

RAPPORT TP3

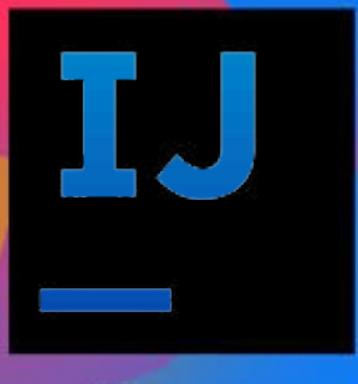
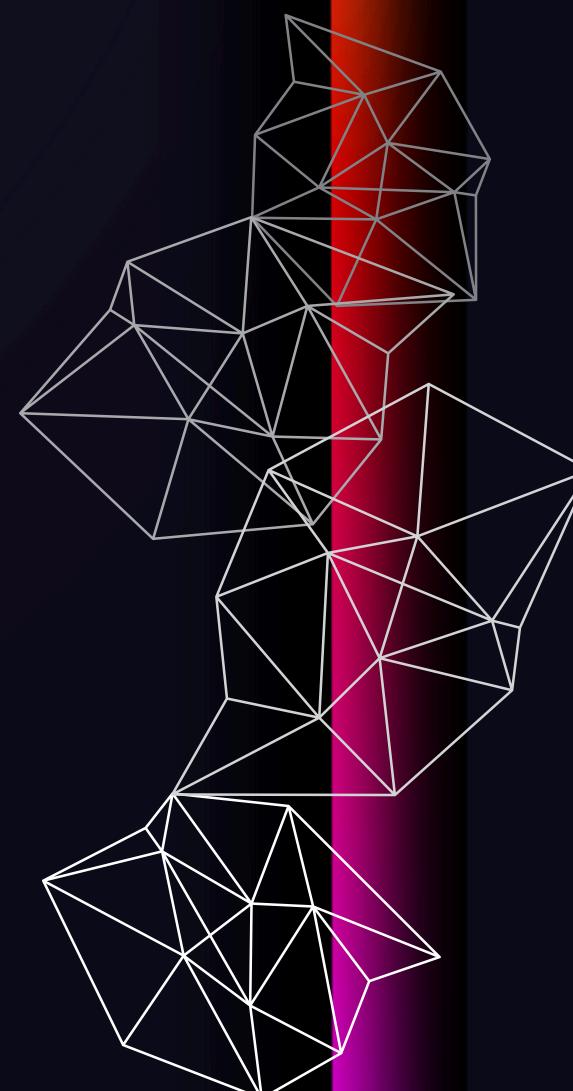
SPRING MVC

SPRING DATA JPA

THYMELEAF

Réalisé par :
Bouchama Hajar

Demandé par :
Mohamed Youssfi



Introduction

Ce rapport documente la création d'une application web Java EE dédiée à la gestion des patients, construite sur les technologies Spring MVC, Thymeleaf et Spring Data JPA. Le projet est découpé en trois parties principales, chacune explorée à travers des tutoriels vidéos.

Dans la première phase, j'ai établi les fondations de l'application en mettant en place les fonctionnalités essentielles telles que l'affichage des patients, la pagination, la recherche et la suppression des patients.

La deuxième partie a été consacrée à l'amélioration de l'esthétique et de l'ergonomie de l'application grâce à l'utilisation de Thymeleaf pour créer des templates réutilisables, ainsi qu'à la mise en place de la validation des formulaires pour garantir l'intégrité des données utilisateur.

Enfin, la dernière phase a été dédiée à la sécurisation de l'application avec l'intégration de Spring Security, offrant des fonctionnalités d'authentification, de gestion des rôles et des autorisations, et renforçant la protection contre les vulnérabilités potentielles.

Ce rapport détaille chaque étape du processus de développement, mettant en lumière les défis rencontrés, les solutions adoptées et les perspectives d'amélioration pour l'avenir de l'application.

Enoncé !

Partie 1 :

<https://www.youtube.com/watch?v=jDm-q-jEbiA>

Créer une application Web JEE basée sur Spring MVC, Thymeleaf et Spring Data JPA qui permet de gérer les patients. L'application doit permettre les fonctionnalités suivantes :

- Afficher les patients
- Faire la pagination
- Chercher les patients
- Supprimer un patient
- Faire des améliorations supplémentaires

Partie 2 :

<https://www.youtube.com/watch?v=eoBE745lDE0>

- Créer une page template
- Faire la validation des formulaires

Partie 3 :

Sécurité avec Spring security : <https://www.youtube.com/watch?v=7VqpC8UD1zM>

Code et explications :

1. Créer une application Web JEE basée sur Spring MVC, Thymeleaf et Spring Data JPA qui permet de gérer les patients.

```
package ma.bouchama.hospital2.entities;

> import ...
16 usages
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
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;

}
```

Dans cette première partie l'entité patient est annotée avec @Entity, ce qui indique qu'elle est mappée à une table dans la base de données. Les annotations Lombok telles que @Data, @NoArgsConstructor, @AllArgsConstructor et @Builder sont utilisées pour générer automatiquement les méthodes getter, setter, constructeurs et le pattern builder. Les champs de l'entité sont annotés avec des contraintes de validation telles que @NotEmpty, @Size et @DecimalMin pour garantir l'intégrité des données. La date de naissance est annotée avec @Temporal pour indiquer le type de données temporelles utilisé en base de données.

```
import ma.bouchama.hospital2.entities.Patient;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

4 usages
public interface PatientRepository extends JpaRepository<Patient, Long> {
    no usages
    Page<Patient> findByNomContains(String keyword, Pageable pageable);

    no usages
    @Query("select p from Patient p where p.nom like :x")
    Page<Patient> chercher(@Param("x") String keyword, Pageable pageable);
}
```

```

@SpringBootApplication
public class Hospital2Application implements CommandLineRunner {
    @Autowired
    private PatientRepository patientRepository;

    public static void main(String[] args) {
        SpringApplication.run(Hospital2Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        Patient patient = new Patient();
        patient.setId(null);
        patient.setNom("hajar");
        patient.setDateNaissance(new Date());
        patient.setScore(20);
        patient.setMalade(false);

        Patient patient2 = new Patient( id: null, nom: "Latifa", new Date(), malade: false, score: 15);

        // On utilise Builder
        Patient patient3 = Patient.builder()
            .nom("Dounia")
            .dateNaissance(new Date())
            .score(10)
            .malade(false)
            .build();
    }
}

```

Ce code représente la classe principale de l'application "Hospital2Application" avec la méthode main qui démarre l'application Spring Boot. L'annotation @SpringBootApplication indique que cette classe est la configuration principale de l'application Spring Boot. La classe implémente également l'interface CommandLineRunner, ce qui signifie que la méthode run sera exécutée au démarrage de l'application. À l'intérieur de cette méthode, des patients fictifs sont ajoutés à la base de données à des fins de démonstration en utilisant le repository PatientRepository.

- Afficher les patients

Liste Patients					
ID	Nom	Date	Malade	Score	
1	Hajar	2024-04-19	false	20	
2	Houria	2024-04-19	true	50	
3	Khadija	2024-04-19	false	17	
4	Fouad	2024-04-19	false	25	

- Faire la pagination

Pckage web :

```

import java.util.List;

@Controller
@AllArgsConstructor
public class PatientController {
    private PatientRepository patientRepository;

    @GetMapping("/index")
    public String index(Model model,
                        @RequestParam(name = "page", defaultValue = "0") int p,
                        @RequestParam(name = "size", defaultValue = "3") int s){
        Page<Patient> pagePatients = patientRepository.findAll(PageRequest.of(p,s));
        model.addAttribute(attributeName: "listPatients",pagePatients.getContent());
        model.addAttribute(attributeName: "pages",new int[pagePatients.getTotalPages()]);
        model.addAttribute(attributeName: "currentPage",p);
        return "patients";
    }
}

```

Ce contrôleur utilise la pagination pour afficher une liste paginée de patients dans une application web, améliorant ainsi l'expérience utilisateur en limitant le nombre d'éléments affichés à la fois.

