

# ***Activité Pratique Spring MVC, Spring Data JPA, Spring Security (EtudiantApp)***

## Sommaire

INTRODUCTION :	1
LES OUTILS DE DEVELOPPEMENTS :	1
CREATION DU PROJET (INITIALISATION) :	1
Structure de projet :	3
Création de la couche Dao :	3
Créer l'entité JPA Etudiant:	3
Créer l'interface EtudiantRepository :	5
Configurer l'unité de persistance : application.properties :	6
Tester la couche DAO :	6
Création de la couche Web :	7
Créer l'application Web qui permet de chercher les etudiants	7
Afficher tous les etudiants:	7
Intégrer BootStrap (webjars) :	8
Chercher des etudiants	8
Basculer vers une base de données MySQL:	9
Faire la pagination :	10
Créer la partie Web qui permet de supprimer un etudiant :	10
saisir et ajouter un etudiant:	11
Créer la partie Web qui permet d'éditer et mettre à jour un etudiant	13
Créer une page template de l'application en utilisant Thymeleaf Layout.	13
Securite :	14
Contextualisation :	18
AJOUT DES MEMBRE AVEC CONFIGURATION DE TYPE USERDETAILSERVICE	21
Conclusion :	25

## INTRODUCTION :

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

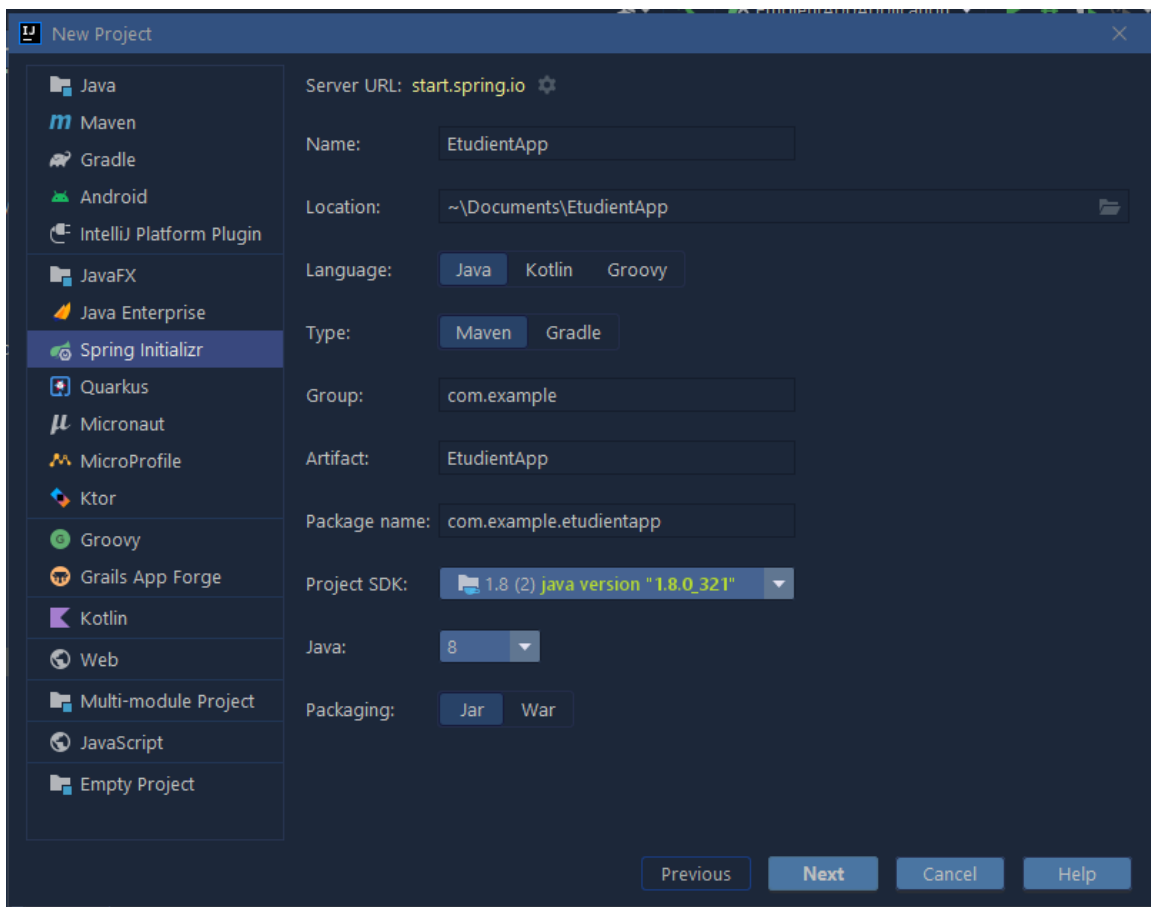
## LES OUTILS DE DEVELOPPEMENTS :

Dans ce tp on va travailler avec IntelliJ Ultimate Edition

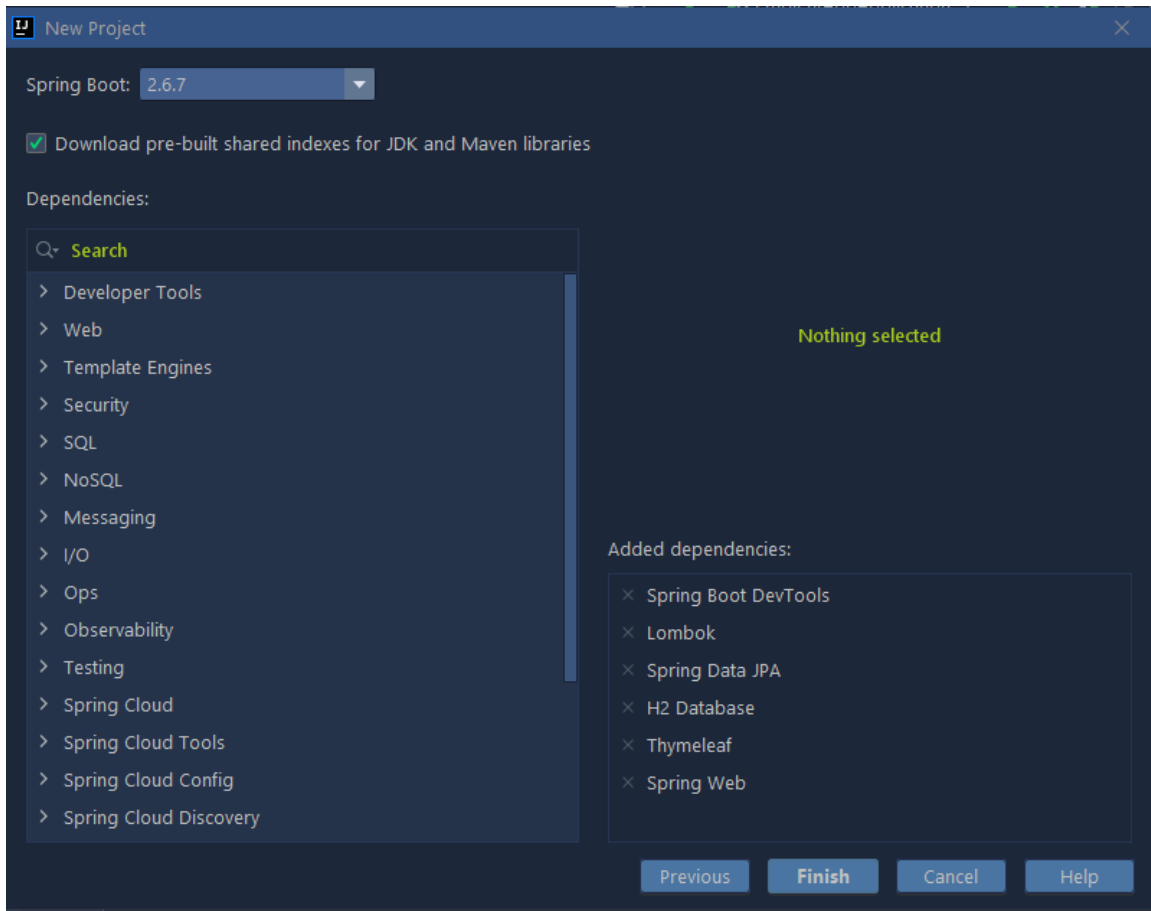
## CREATION DU PROJET (INITIALISATION) :

