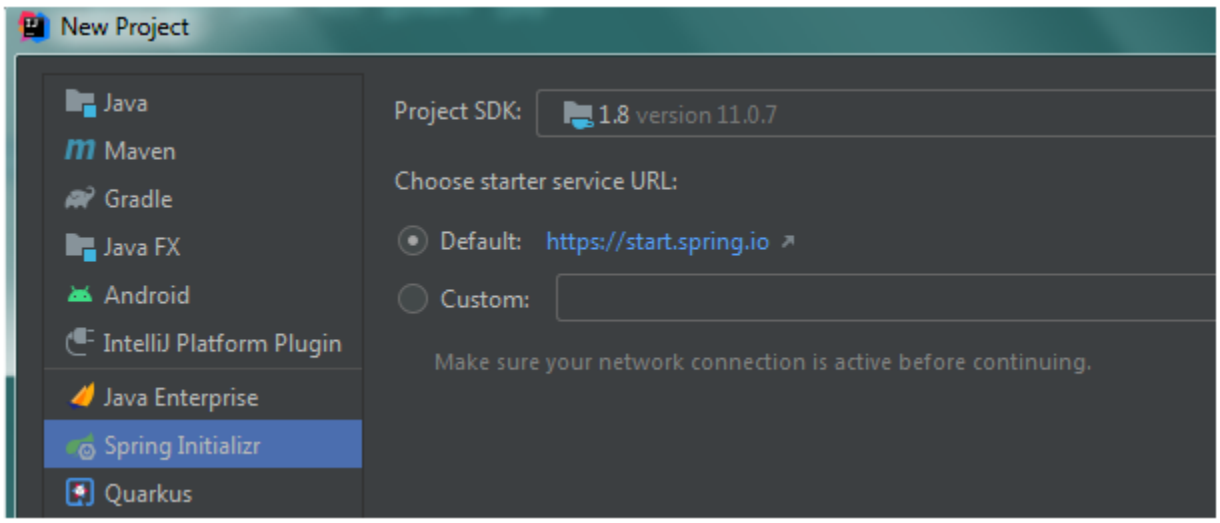
Spring Data JPA.

- Pour utiliser les bases de données relationnelles dans une application Java on a besoin de l'API JDBC (Java Data Base Connection).
- Pour bien géré l'accès aux données il faut faire ce qu'on appelle le Mapping Objet Relationnel (faire la relation entre les tables de la bases de données et les objets Java) en utilisant un framework (ex : Hibarenate).
- Hibernate est un ORM (Object-Relational Mapping) qui implémente la spécification (API) JPA (Java Persistance API) qui permet de standardiser tous ce qu'est Mapping Objet Relationnel.
- Autres implémentations de JPA : TopLink, EclipseLink …
- Spring Data est module de Spring qui va faciliter (à l'extrème) l'utilisation de JPA, il peut etre utilisé avec les bases de données relationnelles (MySQL, Oracle …) et non-relationnelles (MongoDB, Redis …).
- Avec les bases de données relationnelles on va utiliser un module de Spring qui s'appelle Spring Data JPA qui va faire l'ORM basé sur JPA (JPA a été faite pour les bases de données relationnelles).

1- Création de projet Spring

On va créer un projet Spring Boot avec Spring Initializr.