template :

```
<body>
<div class="p-3">
    <div class="card">
        <div class="card-header">Liste Patients</div>
        <div class="card-body">
            <table class="table">
                <thead>
                    <tr>
                        <th>ID</th> <th>Nom</th> <th>Date</th> <th>Malade</th> <th>Score</th>
                    </tr>
                <tr th:each="p:${listPatients}">
                    <td th:text="${p.id}"></td>
                    <td th:text="${p.nom}"></td>
                    <td th:text="${p.dateNaissance}"></td>
                    <td th:text="${p.malade}"></td>
                    <td th:text="${p.score}"></td>
                </tr>
                </thead>
            </table>
            <ul class="nav nav-pills">
                <li th:each="page,status:${pages}">
                    <a th:href="@{/index(page=${status.index})}" th:class="${currentPage==status.index?'btn btn-info ms-1':'btn btn-outline-info ms-1'}" th:text="${status.index}"></a>
                </li>
            </ul>
        </div>
    </div>
</div>
```

- Chercher les patients

Pckage web :

La méthode index du contrôleur PatientController permet la recherche paginée des patients par nom et affiche les résultats dans une vue spécifique.

```
import org.springframework.web.bind.annotation.RequestParam;

import java.util.List;

@Controller
@AllArgsConstructor
public class PatientController {
    private PatientRepository patientRepository;
    @GetMapping("/index")
    public String index(Model model,
                        @RequestParam(name = "page", defaultValue = "0")int p,
                        @RequestParam(name = "size", defaultValue = "3")int s,
                        @RequestParam(name = "keyword", defaultValue = "")String kw) {
        Page<Patient> pagePatients = patientRepository.findByNomContains(kw,PageRequest.of(p,s));
        model.addAttribute("listPatients",pagePatients.getContent());
        model.addAttribute("pages",new int[pagePatients.getTotalPages()]);
        model.addAttribute("currentPage",p);
        model.addAttribute("keyword",kw);
        return "patients";
    }
}
```

template:

```
<div class="card-body">
    <form method="get" th:action="@{index}">
        <label>Keyword : </label>
        <input type="text" name="keyword" th:value="${keyword}">
        <button type="submit" class="btn btn-info"></button>
    </form>
    <table class="table">
        <thead>
            <tr>
                <th>ID</th> <th>Nom</th> <th>Date</th> <th>Malade</th> <th>Score</th>
            </tr>
        </thead>
        <tr th:each="p:${listPatients}">
            <td th:text="${p.id}"></td>
            <td th:text="${p.nom}"></td>
            <td th:text="${p.dateNaissance}"></td>
            <td th:text="${p.malade}"></td>
            <td th:text="${p.score}"></td>
        </tr>
    </thead>
    <table class="nav nav-pills">
        <li th:each="page,status:${pages}">
            <a th:href="@{/index(page=${status.index}, keyword=${keyword})}" th:class="${currentPage==status.index?'btn btn-info ms-1':'btn btn-outline-info ms-1'}" th:text="${status.index}"></a>
        </li>
    </ul>
</div>
```

• Supprimer un patient

Package web

```
}
```

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

```
}
```

cette méthode permet à un administrateur de supprimer un patient spécifique et redirige ensuite l'utilisateur vers la page principale avec les résultats mis à jour.

Template:

```
<button type="submit" class="btn btn-info"></button>
</form>
<table class="table">
    <thead>
        <tr>
            <th>ID</th> <th>Nom</th> <th>Date</th> <th>Malade</th> <th>Score</th>
        </tr>
    </thead>
    <tr th:each="p:${listPatients}">
        <td th:text="${p.id}"></td>
        <td th:text="${p.nom}"></td>
        <td th:text="${p.dateNaissance}"></td>
        <td th:text="${p.malade}"></td>
        <td th:text="${p.score}"></td>
        <td>
            <a onclick="javascript:return confirm('Etes vous sure ?')!" th:href="@{delete(id=${p.id}, keyword=${keyword}, page=${currentPage})}" class="btn btn-danger">Delete</a>
        </td>
    </tr>
</thead>
</table>
<ul class="nav nav-pills">
    <li th:each="page,status:${pages}">
        <a th:href="@{/index(page=${status.index}, keyword=${keyword})}" th:class="${currentPage==status.index?'btn btn-info ms-1':'btn btn-outline-info ms-1'}" th:text="${status.index}"></a>
    </li>
</ul>
```

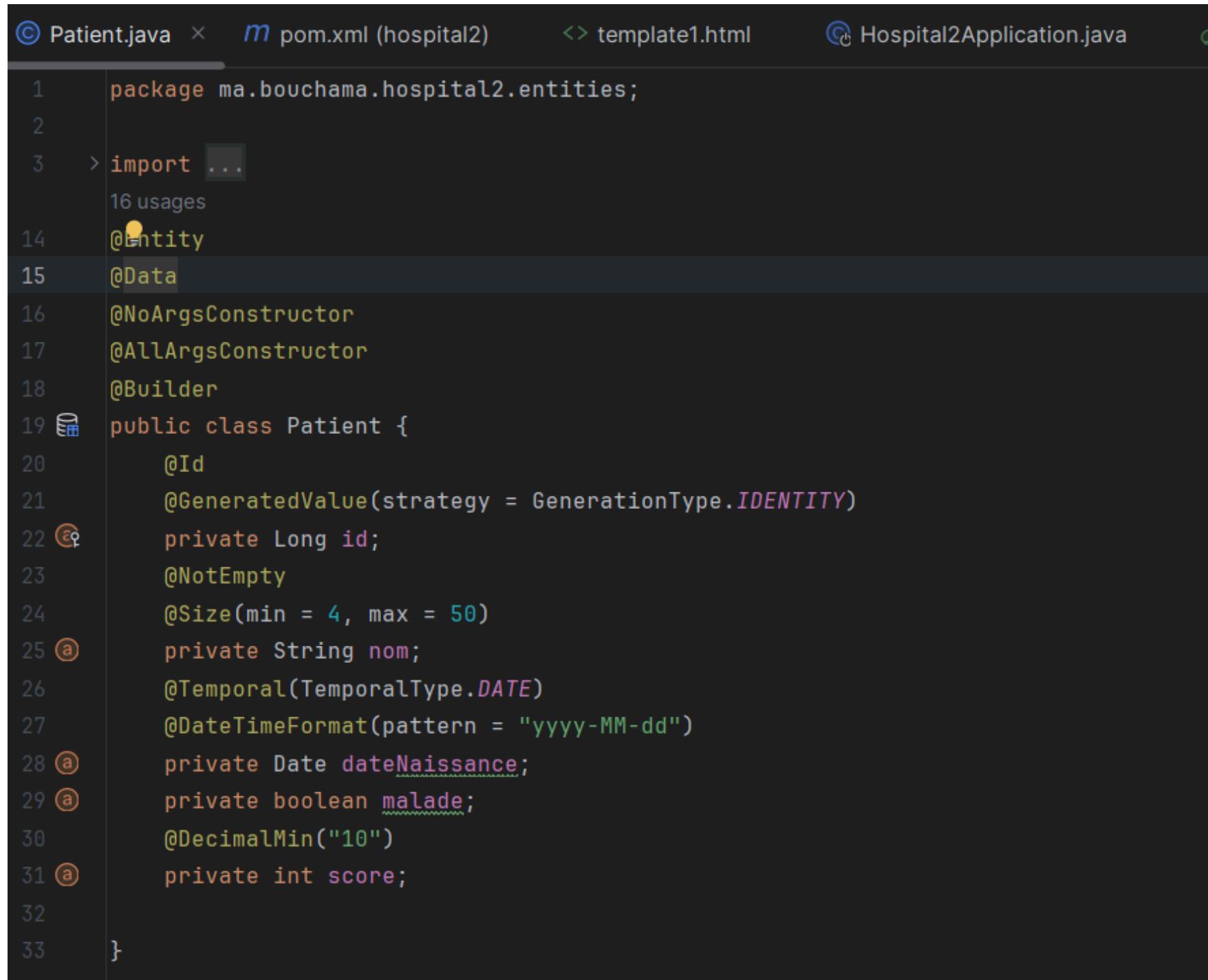
- l'affichage finale de cette partie :

Liste Patients					
ID	Nom	Date	Malade	Score	
1	Hajar	2024-04-19	false	20	
7	Khadija	2024-04-19	false	17	
8	Fouad	2024-04-19	false	25	

Partie 2 :

- Créer une page template
- Faire la validation des formulaires

La classe Patient :