sélectionner les éléments de bases qui constituent notre application. Alors pour le moment on clique sur New-> Spring initializr Project

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



Il suffit de cocher les technologies qu'on souhaite ajouté à notre projet et c'est spring boot qui se charge de les ajouter et configurer dans le projet.

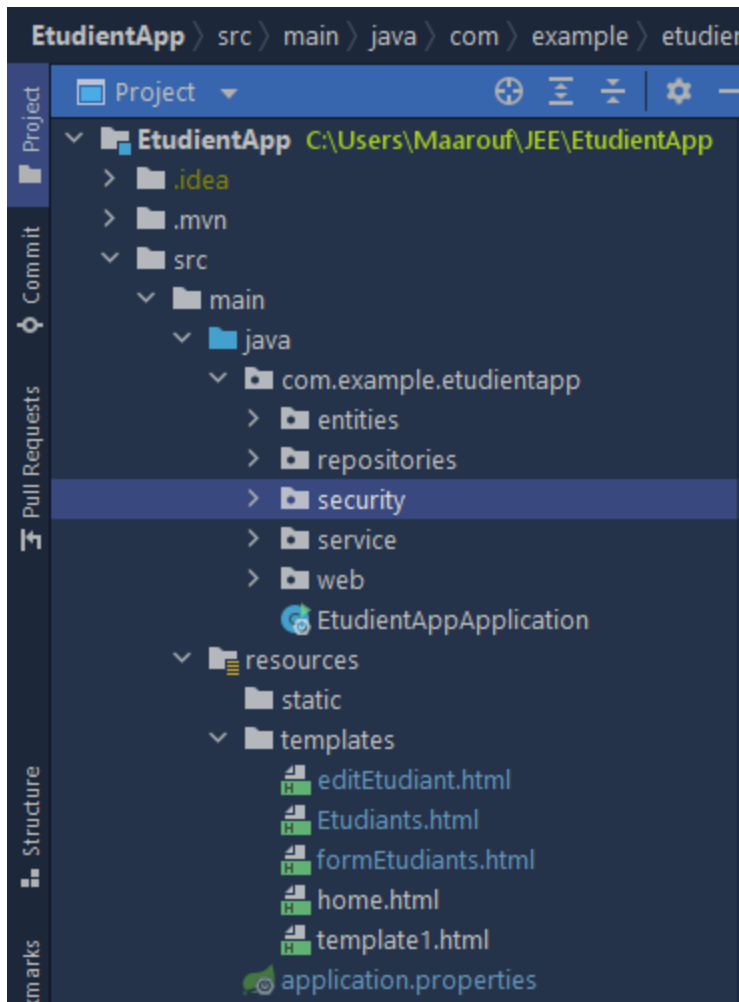


Quant à nous, nous avons ajouté :

- Spring Web : voici l'élément le plus important , c'est celui qui va entraîner la création d'une application spring avec toutes les dépendances y nécessaires.
- Spring DevTools : permettent d'automatiser le redémarrage et le rechargement de votre application lorsque les fichiers sources sont recompilés ou modifiés.
- Lombok : est une API dont le but est de générer à la compilation, du code Java(getters()/setters(), toString()...), à notre place.
- Spring Data JPA : fournit une implémentation de la couche d'accès aux données pour une application **Spring** .
- H2 Database : est un système de gestion de base de données relationnelles écrit en Java. Il peut être intégré à une application Java ou bien fonctionner en mode client-serveur.
- Thymeleaf : est un moteur de template écrit en Java traitant les fichiers XML, XHTML et HTML5.

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

### Structure de projet :



### Création de la couche Dao :

Créer l'entité JPA Etudiant:

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

```

@Entity
@AllArgsConstructor
@NoArgsConstructor
@Data
public class Etudiant {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Size(min = 2,max = 20)
    private String nom;
    @Size(min = 2,max = 20)
    private String prenom ;
    @Email
    private String email;
    @Temporal(TemporalType.DATE)
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private Date dateNaissance;
    @Enumerated(EnumType.STRING)
    private Genre genre ;
    private boolean regle=false;
}

```

- @Entity : specifie que la classe est une entité et qu'elle est mappé à une table de base de données.
- @Id : cette annotations est obligatoire à ajouter dès qu'on ajoute l'annotation @entity, l'annotation @Id spécifi le champs prémaire de la table.
- @GeneratedValue : spécifie comment le sgbd affecte la valeur de l'id.

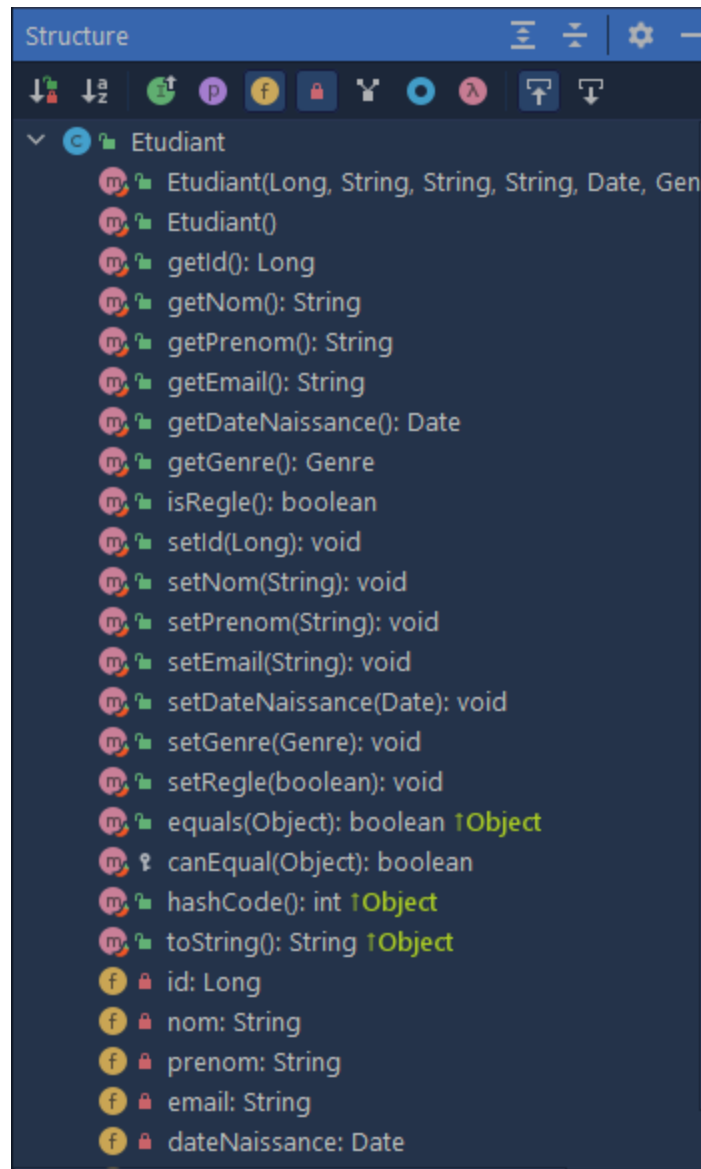
Et voila notre entité est près à etre mappé vers une table dans la bdd.

