FMS ACADEMY BY FMS

FACYLITIES MULTI SERVICES

Une entreprise ordinaire à l'agilité extraordinaire

FORMATION JAVA SPRING ANGULAR

SPRING MVC

mohamed.el-babili@fms-ea.com

33+ 628 111 476

Version : 4.0

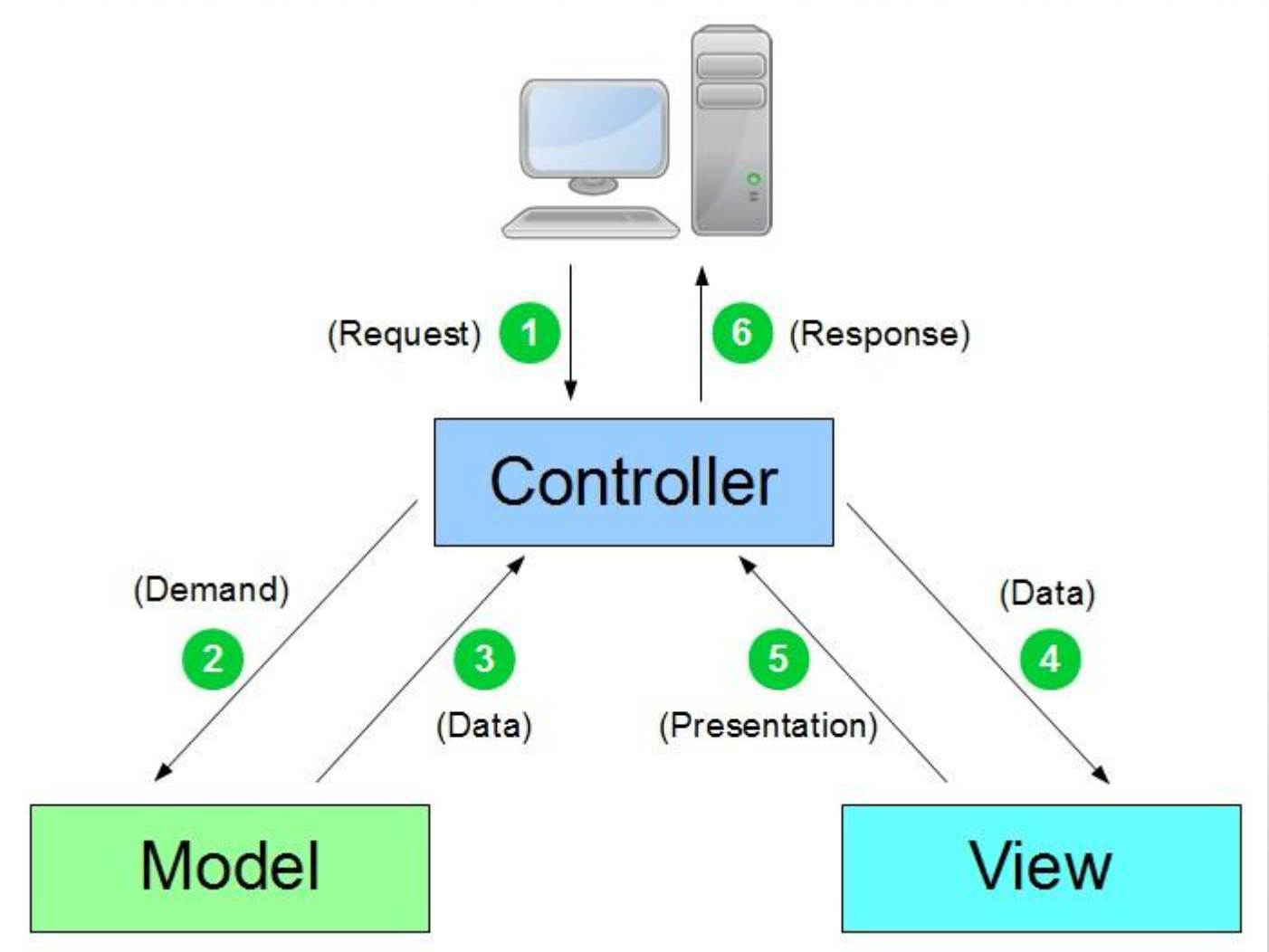
DMAJ : 16/12/24

Module : DEV-SPRI-001
Spring les bases

01 Comprendre l'injection des dépendances et l'inversion de contrôle
02 Exploiter les frameworks Spring et Hibernate/Jpa
03 Réaliser une application Spring Boot utilisant Hibernate et Spring data
04 Réaliser une application web avec Spring Mvc
05 Sécuriser une application web avec Spring Security

SOMMAIRE

PATTERN MVC

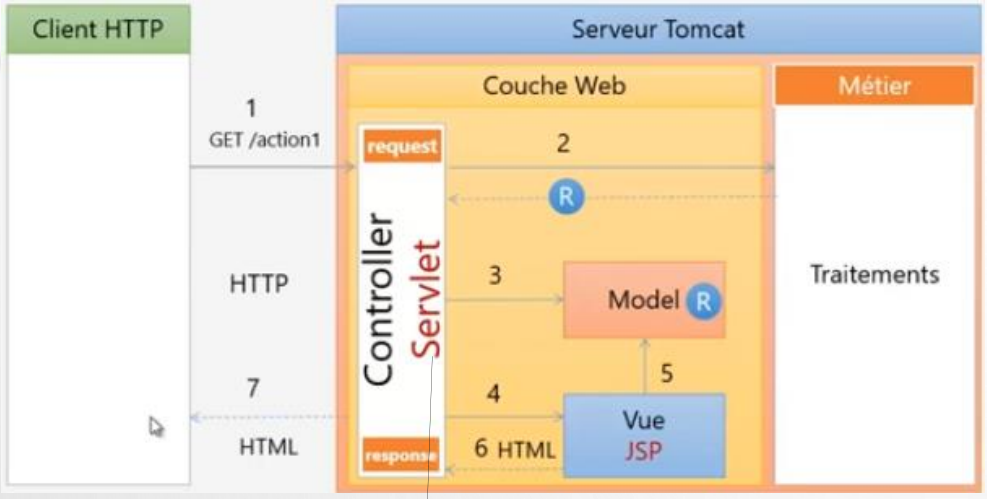


Il existe 2 modèles de rendu de code Html :

- **Côté serveur**
- Coté Client

ARCHITECTURE JAVA EE / SPRING MVC

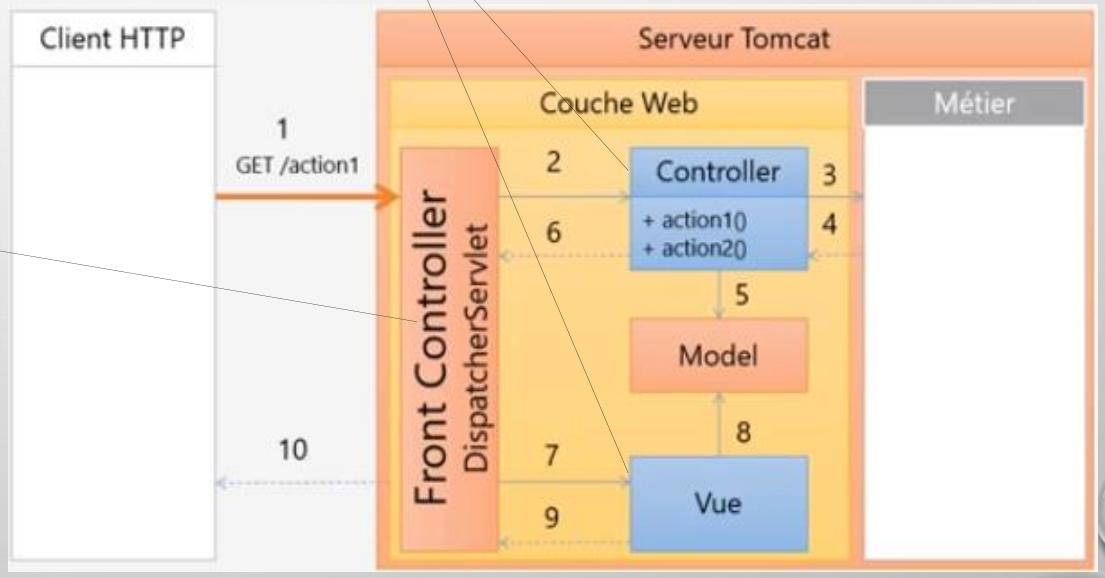
Architecture Java EE



Le développeur doit ici coder uniquement le(s) contrôleur(s) et les vues

Plus besoin de dvp de Servlet, Spring propose par défaut le DispatcherServlet qui fait office de contrôleur, après analyse de la requête, il va orienter vers un contrôleur(classe Java) contenant la bonne méthode (get ou post). Le contrôleur cible peut interroger la couche métier/dao et stocker les résultats dans le model(Map) fourni par Spring avant d'indiquer au DS la vue qu'il faut renvoyer au client Http. La aussi, on n'utilisera pas Jsp mais un moteur de Template : ThymeLeaf

Architecture Spring Mvc



SPÉCIFICATIONS DE L'APPLI À RÉALISER

Soit une appli côté serveur de gestion d'articles dont les **spécifications fonctionnelles** sont :