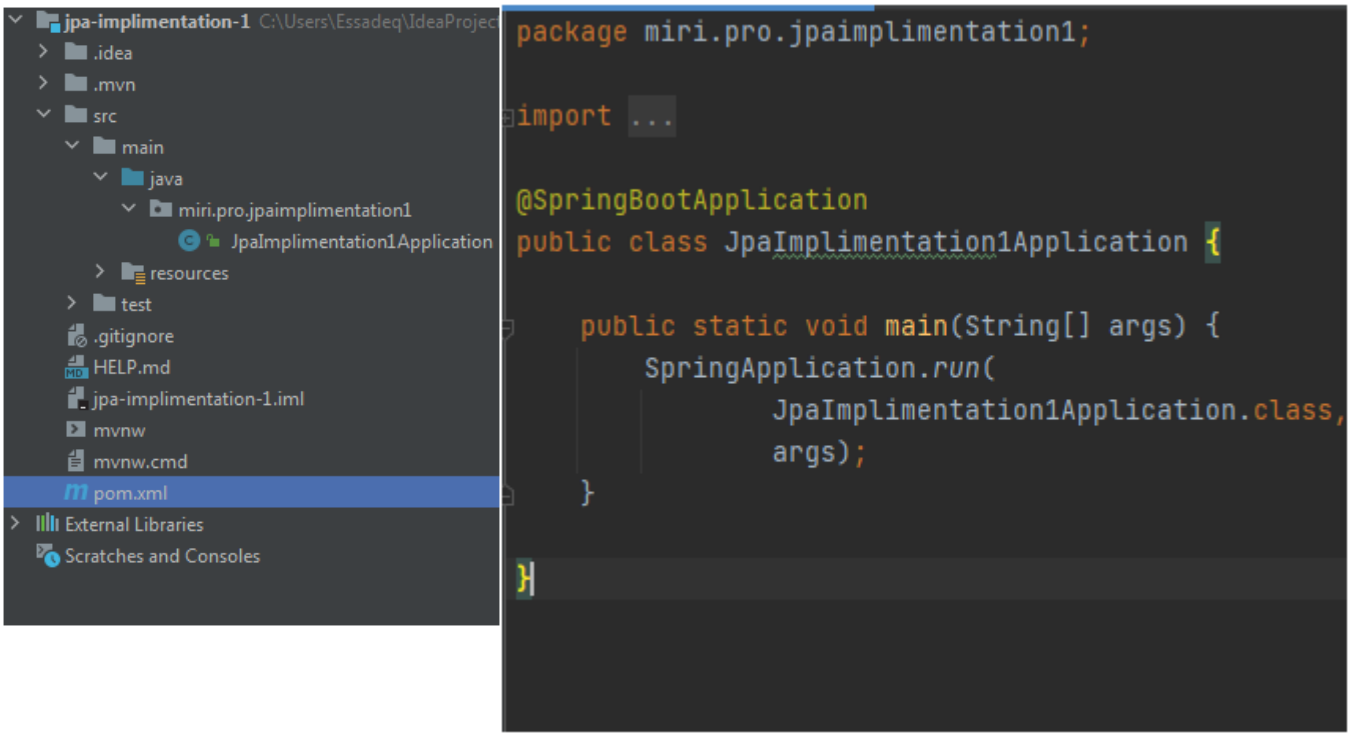


Spring initialzr va faciliter la configuration et réduire l'utilisation des fichiers de configurations.

Les dépendances à télécharger :

- **Spring Data JPA (JPA, Hibernate (par defaut), Spring Data).**
- **H2 Database (SGBD à mémoire).**
- **Spring Web (Spring MVC).**
- **Lombok (Générer les getters et les setters).**

Voilà la structure initiale de l'application :



- Lorsque on démarre l'application donc le conteneur Spring Boot qui va démarrer en premier.
- Le fichier **resources/application.properties** est le fichier de configuration de l'application.
- Supposant que nous allons créer une application de gestion des étudiants, on va créer un package (entities) et à l'intérieur on va déclarer la classe Student.
- Il faut s'assurer que le plugin Lombok est installé dans l'environnement ().
- L'annotation @Data de Lombok permet de générer les getters et les setters de l'ntité.
- Pour qu'elle soit une entité JPA, il faut ajouter l'annotation @ Entity et @ID (Clé primaire) dans la classe.

```java
@Entity // make a JPA entity
@Data // (lombok) generate getters and setters + non-args constructor
@AllArgsConstructor // generate all args constructor
@NoArgsConstructor // generate no args constructor
public class Student {
    @Id // primary key is necessary
    // The value of id will be generated
    // AUTO-INCREMENT
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private Date birthDay;
    private int age;
    private boolean graduated;


}
```

- Notre application va utiliser JPA indirectement : il suffit de créer pour chaque entité une interface EntityRepository qui hérite de l'interface (extends) **JpaRepository** dans le package repositories.
- Spring Boot va automatiquement implémenter les fonctions définies dans cette interface d'une manière extraordinaire juste par le nom de la fonction.

```java
package miri.pro.jpaimplimentation1.repositories;

import miri.pro.jpaimplimentation1.entites.Student;
import org.springframework.data.jpa.repository.JpaRepository;

//JpaRepository<ManagedEntity, PrimaryKeyType>
public interface  StudentRepository extends JpaRepository<Student, Long> {
    // Now without adding any thing, we have the basic functions to deal
    // with the database findAll(), findById(), findAllById() ...

}
```

- Maintenant on a toutes les fonctions de bases pour agir avec la base de données.

- On peut tester ça dans la classe de base (Spring boot container), mais avant il faut configurer la base de données dans le fichier **application.properties**.

```
- spring.datasource.url=jdbc:h2:mem:students-spring-jpa-db
```

