Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

Rapport Java JEE

***Objet** : Spring MVC (Pagination, validation des formulaires, thymeleaf)*

❖ Code source :

https://github.com/Koraiche/EMSI-4IIRG2CC-S8-JEE-/tree/main/WORKSPACE

❖ Pourquoi utiliser Spring MVC :

Le framework **Spring** et en particulier le module **Spring MVC** permettent d'implémenter naturellement un Modèle Vue Contrôleur. L'utilisation des fichiers de configuration XML en font une remarquable implémentation facilement maintenable et évolutive.

❖ Présentation :

1 - Couche DAO avec Spring Data JPA Hibernate :

1.1 – Entity :

```
@Entity
@Table(name="PATIENTS")
@Data @NoArgsConstructor @ToString @AllArgsConstructor
public class Patient {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="NOM",length=25)
    @NotNull @Size(min = 5, max = 15, message = "La taille du nom n'est pas correcte !")
    private String nom;
    @Temporal(TemporalType.DATE)
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private Date dateNaissance;
    @DecimalMin("10")
    private int score;
    private boolean malade;
}
```

La classe patient utilise plusieurs annotations grâce a Lombok comme pour les getters et les setters, cependant ici dans note cas on utilise même des

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

annotations de validations pour les formulaires (javax.validation) comme **@NotNull** dans le champs nom qui signifie que le représentation de notre classe dans la base de données ne vas pas accepter des lignes dont le champ nom est null.

❖D'autres exemples :

@Column : donne des spécifications a la colonne comme la tailles et son nom dans la mémoire.

@Size : spécifie la taille min et max de notre attributs(String), elle nous donne aussi la possibilité afficher un message d'erreurs en cas de non-conformité.

@DecimalMin : spécifie la taille min de notre attribut entier.

@DateTimeFormat(pattern = "yyyy-MM-dd") : spécifie le format de notre date.

1.2– JPA Repository :

```
public interface PatientRepository extends JpaRepository<Patient, Long>{
    public Page<Patient> findByNomContains(String str,Pageable pageable);
    public List<Patient> findByMalade(boolean bool);
    public List<Patient> findByNomContainsAndMalade(String str,boolean bool);
}
```

L'interface PatientRepository représente notre Data Access Layer, chaque prototype va être implémenté par Spring JPA, pour nous fournir l'accès à la base, dans notre projet on aura besoin de trouver des Patients à travers leurs noms pour créer un moteur de recherche.

2 – Spring MVC Server side with thymeleaf :

2.1 – Maven dependencies thymeleaf + validation :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
<dependency>
    <groupId>nz.net.ultraq.thymeleaf</groupId>
    <artifactId>thymeleaf-layout-dialect</artifactId>
</dependency>
```

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

2.2- Pourquoi Thymeleaf ?

Dans les applications Web, **Thymeleaf** vise à être un substitut complet aux JavaServer Pages (JSP) et implémente le concept de modèles naturels: des fichiers modèles qui peuvent être directement ouverts dans les navigateurs et qui s'affichent toujours correctement en tant que pages Web.

2.3- Explication :

En d'autres mot thymeleaf va faire correspondre des routes URL a des Template html via des actions (des fonction mappées).

2.4- Contrôleur :

```java
@Controller
public class PatientsController {
    @Autowired // Injection des dependances
    private PatientRepository patientRepository;

    @GetMapping(path = "/index")
    public String index(){
        return "index";
    }

    @GetMapping(path = "/patients")
    public String list(Model model,
            @RequestParam(name="page", defaultValue="0") int page,
            @RequestParam(name="size", defaultValue="5") int size,
            @RequestParam(name="keyword", defaultValue="") String keyword){
        Page<Patient> pagePatients = patientRepository.findByNomContains(keyword,PageRequest.of(page, size));
        model.addAttribute("liste", pagePatients.getContent());
        model.addAttribute("pages", new int[pagePatients.getTotalPages()]);
        model.addAttribute("currentPage", page);
        model.addAttribute("size", size);
        model.addAttribute("keyword", keyword);
        return "patients";
    }