→ chaque article est défini par son id, sa description, sa marque et son prix

→ L'appli permet d'ajouter en base un article

→ Consulter, mettre à jour, supprimer un article

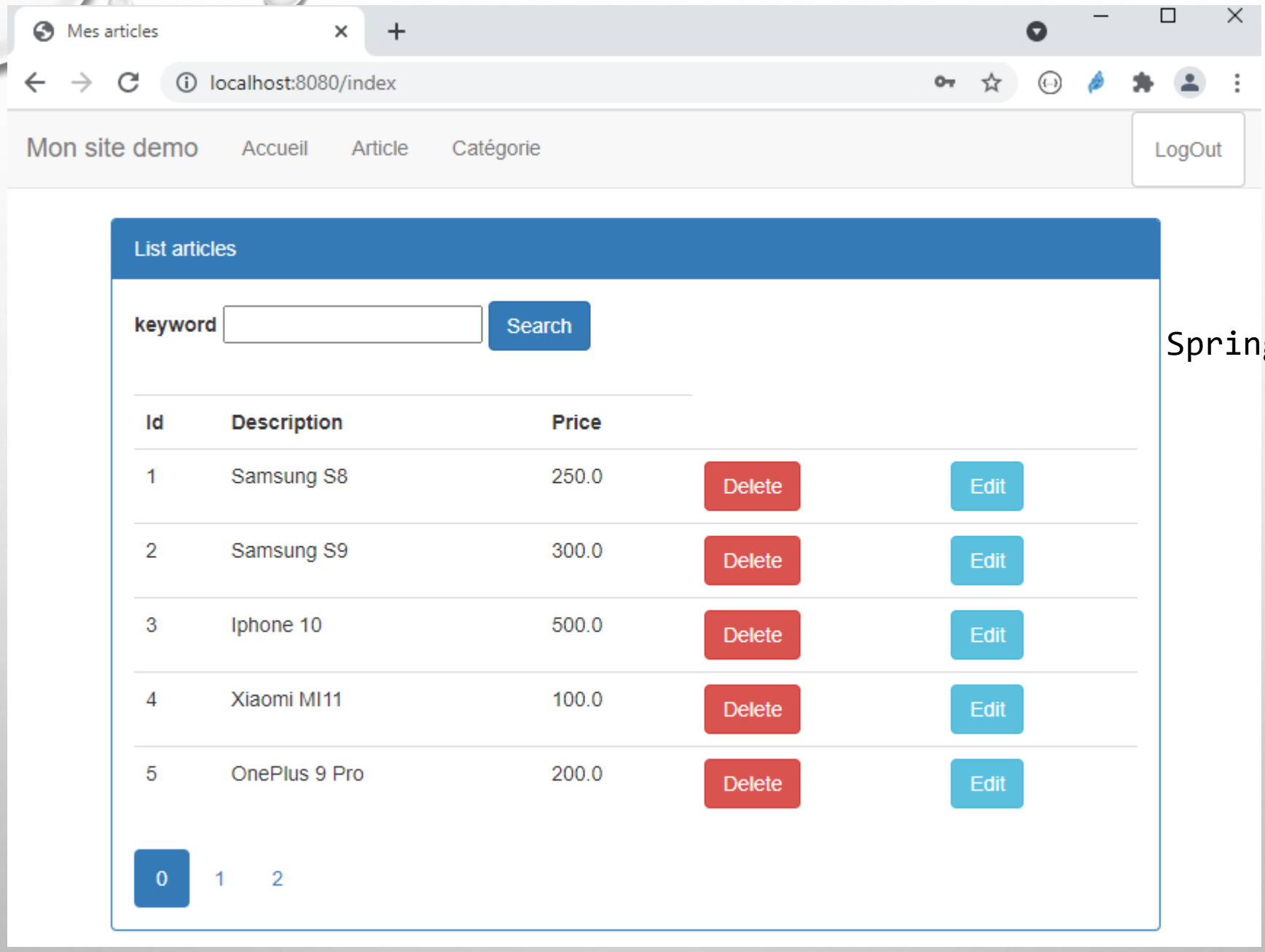
→ Afficher tous les articles à l'aide d'un système de pagination

→ Rechercher des articles à partir d'un mot clé et les afficher

Les spécifications techniques sont :
→ Spring Boot 2.6.0 / Spring Data Jpa Hibernate / Spring Mvc / Thymeleaf / Lombok
→ Java 11 / Eclipse ou IntelliJ / SGBD MariaDB / BootStrap
NB : Quand un numéro de version est manquant, veiller à trouver une version compatible avec les numéros de versions indiquées

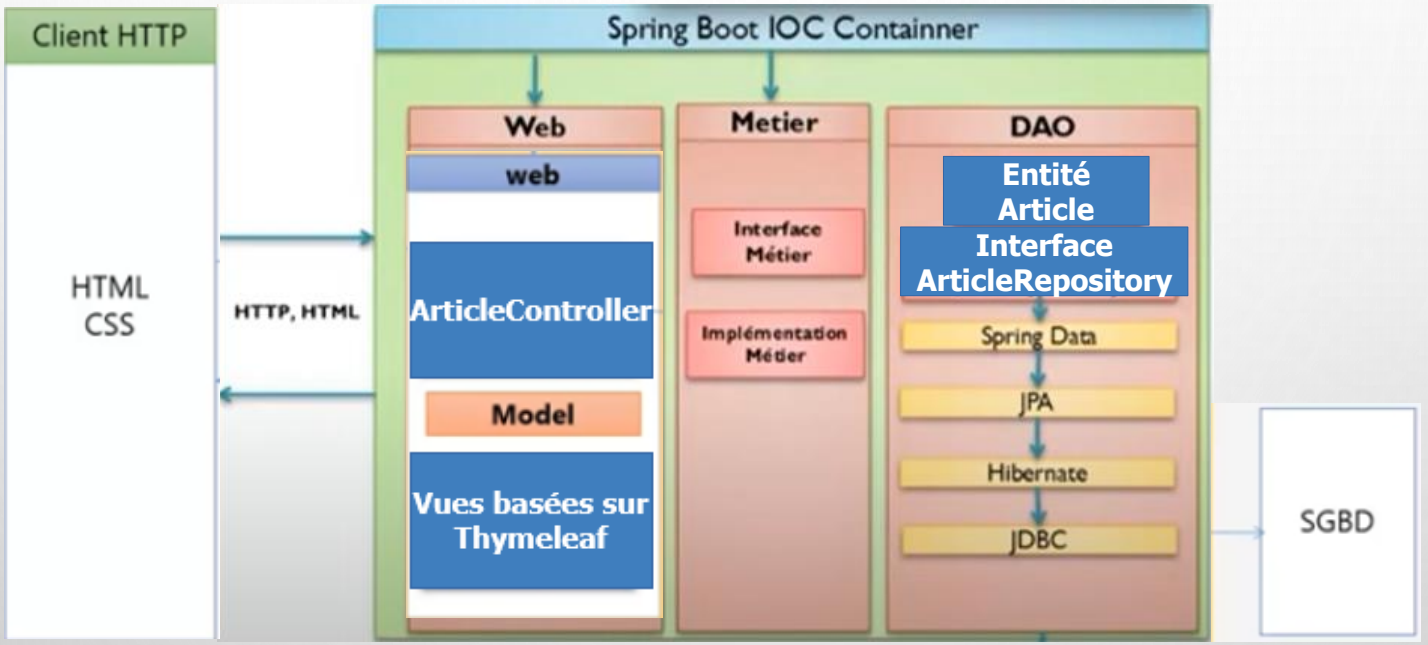
Il existe 2 rôles dans l'application (Spring Security)
→ ADMIN a accès à toutes les pages de l'application
→ USER peut consulter la liste des articles

MAQUETTE/APPLI



SpringShopWeb

ARCHITECTURE DE L'APPLI
(SERVER SIDE)



RÉALISATION DE LA 1ÈRE PHASE DE L'APPLI

1 → New spring starter project : jpa + mariadb + web + thymeleaf + lombok

2 → Compléter le fichier application.Properties (config de l'unité de persistance)

3 → Intégrer votre package entities avec les classes souhaitées puis ajouter les annotations pour obtenir les entités jpa