The screenshot shows a code editor with the Patient.java file open. The file contains Java code defining a Patient entity with various annotations for persistence and validation. The code includes imports, package declaration, entity annotations, and field definitions with their respective annotations like @Id, @GeneratedValue, @Size, @Temporal, @DateTimeFormat, @DecimalMin, and @Builder.

```
1 package ma.bouchama.hospital2.entities;
2
3 > import ...
4     16 usages
5
6 @Entity
7 @Data
8 @NoArgsConstructor
9 @AllArgsConstructor
10 @Builder
11 public class Patient {
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Long id;
15     @NotEmpty
16     @Size(min = 4, max = 50)
17     private String nom;
18     @Temporal(TemporalType.DATE)
19     @DateTimeFormat(pattern = "yyyy-MM-dd")
20     private Date dateNaissance;
21     private boolean malade;
22     @DecimalMin("10")
23     private int score;
24 }
```

l'interface PatientRepository :

```
1 package ma.bouchama.hospital2.repository;
2
3 import ma.bouchama.hospital2.entities.Patient;
4 import org.springframework.data.domain.Page;
5 import org.springframework.data.domain.Pageable;
6 import org.springframework.data.jpa.repository.JpaRepository;
7 import org.springframework.data.jpa.repository.Query;
8 import org.springframework.data.repository.query.Param;
9
10
11 4 usages
12 0  public interface PatientRepository extends JpaRepository<Patient, Long> {
13      1 usage
14          Page<Patient> findByNomContains(String keyword, Pageable pageable);
15
16
17      no usages
18      @Query("select p from Patient p where p.nom like :x")
19      Page<Patient> chercher(@Param("x") String keyword, Pageable pageable);
20  }
```

Ce code définit un repository Spring Data JPA pour l'entité Patient. Il étend l'interface JpaRepository, qui fournit des méthodes CRUD standard pour interagir avec la base de données. Deux méthodes de recherche sont définies : findByNomContains qui recherche les patients dont le nom contient un mot-clé spécifié, et chercher qui utilise une requête JPQL personnalisée pour rechercher les patients dont le nom correspond à un motif spécifié. Les résultats sont paginés grâce à l'objet Pageable, permettant ainsi de récupérer les données par petites portions.

le controller PatientController :

```
1
2
3 @GetMapping("/formPatients")
4 public String formPatients(Model model){
5     model.addAttribute("patient", new Patient());
6     return "formPatients";
7 }
8
9 @PostMapping(path = "/save")
10 public String save(Model model, @Valid Patient patient, BindingResult bindingResult,
11                     @RequestParam(defaultValue = "0") int page,
12                     @RequestParam(defaultValue = "") String keyword){
13     if (bindingResult.hasErrors()) return "formPatients";
14     patientRepository.save(patient);
15     return "redirect:/index?page="+page+"&keyword="+keyword;
16 }
17
18
19 @GetMapping("/editPatient")
20 public String editPatient(Model model, Long id, String keyword, int page){
21     Patient patient=patientRepository.findById(id).orElse(null);
22     if (patient==null) throw new RuntimeException("Patient introuvable");
23     model.addAttribute("patient", patient);
24     model.addAttribute("page", page);
25     model.addAttribute("keyword", keyword);
26     return "editPatient";
27 }
```

La méthode formPatients est utilisée pour afficher le formulaire de création d'un nouveau patient pour les administrateurs. La méthode save est appelée lors de la soumission du formulaire de création ou de modification d'un patient. Si les données saisies ne respectent pas les contraintes de validation, la méthode renvoie le formulaire avec les erreurs. La méthode editPatient récupère les informations d'un patient existant pour les afficher dans un formulaire de modification. Si le patient n'existe pas, une exception est levée.

les templates:

template.html

Elle utilise Thymeleaf, un moteur de template pour Java, pour la gestion dynamique des contenus. La section <head> définit les métadonnées de la page et inclut les liens vers les ressources CSS et JavaScript, notamment la bibliothèque Bootstrap pour le style. La section <body> contient la barre de navigation (<nav>) avec des liens vers différentes pages de l'application, notamment la page d'accueil, les pages de gestion des patients, et le profil utilisateur. Elle intègre également un mécanisme de déconnexion. La section <section> est destinée à être remplie dynamiquement selon le contenu spécifique de chaque page.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <link rel="stylesheet" type="text/css" href="/webjars/bootstrap/5.3.3/css/bootstrap.min.css">
    <script src="/webjars/bootstrap/5.3.3/js/bootstrap.bundle.js"></script>
</head>
<body>
    <!-- Grey with black text -->
    <nav class="navbar navbar-expand-sm bg-dark navbar-dark">
        <div class="container-fluid">
            <ul class="navbar-nav">
                <li class="nav-item">
                    <a class="nav-link active" th:href="@{index}">Home</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Link</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Link</a>
                </li>
                <a class="nav-link" href="#">Link</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Link</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link disabled" href="#">Disabled</a>
                </li>
                <li class="nav-item dropdown">
                    <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
                        Patients
                    </a>
                    <ul class="dropdown-menu">
                        <li><a class="dropdown-item" th:href="@{/formPatients}">Nouveau</a></li>
                        <li><a class="dropdown-item" th:href="@{index}">Chercher</a></li>
                    </ul>
                </li>
            </ul>
            <ul class="navbar-nav">
                <li class="nav-item dropdown">
                    <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
                        [Username]
                    </a>
                    <ul class="dropdown-menu">
                        <li><a class="dropdown-item" th:href="@{/formPatients}">Logout</a></li>
                    </ul>
                </li>
            </ul>
        <ul class="navbar-nav">
            <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
                    [Username]
                </a>
                <ul class="dropdown-menu">
                    <li><a class="dropdown-item" th:href="@{/formPatients}">Logout</a></li>
                </ul>
            </li>
        </ul>
    </div>
</nav>
<section layout:fragment="content1">
</section>
```

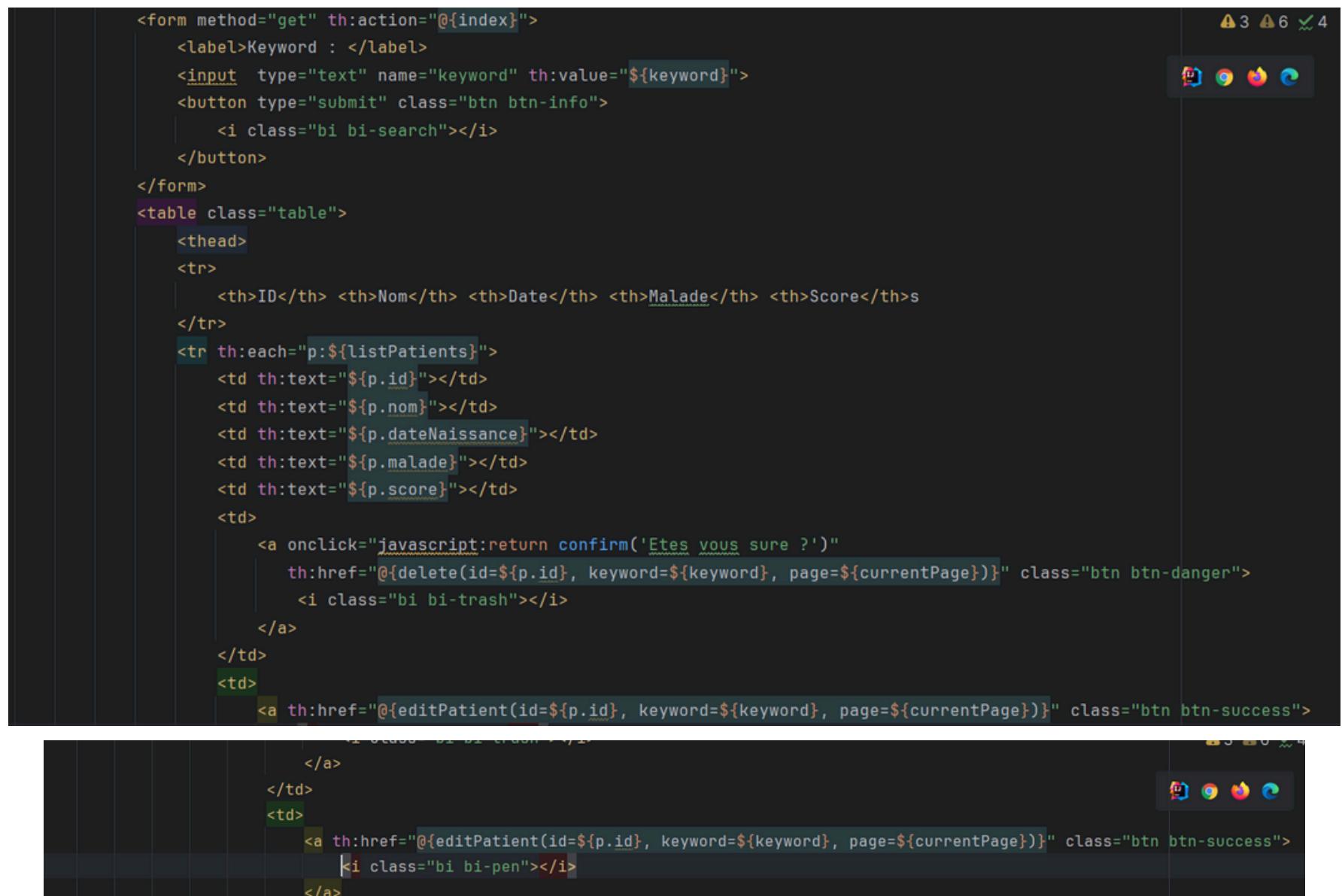
patient.html

La section <head> définit les métadonnées de la page et inclut les liens vers les ressources CSS, notamment la bibliothèque Bootstrap pour le style. La section <body> contient le contenu principal de la page, notamment une liste de patients affichée dans un tableau. Les utilisateurs peuvent rechercher des patients par mot-clé et naviguer entre les pages de résultats. Les administrateurs ont également la possibilité de supprimer ou modifier des patients en fonction de leurs autorisations.

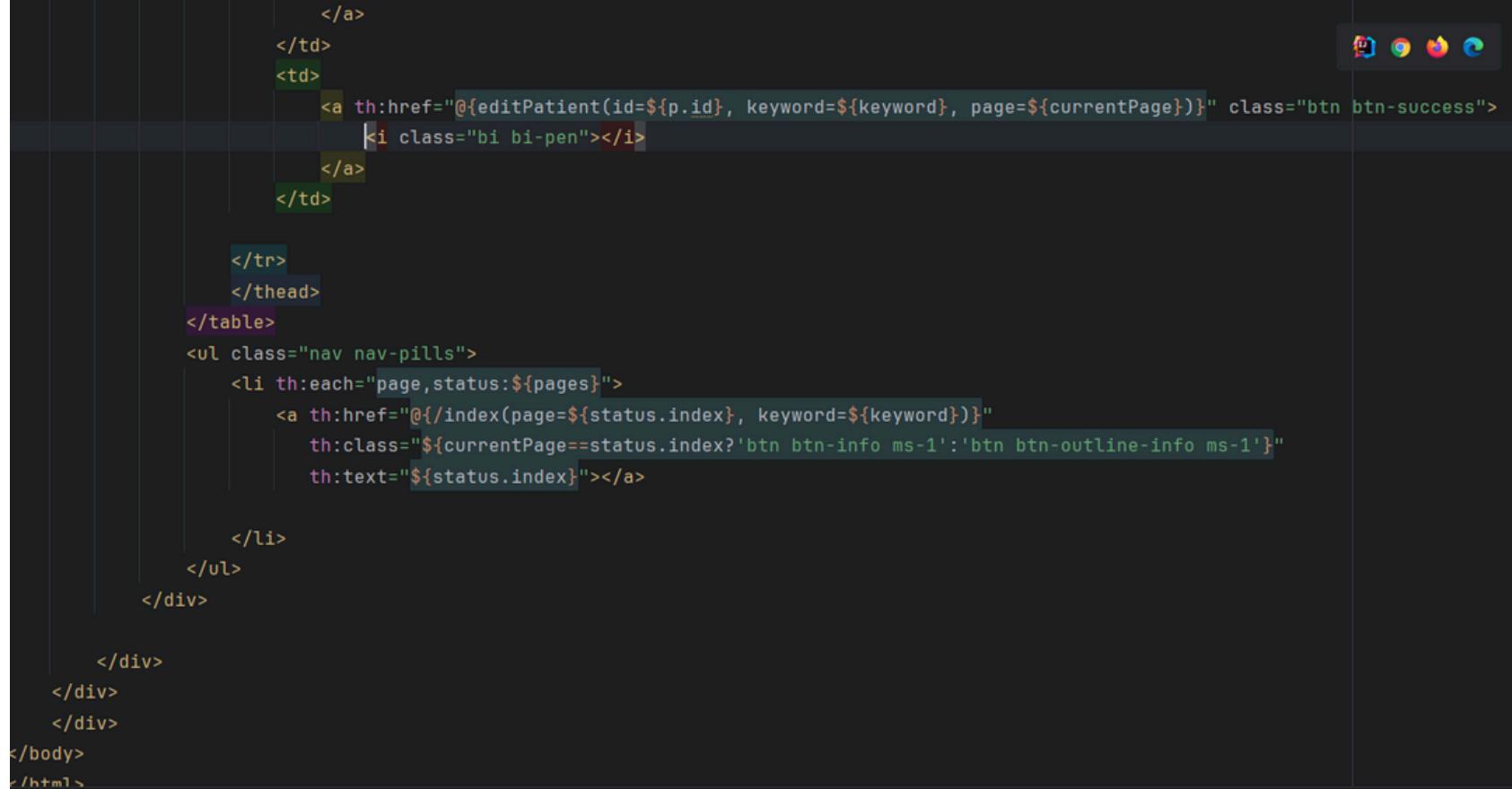
```
<!DOCTYPE html>
<html lang="en">
    xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="template1"

>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <link rel="stylesheet" href="/webjars/bootstrap/5.3.3/css/bootstrap.min.css">
    <link rel="stylesheet" href="/webjars/bootstrap-icons/1.11.3/font/bootstrap-icons.css">
</head>
<body>
    <div layout:fragment="content1">

        <div class="p-3">
            <div class="card">
                <div class="card-header">Liste Patients</div>
                <div class="card-body">
                    <form method="get" th:action="@{index}">
                        <label>Keyword : </label>
                        <input type="text" name="keyword" th:value="${keyword}">
                        <button type="submit" class="btn btn-info">
                            <i class="bi bi-search"></i>
                        </button>
                    </form>
                </div>
            </div>
        </div>
    </div>
</body>
```