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

Alors pour ce faire il suffit d'ajouter les annotations convenable comme c'est figuré ci-dessous :

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

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



On voit bien la présence des methodes et constructeurs souhaité.

#### Créer l'interface `EtudiantRepository` :

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

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

```
public interface EtudiantRepository extends JpaRepository<Etudiant, Long> {
    Etudiant findEtudiantByNom(String nom);
    Page<Etudiant> 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 a pas besoin de la définir ni de l'implémenter. De plus on peut définir 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 `findByNomContains` 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` :

il suffit d'ajouter une ligne qui définit l'URL de cette BDD dans le fichier `application.properties` comme ci-dessous :

```
server.port=8083
spring.datasource.url=jdbc:mysql://localhost:3306/etudiants_db?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MariaDBDialect
spring.jpa.show-sql=true
spring.thymeleaf.cache=false
spring.mvc.hiddenmethod.filter.enabled=true
```

Tester la couche DAO :

Nous modifions la classe `EtudiantAppApplication` pour avoir le résultat suivant:

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

On injecte une `EtudiantRepository` dans la classe et on commence à enregistrer et afficher les étudiants :

```
package com.example.etudiantapp;

import ...

@SpringBootApplication
public class EtudiantAppApplication {

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

    @Bean
    CommandLineRunner commandLineRunner(EtudiantRepository etudiantRepository){
        return args -> {
            etudiantRepository.save(
                new Etudiant( id: null, nom: "Ouassime", prenom: "Maarouf", email: "a@a.com", new Date(), Genre.MASCULIN, regle: false));
            etudiantRepository.save(
                new Etudiant( id: null, nom: "Ali", prenom: "Baba", email: "b@b.com", new Date(), Genre.MASCULIN, regle: false));
            etudiantRepository.save(
                new Etudiant( id: null, nom: "Khadija", prenom: "Sara", email: "c@c.com", new Date(), Genre.FEMININ, regle: true));
            etudiantRepository.save(
                new Etudiant( id: null, nom: "Iman", prenom: "Radi", email: "d@d.com", new Date(), Genre.FEMININ, regle: true));

            etudiantRepository.findAll().forEach(etudiant -> {
                System.out.println(etudiant);
            });
        };
    }
}
```

Et voila les resultat afficher dans le console :

```
Etudiant(id=1, nom=Ouassime, prenom=Maarouf, email=a@a.com, dateNaissance=2022-04-23, genre=MASCULIN, regle=false)
Etudiant(id=2, nom=Ali, prenom=Baba, email=b@b.com, dateNaissance=2022-04-23, genre=MASCULIN, regle=false)
Etudiant(id=3, nom=Khadija, prenom=Sara, email=c@c.com, dateNaissance=2022-04-23, genre=FEMININ, regle=true)
Etudiant(id=4, nom=Iman, prenom=Radi, email=d@d.com, dateNaissance=2022-04-23, genre=FEMININ, regle=true)
```

### Création de la couche Web :

La creation de la couche dao va être facilité avec l'utilisation de du module 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 EtudiantController qui représente un sous contrôleur ou un contrôleur secondaire.

```
@Controller @AllArgsConstructor
public class EtudiantController {
}
~
```

Dans cette classe on créer des methodes qui definit les opérations et on les relie à des url avec @GetMapping()

Créer l'application Web qui permet de chercher les etudiants

Afficher tous les etudiants:

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

```
@GetMapping(path = "/user/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<Etudiant> etudiantPage = etudiantRepository.findByNomContains(keyword, PageRequest.of(page, size));
    model.addAttribute( attributeName: "listEtudiants", etudiantPage.getContent());
    model.addAttribute( attributeName: "pages", new int[etudiantPage.getTotalPages()]);
    model.addAttribute( attributeName: "currentPage", page);
    model.addAttribute( attributeName: "keyword", keyword);
    return "Etudiants";
}
```



```

<table class="table">
  <thead>
    <tr>
      <th>ID</th><th>Nom</th><th>Prenom</th><th>Date</th><th>Genre</th><th>Regle</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="p:${listEtudiants}">
      <td th:text="${p.id}"></td>
      <td th:text="${p.getNom()}"></td>
      <td th:text="${p.getPrenom()}"></td>
      <td th:text="${p.getDateNaissance()}"></td>
      <td th:text="${p.getGenre()}"></td>
      <td th:text="${p.isRegle()}"></td>
    </tr>
  </tbody>
</table>

```

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>

```

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

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

ID	Nom	Prenom	Date	Genre	Regle
1	Ouassime	Maarouf	2022-04-23	MASCULIN	false
2	Ali	Baba	2022-04-23	MASCULIN	false
3	Khadija	Sara	2022-04-23	FEMININ	true
4	Iman	Radi	2022-04-23	FEMININ	true

Chercher des étudiants

Vous observez certainement qu'on a ajouté un nom au champ de texte pour le marquer 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 « /user/index » pour compléter la recherche

```
<form method="get" th:action="@{/user/index}">
  <label>Key Word</label>
  <input type="text" name="keyword" th:value="{keyword}">
  <button type="submit" class="btn btn-primary">Chercher</button>
</form>
```

```
@GetMapping(path = "/user/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<Etudiant> etudiantPage = etudiantRepository.findByNomContains(keyword, PageRequest.of(page, size));
    model.addAttribute("listEtudiants", etudiantPage.getContent());
    model.addAttribute("pages", new int[etudiantPage.getTotalPages()]);
    model.addAttribute("currentPage", page);
    model.addAttribute("keyword", keyword);
    return "Etudiants";
}
```

List des Etudiants

Key Word

ID	Nom	Prenom	Date	Genre	Regle
2	Ali	Baba	2022-04-23	MASCULIN	false

0

Basculer vers une base de données MySQL:

il suffit de changer le fichier application.properties ainsi que créer la base de données dans le SGBDR