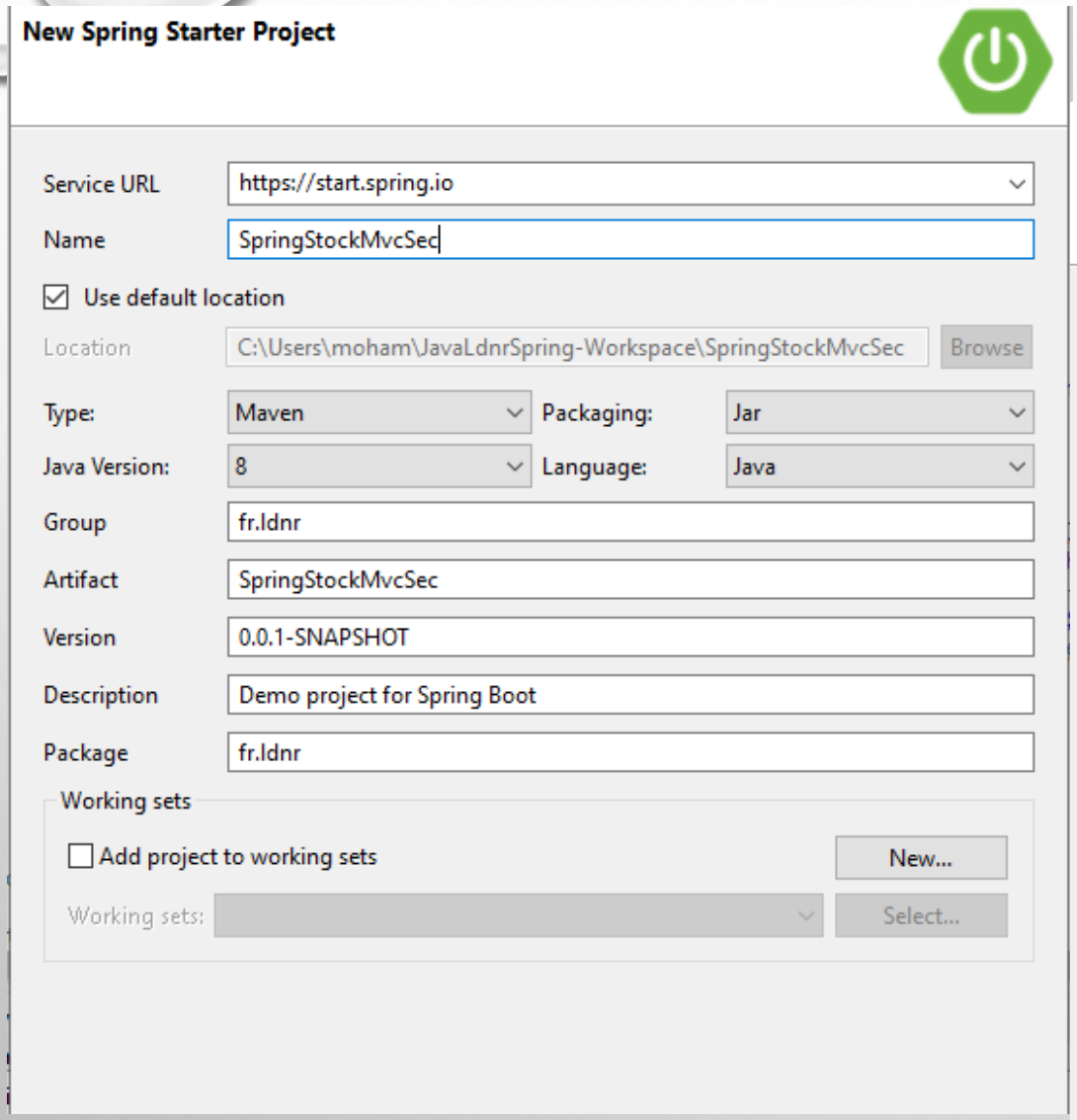
4 → Exécuter l'appli puis vérifier si les tables sont générées (Heidisql, PhpMyAdmin ou en mode console)

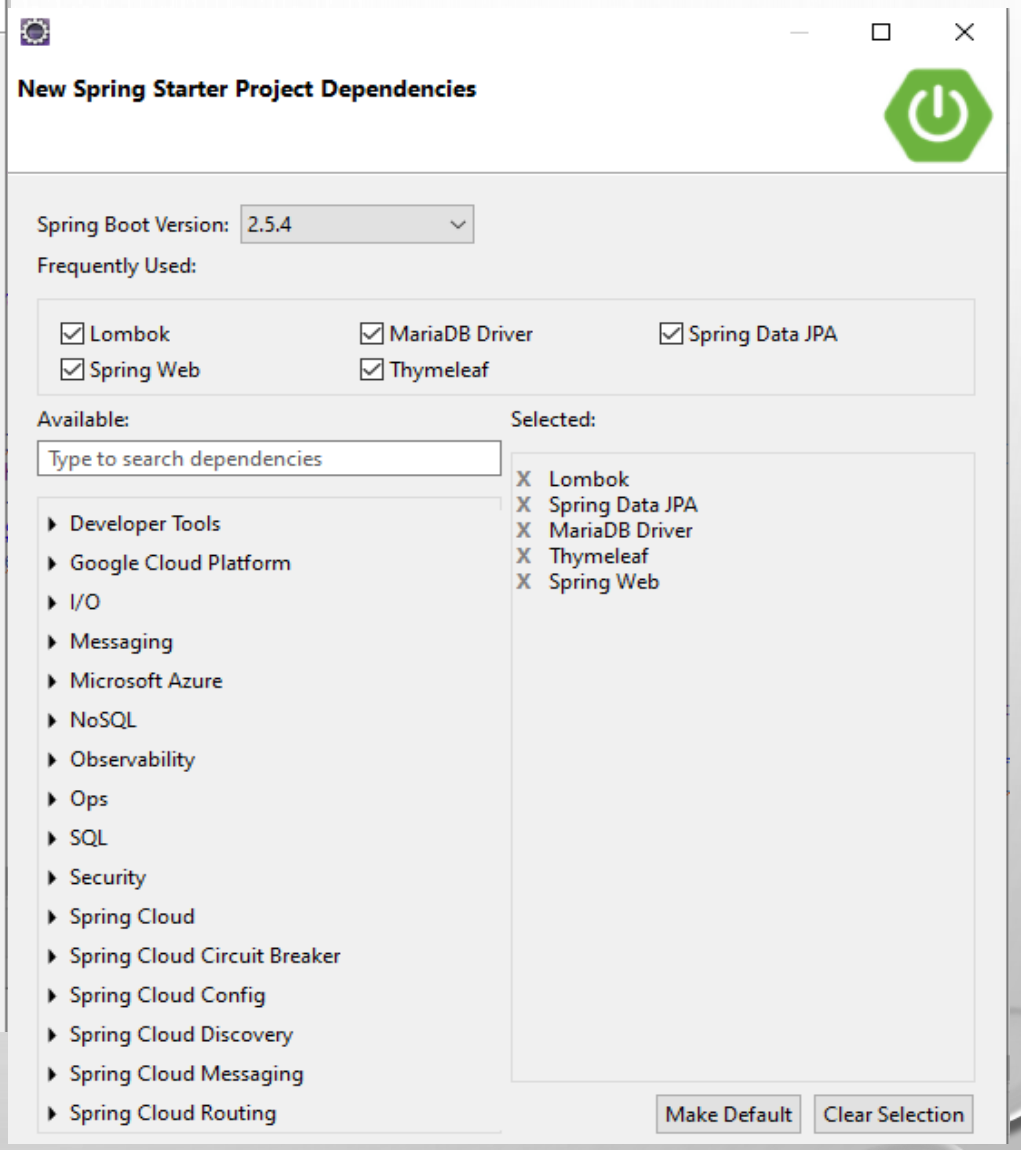
5 → Dans un package fr.fms.dao ajouter vos interfaces JPA qui héritent de jparepository

6 → Injecter les dépendances dans le programme principal puis réaliser un jeu d'essai en insérant des données

7 → Retour sur spring data plus tard cette fois-ci, on va en effet réaliser d'abord la couche web

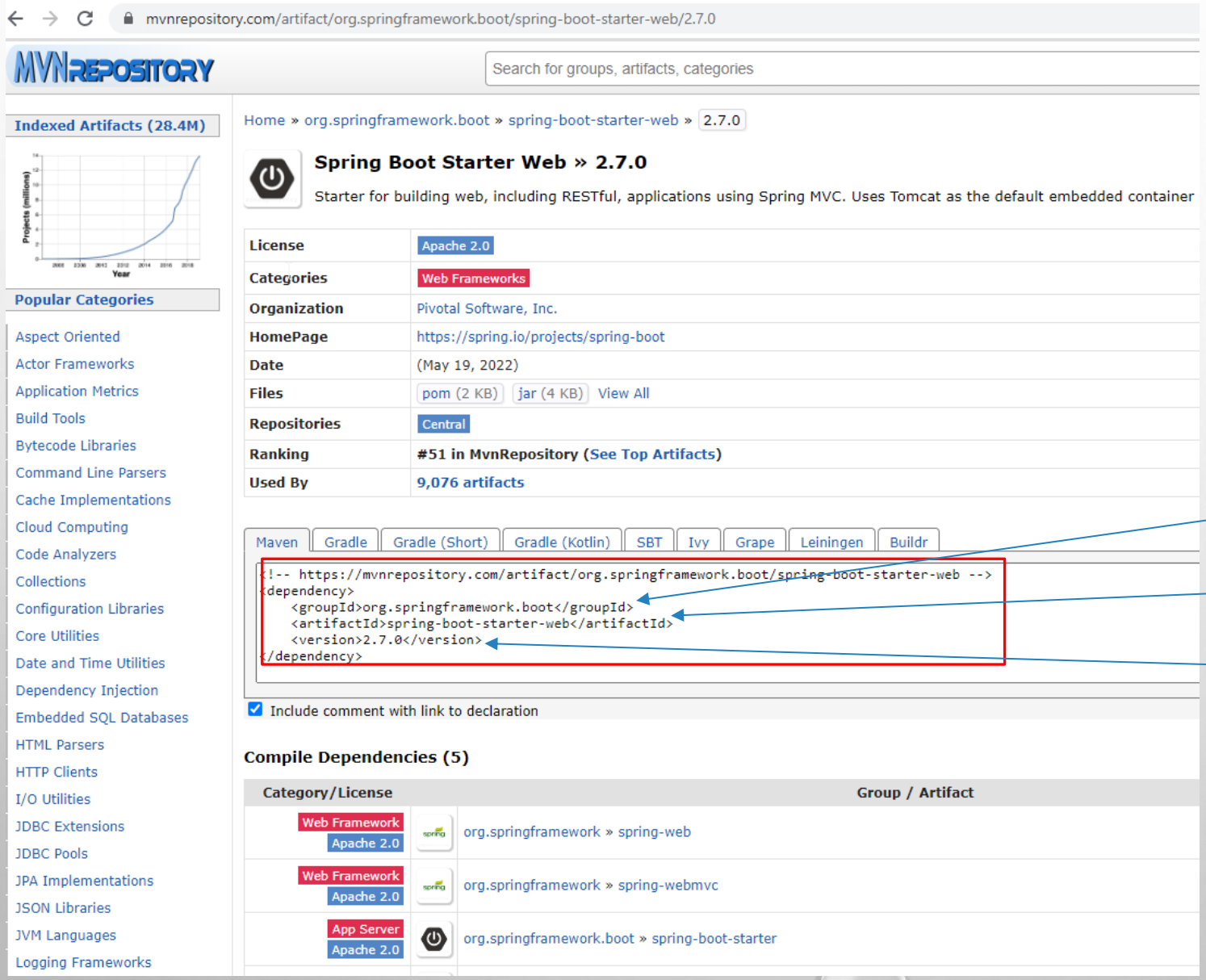
SOLUTION FROM SCRATCH (1 à 6)





