

TP - Spring MVC / Thymeleaf

Contents

INTRODUCTION:	2
CREATION DU PROJET (INITIALISATION) :	2
Création de la couche Dao :	3
Créer l'entité JPA Patient :	3
Créer l'interface PatientRepository :	5
Configurer l'unité de persistance : application.properties :	5
Tester la couche DAO :	5
Création de la couche Web :	7
Créer l'application Web qui permet de chercher les patients	8
Intégrer BootStrap (webjars) :	8
Chercher des patients	10
Basculer vers une base de données MySQL au lieu de H2 :	10
Faire la pagination :	11
Créer la partie Web qui permet de supprimer un patient :	12
Conclusion :	13

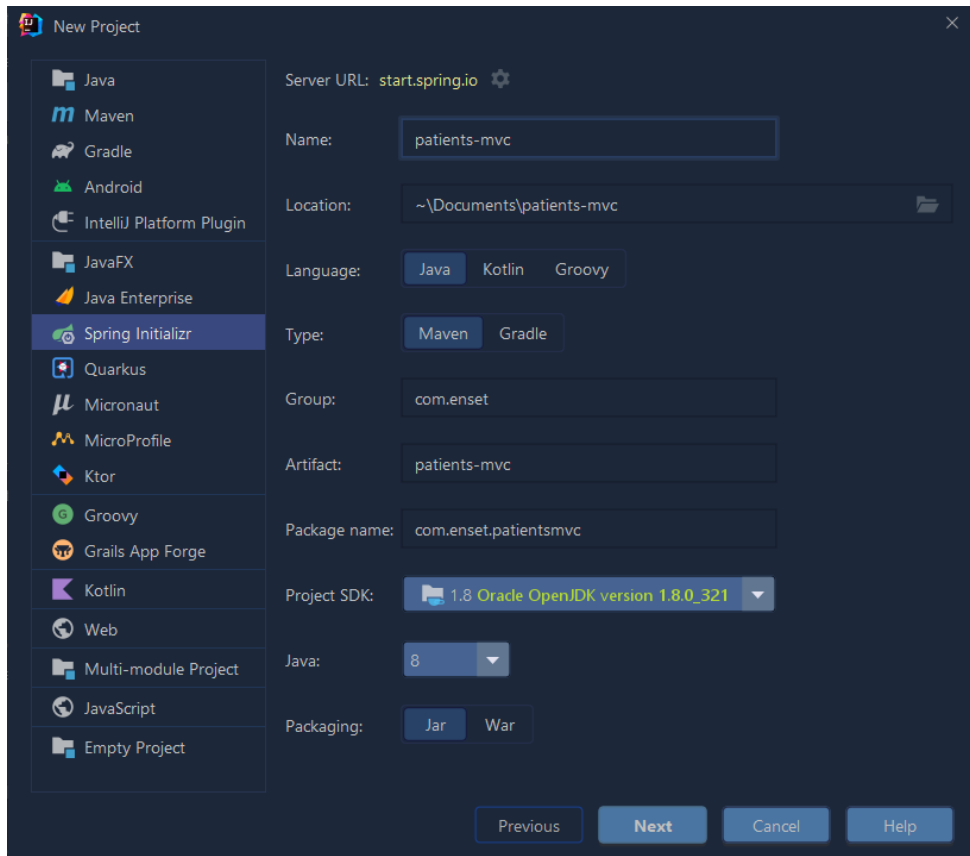
INTRODUCTION:

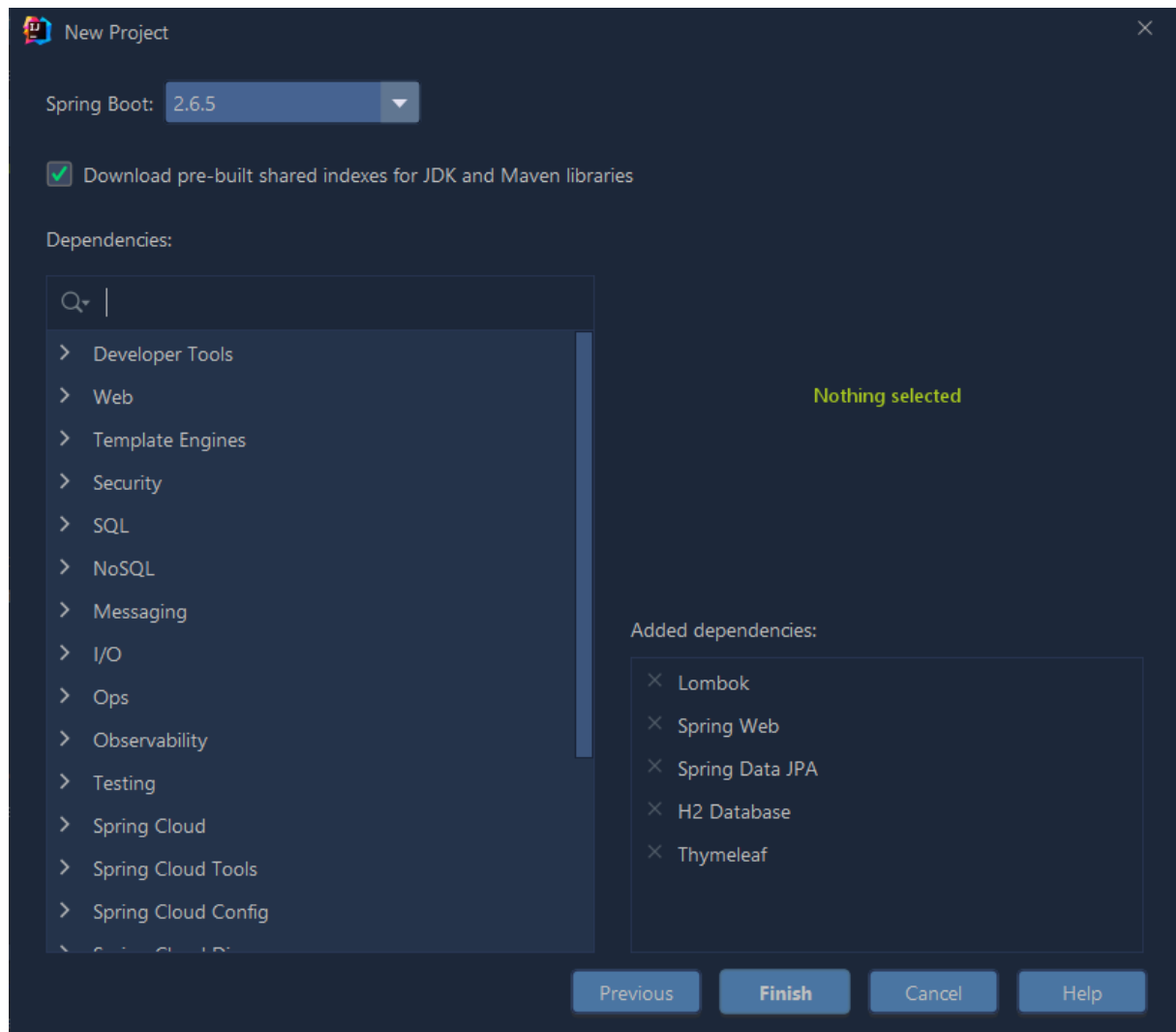
Dans ce deuxième tp nous pratiquerons toutes les nouveaux acquis qui concernent le ORM, Spring Data, Spring MVC

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





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.
- 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.
- Thymeleaf : est un moteur de template écrit en Java traitant les fichiers XML, XHTML et HTML5.