```
<form method="get" th:action="@{index}">
    <label>Keyword : </label>
    <input type="text" name="keyword" th:value="${keyword}">
    <button type="submit" class="btn btn-info">
        <i class="bi bi-search"></i>
    </button>
</form>
<table class="table">
    <thead>
        <tr>
            <th>ID</th> <th>Nom</th> <th>Date</th> <th>Malade</th> <th>Score</th>
        </tr>
    <tbody>
        <tr th:each="p:${listPatients}">
            <td th:text="${p.id}"></td>
            <td th:text="${p.nom}"></td>
            <td th:text="${p.dateNaissance}"></td>
            <td th:text="${p.malade}"></td>
            <td th:text="${p.score}"></td>
            <td>
                <a onclick="javascript:return confirm('Etes vous sure ?')"\>
                    th:href="@{delete(id=${p.id}, keyword=${keyword}, page=${currentPage})}" class="btn btn-danger">
                        <i class="bi bi-trash"></i>
                    </a>
                </td>
                <td>
                    <a th:href="@{editPatient(id=${p.id}, keyword=${keyword}, page=${currentPage})}" class="btn btn-success">
                        <i class="bi bi-pencil"></i>
                    </a>
                </td>
            </td>
        </tr>
    </tbody>
</table>
```



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

editPatient.html

Cette template est conçue pour permettre la modification des informations d'un patient au sein de l'application. Elle utilise Thymeleaf pour la gestion dynamique des contenus. Dans la section <head>, les métadonnées de la page sont définies et les liens vers les ressources CSS, comme Bootstrap, sont inclus. Dans la section <body>, un formulaire est présenté aux utilisateurs pour qu'ils puissent saisir et modifier les détails du patient, tels que son nom, sa date de naissance, son état de maladie et son score. Une fois les modifications terminées, les utilisateurs peuvent enregistrer les données en cliquant sur le bouton "Save".

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

>
<head>
    <meta charset="UTF-8">
    <title>Edit Patients</title>
    <link rel="stylesheet" href="/webjars/bootstrap/5.3.3/css/bootstrap.min.css">
    <link rel="stylesheet" href="/webjars/bootstrap-icons/1.11.3/font/bootstrap-icons.css">
</head>
<body>
    <div layout:fragment="content1">
        <div class="col-md-6 offset-3">
            <form method="post" th:action="@{save(page=${page}, keyword=${keyword})}">
                <div>
                    <label for="id">Id : </label>
                    <label th:text="${patient.id}"></label>
                    <input id="id" class="form-control" type="hidden" name="id" th:value="${patient.id}">
                </div>
                <div>
                    <label for="nom">Nom</label>
                    <input id="nom" class="form-control" type="text" name="nom" th:value="${patient.nom}">
                    <span class="text-danger" th:errors="${patient.nom}"></span>
                    <input id="nom" class="form-control" type="text" name="nom" th:value="${patient.nom}">
                    <span class="text-danger" th:errors="${patient.nom}"></span>
                </div>
                <div>
                    <label>Date Naissance</label>
                    <input class="form-control" type="date" name="dateNaissance" th:value="${patient.dateNaissance}">
                    <span class="text-danger" th:errors="${patient.dateNaissance}"></span>
                </div>
                <div>
                    <label>Malade</label>
                    <input type="checkbox" name="malade" th:checked="${patient.malade}">
                    <span class="text-danger" th:errors="${patient.malade}"></span>
                </div>
                <div>
                    <label>Score</label>
                    <input class="form-control" type="text" name="score" th:value="${patient.score}">
                    <span class="text-danger" th:errors="${patient.score}"></span>
                </div>
                <button type="submit" class="btn btn-primary">Save</button>
            </form>
        </div>
    </div>
<!DOCTYPE html>
<html lang="en"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="template1">

>
<head>
    <meta charset="UTF-8">
    <title>Form Patients</title>
    <link rel="stylesheet" href="/webjars/bootstrap/5.3.3/css/bootstrap.min.css">
    <link rel="stylesheet" href="/webjars/bootstrap-icons/1.11.3/font/bootstrap-icons.css">
</head>
<body>
    <div layout:fragment="content1">
        <div class="col-md-6 offset-3">
            <form method="post" th:action="@{save}">
                <div>
                    <label for="nom">Nom</label>
                    <input id="nom" class="form-control" type="text" name="nom" th:value="${patient.nom}">
                    <span class="text-danger" th:errors="${patient.nom}"></span>
                </div>
                <div>
                    <label>Date Naissance</label>
                    <input class="form-control" type="date" name="dateNaissance" th:value="${patient.dateNaissance}">
                    <span class="text-danger" th:errors="${patient.dateNaissance}"></span>
                </div>
            </form>
        </div>
    </div>

```

Nom

must not be empty

size must be between 4 and 50

Date Naissance

 mm/dd/yyyy CALENDARMalade

Score

 0

must be greater than or equal to 10

Save

Id : 125

Nom

 Hajar

Date Naissance

 08/19/2002 CALENDARMalade

Score

 20Save

Liste Patients

Keyword : hajar

S

ID	Nom	Date	Malade	Score	Delete	Edit
125	Hajar	2024-04-19	false	20	Delete	Edit
129	Hajar	2024-04-19	false	20	Delete	Edit
133	Hajar	2024-04-19	false	20	Delete	Edit

01234567891011121314151617181920212223242526272829303132333435

Liste Patients

Keyword : 

S

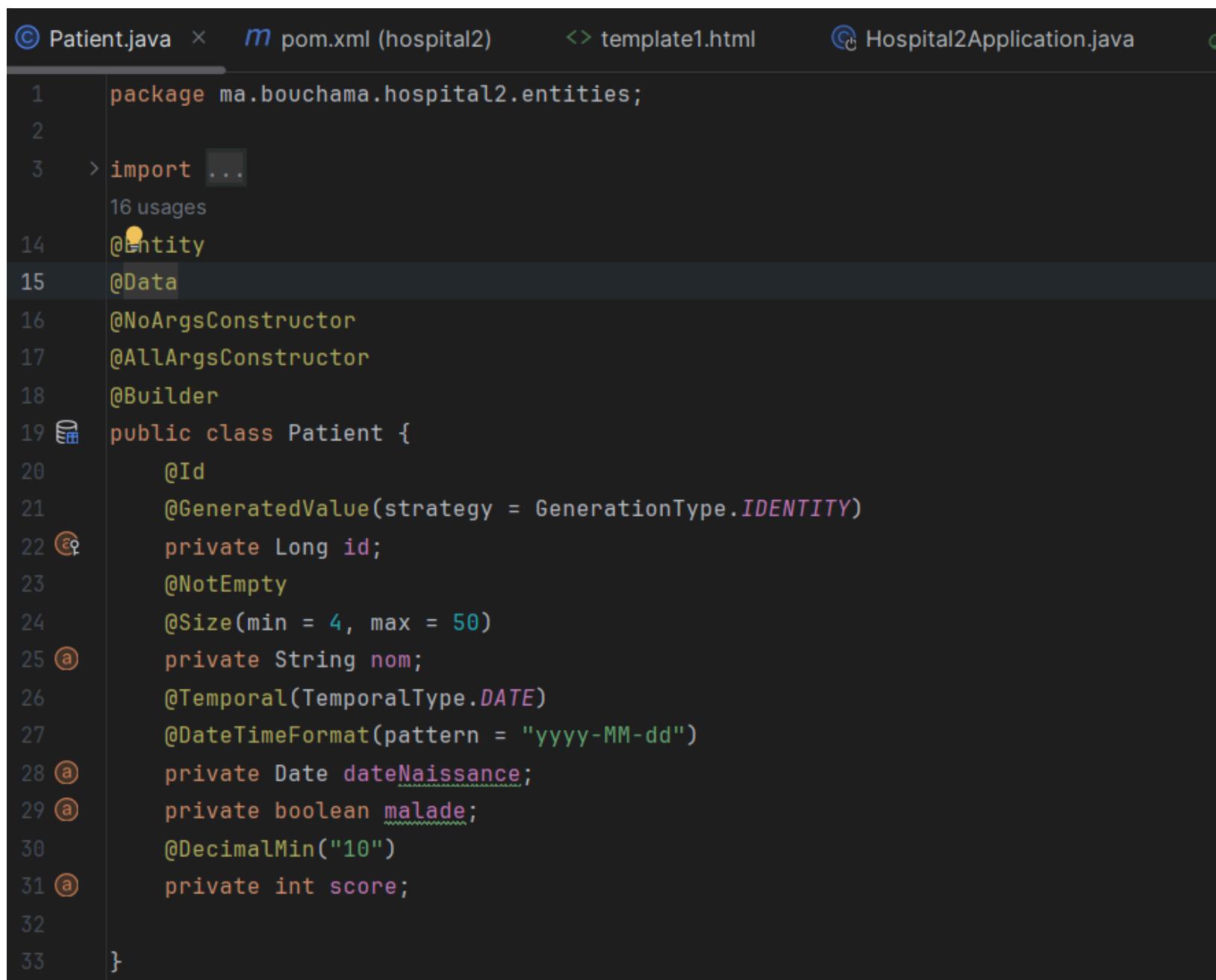
ID	Nom	Date	Malade	Score	Delete	Edit
1	Hajar	2024-04-10	false	20	Delete	Edit
7	Khadija	2024-04-19	true	15	Delete	Edit
8	Fouad	2024-04-19	false	25	Delete	Edit

0123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869707172737475767778798081828384858687888990919293949596979899100101102103104105106107108109110111112113114115116117118119120121122123124125126127128129130131132133134135136137138139140141

Partie 3 :