SOLUTION AVEC PROJET EXISTANT EN UTILISANT MAVEN

Cherchez et ajoutez vous-même les dépendances dans le projet en cours



Ici **Artifact** fait référence aux modules Spring disponibles dans Maven Central Repository. Vous pouvez ajouter un artifact à votre projet en copiant ses coordonnées dans votre fichier **pom.xml**.

Group ID : L'organisation ou le projet auquel appartient l'artifact
Artifact ID : Le nom du module ou de la bibliothèque
Version : La version spécifique de l'artifact que vous souhaitez utiliser

Lombok

```java
@Entity
@Data @NoArgsConstructor @AllArgsConstructor @ToString
public class Article implements Serializable {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    private String description;
    private double price;
}
```

NB : S'agissant de notre javaBean ici, nous avons utilisé Lombok pour générer automatiquement les constructeurs, accesseurs…

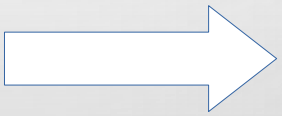
Pour l'utiliser avec Eclipse, vous devez l'ajouter via Help/install new soft/
Saisir https://projectlombok.org/p2
Choisir Lombok...

Si tout va bien, vous devriez obtenir ceci en base :

```
MariaDB [stock]> describe article;
+-------------+-------------+------+-----+---------+----------------+
| Field       | Type        | Null | Key | Default | Extra          |
+-------------+-------------+------+-----+---------+----------------+
| id          | bigint(20)  | NO   | PRI | NULL    | auto_increment |
| description | varchar(255)| YES  |     | NULL    |                |
| price       | double      | NO   |     | NULL    |                |
+-------------+-------------+------+-----+---------+----------------+
3 rows in set (0.078 sec)
```

```java
@SpringBootApplication
public class SpringStockMvcApplication implements CommandLineRunner {
    @Autowired
    ArticleRepository articleRepository;

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

    @Override
    public void run(String... args) throws Exception {
        articleRepository.save(new Article(null,"Samsung S8",250));
        articleRepository.save(new Article(null,"Samsung S9",300));
        articleRepository.save(new Article(null,"Iphone 10",500));

        articleRepository.findAll().forEach(a -> System.out.println(a));
    }
}
```

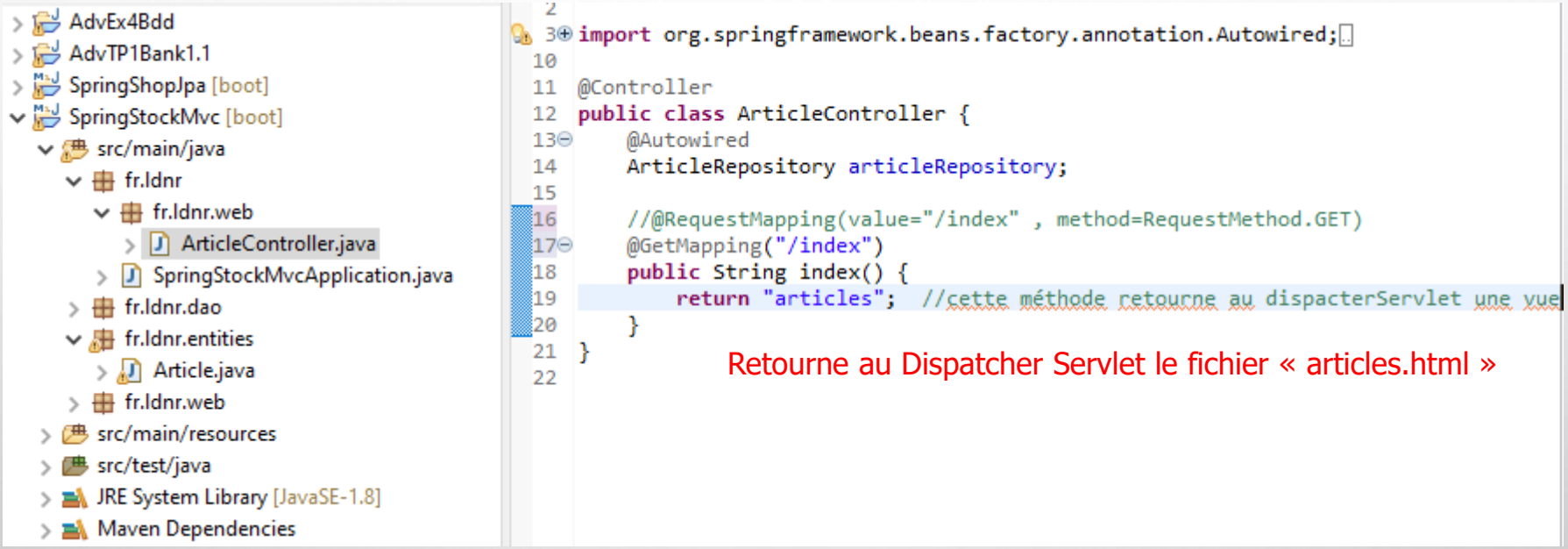
Après qq insertions, nous devrions obtenir ceci :



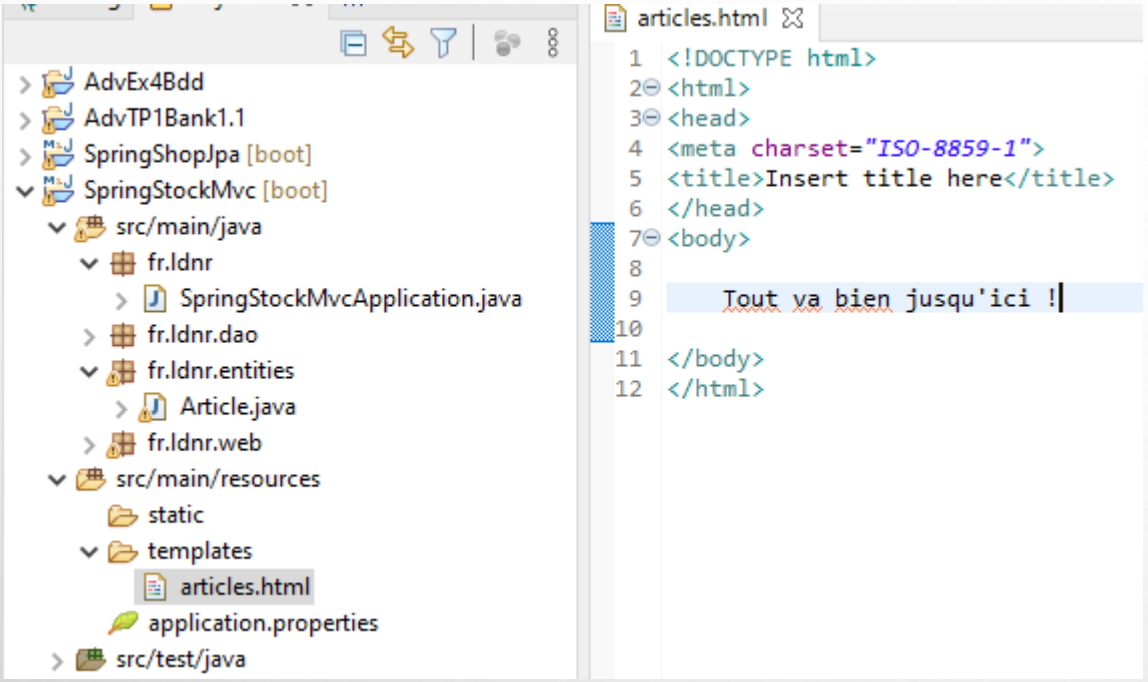
```
+----+-------------+-------+
| id | description | price |
+----+-------------+-------+
|  1 | Samsung S8  |   250 |
|  2 | Samsung S9  |   300 |
|  3 | Iphone 10   |   500 |
+----+-------------+-------+
```

```
Hibernate: insert into article (description, price) values (?, ?)
Hibernate: insert into article (description, price) values (?, ?)
Hibernate: insert into article (description, price) values (?, ?)
Hibernate: select article0_.id as id1_0_, article0_.description as
Article(id=1, description=Samsung S8, price=250.0)
Article(id=2, description=Samsung S9, price=300.0)
Article(id=3, description=Iphone 10, price=500.0)
```