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>
```

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

Table	Action	Lignes	Type	Interclassement	Taille	Perte
app_role	Parcourir Structure Rechercher Insérer Vider Supprimer	2	InnoDB	utf8mb4_general_ci	32,0 kio	-
app_user	Parcourir Structure Rechercher Insérer Vider Supprimer	3	InnoDB	utf8mb4_general_ci	32,0 kio	-
app_user_app_roles	Parcourir Structure Rechercher Insérer Vider Supprimer	4	InnoDB	utf8mb4_general_ci	48,0 kio	-
etudiant	Parcourir Structure Rechercher Insérer Vider Supprimer	4	InnoDB	utf8mb4_general_ci	16,0 kio	-
4 tables	Somme	13	InnoDB	utf8mb4_general_ci	128,0 kio	0 o

Faire la pagination :

La pagination est très importante quand on est susceptible de travailler avec des grandes quantités des données, c'est impratique de charger des milliers d'enregistrement tout à la fois. C'est mieux d'implémenter un système de pagination, et voilà comment on a réussi à afficher les étudiants avec pagination.

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

```
@GetMapping(path = "/user/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<Etudiant> etudiantPage = etudiantRepository.findByNomContains(keyword,PageRequest.of(page, size));
    model.addAttribute( attributeName: "listEtudiants",etudiantPage.getContent());
    model.addAttribute( attributeName: "pages",new int[etudiantPage.getTotalPages()]);
    model.addAttribute( attributeName: "currentPage",page);
    model.addAttribute( attributeName: "keyword",keyword);
    return "Etudiants";
}
```

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 mot-clé en vigueur.

Ces informations sont les fondements de notre pagination et voilà comment on les utilise dans le View Etudiants.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="${status.index}"
            th:href="@{/user/index(page=${status.index},keyword=${keyword})}" ></a>
    </li>
</ul>
```

Voilà le résultat final :

List des Etudiants							
Key Word		<input type="text"/> <input type="button" value="Chercher"/>					
ID	Nom	Prenom	Date	Genre	Regle		
9	Ouassime	Maarouf	2022-04-23	MASCULIN	false	<input type="button" value="Delete"/>	<input type="button" value="Edit"/>
10	Ali	Baba	2022-04-23	MASCULIN	false	<input type="button" value="Delete"/>	<input type="button" value="Edit"/>
11	Khadija	Sara	2022-04-23	FEMININ	true	<input type="button" value="Delete"/>	<input type="button" value="Edit"/>
12	Iman	Radi	2022-04-23	FEMININ	true	<input type="button" value="Delete"/>	<input type="button" value="Edit"/>
<div><input type="button" value="0"/> <input type="button" value="1"/> <input checked="" type="button" value="2"/> <input type="button" value="3"/></div>							

Créer la partie Web qui permet de supprimer un étudiant :

C'est en implémentant ces différentes fonctionnalités qu'on découvre l'étendue des capacités de framework Spring, et comment c'est facile de créer du bon code sans gérer toutes les nuisances, alors pour implémenter la fonctionnalité de suppression d'un étudiant, 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 controller

```
<td sec:authorize="hasAuthority('ADMIN')">
  <form th:action="@{'/admin/delete/{id}/{page}/{keyword}'
        (id=${p.getId()},page=${currentPage},keyword=${keyword})}" th:method="delete" >
    <input type="hidden" name="_method" value="delete" />
    <button onclick="return confirm('are you sure')" type="submit" class="btn btn-danger">
      Delete
    </button>
  </form>
</td>
```

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

```
@RequestMapping(value={"/admin/delete/{id}/{page}/{keyword}",
                      "/admin/delete/{id}/{page}"}, method = RequestMethod.DELETE)
public String delete(@PathVariable Long id,@PathVariable int page,@PathVariable(required = false) String keyword){
    etudiantRepository.deleteById(id);
    if (keyword == null) {
        keyword="";
    }
    return "redirect:/user/index?page="+page+"&keyword="+keyword;
}
```

Il supprime l'étudiant via la méthode fournie par JPA deleteById et fait une redirection vers la page index en gardant les paramètres déjà établis.

saisir et ajouter un étudiant:

tout d'abord on ajoute un bouton ajouter à la nav-bar déjà créée et si un utilisateur clique sur ce bouton on le dirige vers un formulaire de création des étudiants.

Voici l'action évoquée si quelqu'un clique sur le bouton ajouter un étudiant:

```
@GetMapping("/admin/formEtudiants")
public String formPatients(Model model){
    model.addAttribute("etudiant",new Etudiant());
    return "formEtudiants";
}
```

Il crée un nouveau objet étudiant et le l'injecte dans notre modèle pour qu'on puisse l'utiliser pour affecter les valeurs depuis le formulaire.

```

<form method="post" th:action="@{/admin/save}">
  <div><label>Nom</label>
    <input class="form-control" type="text" name="nom" th:value="${etudiant.nom}">
    <span class="text-danger" th:errors="${etudiant.nom}"></span>
  </div><div><label>Prenom</label>
    <input class="form-control" type="text" name="prenom" th:value="${etudiant.prenom}">
    <span class="text-danger" th:errors="${etudiant.prenom}"></span>
  </div><div><label>Email</label>
    <input class="form-control" type="text" name="email" th:value="${etudiant.email}">
    <span class="text-danger" th:errors="${etudiant.email}"></span>
  </div><div><label>Date Naissance</label>
    <input class="form-control" type="date" name="dateNaissance" th:value="${etudiant.dateNaissance}">
    <span class="text-danger" th:errors="${etudiant.dateNaissance}"></span>
  </div><div><label>votre genre ?</label>
    <select name="genre">
      <option th:each="Genre : ${T(com.example.etudiantapp.entities.Genre).values()}"
        th:text="${Genre}"></option>
    </select>
    <span class="text-danger" th:errors="${etudiant.genre}"></span>
  </div><div><label>Regle</label>
    <input type="checkbox" name="regle" th:checked="${etudiant.regle}">
    <span class="text-danger" th:errors="${etudiant.regle}"></span>
  </div><button type="submit" class="btn btn-primary">Save</button>
</form>

```

Maintenant si l'utilisateur clique sur le bouton Save on appelle l'action /admin/save .

```

@PostMapping("/admin/save")
public String save(Model model, @Valid Etudiant etudiant, BindingResult bindingResult,
    @RequestParam(defaultValue = "0") int page,
    @RequestParam(defaultValue = "") String keyword){
    if (bindingResult.hasErrors()) return "formEtudiants";
    etudiantRepository.save(etudiant);
    return "redirect:/user/index?page="+page+"&keyword="+keyword;
}

```

Mais si on veut de plus valider le formulaire avant ajouter le etudiants à la bdd, on doit utiliser l'api de validation de Spring en injectant des annotations dans la classe etudiant:

```

@Size(min = 2,max = 20)
private String nom;
@Size(min = 2,max = 20)
private String prenom ;
>Email
private String email;
@Temporal(TemporalType.DATE)
@DateTimeFormat(pattern = "yyyy-MM-dd")
private Date dateNaissance;

```

Les champs nom et prenom doit être avec une longueur de chaine entre 2 et 20 lettres, la date a une contrainte de pattern .