    @GetMapping(path = "/deletePatient")
    public String deletePatient(Long id, int page, String keyword, int size){
        patientRepository.deleteById(id);
        return "redirect:/patients?page="+page+"&keyword="+keyword+"&size="+size;
    }

    @GetMapping(path = "/deletePatient2")
    public String deletePatient2(Long id, int page, String keyword, int size, Model model){
        patientRepository.deleteById(id);
        return list(model,page, size, keyword);
    }
```

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

```java
@GetMapping(path = "/formPatient")
public String formPatient(Model model) {
    model.addAttribute("patient", new Patient());
    model.addAttribute("mode", "new");
    return "formPatient";
}

@GetMapping(path = "/editPatient")
public String editPatient(Model model, Long id) {
    Patient patient = patientRepository.findById(id).get();
    model.addAttribute("patient", patient);
    model.addAttribute("mode", "edit");
    return "formPatient";
}

@PostMapping(path = "/savePatient")
public String savePatient(Model model, @Valid Patient patient, BindingResult bindingResult){
    if(bindingResult.hasErrors()) return "formPatient";
    patientRepository.save(patient);
    model.addAttribute("patient", patient);
    return "confirmation";
}
```

@Controller : Nous utilisons @Controller en combinaison avec l'annotation @RequestMapping pour les méthodes de gestion de demandes, chaque demande va être dirigé vers un path qui est associé d'une manière unique a une action (fonction) qui fait les traitements nécessaires puis retourne/redirige vers/forward to une vue.

@GetMapping/@PostMapping : Chaque fonction représente une action, une action est sensé gérer une demande/requête, c'est pour cela qu'on utilise @GetMapping pour les requêtes HTTP GET et @PostMapping pour les requêtes HTTP POST.

@RestController : Sert généralement a implémenté des API, pour fournir des retours JSON directe sans passer par thymeleaf et des templates.

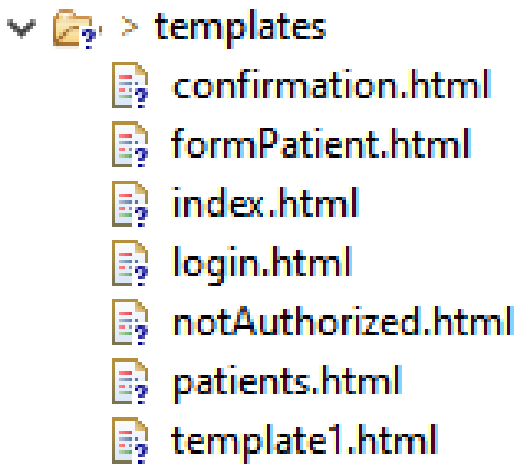
@RequestParam : Sert justement à récupérer des valeurs depuis l'URL grâce à leurs noms.

Model : C'est un objet qui sert à faire passer des valeurs depuis notre contrôleur vers nos templates via sa méthode addAttribute().

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

2.5- Templates :

C'est dossier dans ressources qui va contenir toutes nos vues JSP



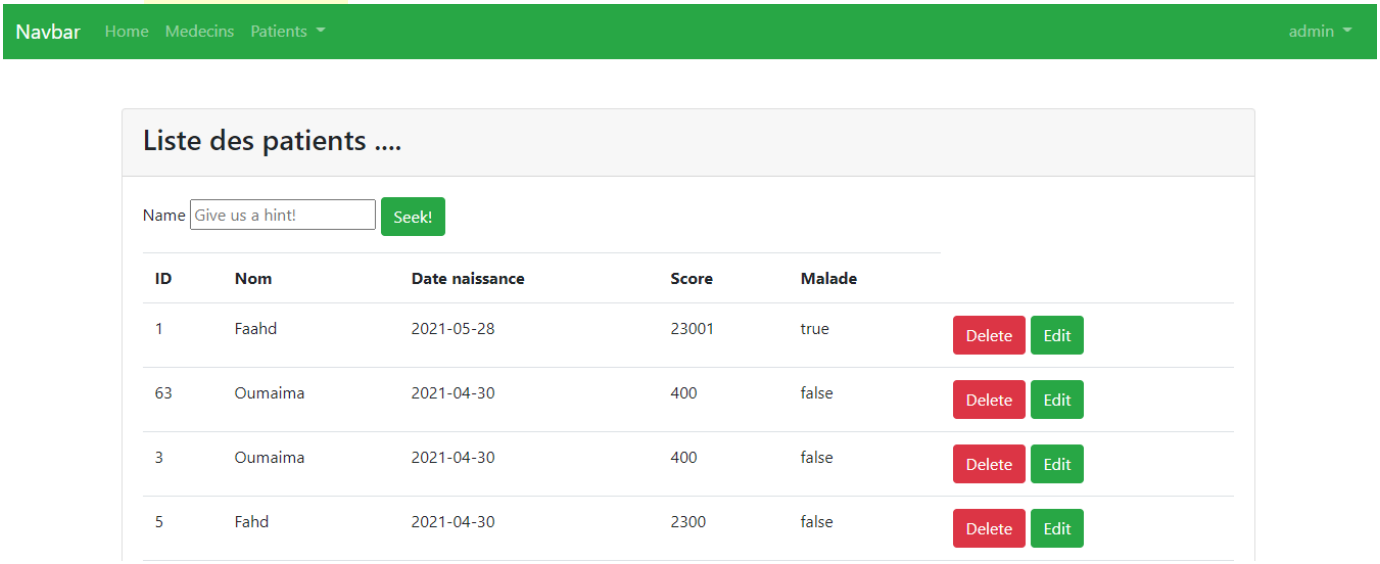
Notre vue template1.html va jouer le rôle de page Maitre qui va contenir toutes les interfaces communes entre les pages. Cela à travers l'inclusion du package thymeleaf dans note fichier html.