7: AJOUTER UN CONTRÔLEUR DANS LA COUCHE WEB



8: AJOUTER UNE VUE DANS LES RESSOURCES/TEMPLATES



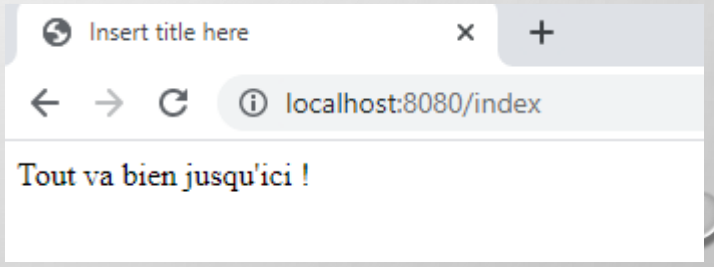
Important pour la suite :

- N'oubliez pas de mettre entre commentaires vos insertions en base pour éviter les doublons (mode update)

- Privilégiez le stop and run afin d'éviter tout conflit sur le port du serveur http

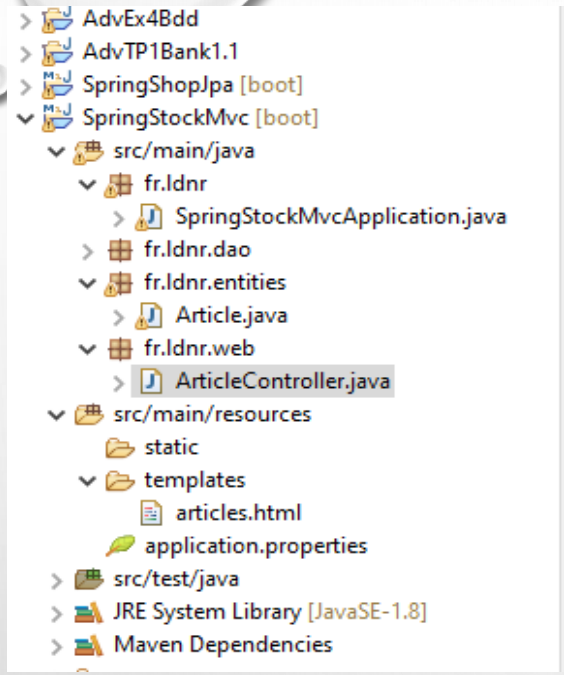
- Vous pouvez changer le numéro de port dans le fichier app.properties, ex : Server.port=8081

Lancer l'appli et une page web
Avec l'URL suivante :



NB : contrairement au dossier templates, le dossier static contient les éléments qui ne nécessitent pas de traitement côté serveur (page html statique, css...)

9 : INSERTION DES DONNÉES DANS LE MODÈLE ET UTILISATION DE THYMELEAF POUR LES AFFICHER DANS LA VUE



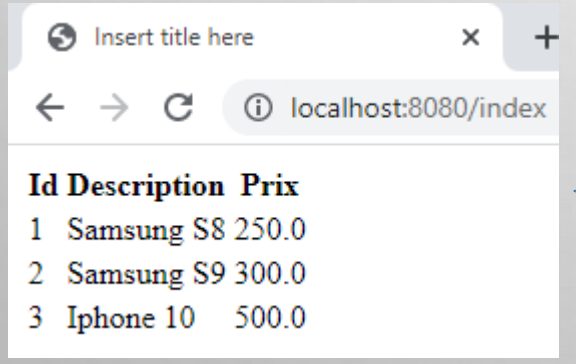

```java
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

import fr.ldnr.dao.ArticleRepository;
import fr.ldnr.entities.Article;

@Controller
public class ArticleController {
    @Autowired
    ArticleRepository articleRepository;

    //@RequestMapping(value="/index" , method=RequestMethod.GET)
    @GetMapping("/index")
    public String index(Model model) {  //le model est fourni par spring, je peux l'utiliser comme ci
        List<Article> articles = articleRepository.findAll();   //récupère tous les articles
        model.addAttribute("listArticle",articles);     //insertion de tous les articles dans le model
                                                        //accessible via l'attribut "listArticle"
        return "articles";  //cette méthode retourne au dispacterServlet la vue articles.html
    }
}
```

Thymeleaf est un moteur de template basé sur le modèle Mvc, il récupère ici les données directement du model pour les insérer dans la vue

```html
<!DOCTYPE html>
<html xmlns:th="http://thymeleaf.org">
<head>Attribut XML permettant d'associer un préfixe th à l'URI spécifié
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

    <table>
        <tr>
            <th>Id</th> <th>Description</th> <th>Prix</th>
        </tr>
        <tr th:each="a:${listArticle}">
            <td th:text="${a.id}"> </td>
            <td th:text="${a.description}"> </td>
            <td th:text="${a.price}"> </td>
        </tr>

    </table>

</body>
</html>
```

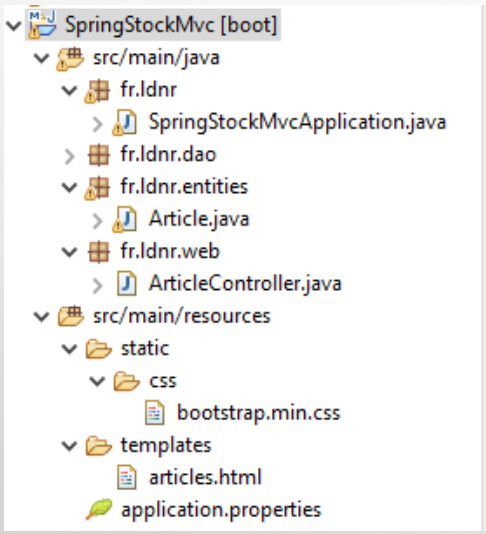
Permettant à Thymeleaf d'être utilisé sur le document Via le préfixe

Exécution



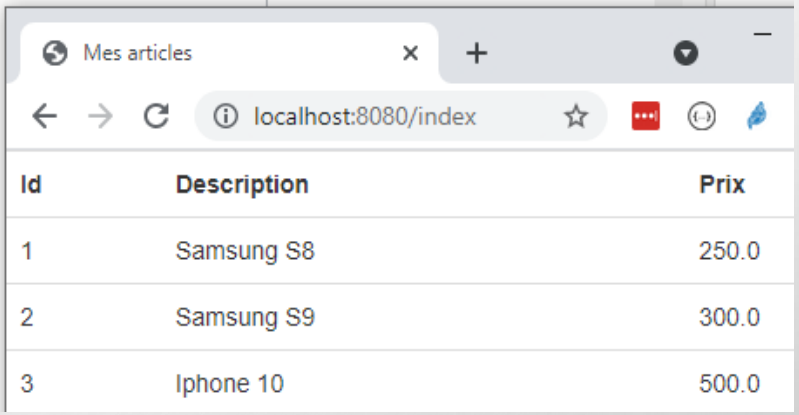
ÉTAPE 10 : UTILISATION DE BOOTSTRAP

Ajouter Bootstrap à votre projet :
- dans le repertoire static, ajouter un dossier css
- y ajouter le fichier Bootstrap.min.css (version 3 ici)
- ajouter le lien à votre page html
- redémarrer l'appli




```
6  <title>Mes articles</title>     <!-- Ide a besoin de ce lien -->
7  <link rel="stylesheet" href="../static/css/bootstrap.min.css"
8                         th:href="@{css/bootstrap.min.css}">
9                                  <!-- thymeleaf a besoin de celui-ci -->
10 </head>
11
12 <body>
13
14     <table class = "table">
15         <tr>
16             <th>Id</th> <th>Description</th> <th>Prix</th>
17         </tr>
18         <tr th:each="a:${listArticle}">
19             <td th:text="${a.id}"> </td>
20             <td th:text="${a.description}"> </td>
21             <td th:text="${a.price}"> </td>
22         </tr>
23
24     </table>
```

th:href est un attribut spécifique à Thymeleaf permettant de définir l'URL cible d'un lien Html, renvoyant ici vers le fichier css



Lors d'une modification du code HTML, il faut redémarrer l'application pour voir les changements, car Thymeleaf utilise un cache pour optimiser le chargement des pages. C'est utile en production, mais gênant en développement. Il existe 2 solutions pour contourner cela :

- **Désactiver le cache** en ajoutant dans app.properties : spring.thymeleaf.cache = false

- **Utiliser la dépendance DevTools** : Spring désactivera les caches et redémarrera automatiquement l'application à chaque modification.