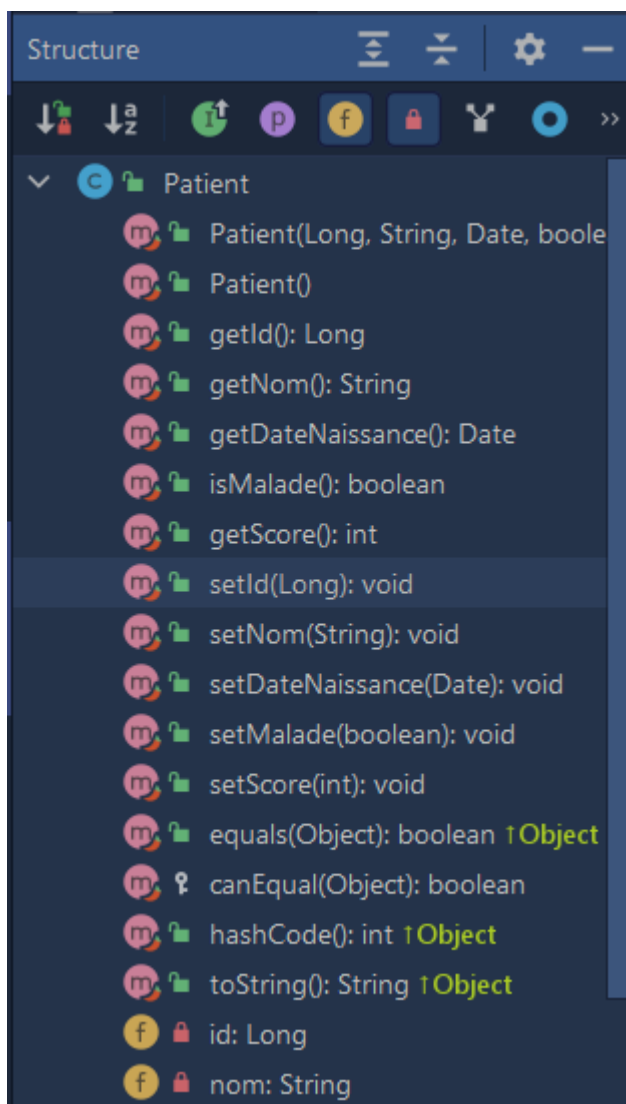
Création de la couche Dao :

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 com.enset.patientsmvc.entities;
import ...
@Entity @AllArgsConstructor @NoArgsConstructor @Data
public class Patient {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    @Temporal(TemporalType.DATE)
    private Date dateNaissance;
    private boolean malade ;
    private int score;
}
```

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



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'autres méthodes 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 résultat final semble à la figure ci-dessous :

```
package com.enset.patientsmvc.repositories;

import ...

public interface PatientRepository extends JpaRepository<Patient, Long> {

    Page<Patient> findByNomContains(String keyword, Pageable pageable);

}
```

Alors dans cet exemple la méthode findAll() qui est une méthode « habituelle » dans les classes DAO est fournie par défaut, et on n'a pas besoin de la définir ni de l'implémenter. De plus on peut définir d'autres méthodes, en cas de besoin, et c'est Spring qui analyse le nom de la méthode définit et ses paramètres pour l'implémenter. (comme findAllByNameContains qui se traduit vers une requête qui retourne toutes les enregistrements qui ont une valeur dans le champ name qui contient la chaîne name fournie en paramètres)

Configurer l'unité de persistance : application.properties :

Pour le moment nous travaillons avec une base de données 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:patients_db
spring.h2.console.enabled=true
server.port=8083
```

Tester la couche DAO :

Nous modifions la classe **PatientsMvcApplication** pour avoir le résultat suivant:

```
package com.enset.patientsmvc;
import ...
@SpringBootApplication
public class PatientsMvcApplication {
    public static void main(String[] args) { SpringApplication.run(PatientsMvcApplication.class, args); }
    @Bean
    CommandLineRunner commandLineRunner(PatientRepository patientRepository){
        return args -> {
            patientRepository.save(new Patient( id: null, nom: "Ouassime", new Date(), malade: false, score: 20));
            patientRepository.save(new Patient( id: null, nom: "Ali", new Date(), malade: true, score: 30));
            patientRepository.save(new Patient( id: null, nom: "Khadija", new Date(), malade: true, score: 10));
            patientRepository.save(new Patient( id: null, nom: "Ikram", new Date(), malade: false, score: 9));

            patientRepository.findAll().forEach(patient -> {
                System.out.println(patient.getNom());
            });
        };
    }
}
```