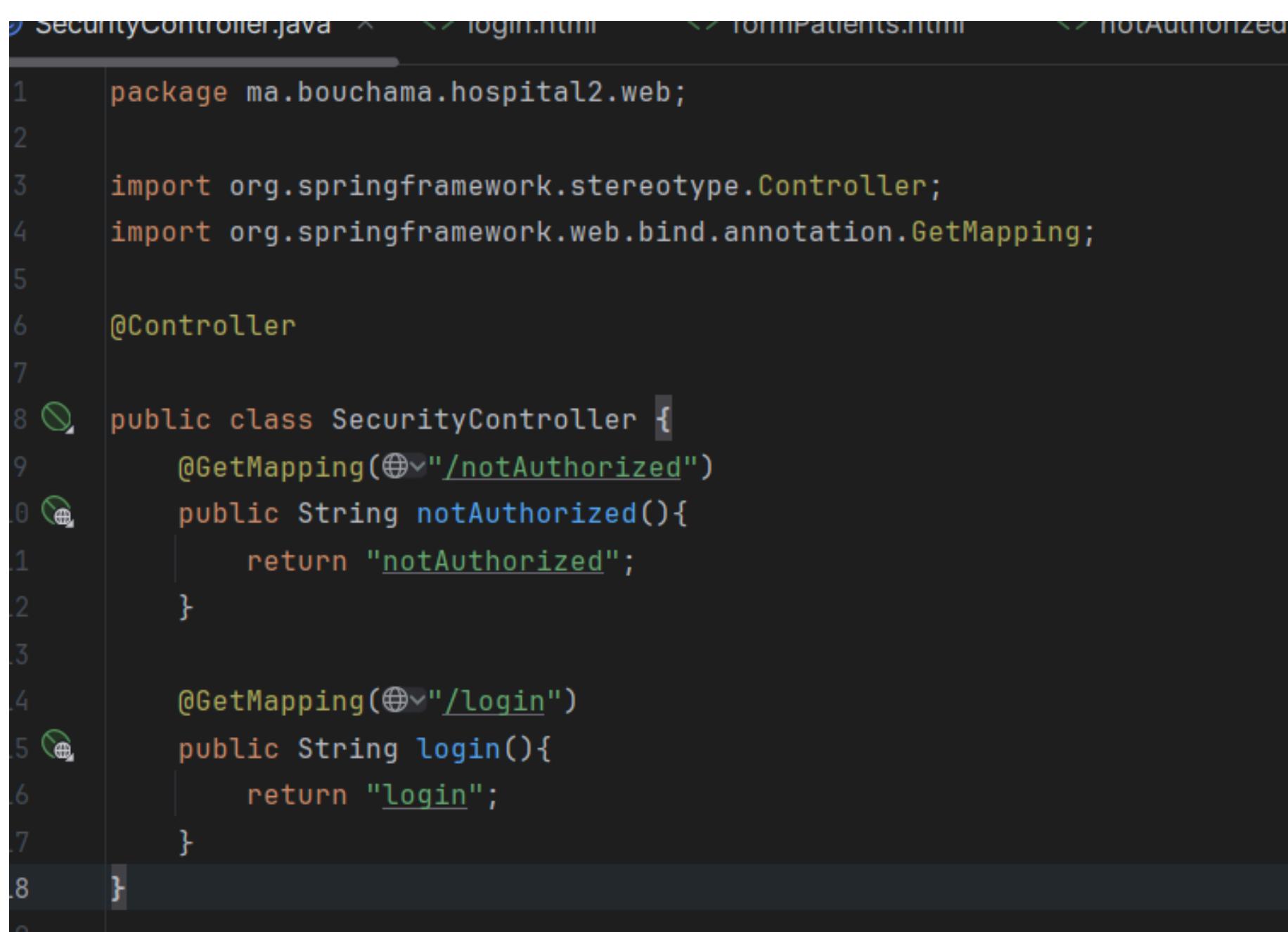
La classe Patient :

Cette classe représente l'entité Patient de l'application. Elle est annotée avec @Entity pour être gérée par JPA. Les annotations @Data, @NoArgsConstructor, @AllArgsConstructor, et @Builder sont de Lombok pour générer automatiquement des méthodes de getter, setter, constructeurs, et le pattern builder. Les champs de la classe sont annotés avec des contraintes de validation telles que @NotEmpty, @Size, et @DecimalMin pour garantir l'intégrité des données. La date de naissance est annotée avec @Temporal et @DateTimeFormat pour définir le format de date attendu.



```
1 package ma.bouchama.hospital2.entities;
2
3 > import ...;
4
5 @Entity
6 @Data
7 @NoArgsConstructor
8 @AllArgsConstructor
9 @Builder
10 public class Patient {
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private Long id;
14     @NotEmpty
15     @Size(min = 4, max = 50)
16     private String nom;
17     @Temporal(TemporalType.DATE)
18     @DateTimeFormat(pattern = "yyyy-MM-dd")
19     private Date dateNaissance;
20     private boolean malade;
21     @DecimalMin("10")
22     private int score;
23 }
24
25 }
```

SecurityController :



```
1 package ma.bouchama.hospital2.web;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.GetMapping;
5
6 @Controller
7
8 public class SecurityController {
9     @GetMapping("/notAuthorized")
10    public String notAuthorized(){
11        return "notAuthorized";
12    }
13
14    @GetMapping("/login")
15    public String login(){
16        return "login";
17    }
18 }
```

Ce contrôleur gère les requêtes liées à la sécurité de l'application. Il définit deux méthodes pour les URL "/notAuthorized" et "/login". Lorsqu'un utilisateur tente d'accéder à une ressource non autorisée, il est redirigé vers la page "notAuthorized". La méthode "login" est utilisée pour afficher la page de connexion de l'application.

PatientController :

```
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.List;

@Controller
@AllArgsConstructor
public class PatientController {
    private PatientRepository patientRepository;
    @GetMapping("/user/index")
    public String index(Model model,
                        @RequestParam(name = "page", defaultValue = "0") int p,
                        @RequestParam(name = "size", defaultValue = "3") int s,
                        @RequestParam(name = "keyword", defaultValue = "") String kw
    ) {
        Page<Patient> pagePatients = patientRepository.findByNomContains(kw, PageRequest.of(p, s));
        model.addAttribute("listPatients", pagePatients.getContent());
        model.addAttribute("pages", new int[pagePatients.getTotalPages()]);
        model.addAttribute("currentPage", p);
        model.addAttribute("keyword", kw);
        return "patients";
    }
    @GetMapping("/admin/delete")
    @PreAuthorize("hasRole('ROLE_ADMIN')")
    public String delete(Long id, String keyword, int page) {
        patientRepository.deleteById(id);
        return "redirect:/user/index?page=" + page + "&keyword=" + keyword;
    }
}
```

```
@GetMapping("/")
public String home(){
    return "redirect:/user/index";
}
no usages
@ResponseBody
public List<Patient> lisPatients(){
    return patientRepository.findAll();
}
@GetMapping("/admin/formPatients")
public String formPatients(Model model){
    model.addAttribute("patient", new Patient());
    return "formPatients";
}
@PostMapping(path = "/admin/save")
@PreAuthorize("hasRole('ROLE_ADMIN')")
public String save(Model model, @Valid Patient patient, BindingResult bindingResult,
                   @RequestParam(defaultValue = "0") int page,
                   @RequestParam(defaultValue = "") String keyword){
    if (bindingResult.hasErrors()) return "formPatients";
    patientRepository.save(patient);
    return "formPatients";
}

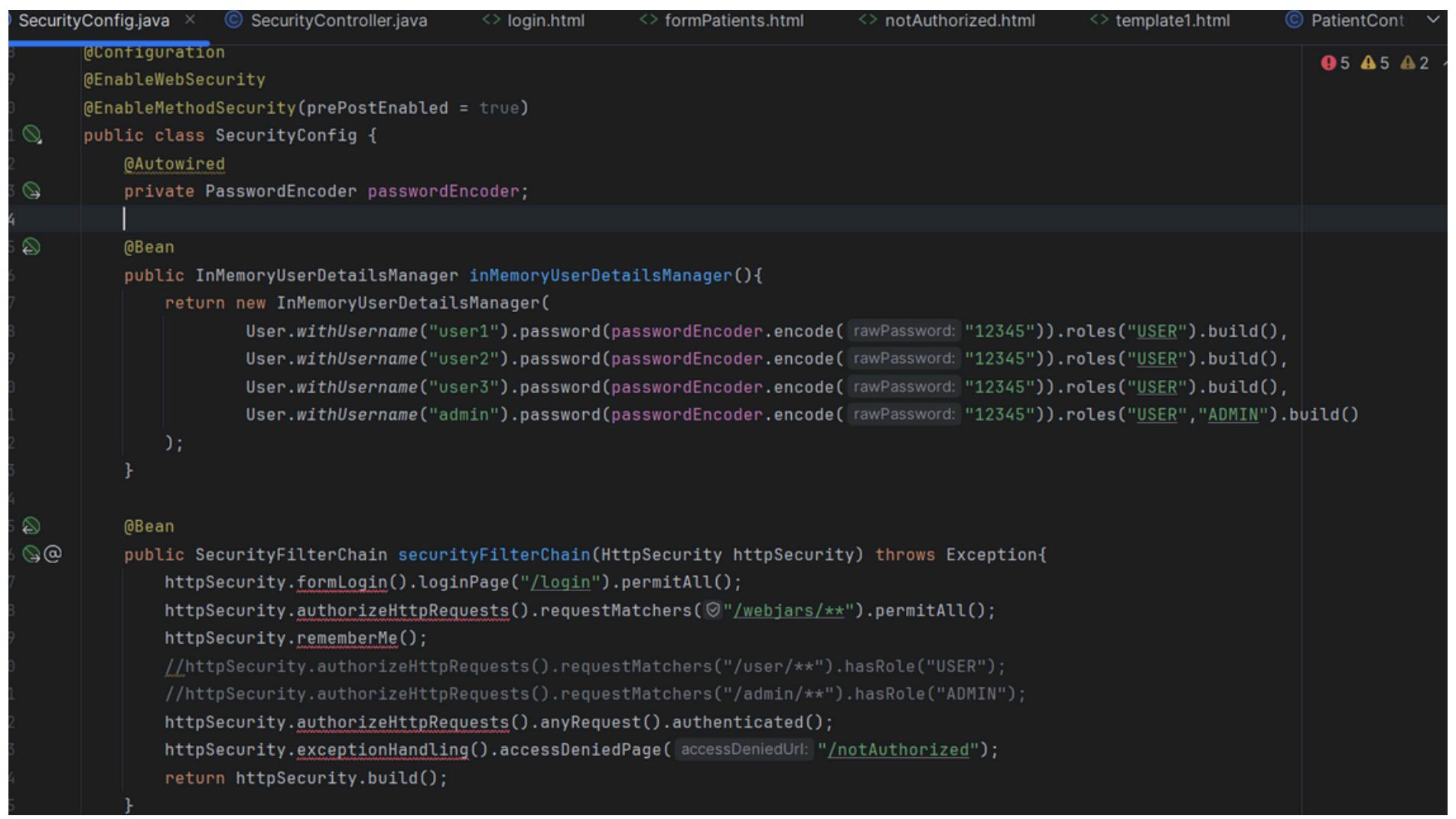
@GetMapping("/admin/editPatient")
@PreAuthorize("hasRole('ROLE_ADMIN')")
public String formPatients(Model model, Long id, String keyword, int page){
    Patient patient=patientRepository.findById(id).orElse(null);
    if (patient==null) throw new RuntimeException("Patient introuvable");
    model.addAttribute("patient", patient);
    model.addAttribute("page", page);
    model.addAttribute("keyword", keyword);
    return "editPatient";
}

}
```