10 : UTILISATION DE BOOTSTRAP

```
<body>
    <div class = "container">
        <div class="panel panel-primary">
            <div class="panel-heading">Liste des articles</div>
            <div class="panel-body">
                <table class = "table">
                    <tr>
                        <th>Id</th> <th>Description</th> <th>Prix</th>
                    </tr>
                    <tr th:each="a:${listArticle}">
                        <td th:text="${a.id}"> </td>
                        <td th:text="${a.description}"> </td>
                        <td th:text="${a.price}"> </td>
                    </tr>
                </table>
            </div>
        </div>
    </div>

</body>
```

Liste des articles

Id	Description	Prix
1	Samsung S8	250.0
2	Samsung S9	300.0
3	Iphone 10	500.0

11 : PAGINATION

La vue ici peut solliciter notre contrôleur pour lui indiquer que telle page doit être affichée : **http://localhost:8080/index?page=1**
Reste à celui-ci de renvoyer la bonne page avec un nombre limité d'articles par pages comme ci-dessous

```
@GetMapping("/index")                    //dans une servlet on utilisait request.getParameter("page")
public String index(Model model, @RequestParam(name="page" , defaultValue = "0") int page) {
    Page<Article> articles = articleRepository.findAll(PageRequest.of(page, 5));
    //en retour, au lieu d'une liste d'articles, on a tous les articles formatés en page pointant sur la page demandée
    model.addAttribute("listArticle",articles.getContent());    //pour récupérer sous forme de liste la page pointée

    //pour afficher des liens de pagination permettant à l'utilisateur de passer d'une page à l'autre, il faut :
    //- récupérer le nombre total de pages
    //- l'injecter dans le model sous forme de tableau d'entier
    //- sur la partie html il suffira de boucler sur ce tableau pour afficher toutes les pages
    model.addAttribute("pages", new int[articles.getTotalPages()]);

    //s'agissant de l'activation des liens de navigation, il faut transmettre à la vue la page courante
    //thymeleaf pourra delors vérifier si la page courante est égal à l'index de la page active
    model.addAttribute("currentPage",page);

    return "articles";
}
```

3

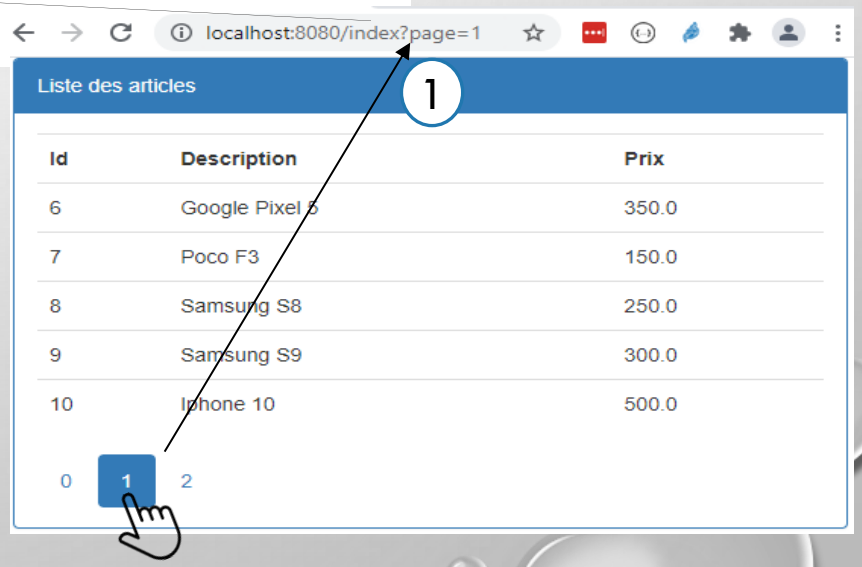
4 Model

5

```
<ul class = "nav nav-pills">            <!-- alignement de la liste -->
    <li th:class="${currentPage==status.index}?'active':''"  th:each="page,status:${pages}">
        <!-- 4 activer cette balise si condition v -->      <!-- 1 pour chaque indice de notre tableau "de pages"-->
        <a th:href="@{/index(page=${status.index})}" th:text="${status.index}"></a>
        <!-- 3 lien vers un indice/"page"            2 afficher l'indice du tableau -->
    </li>
</ul>
```

2

/index?page=0
/index?page=1
/index?page=2



1

En résumé, pour chaque balise <li> de notre boucle, nous affichons l'index compris entre 0 et le nombre total de pages (-1). Chaque index renvoie vers la page correspondante et l'index de la page courante est activé ou visible.

DispatcherServlet & Thymeleaf

- Le DispatcherServlet reçoit une requête HTTP pour une URL spécifique.
- Le DS consulte la configuration de l'application pour déterminer quel contrôleur doit gérer cette requête.
- Le DS appelle la méthode correspondante du contrôleur.
- Le contrôleur traite la demande, injecte le modèle dans la méthode et lui ajoute des attributs puis renvoie simplement le nom de la vue à afficher.
- Le DS récupère le nom de la vue renvoyé par le contrôleur et recherche la vue html correspondante.
- Une fois la vue trouvée, le DS transmet le modèle à la vue pour le rendu.
- La vue utilise les données du modèle pour générer la réponse HTML renvoyée au client.

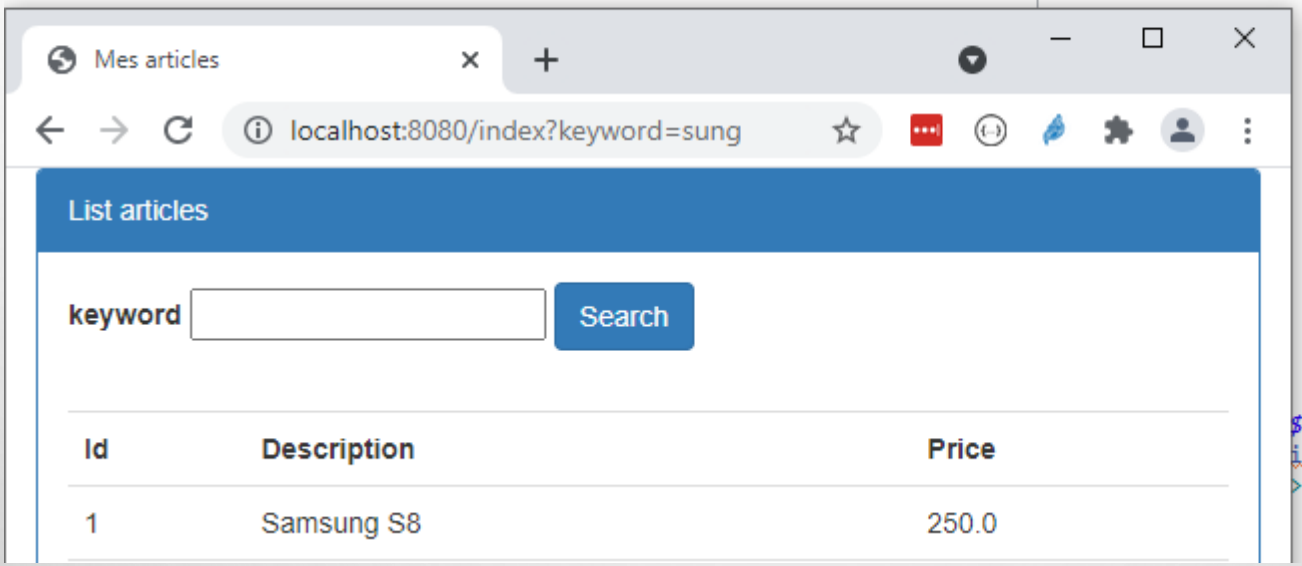
- La vue HTML (page HTML enrichie de balises Thymeleaf) contient des attributs Thymeleaf qui indiquent comment les données du modèle doivent être rendues dans la vue.
- Lorsque le navigateur demande cette vue, le serveur interprète d'abord le code HTML de la vue.
- Lorsqu'il rencontre des attributs Thymeleaf, Th entre en jeu pour traiter ces attributs.
- Thymeleaf extrait les données du modèle transmises par le DispatcherServlet et les insère dynamiquement dans les positions correspondantes de la vue HTML en fonction des directives Th spécifiées.
- Le code HTML final, enrichi des données du modèle, est renvoyé au navigateur de l'utilisateur

12 : UTILISATION DE SPRING DATA

Ensuite, on souhaite ajouter un formulaire permettant d'afficher tous les articles contenant un mot clé

12.1

```
<div class="panel-heading">List articles</div>

<div class="panel-body">
    <form th:action="@{/index}" method="get">       <!-- validation du formulaire -> appel de la méthode index -->
        <label>keyword</label>
        <input type="text" name="keyword">
        <button type="submit" class="btn btn-primary"> Search </button>
    </form>
</div>
```




```
${pages}">
ice de notre tableau "de pages"-->
>
 -->
```

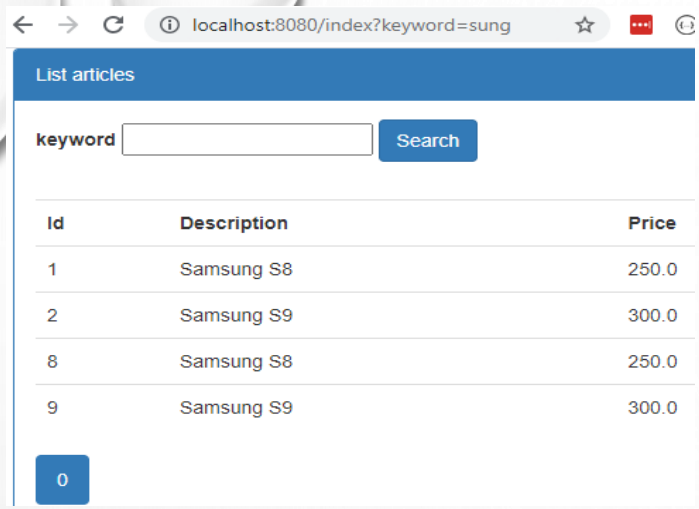
12.2

```
public interface ArticleRepository extends JpaRepository<Article, Long> {
    Page<Article>   findByDescriptionContains(String description , Pageable pageable);
}
```