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

Resultat en console :

```
2022-03-27 09:12:43.910 INFO 9328 --- [
Ouassime
Ali
Khadija
Ikram
```

Resultat dans h2 :

```
jdbc:h2:mem:patients_db
PATIENT
+ ID
+ DATE_NAISSANCE
+ MALADE
+ NOM
+ SCORE
+ Indexes
+ INFORMATION_SCHEMA
+ Sequences
+ Users
H2 1.4.200 (2019-10-14)
```

SELECT * FROM PATIENT PATIENT;

ID	DATE_NAISSANCE	MALADE	NOM	SCORE
1	2022-03-27	FALSE	Ouassime	20
2	2022-03-27	TRUE	Ali	30
3	2022-03-27	TRUE	Khadija	10
4	2022-03-27	FALSE	Ikram	9

(4 rows, 9 ms)

Création de la couche Web :

La creation de la couche dao va être facilité avec l'utilisation de du SPRING MVC, c'est spring qui gère la creation du servlet, et nous fournit toutes ses fonctionnalité 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 executer une opération associée à chaque action. Ces opérations sont implémentées dans une classe appelée PatientController qui représente un sous contrôleur ou un contrôleur secondaire.

```
package com.enset.patientsmvc.web;
+import ...
@Controller @AllArgsConstructor
public class PatientController {
    PatientRepository patientRepository;
    @GetMapping(path = "/index")
    public String patients(Model model,
        @RequestParam(name = "page",defaultValue = "0") int page,
        @RequestParam(name = "size",defaultValue = "4") int size,
        @RequestParam(name = "keyword",defaultValue = "") String
keyword) {
        //Page<Patient> patientPages = patientRepository.findAll(PageRequest.of(page,
size));
        Page<Patient> patientPages =
patientRepository.findByNomContains(keyword,PageRequest.of(page, size));
        model.addAttribute("listPatients",patientPages.getContent());
        model.addAttribute("pages",new int[patientPages.getTotalPages()]);
        model.addAttribute("currentPage",page);
        model.addAttribute("keyword",keyword);
        return "patients";
    }
    @GetMapping("/delete")
    public String delete(Long id,String keyword,int page) {
        patientRepository.deleteById(id);
        return "redirect:/index?page="+page+"&keyword="+keyword;
    }
    @GetMapping("/")
    public String home() {
        return "redirect:/index";
    }
    @GetMapping("/patients") @ResponseBody
    public List<Patient> patientsList() {
        return patientRepository.findAll();
    }
}
```

Créer l'application Web qui permet de chercher les patients

```
<tr th:each="p:${listPatients}">
  <td th:text="${p.id}"></td>
  <td th:text="${p.getNom()}"></td>
  <td th:text="${p.getDateNaissance()}"></td>
  <td th:text="${p.isMalade()}"></td>
  <td th:text="${p.getScore()}"></td>
</tr>
```

← → ↻ ⓘ localhost:8082/index

List des Patients

ID	Nom	Date	Malade	Score
1	Ouassime	2022-03-27	false	20
2	Ali	2022-03-27	true	30
3	Khadija	2022-03-27	true	10
4	Ikram	2022-03-27	false	9

Intégrer BootStrap (webjars) :

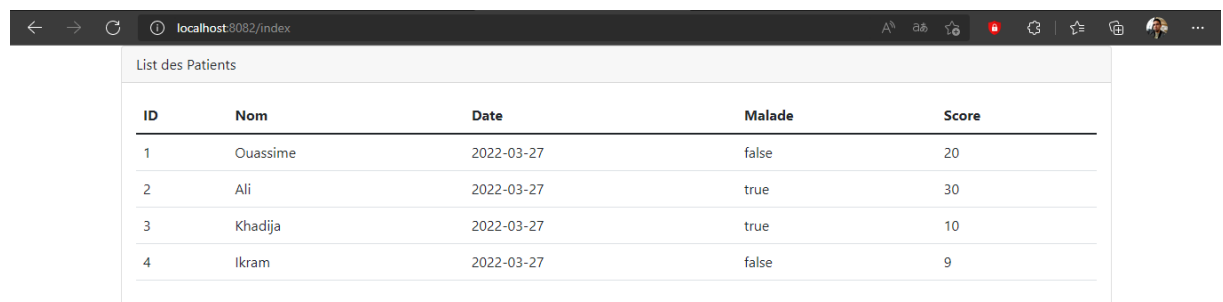
Pour ce faire il faut ajouter la dependance Mavend de Bootstrap et on ajoute une balise de type link à la page HTML :

```
<head>
  <meta charset="UTF-8">
  <title>Title</title>
  <link rel="stylesheet"
href="/webjars/bootstrap/5.1.3/css/bootstrap.min.css">
</head>
```

```
<!-- https://mvnrepository.com/artifact/org.webjars/bootstrap -->
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>5.1.3</version>
</dependency>
```