les annotations ci-dessus **@PreAuthorize** sont utilisées pour restreindre l'accès aux fonctionnalités spécifiques aux utilisateurs ayant le rôle "ROLE_ADMIN". Ces annotations garantissent que seuls les utilisateurs avec les privilèges appropriés peuvent effectuer des actions telles que la suppression (delete) et la sauvegarde (save) des patients. Cela renforce la sécurité de l'application en limitant les actions sensibles aux seuls utilisateurs autorisés.

SecurityConfig :

Ce code configure la sécurité de notre application Spring Boot. Il utilise Spring Security pour gérer l'authentification et l'autorisation des utilisateurs. Les utilisateurs sont stockés en mémoire avec leurs noms d'utilisateur, mots de passe et rôles associés. Les URL "/login" et "/webjars/**" sont accessibles publiquement, tandis que toutes les autres requêtes nécessitent une authentification. Un utilisateur avec le rôle "ADMIN" a accès à certaines ressources supplémentaires.



```

1  @Configuration
2  @EnableWebSecurity
3  @EnableMethodSecurity(prePostEnabled = true)
4  public class SecurityConfig {
5
6      @Autowired
7      private PasswordEncoder passwordEncoder;
8
9
10     @Bean
11     public InMemoryUserDetailsManager inMemoryUserDetailsManager(){
12         return new InMemoryUserDetailsManager(
13             User.withUsername("user1").password(passwordEncoder.encode("12345")).roles("USER").build(),
14             User.withUsername("user2").password(passwordEncoder.encode("12345")).roles("USER").build(),
15             User.withUsername("user3").password(passwordEncoder.encode("12345")).roles("USER").build(),
16             User.withUsername("admin").password(passwordEncoder.encode("12345")).roles("USER", "ADMIN").build()
17         );
18     }
19
20
21     @Bean
22     public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception{
23         httpSecurity.formLogin().loginPage("/login").permitAll();
24         httpSecurity.authorizeHttpRequests().requestMatchers(ignored("/webjars/**")).permitAll();
25         httpSecurity.rememberMe();
26         //httpSecurity.authorizeHttpRequests().requestMatchers("/user/**").hasRole("USER");
27         //httpSecurity.authorizeHttpRequests().requestMatchers("/admin/**").hasRole("ADMIN");
28         httpSecurity.authorizeHttpRequests().anyRequest().authenticated();
29         httpSecurity.exceptionHandling().accessDeniedPage(accessDeniedUrl: "/notAuthorized");
30         return httpSecurity.build();
31     }
32 }

```

La partie du mise à jour :

Les instructions SQL ci-dessous sont utilisées pour définir les structures des tables "**users**" et "**authorities**" dans la base de données. La table "users" stocke les informations essentielles des utilisateurs, telles que le nom d'utilisateur, le mot de passe et un indicateur de statut activé/désactivé. La table "authorities" est conçue pour associer des rôles ou des autorisations spécifiques aux utilisateurs. Une contrainte de clé étrangère assure l'intégrité référentielle entre ces deux tables, garantissant que chaque autorité est attribuée à un utilisateur existant. De plus, un index unique sur la table "authorities" améliore les performances des requêtes et garantit que chaque paire nom d'utilisateur-autorité est unique, évitant ainsi les doublons.

```

create table if not exists users(username varchar(50) not null primary key, password varchar(500) not null,
                                enabled boolean not null);
create table if not exists authorities (username varchar(50) not null, authority varchar(50) not null,
                                         constraint fk_authorities_users foreign key(username) references users(username));
create unique index if not exists ix_auth_username on authorities (username, authority);

```

```

spring.application.name=hospital2
server.port =8084
#spring.datasource.url=jdbc:h2:mem:hospital2-db
#spring.h2.console.enabled=true
spring.datasource.url=jdbc:mysql://localhost:3306/hospital2_db?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.mvc.format.date=dd/MM/yyyy

```

HospitalApplication :

dans notre classe d'application nous avons défini un bean CommandLineRunner en utilisant la méthode CommandLineRunner pour initialiser des utilisateurs dans l'application au démarrage. Il vérifie l'existence de certains utilisateurs (user1, user2 et admin) dans le gestionnaire JdbcUserDetailsManager, qui gère les détails des utilisateurs et des rôles dans une base de données JDBC. Si ces utilisateurs n'existent pas, ils sont créés avec des mots de passe encodés et des rôles spécifiques. user1 et user2 sont créés avec le rôle USER, tandis que admin est créé avec les rôles **USER** et **ADMIN**. Cela permet de s'assurer que des comptes utilisateur de base sont disponibles dans l'application dès son démarrage, facilitant ainsi les tests et la gestion initiale des utilisateurs.

```

// @Bean
no usages new *
CommandLineRunner commandLineRunner(JdbcUserDetailsManager jdbcUserDetailsManager, PasswordEncoder passwordEncoder)
{
    return args -> {
        if (!jdbcUserDetailsManager.userExists( username: "user1") ) {
            jdbcUserDetailsManager.createUser(User.withUsername("user1")
                .password(passwordEncoder.encode( rawPassword: "0000"))
                .roles("USER").build());
        }

        if (!jdbcUserDetailsManager.userExists( username: "user2") ) {
            jdbcUserDetailsManager.createUser(User.withUsername("user2")
                .password(passwordEncoder.encode( rawPassword: "0000"))
                .roles("USER").build());
        }

        if (!jdbcUserDetailsManager.userExists( username: "admin") ) {
            jdbcUserDetailsManager.createUser(User.withUsername("admin")
                .password(passwordEncoder.encode( rawPassword: "0000"))
                .roles("USER", "ADMIN").build());
        }
    };
}