Il faut aussi ajouter un objet de type ResultBinding à la signature de notre action, c'est Spring qui gere l'injection de la dépendance convenable dans cette parametre et nous on peut l'utiliser directement comme ci-dessous :

```
@PostMapping("/admin/save")
public String save(Model model, @Valid Etudiant etudiant, BindingResult bindingResult,
    @RequestParam(defaultValue = "0") int page,
    @RequestParam(defaultValue = "") String keyword){
    if (bindingResult.hasErrors()) return "formEtudiants";
    etudiantRepository.save(etudiant);
    return "redirect:/user/index?page="+page+"&keyword="+keyword;
}
```

Dans notre formulaire on peut ajouter des span qui affiche un message d'erreur si nécessaire.

```
<div>
    <label>Email</label>
    <input class="form-control" type="text" name="email" th:value="${etudiant.email}">
    <span class="text-danger" th:errors="${etudiant.email}"></span>
</div>
```

Nom

W

la taille doit être comprise entre 2 et 20

Prenom

TABBAKH

Email

2

doit être une adresse électronique syntaxiquement correcte

Créer la partie Web qui permet d'éditer et mettre à jour un etudiant

Pour ce faire on avance de la même manière qu'on fait avec la suppression d'un etudiant, alors dans la table d'affichage on ajoute une nouvelle colonne et dedans on crée un bouton appeler Edit, le clique sur ce bouton invoque l'action suivante :

```
@GetMapping("/admin/editEtudiant")
public String editPatient(Model model, Long id, int page, String keyword){
    Etudiant etudiant = etudiantRepository.findById(id).orElse( other: null);
    if (etudiant == null) throw new RuntimeException("Etudiant Introuvable");
    model.addAttribute( attributeName: "etudiant", etudiant);
    model.addAttribute( attributeName: "currentPage", page);
    model.addAttribute( attributeName: "keyword", keyword);
    return "editEtudiant";
}
```

Il faut que récupérer le etudiant equivalent au Id fourni et le passe dans le module, finalement il renvoi la vue editEtudiant.html

Par la suite le même formulaire de creation s'affiche mais cette fois il est déjà populé avec les informations du etudiant choisi.

Il suffit maintenant de modifier les informations necessaires et cliquer sur le bouton Save.

Créer une page template de l'application en utilisant Thymeleaf Layout.

Tout d'abord il faut ajouter la dépendence de layout thymeleaf dans le fichier pom.xml

on va créer une page template layout.html avec Thymeleaf et inject le contenu selon la page demandé

dans layout.html on ajoute les noms d'espaces suivants :

```
<!DOCTYPE html>
<html xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      xmlns:th="http://www.thymeleaf.org">
<head>
```

Dans le body de ce fichier on ajoute le contenu qui est statique dans notre template (pour nous c'est le nav-bar)

et au-dessous du navbar on ajoute une section dans laquelle thymeleaf inject le contenu désirable.

```
<div layout:fragment="content1"></div>
```

Et on finit notre html comme d'habitude.

Maintenant pour les autres pages il faut ajouter les meme namespace mais il faut indiquer au thymeleaf que le contenu de cette page est à injecter dans une certaine template, pour ce faire :

```
<!DOCTYPE html>
<html lang="en" layout:decorate="template1"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      xmlns:th="http://www.thymeleaf.org">
<head>
```

Et

```
<div layout:fragment="content1">
  <div class="container mt-2">
    <div class="card">
      <div class="card-header">List des Etudiants</div>
      <div class="card-body">
        <form method="get" th:action="@{/user/index}">
          <label>Key Word</label>
          <input type="text" name="keyword" th:value="${keyword}">
          <button type="submit" class="btn btn-primary">Chercher</button>
        </form>
      </div>
    </div>
  </div>
</div>
```

Comme ça ThymeLeaf comprend qu'il doit injecter ce fragment dans une section nommée « content1 » dans notre template « layout »

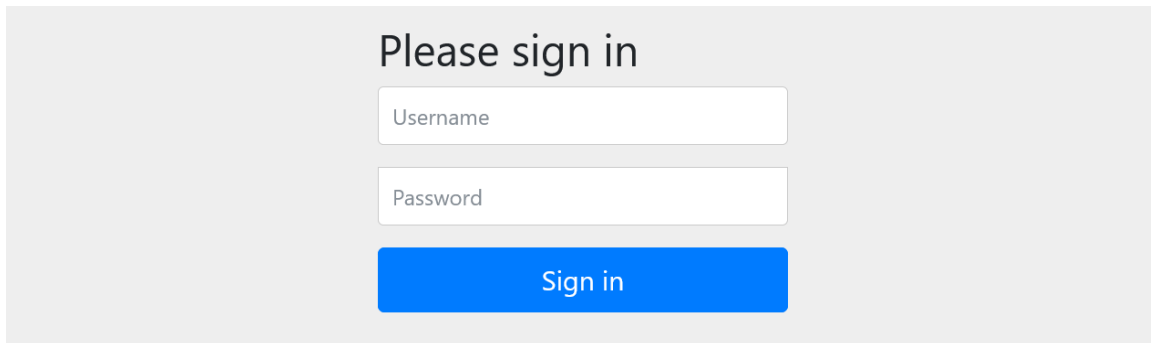
### Securite :

Pour sécuriser notre application on va utiliser Spring Security est un module de Spring qui permet de sécuriser les applications web.

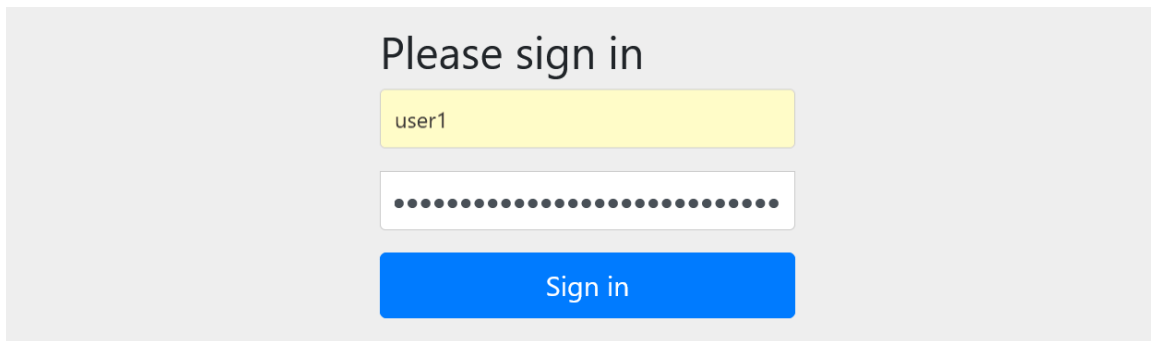
En bref on va configurer des filtres qui vont intercepter les requêtes http et vérifier si l'utilisateur qui vient d'envoyer existe et s'il a le droit d'exécuter cette action sinon il renvoie une page avec le message 403 (ACCESS FORBIDEN)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Dès qu'on ajoute ces dépendances à notre application nous allons remarquer la nécessité de s'authentifier afin d'accéder à l'application. En fait c'est Spring Security qui installe par défaut un filtre qui spécifie que toutes les requêtes http devront être fait par un utilisateur authentifié.

A screenshot of the Spring Security default login page. It has a light gray background. At the top, the text "Please sign in" is centered in a dark gray font. Below this, there are two white input fields with gray borders. The first field is labeled "Username" and the second is labeled "Password". Below the password field is a blue button with the text "Sign in" in white.