```
<html
      xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity5"
      >
```

La vue liste patients est en relation avec l'action :

```
@GetMapping(path = "/patients")
      public String list(Model model,
                  @RequestParam(name="page", defaultValue="0") int page,
                  @RequestParam(name="size", defaultValue="5") int size,
                  @RequestParam(name="keyword", defaultValue="") String keyword){
            Page<Patient> pagePatients =
patientRepository.findByNomContains(keyword,PageRequest.of(page, size));
            model.addAttribute("liste", pagePatients.getContent());
            model.addAttribute("pages", new int[pagePatients.getTotalPages()]);
            model.addAttribute("currentPage", page);
            model.addAttribute("size", size);
            model.addAttribute("keyword", keyword);
            return "patients";
              }
```

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE



Le bouton Delete est en relation avec l'action :

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

	@GetMapping(path = "/deletePatient2")
	public String deletePatient2(Long id, int page, String keyword, int size, Model model){
		patientRepository.deleteById(id);
		return list(model,page, size, keyword);
		  }
```

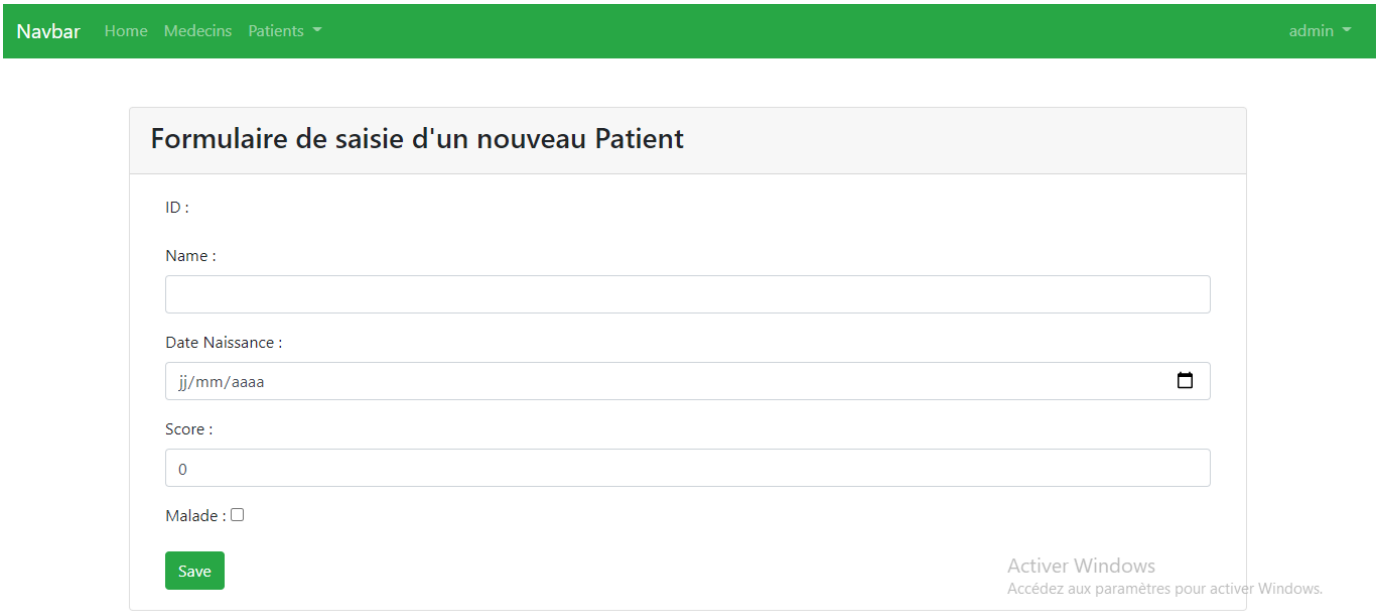
Le bouton Edit ainsi que la vue ajouter et le formulaire sont tous les trois en relation avec les actions :

```
@GetMapping(path = "/formPatient")
	public String formPatient(Model model) {
		model.addAttribute("patient", new Patient());
		model.addAttribute("mode", "new");
		return "formPatient";
	}

	@GetMapping(path = "/editPatient")
	public String editPatient(Model model, Long id) {
		Patient patient = patientRepository.findById(id).get();
		model.addAttribute("patient", patient);
		model.addAttribute("mode", "edit");
		return "formPatient";
	}

	@PostMapping(path = "/savePatient")
```

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2

Site : CENTRE

```
    public String savePatient(Model model, @Valid Patient patient, BindingResult
bindingResult){