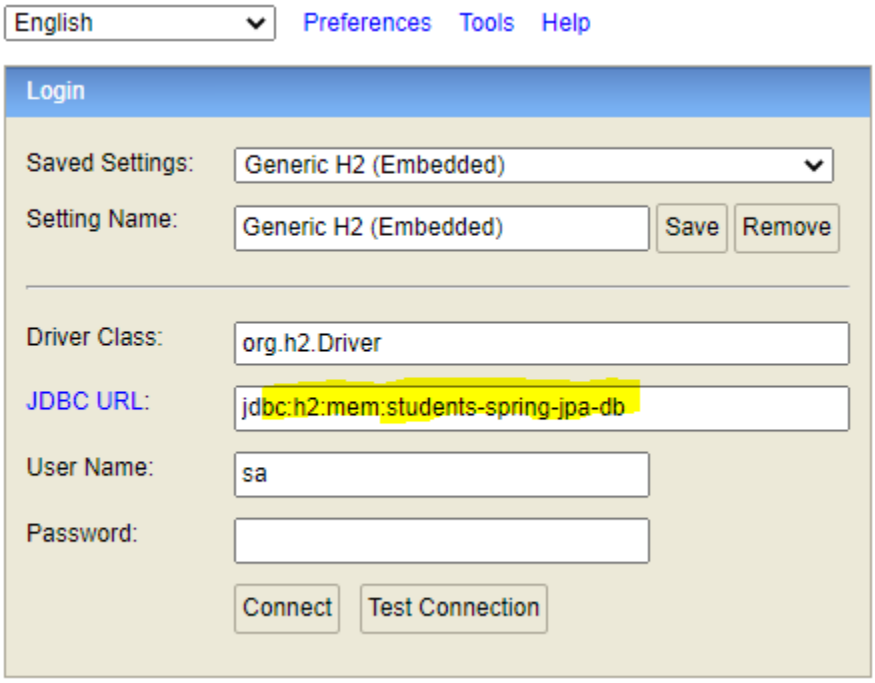
- On démarre l'application (Main class), Spring Boot va scanner toutes les classes et les interfaces qui sont à l'intérieure du package de base **miri.pro.jpaimplimentation1;** (Le package de même niveau avec la classe principale).
- Par défaut l'application démarre sur Tomcat embedded server sur le port 8080. On peut le modifier :

```
- server.port=8082
```

- On peut activer l'interface web du SGBD H2 par :

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

- Ce qui ne donne la possibilité de contrôler nos bases de données H2 à partir de l'adresse : http://localhost:8082/h2-console/

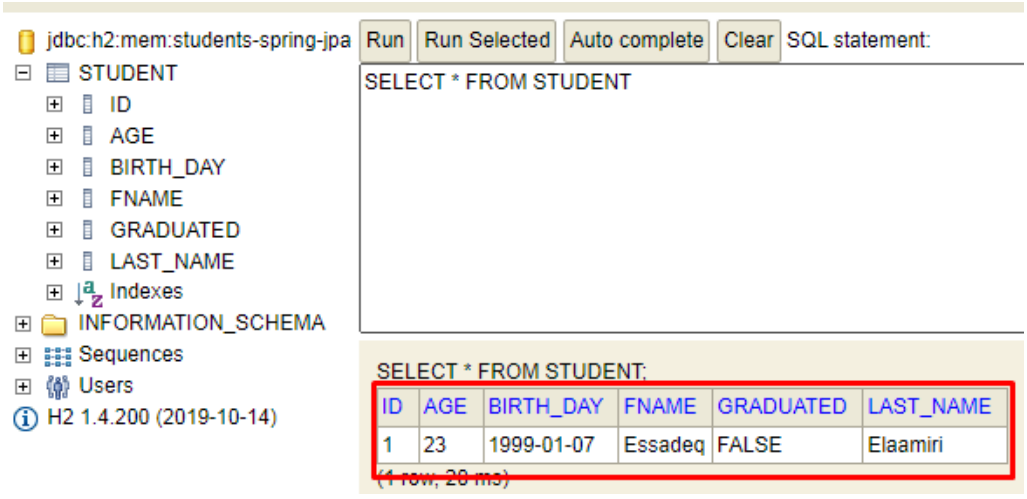


- On peut observer que la base de données a été créé et il y a une table ‘STUDENT’ qui a été générée automatiquement (par Hibernate) avec des colonnes de mêmes noms que les attributs de l’entité Student.
- On peut modifier ou bien contrôler dans le comportement de génération des tables par l’utilisation des annotations dans l’entité, par exemple s’assurer que le type de la colonne ‘BirthDate’ dans la base de données est bien Date, ou bien le nom doit être de 60 caractères et pas 255 qui est par défaut.

```java
@Entity // make a JPA entity
@Data // (lombok) generate getters and setters + non-args constructor
@AllArgsConstructor // generate all args constructor
@NoArgsConstructor // generate no args constructor
public class Student {
    @Id // primary key is necessary
    // The value of id will be generated
    // AUTO-INCREMENT
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "FNAME", length = 50)
    private String firstName;
    private String lastName;
    @Temporal(TemporalType.DATE)
    private Date birthDay;
    private int age;
    private boolean graduated;
```

- Maintenant on va insérer quelques entrées à la base de données, pour faire ça on va notre code dans la fonction **run()** de l’interface **CommandLineRunner** qu’on va implémenter par la classe de base de l’application.
- Spring boot va automatiquement faire appel à la méthode run() après le démarrage de l’application.
- Pour réagir avec la base de données on va avoir besoin d’un objet de **StudentRepository** dans ce cas on va utiliser l’injection de dépendance, Spring va se charger d’injecter une implémentation de l’interface.

```java
@SpringBootApplication
public class JpaImplimentation1Application implements CommandLineRunner {
    //Dependency injection
    /*
     means: please find me in implementation of this interface
     and inject it here in this field
     Spring will take care of that, it creates the implementation
     by itself,
     */
    @Autowired
    private StudentRepository studentRepository;
    public static void main(String[] args) {
        SpringApplication.run(
                JpaImplimentation1Application.class,
                args);
    }
    @Override
    public void run(String... args) throws Exception {
        // using DB functions
        studentRepository.save(
                new Student( id: null
                , firstName: "Essadeq"
                , lastName: "Elaamiri"
                , new Date(Date.valueOf("1999-01-07").getTime())
                , age: 23
                , graduated: false));
    }
}
```



