman");

RendezVous rendezVous = new RendezVous();
rendezVous.setDate(new Date());
rendezVous.setAnnule(StatusRDV.DONE);
rendezVous.setMedecin(medecin);
rendezVous.setPatient(patient1);
iHospitalService.saveRDV(rendezVous);
```

Figure 14 : Création d'un rendezVous

Et après on va créer une consultation qui concerne un rendezVous

```
Consultation consultation = new Consultation();
RendezVous rendezVous1 = rendezVousRepository.findAll().get(0);
consultation.setDateConsultation(rendezVous1.getDate());
consultation.setRendezVous(rendezVous1);
consultation.setRapport("Rapport de la consultation ....");
iHospitalService.saveConsultation(consultation);
```

Figure 15 : création d'une consultation

On execution notre application on vois bien dans la base de donnée les données.

SELECT * FROM PATIENT;

ID	DATE_NAISSANCE	MALADE	NOM
1	2022-03-31	FALSE	issam
2	2022-03-31	FALSE	younes
3	2022-03-31	FALSE	wiam

(3 rows, 2 ms)

SELECT * FROM MEDECIN;

ID	EMAIL	NOM	SPECIALITE
1	ayman@gmail.com	ayman	Cardio
2	hanane@gmail.com	hanane	Dentiste
3	yasmine@gmail.com	yasmine	Dentiste

(3 rows, 3 ms)

SELECT * FROM CONSULTATION;

ID	DATE_CONSULTATION	RAPPORT	RENDEZ_VOUS_ID
1	2022-03-31 00:00:00	Rapport de la consultation	c36888ff-7a93-4238-a676-b04f19e44f73

(1 row, 2 ms)

SELECT * FROM RENDEZ_VOUS;

ID	ANNULE	DATE	MEDECIN_ID	PATIENT_ID
c36888ff-7a93-4238-a676-b04f19e44f73	DONE	2022-03-31	1	1

(1 row, 1 ms)

Par exemple souhaitons afficher ces données dans une page web pour cela on va créer un controller qui contient une méthode qui va retourner la liste des patients

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

Puisque ce sont des relations bidirectionnelle on va utiliser l'annotaion JsonProperty pour qu'on evite a chaque fois d'avoir plusieurs enregistrements

```java
@JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
```

Et voila les données sont bien affiché dans notre page web