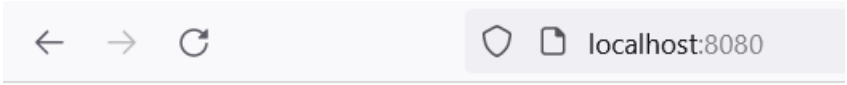
TP n°5 en Systèmes distribués (Thymeleaf)

Application CRUD Spring Boot avec Thymeleaf

Thymeleaf est un moteur de template Java côté serveur pour les environnements web, capable de traiter HTML, XML, JavaScript, CSS et même du texte brut.
Il est plus puissant que JSP et responsable du rendu dynamique du contenu sur l'interface utilisateur.
Le moteur permet un travail parallèle des développeurs backend et frontend sur une même vue.
Il est principalement utilisé avec Spring MVC dans la creation des applications web

Thymeleaf peut être utilisé pour remplacer **JSP** sur la couche **View** (View Layer) de l'application **Web MVC**. **Thymeleaf** est un logiciel à code source ouvert (open source) sous licence **Apache 2.0**.



Etudiant List

Ajout nouvel Etudiant

Nom	Prenom	Reffil	Action
Aain	mark	F1	Edit Delete
Smith	Bob	F1	Edit Delete

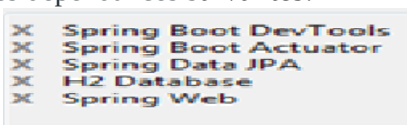
Exercice : Effectuer des opérations crud sur le jeu de données Employe et Departement du projet microservice de tp4.

1. departement-service

1.1 Ajouter au projet la dependance Thymeleaf

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

et les dependances suivantes:



- Spring Web (Créez des applications Web, y compris RESTful, à l'aide de **Spring** MVC. Utilise Apache Tomcat comme conteneur incorporé par défaut.)
- **Spring Data JPA** (Persister les données dans les magasins SQL avec l'API Java Persistence à l'aide de Spring Data et Hibernate.)
- **Spring Boot Devtools** (fournit des redémarrages rapides des applications, LiveReload et des configurations pour une expérience de développement améliorée)
- Pilote H2
- **Thymeleaf** (moteur de template Java côté serveur pour les environnements web et autonomes. Permet d'afficher correctement le HTML dans les navigateurs et sous forme de prototypes statiques.)

1.2 Redefinir les classes du projet :

application.proprietes : ajouter les parametres permettant d'acces a la base H2

Classe Departement.java

```
@Entity
public class Departement {
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	private long id;
	private String nom;

	// constructeur + getteurs + setters
}
```

DepartementRepository.java

```
@Repository
public interface DepartementRepository extends JpaRepository<Departement, Long>{
	List<Departement> findByNom(String nom);
}
```

DepartementNotFoundException.java

```
public class DepartementNotFoundException extends RuntimeException {
	private static final long serialVersionUID = 1L;
	public DepartementNotFoundException(String exception) {
		super(exception);
	}
}
```

DepartementController .java

```
@Controller
public class DepartementController {

	@Autowired private DepartementRepository deptRepository;

	@GetMapping("/")
	public String HomePage(Model model) {
    model.addAttribute("alldeptlist", deptRepository.findAll());
    return "index";  // page index.html
  }

  @GetMapping("/addnew")
  public String addNewDepartement(Model model) {
    Departement departement = new Departement();
    model.addAttribute("departement", departement);
    return "add-departement";  //page add-departement.html
  }

  @PostMapping("/save")
  public String saveDepartement(@ModelAttribute("departement") Departement departement) {
		deptRepository.save(departement);
    return "redirect:/";
  }

  @GetMapping("/UpdateDepartement/{id}")
  public String updateForm(@PathVariable("id") long id, Model model) {
		Optional<Departement> departement = deptRepository.findById(id);
    model.addAttribute("departement", departement);
    return "update-departement";
  }
```

```
    @GetMapping("/deleteDepartement/{id}")
    public String deleteById(@PathVariable(value = "id") long id) {
            deptRepository.deleteById(id);
        return "redirect:/";
    }
}
```

Classe DepartementController
Il s'agit de la classe de contrôleur, elle contrôle essentiellement le flux de données. Il contrôle le flux de données dans l'objet du modèle et met à jour la vue chaque fois que les données changent.