- Autres fonctions : 40 :00

```
-   List<Student> studentList = studentRepository.findAll();
-   Student student =  studentRepository.findById(4L).get();
-   Student student2 =  studentRepository.findById(5L).orElse(null);
-   student2.setAge(100); // upadate object
    studentRepository.save(student2); // save updated object => update
    entry in the DB
-   studentRepository.deleteById(7L);
```

- Pagination:

```
-   Page<Student> studentPage = studentRepository.findAll(PageRequest.of(2,
    10));
```

```
-   // pagination
    Page<Student> studentPage = studentRepository.findAll(PageRequest.of(2,
    10));
    System.out.println("Total pages: "+ studentPage.getTotalPages());
    System.out.println("Total elements: "+ studentPage.getTotalElements());
    System.out.println("Page number: "+ studentPage.getNumber());
    List<Student> studentList = studentPage.getContent();
    displayList(studentPage);
```

- On peut ajouter nos fonctions personnalisées à l'interface StudentRepository. Il suffit de la déclarer, pas la peine de la définir hhh. A partir du nom de la fonction, Spring va se charger de l'implémenter.

```
-//adding some functions
public List<Student> findAllByGraduated(boolean isGraduated);
public Page<Student> findAllByGraduated(boolean isGraduated, Pageable
pageable);
public List<Student> findAllByGraduatedAndAgeIsLessThan(boolean isGraduated,
int age );
public  List<Student>
findAllByGraduatedAndAgeGreaterThanEqualAndFirstNameContains(boolean
isGraduated, int age, String str);
```

- Le <u>problème</u> ici c'est qu'on peut arriver à des noms très longs, et pour dépasser cela, on peut utiliser ce qu'on appelle **HQL** (Hibernate query language) [https://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/queryhql.html] dans l'annotation @Query pour dire au SpringData comment interprète la fonction (le résultat à retourner), et utiliser un nom qui cours et significatif.

```
 @Query("select student from Student student where
student.graduated=:isGraduated and student.age>=:age and student.firstName
like %:key%")
public List<Student> findStudentsWithHQL(boolean isGraduated, int age, String
key);
```

- L'annotation @Param nous aider si les noms des paramètres de Query sont pas les même de la fonction.

```
@Query("select student from Student student where
student.graduated=:isGraduated and student.age>=:age and student.firstName
like %:key%")
public List<Student> findStudentsWithHQL(@Param("isGraduated") boolean
isGraduated, @Param("age") int age, @Param("key") String key);
```

- On pas visualiser nos requêtes SQL dans le terminale, si on a activé la propriété suivante de le fichier application.properties.

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

- Basculer vers MySQL, en ajoutant la dépendance :