Et voilà, Bootstrap est prêt pour être utilisé dans notre projet.

Nous allons donc ajouter une nav bar et un champ de text plus une bouton pour faire des recherche par nom dans la base de données, et finalement on va styler notre table html à l'aide de bootstrap.




```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <link rel="stylesheet" href="/webjars/bootstrap/5.1.3/css/bootstrap.min.css">
</head>
<body>
    <div class="container mt-2">
        <div class="card">
            <div class="card-header">List des Patients</div>
            <div class="card-body">
                <form method="get" th:action="@{index}">
                    <label>Key Word</label>
                    <input type="text" name="keyword" th:value="${keyword}">
                    <button type="submit" class="btn btn-primary">Chercher</button>
                </form>
                <table class="table">
                    <thead>
                        <tr>
                            <th>ID</th><th>Nom</th><th>Date</th><th>Malade</th><th>Score</th>
                        </tr>
                    </thead>
                    <tbody>
                        <tr th:each="p:${listPatients}">
                            <td th:text="${p.id}"></td>
                            <td th:text="${p.getNom()}"></td>
                            <td th:text="${p.getDateNaissance()}"></td>
                            <td th:text="${p.isMalade()}"></td>
                            <td th:text="${p.getScore()}"></td>
                            <td>
                                <a class="btn btn-danger" onclick="return confirm('Etes vous
sure ?') "
th:href="@{delete(id=${p.id},keyword=${keyword},page=${currentPage})}">
                                    Delete
                                </a>
                            </td>
                        </tr>
                    </tbody>
                </table>
                <ul class="nav nav-pills">
                    <li th:each="page,status:${pages}">
                        <a th:class="${status.index==currentPage?'btn btn-primary ms-1'
: 'btn btn-outline-
primary ms-1'}"
th:text="${status.index}"
th:href="@{/index(page=${status.index},keyword=${keyword})}"
></a>
                    </li>
                </ul>
            </div>
        </div>
    </div>
</body>
</html>

```

Chercher des patients

Vous observez certainement qu'on a ajouté un nom au champ de texte pour le marqué et on vient de définir une action à faire lors de la soumission du form

Il suffit maintenant de modifier le code l'opération dans le contrôleur qui est mapé au « /index » pour compléter la recherche

```
@GetMapping(path = "/index")
public String patients(Model model,
    @RequestParam(name = "page", defaultValue = "0") int
    page,
    @RequestParam(name = "size", defaultValue = "4") int
    size,
    @RequestParam(name = "keyword", defaultValue = "")
    String keyword) {
    //Page<Patient> patientPages =
    patientRepository.findAll(PageRequest.of(page, size));
    Page<Patient> patientPages =
    patientRepository.findByNomContains(keyword, PageRequest.of(page, size));
    model.addAttribute("listPatients", patientPages.getContent());
    model.addAttribute("pages", new int[patientPages.getTotalPages()]);
    model.addAttribute("currentPage", page);
    model.addAttribute("keyword", keyword);
    return "patients";
}
```

List des Patients

Key Word

Ou

Chercher

ID	Nom	Date	Malade	Score	
5	Ouassime	2022-03-27	false	20	Delete
9	Ouassime	2022-03-27	false	20	Delete
13	Ouassime	2022-03-27	false	20	Delete
17	Ouassime	2022-03-27	false	20	Delete

0

1

2

3

4

5

6

7

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).