            if(bindingResult.hasErrors()) return "formPatient";
            patientRepository.save(patient);
            model.addAttribute("patient", patient);
            return "confirmation";
              }
```



2.6- Bootstrap:

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>4.1.3</version>
</dependency>
```

```
<link rel="stylesheet" type="text/css" href="/webjars/bootstrap/4.1.3/css/bootstrap.min.css"/>
<script src="/webjars/jquery/3.4.1/jquery.min.js" ></script>
<script src="/webjars/bootstrap/4.1.3/js/bootstrap.min.js" ></script>
```

2.6- Pagination :

Pour des raisons d'optimisation on évite de récupérer toutes les lignes d'une table a la fois, pour cela on récupère page par page pour alléger la mémoire et éviter le blocage du navigateur en cas où il y'a trop de lignes.

Nom & prénom : Fahd KORAICHE
Classe : 4IIR G2
Site : CENTRE

```
public Page<Patient> findByNomContains(String str,Pageable pageable);
```

Pour cela on aura besoin d'un objet générique Page ainsi qu'une interface Pageable dans notre prototype dans le repository.

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

</ul>
```

Dans notre Template on va ajouter une liste dans laquelle on va afficher les pages sous forme de boutons avec des actions de pagination gérées dans le contrôleur selon l'indice et la taille de la page demandé en plus du mot clé recherché.

```
@GetMapping(path = "/patients")
       public String list(Model model,
                     @RequestParam(name="page", defaultValue="0") int page,
                     @RequestParam(name="size", defaultValue="5") int size,
                     @RequestParam(name="keyword", defaultValue="") String keyword){
              Page<Patient> pagePatients =
patientRepository.findByNomContains(keyword,PageRequest.of(page, size));
              model.addAttribute("liste", pagePatients.getContent());
              model.addAttribute("pages", new int[pagePatients.getTotalPages()]);
              model.addAttribute("currentPage", page);
              model.addAttribute("size", size);
              model.addAttribute("keyword", keyword);
              return "patients";
                }
```