```

AccountService

Le service AccountService définit des méthodes pour ajouter de nouveaux utilisateurs et rôles, attribuer des rôles aux utilisateurs, retirer des rôles et charger des utilisateurs par leur nom d'utilisateur.

```

AccountService.java ✘ AccountServiceimpl.java ✘ Hospital2Application.java ✘ SecurityConfig.java ✘ User
package ma.bouchama.hospital2.security.service;

import ma.bouchama.hospital2.security.entities.AppRole;
import ma.bouchama.hospital2.security.entities.AppUser;

4 usages 1 implementation new *
public interface AccountService {
    3 usages 1 implementation new *
    AppUser addNewUser(String username, String password, String email, String confirmPassword);
    2 usages 1 implementation new *
    AppRole addNewRole(String role);
    3 usages 1 implementation new *
    void addRoleToUser(String username, String role);
    no usages 1 implementation new *
    void removeRoleFromUser(String username, String role);
    1 usage 1 implementation new *
    AppUser loadUserByUsername(String username);
}

```

AccountServiceImpl

La classe AccountServiceImpl implémente l'interface AccountService et fournit des fonctionnalités essentielles telles que l'ajout de nouveaux utilisateurs, l'ajout de nouveaux rôles, l'attribution et la suppression de rôles aux utilisateurs, ainsi que le chargement des utilisateurs par leur nom d'utilisateur

AccountService.java AccountServiceImpl.java Hospital2Application.java SecurityConfig.java UserD

```
package ma.bouchama.hospital2.security.service;

import lombok.AllArgsConstructor;
import ma.bouchama.hospital2.security.entities.AppRole;
import ma.bouchama.hospital2.security.entities.AppUser;
import ma.bouchama.hospital2.security.repo.AppRoleRepository;
import ma.bouchama.hospital2.security.repo.AppUserRepository;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import jakarta.transaction.Transactional;

import java.util.UUID;
new *
@Service @AllArgsConstructor
public class AccountServiceImpl implements AccountService
{
    private AppUserRepository appUserRepository;
    private AppRoleRepository appRoleRepository;
    private PasswordEncoder passwordEncoder;
    3 usages new *
    @Override

@Override
public AppUser addNewUser(String username, String password, String email, String confirmPassword)
{
    AppUser appUser=appUserRepository.findByUsername(username);
    if(appUser!=null) throw new RuntimeException("ce utilisateur existe déjà");
    if(!password.equals(confirmPassword)) throw new RuntimeException("mot de passe incorrecte");
    appUser=AppUser.builder()
        .userId(UUID.randomUUID().toString())
        .username (username)
        .password(passwordEncoder.encode(password))
        .email(email)
        .build();
    return appUserRepository.save(appUser);
}

2 usages new *
@Override
public AppRole addNewRole(String role)
{
    AppRole appRole = appRoleRepository.findById(role).orElse( other: null);
    if(appRole!=null) throw new RuntimeException("Ce rôle existe déjà");
    appRole = AppRole.builder()
        .role(role)
        .build();
    return appRoleRepository.save(appRole);
}

3 usages new *
@Override
public void addRoleToUser(String username, String role)
{
    AppUser appUser = appUserRepository.findByUsername(username);
    AppRole appRole = appRoleRepository.findById(role).get();
    appUser.getRoles().add(appRole);
    //appUserRepository.save(appUser);
}

no usages new *
@Override
public void removeRoleFromUser(String username, String role)
{
    AppUser appUser = appUserRepository.findByUsername(username);
    AppRole appRole = appRoleRepository.findById(role).get();
    appUser.getRoles().remove(appRole);
}
```

HospitalApplication

Dans la classe HospitalApplication on a défini un bean CommandLineRunner qui initialise des rôles et des utilisateurs lors du démarrage de l'application. Ce bean utilise le service AccountService pour ajouter de nouveaux rôles (USER et ADMIN) et créer trois nouveaux utilisateurs (user11, user22, et admin0) avec des identifiants et des mots de passe spécifiés. Les rôles sont ensuite attribués aux utilisateurs appropriés : user11 et user22 se voient attribuer le rôle USER, tandis que admin0 se voit attribuer le rôle ADMIN. Cette initialisation permet de s'assurer que des utilisateurs de base et leurs rôles associés sont disponibles dès le démarrage de l'application, facilitant ainsi la gestion et les tests initiaux des utilisateurs et des rôles dans le système.

```
new *
@Bean
CommandLineRunner commandLineRunnerUserDetails(AccountService accountService)
{
    return args->{
        accountService.addNewRole("USER");
        accountService.addNewRole("ADMIN");

        accountService.addNewUser( username: "user11", password: "12345", email: "user11@gmail.com", confirmPassword: "12345"
        accountService.addNewUser( username: "user22", password: "12345", email: "user22@gmail.com", confirmPassword: "12345"
        accountService.addNewUser( username: "admin0", password: "12345", email: "admin0@gmail.com", confirmPassword: "12345"

        accountService.addRoleToUser( username: "user11", role: "USER");
        accountService.addRoleToUser( username: "user22", role: "USER");
        accountService.addRoleToUser( username: "admin0", role: "ADMIN");
    };
}

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

ID	Nom	Date	Malade	Score
10	Houria	2024-04-19	true	50
12	Fouad	2024-04-19	false	25
13	Hajar	2024-04-19	false	20

ID	Nom	Date	Malade	Score		
10	Houria	2024-04-19	true	50		
12	Fouad	2024-04-19	false	25		
13	Hajar	2024-04-19	false	20		

UserDetailServiceImpl :

Le code présenté implémente un service appelé UserDetailServiceImpl qui agit en tant que service de gestion des détails des utilisateurs dans le contexte de la sécurité d'une application hospitalière. Ce service, annoté avec @Service pour être géré par Spring, utilise @AllArgsConstructor pour l'injection de dépendances via le constructeur, notamment avec l'AccountService. La méthode loadUserByUsername est implémentée pour récupérer les détails d'un utilisateur à partir de son nom d'utilisateur. Lorsqu'un utilisateur est recherché, le service AccountService est utilisé pour obtenir ses informations à partir de la base de données. Si aucun utilisateur correspondant n'est trouvé, une exception UsernameNotFoundException est levée pour indiquer que l'utilisateur n'existe pas. Ensuite, les rôles de l'utilisateur sont extraits et convertis en un tableau de chaînes. Enfin, un objet UserDetails est créé à partir des informations récupérées de l'utilisateur, telles que le nom d'utilisateur, le mot de passe et les rôles, en utilisant la classe utilitaire User fournie par Spring Security.

```
import org.springframework.stereotype.Service;

new *
@Service
@AllArgsConstructor
public class UserDetailServiceImpl implements UserDetailsService {
    private AccountService accountService;
    no usages new *
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
    {
        AppUser appUser = accountService.loadUserByUsername(username);
        if(appUser==null) throw new UsernameNotFoundException(String.format("User %s not found", username));
        String[] roles = appUser.getRoles().stream().map(u->u.getRole()).toArray(String[]::new);
        UserDetails userDetails= User
            .withUsername (appUser.getUsername())
            .password(appUser.getPassword())
            .roles(roles).build();
        return userDetails;
    }
}
```

Conclusion :

Ce TP a abouti à la conception et à la mise en œuvre d'une application Web JEE dédiée à la gestion des patients, en utilisant des technologies modernes telles que Spring MVC, Thymeleaf et Spring Data JPA. Chaque partie du TP a progressivement développé les fonctionnalités essentielles, telles que l'affichage, la pagination, la recherche et la suppression des patients, tout en intégrant des améliorations supplémentaires pour une meilleure expérience utilisateur. La seconde partie a souligné l'importance des templates pour une interface utilisateur cohérente et a introduit la validation des formulaires pour garantir l'intégrité des données saisies. Enfin, la troisième partie a mis en lumière l'aspect crucial de la sécurité avec Spring Security, assurant ainsi la protection des ressources sensibles et l'accès restreint aux fonctionnalités en fonction des autorisations. Dans l'ensemble, ce TP a offert une expérience complète en matière de développement d'applications web modernes avec Spring, en couvrant à la fois les fonctionnalités fondamentales et les aspects avancés tels que la sécurité et la validation des données, permettant ainsi aux développeurs d'acquérir des compétences pratiques pour créer des applications robustes et sécurisées.