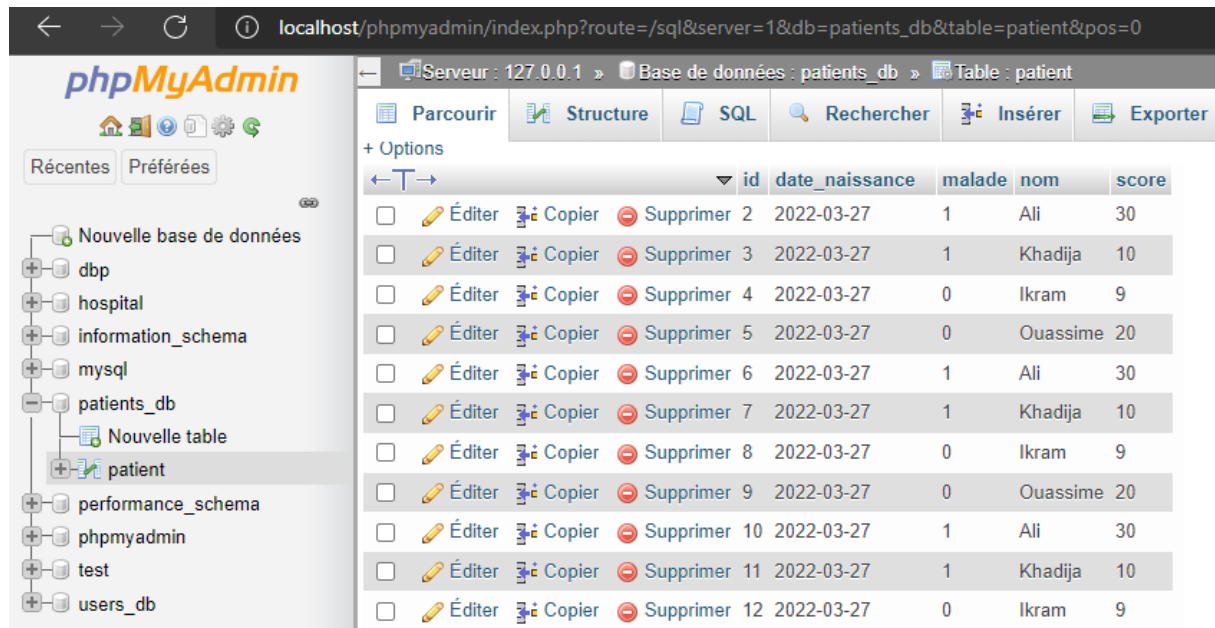
```
server.port=8082
spring.datasource.url=jdbc:mysql://localhost:3306/patients_db?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create
spring.jpa.properties.hibernate.dialect=
org.hibernate.dialect.MariaDBDialect
spring.jpa.show-sql=true
```

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

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.28</version>
</dependency>
```

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

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



	id	date_naissance	malade	nom	score
<input type="checkbox"/>	2	2022-03-27	1	Ali	30
<input type="checkbox"/>	3	2022-03-27	1	Khadija	10
<input type="checkbox"/>	4	2022-03-27	0	Ikram	9
<input type="checkbox"/>	5	2022-03-27	0	Ouassime	20
<input type="checkbox"/>	6	2022-03-27	1	Ali	30
<input type="checkbox"/>	7	2022-03-27	1	Khadija	10
<input type="checkbox"/>	8	2022-03-27	0	Ikram	9
<input type="checkbox"/>	9	2022-03-27	0	Ouassime	20
<input type="checkbox"/>	10	2022-03-27	1	Ali	30
<input type="checkbox"/>	11	2022-03-27	1	Khadija	10
<input type="checkbox"/>	12	2022-03-27	0	Ikram	9

Faire la pagination :

La pagination est très importante quand on est susceptible de travailler avec des grandes quantités des données, c'est impraticable de charger des milliers d'enregistrement tout à la fois. C'est lieux d'implementer un système de pagination, et voila comment on a réussi à afficher les patients avec pagination.

Tout d'abord il faut ajouter des modification à l'action listPatient qui deviendra comme ci-dessous :