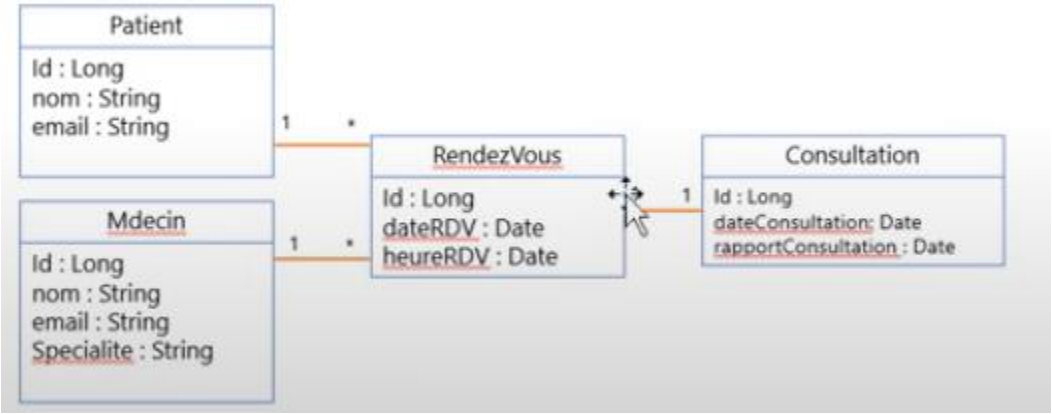
```
 <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

- Et en modifiant la configuration dans le fichier de config:

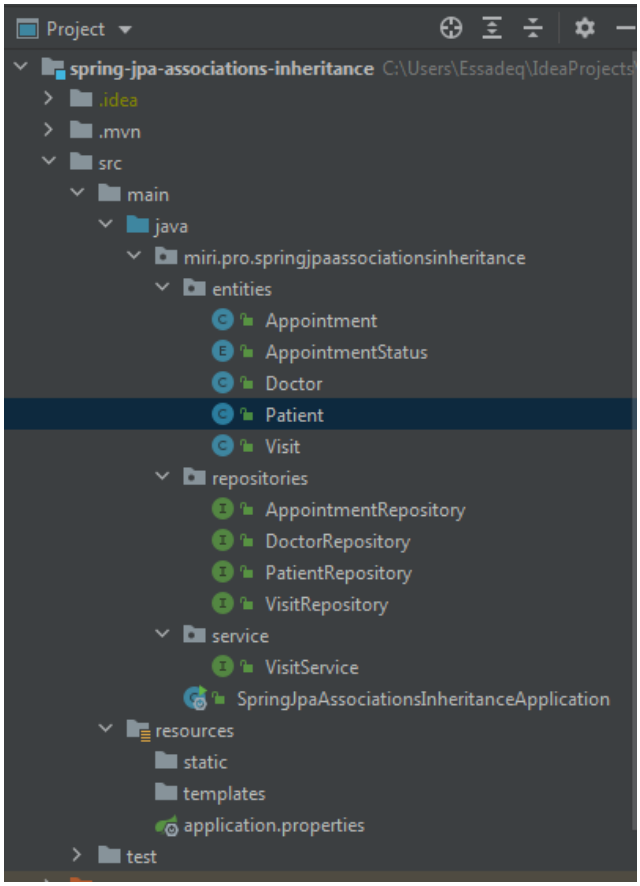
```
#spring.datasource.url=jdbc:h2:mem:students-spring-jpa-db
#create db if not found
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/students-db-
test-spring-data?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
server.port=8082
#spring.h2.console.enabled=true
# create : destroy the previous schema and create a new one
# update : make changes id necessary
#spring.jpa.hibernate.ddl-auto=update ## error hhh
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MariaDBDialect
spring.jpa.show-sql=true
```

Hibernate Spring Data, Mapping des association

- On peut initializer un projet Spring Boot via : https://start.spring.io (Spring Initialzr).
- Voilà le diagramme d'utilisation utilisé dans ces exemples :



- Voilà la structure du code de l'exemple



- Les associations sont bidirectionnelles, par exemple un rendez-vous (Appointment) est concerné à un Patient et un patient est concerné par un ou plusieurs rendez-vous (OneToMany), donc on

va avoir une collection des rendez-vous (@OneToMany) dans la classe Patient, et un objet représentant d'un Patient dans la classe rendez-vous (@ManyToOne), et pour que Spring peut savoir que c'est la même association il faut ajouter (**mappedBy = "patient"**) dans une des deux côtés. C'est comme je dis à Spring que j'ai un attribut qui s'appelle 'patient' dans l'autre coté (c à d dans la classe Appointment).

```java
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Patient {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String email;
    @Temporal(TemporalType.DATE)
    private Date birth;
    @OneToMany(mappedBy = "patient")
    private Collection<Appointment> appointmentCollection;
}
```

```java
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Appointment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Temporal(TemporalType.DATE)
    private Date date;
    private boolean canceled;
    @Enumerated(EnumType.STRING)
    private AppointmentStatus status;
    @ManyToOne
    private Patient patient;
    @ManyToOne
    private Doctor doctor;
    @OneToOne(mappedBy = "apointement")
    private Visit visit;

}
```

- Les types énumérés sont sauvegardés par défaut en forme des entiers (0, 1, 2 …) dans la base de données, et pour les sauvegarder en forme de string (Ses noms) on peut utiliser l'annotation (**@Enumerated(EnumType.STRING)**).

```java
    private boolean canceled;
    @Enumerated(EnumType.STRING)
    private AppointmentStatus status;
    @ManyToOne
```

- L'attribut '**fetch**' des annotations des associations peut prendre soit 'FetchType.LAZY' ou bien ' FetchType.EAGER'. La première dit à Spring de ne pas charger les données dans l'attribut que lors on a besoin (par exemple lorsque on fait appel au objet), par contre la deuxième charge tous les données (dans cette exemple la collection va être remplis des données de la base de donnée lorsque on fait une sélection…)(Généralement utilisé s'il y a une relation forte, on va avoir besoins les données, ou bien une relation de composition) .

```java
@OneToOne(mappedBy = "apointement", fetch= FetchType.EAGER)
private Visit visit;
```

- Avec EAGER il faut initialiser la collection (new ArrayList<>())
- JPA: n'accepte pas deux EAGER successifs, pour éviter de charger toute la BD.
- Si il y a une relation @OneToOne : la clé étrangère va être placée dans la classe (Table) qui ne possède pas de 'mappeBy'. Dans l'exemple suivant la clé étrangère va être placée dans la table qui représente la classe consultation (Visit).