Alors même si on n'a pas encore créé un utilisateur, nous pouvons se connecter à l'aide de mot de passe que Spring nous génère. (il l'affiche au console)

A screenshot of the Spring Security default login page, similar to the one above. The "Username" field is now filled with the text "user1" and has a yellow background. The "Password" field is filled with 20 black dots. The "Sign in" button remains blue with white text.

Et maintenant nous sommes authentifiés.

Alors pour bien configurer Spring Security on crée une classe de configuration (qui utilise la classe `@Configuration` et qui étend la classe ***WebSecurityConfigurerAdapter***)

La première chose à faire est de redéfinir la méthode configure :

La ligne ci-dessous permet d'appliquer un filtre qui exige l'authentification pour toutes les requêtes http.



```

@Configuration @EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private DataSource dataSource;
    @Autowired
    private UserDetailsServiceImpl userDetailsService;
    @Autowired
    private PasswordEncoder passwordEncoder;
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService);
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.formLogin();
        http.authorizeRequests()
            .antMatchers( ...antPatterns: "/webjars/**").permitAll();
        http.authorizeRequests().antMatchers( ...antPatterns: "/").permitAll();
        // http.authorizeRequests().antMatchers("/admin/**").hasRole("ADMIN");
        http.authorizeRequests().antMatchers( ...antPatterns: "/admin/**").hasAuthority("ADMIN");
        //http.authorizeRequests().antMatchers("/user/**").hasRole("USER");
        http.authorizeRequests().antMatchers( ...antPatterns: "/user/**").hasAuthority("USER");
        http.authorizeRequests().anyRequest().authenticated();
        http.exceptionHandling().accessDeniedPage("/403");
    }
}

```

Si par la suite on redémarre notre app et on essaye d'accéder à n'importe quelle ressource (par exemple : *localhost :8083/admin/save*) sans s'authentifier nous rencontrons une erreur 403 (accès interdit). Afin que l'application redirige les utilisateurs qui ne sont pas authentifiés vers le formulaire de login on utilise la méthode `http.login()`

*Http.login* fournit un formulaire de connexion qui est déjà créé par Spring. Si on souhaite ajouter notre propre formulaire de login on peut le faire en remplaçant cette ligne par la suivante : ***http.formLogin.loginPage(« /loginVie »)***

On peut aussi utiliser ***http.httpBasic()*** au lieu de ***http.formLogin()*** cela nous engendre un formulaire qui appartient au protocole http (il flotte, crée avec js et non pas html)

Pour le moment, la page login par défaut est suffisante.

Maintenant vient le part d'ajouter des utilisateurs pour qu'on puisse visualiser le fonctionnement de l'application. Pour ce faire on doit redéfinir la deuxième méthode configure :

**b) AJOUT DES MEMBRES :**

```
// @Bean
CommandLineRunner saveUsers(SecurityService securityService){
    return args → {
        securityService.saveNewUser( userName: "ouassime", password: "1234", rePssword: "1234");
        securityService.saveNewUser( userName: "maarouf", password: "1234", rePssword: "1234");
        securityService.saveNewUser( userName: "wissam", password: "1234", rePssword: "1234");

        securityService.saveNewRole( roleName: "USER", description: "");
        securityService.saveNewRole( roleName: "ADMIN", description: "");

        securityService.addRoleToUser( userName: "maarouf", roleName: "ADMIN");
        securityService.addRoleToUser( userName: "maarouf", roleName: "USER");
        securityService.addRoleToUser( userName: "ouassime", roleName: "USER");
        securityService.addRoleToUser( userName: "wissam", roleName: "USER");
    };
}
```

On vient d'ajouter 3 utilisateurs qui peuvent s'authentifier à notre application, mais il y a un seul problème, ce problème concerne l'encodage des mots de passe qui est activé par défaut dans Spring Security, maintenant qu'on crée nos utilisateurs de cette manière Spring ne peut pas comparer leurs mots de passe aux celles saisies par un utilisateur dans le formulaire, puisqu'une est encodé est l'autre non.

Une solution initiale consiste à spécifier que le mot de passe n'est pas encodé, alors on utilise le **Password Storage Format** pour spécifier l'encodage de nos mots de passe (NoOpPasswordEncoder ou {noop} signifie l'absence d'encodage)

On vient de créer trois : 2 utilisateurs et un administrateur, ces comptes sont des noms, des mots de passes et des rôles, et ils sont désormais disponible pour l'utilisation.

Mais il faut quand même avoir une stratégie d'encodage de mot de passe qui est claire est recommandé.

Donc on crée une méthode qui permet de renvoyer une implémentation de l'algorithme BCrypt pour l'encodage :

```
@Bean
PasswordEncoder passwordEncoder(){ return new BCryptPasswordEncoder();
}
```

Le login est parfait pour le moment, on peut avancer vers l'implémentation du *log out*. C'est très simple, il suffit de localiser le bouton *log out* dans le code source et de le modifier pour qu'il ressemble à la figure ci-dessous :

```
<li><a class="dropdown-item" th:href="@{/logout}">Logout</a></li>
```

Par la suite on va créer des filtres qui à base des ressources qu'un utilisateur demande consultation et à base du rôle de cet utilisateur on prend la décision de l'accepter ou refuser.

Dans la méthode configure on spécifie que seuls les admins peuvent supprimer ou modifier ou ajouter des ressources, tous les autres utilisateurs peuvent seulement les consulter.

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.formLogin();
    http.authorizeRequests()
        .antMatchers( ...antPatterns: "/webjars/**").permitAll();
    http.authorizeRequests().antMatchers( ...antPatterns: "/").permitAll();
    // http.authorizeRequests().antMatchers("/admin/**").hasRole("ADMIN");
    http.authorizeRequests().antMatchers( ...antPatterns: "/admin/**").hasAuthority("ADMIN");
    //http.authorizeRequests().antMatchers("/user/**").hasRole("USER");
    http.authorizeRequests().antMatchers( ...antPatterns: "/user/**").hasAuthority("USER");
    http.authorizeRequests().anyRequest().authenticated();
    http.exceptionHandling().accessDeniedPage("/403");
}

```

Cette distinction est faite à base des urls passés, les url de modification et suppression et d'ajout sont ceux qui commencent par **/admin** alors on exige que ces actions doivent être appelé par des utilisateurs avec le rôle admin.

On spécifie aussi qu'un utilisateur doit être authentifié pour afficher les ressource (affichage de liste des étudiants), sinon il se contente par la page d'accueil.

Maintenant si un utilisateur veux accéder à un ressource, on le renvoie un formulaire de login. Sinon si il essaye d'accéder à la ressource via l'url on l'affiche une page 403.

On peut même configurer cette page pour visualiser un message de notre choix. Pour ce faire il suffit d'ajouter la ligne suivante dans la méthode configure, et créer le view *403.html* qui prend la place de cette page que Spring nous a fourni par default.