```
@GetMapping(path = "/index")
public String patients(Model model,
    @RequestParam(name = "page",defaultValue = "0") int page,
    @RequestParam(name = "size",defaultValue = "4") int size,
    @RequestParam(name = "keyword",defaultValue = "") String keyword){
    //Page<Patient> patientPages = patientRepository.findAll(PageRequest.of(page, size));
    Page<Patient> patientPages =
    patientRepository.findByNomContains(keyword,PageRequest.of(page, size));
    model.addAttribute("listPatients",patientPages.getContent());
    model.addAttribute("pages",new int[patientPages.getTotalPages()]);
    model.addAttribute("currentPage",page);
    model.addAttribute("keyword",keyword);
    return "patients";
}
```

On a ajouté au module les informations nécessaires à la pagination, ces infos sont : le nombre total de la page, la taille de chaque page et le motClé en vigueur.

Ces informations sont les fondement de notre pagination et voila comment on a les utiliser dans le View patients.html

Sous la table on ajoute une liste non ordonnée comme ci-dessous :

```
<ul class="nav nav-pills">
  <li th:each="page,status:${pages}">
    <a th:class="${status.index==currentPage?'btn btn-primary ms-1'
      : 'btn btn-outline-primary ms-1'}"
      th:text="${status.index}"
      th:href="@{/index(page=${status.index},keyword=${keyword})}" ></a>
  </li>
</ul>
```

The screenshot shows a web application running on localhost:8083/index?keyword=. It features a search bar with the text "Chercher" and a table titled "List des Patients". The table has columns for ID, Nom, Date, Malade, and Score. Below the table is a pagination control showing a range of page numbers from 0 to 32, with the current page highlighted as 0.

ID	Nom	Date	Malade	Score
2	Ali	2022-03-27	true	30
3	Khadija	2022-03-27	true	10
4	Ikram	2022-03-27	false	9
5	Ouassime	2022-03-27	false	20

Créer la partie Web qui permet de supprimer un patient :

C'est on implémentant ces différents fonctionnalités qu'on découvre le l'étend des capacités de framework Spring, et comment c'est facile de créer du bon code sans gérer tous les nuisances, alors pour implémenter la fonctionnalité de suppression d'un patient, on ajoute un bouton delete dans chaque ligne du tableau et si ce bouton est cliqué il suffit d'appeler l'action convenable dans notre contrôleur

```
<td>
  <a class="btn btn-danger" onclick="return confirm('Etes vous sure ?') "
  th:href="@{delete(id=${p.id},keyword=${keyword},page=${currentPage})}">
    Delete
  </a>
</td>
```

A chaque fois que le bouton est cliqué on appelle l'action deletePatient, et voilà ce que cette méthode fait :

```
@GetMapping("/delete")
public String delete(Long id,String keyword,int page){
    patientRepository.deleteById(id);
    return "redirect:/index?page="+page+"&keyword="+keyword;
}
```

Il supprime le patient via la methode fourni par JPA deletById et fait une redirection vers la page index on gardant les paramètres déjà établis.

The screenshot shows a web browser at `localhost:8083/index?keyword=`. The page title is "List des Patients". There is a search bar with the placeholder "Key Word" and a "Chercher" button. A modal dialog is open, titled "localhost:8083 indique", with the text "Etes vous sure ?" and buttons "OK" and "Annuler". Below the dialog is a table of patients:

ID	Nom	Date	Malade	Score	
2	Ali	2022-03-27	true	30	Delete
3	Khadija	2022-03-27	true	10	Delete
4	Ikram	2022-03-27	false	9	Delete
5	Ouassime	2022-03-27	false	20	Delete

At the bottom of the page, there is a pagination control showing a range of numbers from 0 to 32, with 0 selected.

Conclusion :

Dans ce TP nous avons vu comment utiliser spring data et spring mvc pour crée une petite application de gestion des patients (pas complète) et nous avons réussi à faire la partie web qui permet à un utilisateur de voir les patients et chercher et aussi supprimer un patient