```java
public class Appointment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Temporal(TemporalType.DATE)
    private Date date;
    private boolean canceled;
    @Enumerated(EnumType.STRING)
    private AppointmentStatus status;
    @ManyToOne
    private Patient patient;
    @ManyToOne
    private Doctor doctor;
    @OneToOne(mappedBy = "appointment", fetch= FetchType.LAZY)
    private Visit visit;

}
```

```java
public class Visit { // consultation
    @javax.persistence.Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Temporal(TemporalType.DATE)
    private Date visitDate;
    private String consultationReport;
    @OneToOne
    private Appointment appointment;

}
```

- Dans la classe principale de l'application (SpringJpaAssociationsInheritanceApplication), pour exécuter des instructions au démarrage il y a deux possibilité :
  - Soit on implémente l'interface **CommandLineRunner** et sa méthode **run**().
  - Soit on crée une méthode qui retourne un objet de type **CommandLineRunner,** et on ajoutant l'annotation @Bean, qui dit à Spring que
    1. Exécuter cette méthode au démarrage.
    2. La méthode va retourner un objet, et ce dernier va être un composant Spring (va être dans le contexte parmi la liste des objets composants qui peuvent être injecter avec @autowired).

```
public class SpringJpaAssociationsInheritanceApplication {

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

    @Bean
    CommandLineRunner start(){
        return args -> {

            Stream.of("Essadeq", "Mariam", "Oumaima", "Ali").
                    forEach((name)->{
                        Doctor doctor;
                        doctor = new Doctor();
                        doctor.setName(name);
                        doctor.setEmail(name.concat("@gmail.com"));
                        doctor.setSpeciality(Stream.of("Dentist", "Cardio", "Psycho").fi

                    });
        };
    }
}
```

- C'est le même code que celui le suivant, on a changé juste le nom de la fonction et on n'a pas utilisé la fonction lambda :

```
@Bean
CommandLineRunner run(){
    return new CommandLineRunner() {
        @Override
        public void run(String... args) throws Exception {
            Stream.of("Essadeq", "Mariam", "Oumaima", "Ali").
                    forEach((name)->{
                        Doctor doctor;
                        doctor = new Doctor();
                        doctor.setName(name);
                        doctor.setEmail(name.concat("@gmail.com"));
                        doctor.setSpeciality(Stream.of("Dentist", "Cardio", "Psycho").fin
                    });
        }
    };
}
}
```

- Maintenant si on a besoin d'un objet, il suffit de le déclarer dans les paramètres de la fonction start(), et Spring va se charger de l'injecter.
- Pour bien structurer notre code on va migrer le code métier vers le package **service** ou on va déposer nos interfaces et classes métier **(on ajout l'annotation @Service au classes).** Et au lieu d'interagir avec les repositories directement on va avoir la possibilité de faire des traitements avant d'interagir avec la base de données.
- Au lieu d'utiliser @Autowired avec un nombre des attributs, il est mieux de les utiliser comme paramètres le constructeur.

```java
public class HospitalServiceImp implements IHospitalService{

    //@Autowired
    private PatientRepository patientRepository;
    //@Autowired
    private DoctorRepository doctorRepository;
    //@Autowired
    private VisitRepository visitRepository;
    //@Autowired
    private AppointmentRepository appointmentRepository;

    public HospitalServiceImp(PatientRepository patientRepository,
                              DoctorRepository doctorRepository,
                              VisitRepository visitRepository,
                              AppointmentRepository appointmentRepository)
    {
        this.patientRepository = patientRepository;
        this.doctorRepository = doctorRepository;
        this.visitRepository = visitRepository;
        this.appointmentRepository = appointmentRepository;
    }

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

```java
@Override
public Appointment saveAppointment(Appointment appointment) {
    appointment.setId(UUID.randomUUID().toString()); //generate unique id
    return appointmentRepository.save(appointment);
}
```

- Supposant que on a besoin de consulter la liste des patients sur notre navigateur, pour faire cela on va créer une couche (package) web qui va contenir nos contrôleurs, et créer notre **PatientController** avec l'annotation **@RestController**.

```java
@RestController
public class PatientController {

    @Autowired // injection
    PatientRepository patientRepository;

    //le path (route) pour acceder à cette fonction
    @GetMapping("/patients") // on va recevoir une liste sous fome de JSON
    public List<Patient> getPatientsList(){
        return patientRepository.findAll();
    }
}
```

- Le problème ici, c'est que on va avoir une récursivité de données puisque on utilise des associations bidirectionnelles, Spring convertis le tous les attributs de patient en format json, et donc la liste des Appointment aussi, et lorsqu'il convertit les objets Appointment de la collection il va reconvertis l'objet Patient qu'est attribut d'Appointement.
- Pour éviter ce problème, on peut dire au Spring de ne pas convertir les objets Patient d'Appointement par l'annotation **@JsonProperty(access = JsonProperty.Access.WRITE_ONLY).** Pour que l'attribut soit accessible seulement en lecture, dans ce cas-là on va avoir la liste des Appointments sans avoir le Patient de chacune, la même chose avec l'attribut **Appointments** collection dans le Bean **Doctor**, et l'attribut dans le Bean **Visit**.
- Dans la class Doctor :

```java
-   @OneToMany(mappedBy = "doctor")
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Collection<Appointment> appointmentCollection;
```