```
http.exceptionHandling().accessDeniedPage("/403");
```

```

<!DOCTYPE html>
<html lang="en" layout:decorate="template1"
  xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
  xmlns:th="http://www.thymeleaf.org"
  xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
  <link rel="stylesheet" href="/webjars/bootstrap/5.1.3/css/bootstrap.min.css">
</head>
<body>
  <div layout:fragment="content1">
    <div class="container">
      <h2 class="text-danger">Vous n'etes pas autorise a acceder a cette ressource</h2>
    </div>
  </div>
</body>
</html>

```

Contextualisation :

Pour contextualiser notre application web nous allons implémenter les règles suivantes :

- Il y a trois types d'utilisateur : visiteur, utilisateur, admin.
- Les visiteurs ont accès uniquement à la page d'accueil.

- Les utilisateurs peuvent afficher la liste des étudiants et utiliser la fonctionnalité de recherche.
- Les admins peuvent faire tous ce qui précède ainsi qu'ajouter, modifier et supprimer les étudiants.

Ces simples règles peuvent se traduire vers les tâches suivantes :

1. Afficher le nom de l'utilisateur connecté.
2. Afficher log in si un visiteur est connecté.
3. Afficher log out si un utilisateur ou admin est connecté.
4. Afficher les boutons ajouter, supprimer et modifier seulement si l'utilisateur connecté est un admin.

Pour contextualiser notre application nous allons utiliser une dépendance qui permet au *Thymeleaf* de communiquer avec le contexte de *Spring Security* et elle nous fournit quelques directives qui facilitent la tâche de contextualisation.

```
<dependency>
  <groupId>org.thymeleaf.extras</groupId>
  <artifactId>thymeleaf-extras-springsecurity5</artifactId>
</dependency>
```

Maintenant que la dépendance est déclarée nous allons ajouter son namespace dans les fichiers de view qui nous intéressent.

```
xmlns:sec="http://www.thymeleaf.org/extras/spring-security"
```

Réalisation des tâches de contextualisation :

1. Le nom de l'utilisateur connecté et affichage de login/logout conditionnel :  
Dans le fichier layout.html on effectue les modifications suivantes.  
**sec:authentication= « name »** : renvoie le nom d'utilisateur connecté.  
**sec:authorize := « »** on l'utilise pour spécifier la condition d'affichage d'une certaine balise html(généralement un span)  
**isAuthenticated** : renvoie true si l'utilisateur est authentifié(pas visiteur),  
**isAnonymous** : renvoie true si l'utilisateur n'est pas authentifié(un visiteur),

```
<ul class="navbar-nav">
  <li class="nav-item dropdown" sec:authorize="isAuthenticated()" >
    <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
      <span sec:authentication="name"></span>
    </a>
    <ul class="dropdown-menu" >
      <li><a class="dropdown-item" th:href="@{/logout}">Logout</a></li>
    </ul>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#" sec:authorize="!isAuthenticated()" th:href="@{/login}">Login</a>
  </li>
</ul>
```

2. Contrôler l'affichage des boutons ajouter, modifier et supprimer :  
Dans le fichier layout.html, on change le dropdown menu pour qu'il affiche l'action ajouter uniquement pour les admins.

```

<li class="nav-item dropdown">
  <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">Etudiants</a>
  <ul class="dropdown-menu">
    <li sec:authorize="hasAuthority('ADMIN')">
      <a class="dropdown-item" th:href="@{/admin/formEtudiants}">Nouveau</a>
    </li>
    <li><a class="dropdown-item" th:href="@{/user/index}">Chercher</a></li>
  </ul>
</li>

```

**hasAuthority('ROLE')** : renvoie true si l'utilisateur a le rôle mentionné entre les parenthèses.

Dans ce cas seuls les utilisateurs qui ont le rôle administrateur peuvent voir le lien d'ajouter.(Nouveau)

De même dans la page Etudiants.html seuls les admins peuvent voir les boutons de modification et suppression.

```

<table class="table">
  <thead>
    <tr><th>ID</th><th>Nom</th><th>Prenom</th><th>Date</th><th>Genre</th><th>Regle</th></tr>
  </thead>
  <tbody>
    <tr th:each="p:${listEtudiants}">
      <td th:text="${p.id}"></td>
      <td th:text="${p.getNom()}"></td>
      <td th:text="${p.getPrenom()}"></td>
      <td th:text="${p.getDateNaissance()}"></td>
      <td th:text="${p.getGenre()}"></td>
      <td th:text="${p.isRegle()}"></td>
      <td sec:authorize="hasAuthority('ADMIN')">
        <form th:action="@{/admin/delete/{id}/{page}/{keyword}"
              (id=${p.getId()},page=${currentPage},keyword=${keyword})" th:method="delete" >
          <input type="hidden" name="_method" value="delete" />
          <button onclick="return confirm('are you sure')" type="submit" class="btn btn-danger">
            Delete
          </button>
        </form>
      </td>
      <td sec:authorize="hasAuthority('ADMIN')">
        <a class="btn btn-success"
          th:href="@{/admin/editEtudiant(id=${p.id},keyword=${keyword},page=${currentPage})}">
          Edit
        </a>
      </td>
    </tr>
  </tbody>
</table>

```

Ce qu'un admin voit :



Key Word <input type="text"/> <button>Chercher</button>						
ID	Nom	Prenom	Date	Genre	Regle	
1	Ouassime	Maarouf	2022-04-23	MASCULIN	false	<button>Delete</button> <button>Edit</button>
2	Ali	Baba	2022-04-23	MASCULIN	false	<button>Delete</button> <button>Edit</button>
3	Khadija	Sara	2022-04-23	FEMININ	true	<button>Delete</button> <button>Edit</button>
4	Iman	Radi	2022-04-23	FEMININ	true	<button>Delete</button> <button>Edit</button>

0 1 2 3

Ce qu'un utilisateur voit :

The screenshot shows a web browser at localhost:8083/user/index. The page has a dark header with 'Home' and 'Etudiants' links, and a user profile 'ouassime'. The main content area is titled 'List des Etudiants' and contains a search bar with the text 'Key Word' and a 'Chercher' button. Below the search bar is a table with the following columns: ID, Nom, Prenom, Date, Genre, and Regle. The table contains four rows of student data. At the bottom of the table, there are pagination buttons for 0, 1, 2, and 3.

ID	Nom	Prenom	Date	Genre	Regle
1	Ouassime	Maarouf	2022-04-23	MASCULIN	false
2	Ali	Baba	2022-04-23	MASCULIN	false
3	Khadija	Sara	2022-04-23	FEMININ	true
4	Iman	Radi	2022-04-23	FEMININ	true

### c) AJOUT DES MEMBRE AVEC CONFIGURATION DE TYPE USERDETAILSERVICE

Notre but primaire est de connecter l'application Spring à une base de données avec **JPA** et retourner les informations de l'utilisateur : si un utilisateur essaie de se connecter à l'app, on va vérifier l'existence de cet utilisateur dans la bdd, ensuite on vérifie son mot de passe et ses rôles.

Table AppUser :

				user_id	active	password	username
<input type="checkbox"/>	Éditer	Copier	Supprimer	565b5ea3-7ae1-4b3d-9d08-312bdbc4b27f	1	\$2a\$10\$X3T4s9lwR9IRfPA8hhB/e9qPLvNJEbHRSNXI1XtXr...	ouassime
<input type="checkbox"/>	Éditer	Copier	Supprimer	8d557d79-3391-4fe7-acab-fff77ac1973b	1	\$2a\$10\$uIFZ6a0Z8k1ieYbxUrB70eyQN/5ZxbWB6k.N9571Amt...	maarouf
<input type="checkbox"/>	Éditer	Copier	Supprimer	be115a43-bdc9-4628-85d1-225267ff13e4	1	\$2a\$10\$501ykn2VnjAK3jopXjVFPuclKA6juNznCmCmFRDvw9...	wissam

Table AppRole :

				role_id	description	role_name
<input type="checkbox"/>	Éditer	Copier	Supprimer	1		USER
<input type="checkbox"/>	Éditer	Copier	Supprimer	2		ADMIN

Table app\_user\_app\_roles :

app_user_user_id	app_roles_role_id
8d557d79-3391-4fe7-acab-fff77ac1973b	2
8d557d79-3391-4fe7-acab-fff77ac1973b	1
565b5ea3-7ae1-4b3d-9d08-312bdbc4b27f	1
be115a43-bdc9-4628-85d1-225267ff13e4	1

Alors c'est depuis cette table qu'on extrait les informations nécessaires pour l'authentification d'un utilisateur.

Dans notre classe de configuration SecurityConfig.java on garde l'autorisation explicitée dans la méthode **configure(HttpSecurity http)** telle quelle.

Dans la méthode d'authentification **`configure(AuthenticationBuilderManager auth)`**, on implémente une authentification basé sur **`UserDetailsService`**.

**`UserDetailsService`** est l'interface principale qui charge les informations spécifiques à un utilisateur.

Elle est utilisée dans Spring comme étant un **`DAO`** pour les objets utilisateurs. L'interface exige la redefinition d'une seule méthode (de mode lecture seulement).

Avant de spécifier le type d'authentification dans la methode **`configure(AuthenticationBuilderManager auth)`**, il faut qu'on implémente l'interface **`UserDetailsService`** .

On crée une nouvelle classe dans le package « `com.example.etudiantapp.security` », cette classe implémente **`UserDetailsService`** et redéfinit la seule methode **`loadUserByUsername(String s)`**.

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private SecurityService securityService;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        AppUser appUser = securityService.loadUserByUsername(username);
        Collection<GrantedAuthority> authorities1 =
            appUser.getAppRoles().stream().map(role -> new SimpleGrantedAuthority(role.getRoleName()))
                .collect(Collectors.toList());
        User user = new User(appUser.getUsername(), appUser.getPassword(), authorities1);
        return user;
    }
}
```