Veiller ici à bien choisir les imports

```
@GetMapping("/index")
public String index(Model model, @RequestParam(name="page" , defaultValue = "0") int page,
                                 @RequestParam(name="keyword" , defaultValue = "") String kw) {
   Page<Article> articles = articleRepository.findByDescriptionContains(kw , PageRequest.of(page, 5));
```

12.3



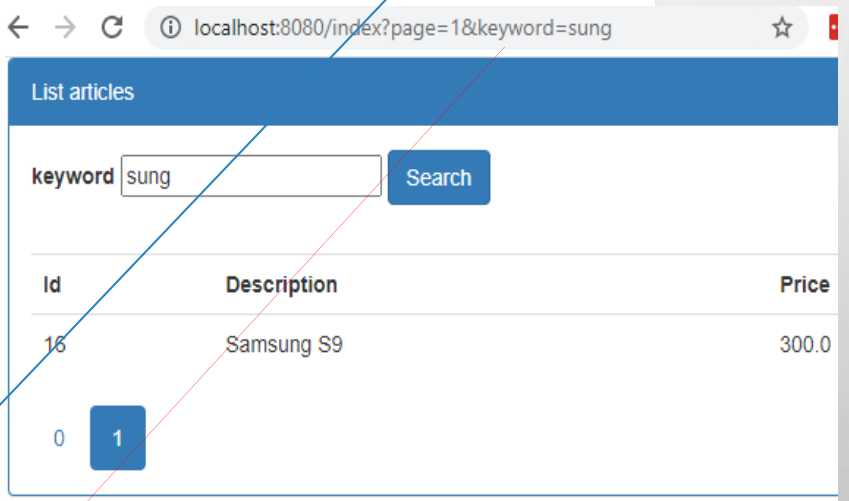
On souhaite au passage que le mot clé reste dans la zone de saisie lorsqu'on passe à une autre page :

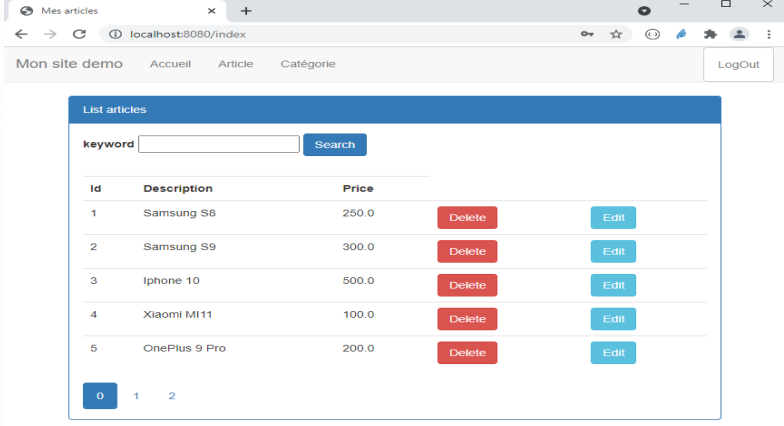
1/ Dans le contrôleur, il faut ajouter un attribut correspondant dans le model.

```
model.addAttribute("keyword",kw);
```

2/ Dans la vue, il faut afficher le mot clé et le rajouter dans la pagination pour nous permettre de naviguer sur le résultat de la recherche.

```
<div class="panel-body">
    <form th:action="@{/index}" method="get">        <!-- validation du formulaire -> appel de la méthode index -->
        <label>keyword</label>
        <input type="text" name="keyword" th:value="${keyword}">
        <button type="submit" class="btn btn-primary"> Search </button>
    </form>
</div>

<div class="panel-body">
    <table class = "table">
        <tr>
            <th>Id</th> <th>Description</th> <th>Price</th>
        </tr>
        <tr th:each="a:${listArticle}">
            <td th:text="${a.id}"> </td>
            <td th:text="${a.description}"> </td>
            <td th:text="${a.price}"> </td>
        </tr>
    </table>

    <ul class = "nav nav-pills">            <!-- alignement de la liste -->
        <li th:class="${currentPage==status.index}?'active':''"   th:each="page,status:${pages}">
            <!-- 4 activer cette balise si condition v -->     <!-- 1 pour chaque indice de notre tableau "de pages"-->
            <a th:href="@{/index(page=${status.index},keyword=${keyword})}" th:text="${status.index}"></a>
            <!-- 3 lien vers un indice/"page"       5 motclé=motclé courant     2 afficher l'indice du tableau -->
        </li>
    </ul>

</div>
```





12.4

On souhaite maintenant ajouter l'option de suppression d'un article :

- ajouter le lien correspondant dans la vue + ajouter les infos pour rester dans la même page

- ajouter la méthode dans le contrôleur, y insérer enfin la redirection vers la page principale

12.4.1

```
<tr th:each="a:${listArticle}">
    <td th:text="${a.id}"> </td>
    <td th:text="${a.description}"> </td>
    <td th:text="${a.price}"> </td>
    <td>                                <!-- après une suppression, pour garder le même contexte ou page -->
        <a th:href="@{/delete(id=${a.id} , page=${currentPage , keyword=${keyword})}" >Delete</a>
    </td>
</tr>
```

Manque qqchose ici ?

12.4.2

```
@GetMapping("/delete")      //on peut ne pas préciser le paramètre de la requete, il va rechercher sur la base id
public String delete(Long id, int page, String keyword) {

    articleRepository.deleteById(id);

    return "redirect:/index?page="+page+"&keyword="+keyword;
}
```

13 : AJOUT DU GESTIONNAIRE DE TEMPLATE THYMELEAF

Jusqu'ici, nous avons utilisé Thymeleaf pour gérer la Spring EL permettant l'interaction avec le contrôleur via le model. Il est aussi possible de l'utiliser comme générateur de template. En effet, un site est constitué de nombreuses pages html qui affichent les mêmes données statiques notamment dans le header, footer, menus…

Pour ce faire, il faut ajouter :

1/ une page template « layout.html » par ex qui contiendra tous les éléments commun

2/ les namespaces dans la page en question (th & layout/ajout dépendances dans pom.xml)

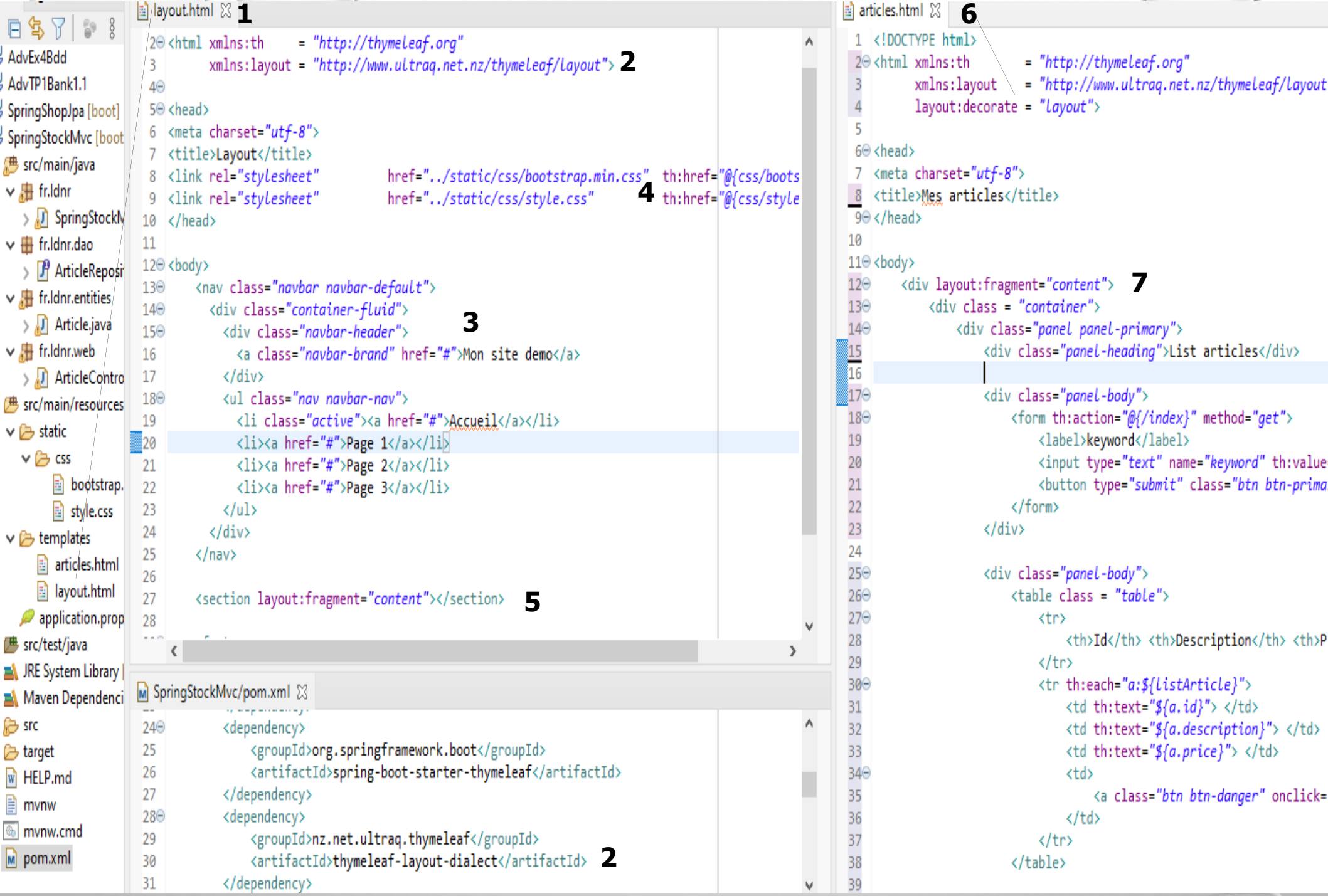
3/ les headers (barre de navigation par ex) et footers statiques