Lorsque l'utilisateur tape l'URL **localhost :8080/** sur le navigateur, la requête va à la méthode **HomePage()** et dans cette méthode, nous récupérons la liste des departements et l'ajoutons dans la fenêtre modale avec clé, paire valeur et retournons la page **index.html**. Dans la page index.html, la clé **(alldeptlist)** est identifiée comme un objet java et Thymeleaf parcourt la liste et génère du contenu dynamique selon le modèle fourni par l'utilisateur.

- **/addNew –** lorsque l'utilisateur clique sur le bouton **Ajouter un departement**, la requête est envoyée à **la méthode addNewDepartement().** Et dans cette méthode, nous créons simplement l'objet vide de departement et le renvoyons à **add-departement.html** afin que l'utilisateur puisse remplir les données dans cet objet vide et lorsque l'utilisateur appuie sur le bouton **enregistrer que /save** mapping s'exécute et obtienne l'objet de departement et enregistre cet objet dans la base de données.
- **/UpdateDepartement/{id} –** Ce mappage permet de mettre à jour les données existantes sur les departements.
- **/deleteDepartement/{id} –** Ce mappage permet de supprimer les données existantes des departements.

Thymeleaf Template est un fichier modèle. Son contenu est au format **XML/XHTML/HTML5**. Nous allons créer 3 fichiers et les placer dans le répertoire **src/main/resources/templates** :

Index.html

```
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>List Departement</title>
</head>
<body>
        <h3>Departement List</h3>
        <a th:href="@{/addnew}" >Ajout nouvel Departement</a>
          <table border="1" >
             <thead>
              <tr>
                <th>Nom</th>
                <th>Action</th>
              </tr>
            </thead>
            <tbody>
            <tr th:each="departement:${alldeptlist}">
                  <td th:text="${departement.nom}"></td>
                  <td> <a th:href="@{/UpdateDepartement/{id}(id=${departement.id})}">Edit</a>
                  <a th:href="@{/deleteDepartement/{id}(id=${departement.id})}">Delete</a>
                 </tr>
        </tbody>
      </table>
```

```
</div>
</body>
```

Tous les fichiers **HTML** doivent déclarer l'utilisation de **Thymeleaf Namespace**:

```
<!-- Thymeleaf Namespace -->
<html xmlns:th="http://www.thymeleaf.org">
```

Dans les fichiers modèles (Template file), il y a des **Thymeleaf Marker** (des marqueurs de Thymeleaf) qui sont des instructions aidant les données de processus de **Thymeleaf Engine**. **Thymeleaf Engine** analyse les fichiers modèles (Template file) et les combine avec les données Java pour générer (generate) un nouveau document.

Thymeleaf Marker

```
<tr th:each="departement:${alldeptlist}">
<td th:text="${departement.nom}"></td>
```

add-departement.html

```
<body>
  <div>
    <h1>Gestion Departement</h1>
    <hr>
    <h2>Save Departement</h2>
    <form action="#" th:action="@{/save}" th:object="${departement}" method="POST">
      <input type="text" th:field="*{nom}" placeholder="Nom Departement">
      <button type="submit">Enregistrer Departement</button>
    </form>
    <hr>
    <a th:href="@{/}"> List Departement</a>
  </div>
</body>
```

update-departement.html

```
<body>
  <div>
    <h1>Gestion Departementt</h1>
    <hr>
    <h2>Edit Departement</h2>
    <form action="#" th:action="@{/save}" th:object="${departement}"  method="POST">
      <input type="hidden" th:field="*{id}" />
      <input type="text" th:field="*{Nom}">
      <button type="submit"> Edit Departement</button>
    </form>
    <hr>
    <a th:href = "@{/}"> List Departement</a>
  </div>
</body>
```

2. emplote-service

Ajouter les dependances et redefinir les classes du projet
Ajouter les vues : index.html, add-employe, et update-employe

3. Gateway-service

Reconfigurer le projet, tester l'ensemble des templates.