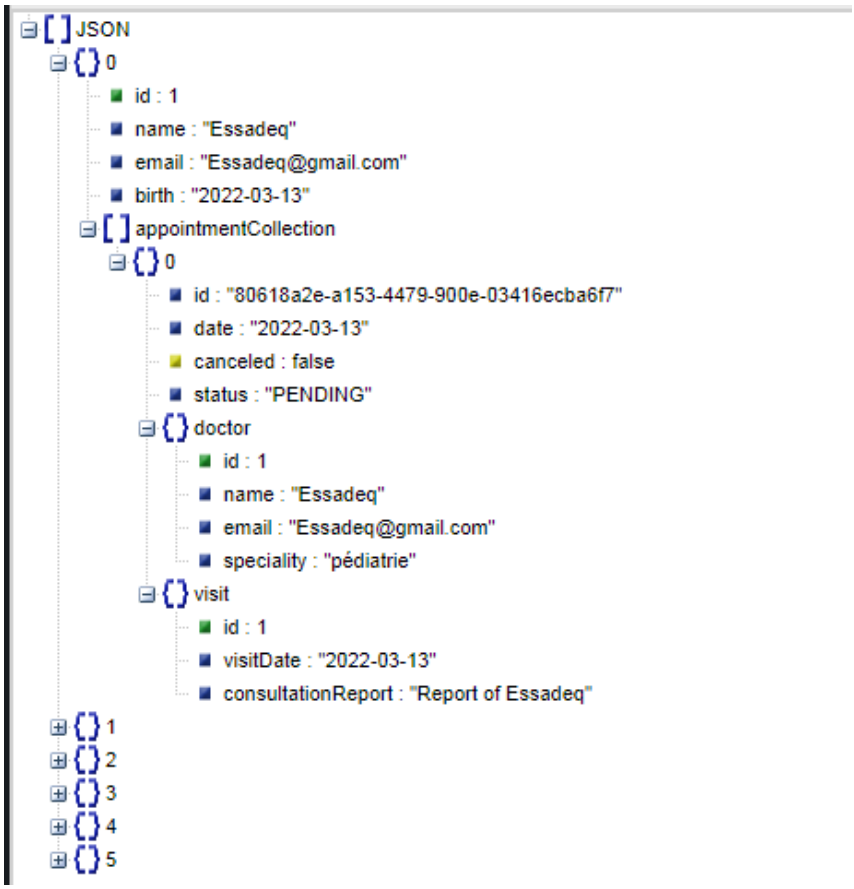
- Dans la classe Visit:

```java
-   @OneToOne
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Appointment appointment;
```

- Et dans la classe Appointement:

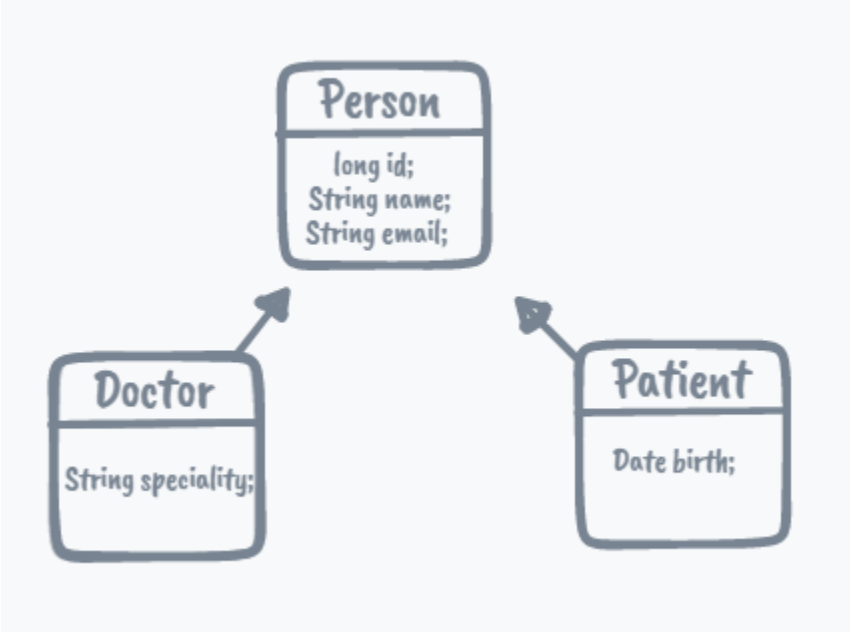
```java
public class Appointment {
    @Id
    //@GeneratedValue(strategy = GenerationType.IDENTITY)
    private String id;
    @Temporal(TemporalType.DATE)
    private Date date;
    private boolean canceled;
    @Enumerated(EnumType.STRING)
    private AppointmentStatus status;
    @ManyToOne
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Patient patient;
    @ManyToOne
    private Doctor doctor;
    @OneToOne(mappedBy = "appointment", fetch= FetchType.LAZY)
    private Visit visit;

}
```

- Et voilà le résultat sur le lien : http://localhost:8080/patients



L'héritage et les associations

- Supposant, qu'on va utiliser une classe **Person** pour être une classe mère de la classe Patient et Doctor afin d'accumuler les propriétés communes.