On peut implémenter le logique de cette methode de n'importe quelle façon qu'on veut(chargement des infors des utilisateur depuis fichier txt, api, hard coded...), mais quant à ce tp, on va utiliser JPA, donc il faut qu'on :

- créer une entité AppUser mappé à la table qu'on a créé avant.

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class AppUser {
    @Id
    private String userId;
    @Column(unique = true)
    private String username;
    private String password;
    private boolean active;
    @ManyToMany(fetch = FetchType.EAGER)
    private List<AppRole> appRoles = new ArrayList<>();
}
```

- créer une entité AppRole mappé à la table qu'on a créé avant.

```

public class AppRole {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long roleId;
    @Column(unique = true)
    private String roleName;
    private String description;
}

```

```

@Service @Slf4j @AllArgsConstructor
public class SecurityServiceImpl implements SecurityService {
    private AppUserRepository appUserRepository;
    private AppRoleRepository appRoleRepository;
    private PasswordEncoder passwordEncoder;
    @Override
    public AppUser saveNewUser(String username, String password, String
rePassword) {
        if (!password.equals(rePassword))
            throw new RuntimeException("password not match");
        String hashedPWD = passwordEncoder.encode(password);
        AppUser appUser = new AppUser();
        appUser.setUserId(UUID.randomUUID().toString());
        appUser.setUsername(username);
        appUser.setPassword(hashedPWD);
        appUser.setActive(true);
        AppUser savedAppUser = appUserRepository.save(appUser);
        return savedAppUser;
    }

    @Override
    public AppRole saveNewRole(String roleName, String description) {
        AppRole appRole = appRoleRepository.findByRoleName(roleName);
        if (appRole != null)
            throw new RuntimeException("Role "+roleName+"Already
exist");
        appRole = new AppRole();
        appRole.setRoleName(roleName);
        appRole.setDescription(description);
        AppRole savedAppRole = appRoleRepository.save(appRole);
        return savedAppRole;
    }

    @Transactional
    @Override
    public void addRoleToUser(String username, String roleName) {
        AppUser appUser = appUserRepository.findByUsername(username);
        if (appUser == null)
            throw new RuntimeException("User not found");
        AppRole appRole = appRoleRepository.findByRoleName(roleName);
        if (appRole == null)
            throw new RuntimeException("Role not found");
        appUser.getAppRoles().add(appRole);
    }

    @Override
    public void removeRoleFromUser(String username, String roleName) {

```



```

        AppUser appUser = appUserRepository.findByUsername(username);
        if (appUser == null)
            throw new RuntimeException("User not found");
        AppRole appRole = appRoleRepository.findByRoleName(roleName);
        if (appRole == null)
            throw new RuntimeException("Role not found");
        appUser.getAppRoles().remove(appRole);
    }

    @Override
    public AppUser loadUserByUsername(String username) {
        return appUserRepository.findByUsername(username);
    }

```

On constate que la classe **SecurityServiceImpl** inclut des informations supplémentaires qu'on a pas défini dans la classe AppUser

Ensuite il faut créer la JpaRepository responsable de la persistance des objets AppUser dans la table user\_details.

```

public interface AppUserRepository extends JpaRepository<AppUser,String> {

    AppUser findByUsername(String username);

}

```

Finalement on retourne vers le package « com.example.etudiantapp.security », dans la classe **UserDetailsServiceImpl** et dans la methode **loadUserByUsername(String s)**, on pointe vers notre **AppUserRepository** et on cherche l'utilisateur dans les tables qui se trouve dans la bdd.

```

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private SecurityService securityService;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        AppUser appUser = securityService.loadUserByUsername(username);
        Collection<GrantedAuthority> authorities1 =
            appUser.getAppRoles().stream().map(role -> new SimpleGrantedAuthority(role.getRoleName()))
                .collect(Collectors.toList());
        User user = new User(appUser.getUsername(), appUser.getPassword(), authorities1);
        return user;
    }
}

```

Après redemarrage du serveur, on saisi les nouvelle informations de l'utilisteur :

## Please sign in

You have been signed out

maarouf

....

Sign in

Home Etudiants ▼

maarouf ▼

List des Etudiants

Key Word

Chercher

ID	Nom	Prenom	Date	Genre	Regle		
1	Ouassime	Maarouf	2022-04-23	MASCULIN	false	Delete	Edit
2	Ali	Baba	2022-04-23	MASCULIN	false	Delete	Edit
3	Khadija	Sara	2022-04-23	FEMININ	true	Delete	Edit
4	Iman	Radi	2022-04-23	FEMININ	true	Delete	Edit

0

1

2

3

### Conclusion :

A la fin de ce tp nous avons arriver a cree une application complet en utlisons lensemble des outiles quo n a déjà cite , et on securiser notre application avec Spring Security qu est un outil genial qui facilite l'authentification et l'autorisation des utilisateurs, ainsi que la contextualisation des views de notre application.