- Pour implémenter cette association de l'héritage on a trois possibilités :
  - **SINGLE_TABLE** : utiliser une seule table avec **Discriminator column** (qui va nous dire l'entrée de quel type), c'est plus rapide mais on va perdre de l'espace (les colonnes différentes qui vont rester nulles).
  - **TABLE_PER_CLASS** : on aura pas des colonnes nulles, mais on va avoir des colonnes qui sont répétés, et si on chercher une personne, je dois chercher dans les deux classes (Tables), limité aussi en terme d'agrégations, calcules ...[utile s'il y a une grande différences de gestions et des colonnes].
  - **JOINED_TABLE**: (Transforme l'héritage à une association) table pour la classe de base, et des tables pour les classes filles, ça va limiter tous les inconvénients des deux stratégies précédentes.
- l'annotation @Inheritance dispose de l'attribut strategy pour préciser la stratégie utilisée dans le modèle de données. Cette stratégie est une énumération du type InheritanceType et accepte les valeurs : `SINGLE_TABLE, JOINED, TABLE_PER_CLASS.`
- Lien utile : https://gayerie.dev/epsi-b3-orm/javaee_orm/jpa_inheritance.html
- Dans le cas de JOINED par exemple :

```java
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Inheritance(strategy = InheritanceType.JOINED)
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String email;
}
```

```java
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Patient extends Person{

    @Temporal(TemporalType.DATE)
    private Date birth;
    @OneToMany(mappedBy = "patient")
    private Collection<Appointment> appointmentCollection;
}
```

```java
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Doctor extends Person{

    private String speciality;

    @OneToMany(mappedBy = "doctor")
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Collection<Appointment> appointmentCollection;
}
```

- Voici les tables résultantes :

SELECT * FROM PERSON;

ID	EMAIL	NAME
1	Essadeq@gmail.com	Essadeq
2	Mariam@gmail.com	Mariam
3	Oumaima@gmail.com	Oumaima
4	Ali@gmail.com	Ali
5	Salma@gmail.com	Salma
6	Ahmed@gmail.com	Ahmed
7	Essadeq@gmail.com	Essadeq
8	Mariam@gmail.com	Mariam
9	Oumaima@gmail.com	Oumaima
10	Ali@gmail.com	Ali
11	Salma@gmail.com	Salma
12	Ahmed@gmail.com	Ahmed

(12 rows, 7 ms)

SELECT * FROM PATIENT;

BIRTH	ID
2022-03-13	7
2022-03-13	8
2022-03-13	9
2022-03-13	10
2022-03-13	11
2022-03-13	12

(6 rows, 8 ms)

SELECT * FROM DOCTOR;

SPECIALITY	ID
pédiatrie	1
neurochirurgie	2
ophtalmologie	3
ophtalmologie	4
gynécologie obstétrique	5
anesthésie-réanimation	6

(6 rows, 13 ms)

- Pour les autres stratégies il suffit de changer le type dans l'annotation : **@Inheritance(strategy = InheritanceType.**X).

```
- @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
```

- Dans ce cas Spring utilise **DTYPE** comme colonne discriminateur, par default il utilise les noms des classes, on peut changer ça par l'annotation **@DiscriminatorColumn.**

SELECT * FROM PERSON;

DTYPE	ID	EMAIL	NAME	SPECIALITY	BIRTH
Doctor	1	Essadeq@gmail.com	Essadeq	pneumologie	*null*
Doctor	2	Mariam@gmail.com	Mariam	anesthésie-réanimation	*null*
Doctor	3	Oumaima@gmail.com	Oumaima	neurochirurgie	*null*
Doctor	4	Ali@gmail.com	Ali	ophtalmologie	*null*
Doctor	5	Salma@gmail.com	Salma	pédiatrie	*null*
Doctor	6	Ahmed@gmail.com	Ahmed	neurochirurgie	*null*
Patient	7	Essadeq@gmail.com	Essadeq	*null*	2022-03-13
Patient	8	Mariam@gmail.com	Mariam	*null*	2022-03-13
Patient	9	Oumaima@gmail.com	Oumaima	*null*	2022-03-13
Patient	10	Ali@gmail.com	Ali	*null*	2022-03-13
Patient	11	Salma@gmail.com	Salma	*null*	2022-03-13
Patient	12	Ahmed@gmail.com	Ahmed	*null*	2022-03-13

(12 rows, 7 ms)

- `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`
- Dans ce cas on va avoir l'exception suivante: **Cannot use identity column key generation with <union-subclass> mapping for:,** et c'est normal car on a définie la clé primaire dans la classe mère est on a la définir comme **AUTO_INCREAMENT**, logiquement comment Spring va ajouter dans deux tables séparées en respectant l'auto incrémentation ? la solutions ici soit de donner un ID pour chaque classe fille, ou bien :
- `//@GeneratedValue(strategy = GenerationType.IDENTITY) To @GeneratedValue(strategy = GenerationType.TABLE)`
- Dans la classe mère.