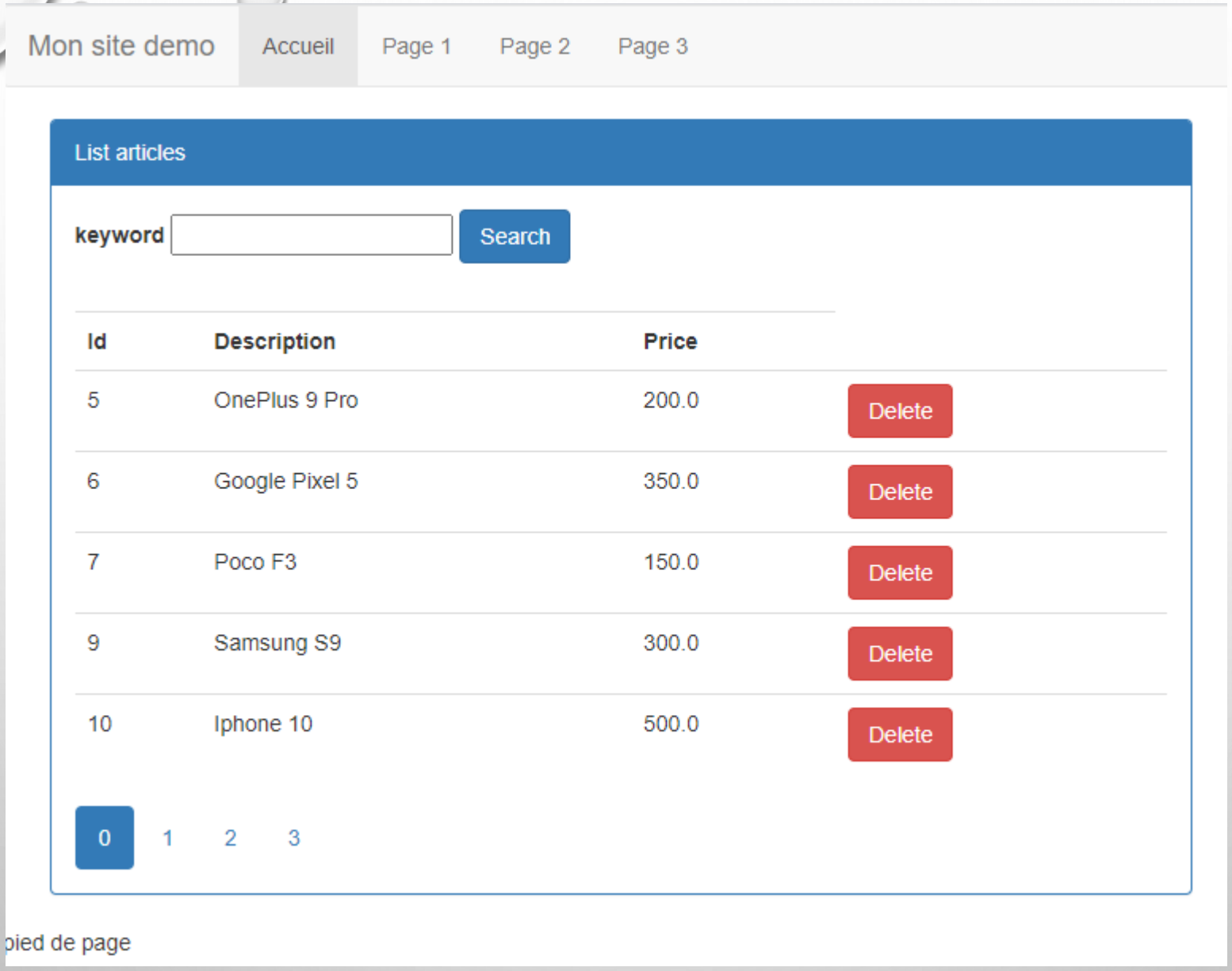
4/ les liens css s'il y en a

5/ dans le body une section layout:fragment pour les contenus dynamique

6/ ajouter une page html et demander à la décorer avec la page template (layout:decorator)

7/ ajouter dans cette page html, une section fragment qui contiendra les éléments dynamiques





ÉTAPE 13.1 : AJOUT D'UNE PAGE AVEC FORMULAIRE DE SAISI D'UN NOUVEL ARTICLE

1/ ajouter le lien vers le contrôleur dans la page template

```
<li><a th:href="@{/index}">Accueil</a></li>
<li><a th:href="@{/article}">Article</a></li>
```

2/ ajouter la méthode dans le contrôleur qui va renvoyer vers une nouvelle page article.html

3/ ajouter cette nouvelle page article.html dans le répertoire templates

4/ décorer celle-ci avec la page template + fragment comme vu auparavant

5/ ajouter un formulaire

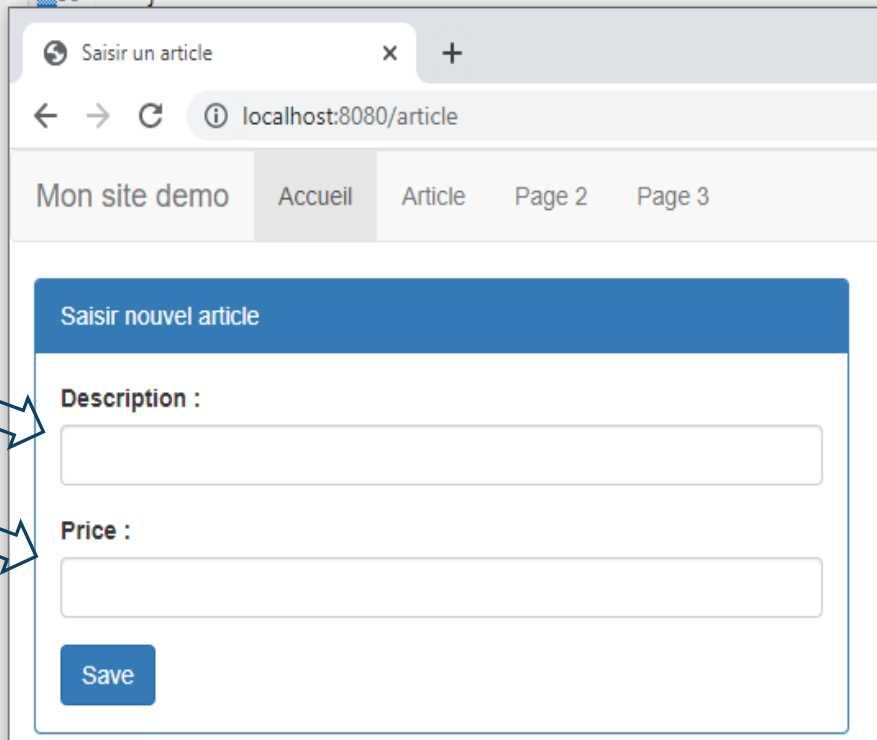
articles.html | article.html

```
<!DOCTYPE html>
<html xmlns:th="http://thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="layout">
<head>
<meta charset="utf-8">
<title>Saisir un article</title>
</head>

<body>
    <div layout:fragment="content">
        <div class="col-md-4 col-xs-8">
            <div class="panel panel-primary">
                <div class="panel-heading">Saisir nouvel article</div>
                <div class="panel-body">
                    <form th:action="@{/save}" method="post">
                            <div class="form-group">
                                <label class="control-label"> Description : </label> <input
                                    class="form-control" type="text" name="description">
                            </div>
                            <div class="form-group">
                                <label class="control-label"> Price : </label> <input
                                    class="form-control" type="text" name="price">
                            </div>
                            <button type="submit" class="btn btn-primary">Save</button>
                    </form>
                </div>
            </div>
        </div>
    </div>
</body>

</html>
```

NB : Attention à ne pas confondre @{/lien} et ${valeur}

ArticleController.java

```
52
53    @GetMapping("/article")
54    public String article() {
55        return "article";
56    }
```



6/ Ajouter la méthode dans le contrôleur et tester

```
@PostMapping("/save")
public String save(Article article) {
    articleRepository.save(article);
    return "article";
}
```

ÉTAPE 13.2 : VALIDATION DU FORMULAIRE

7/ Il s'agit ici d'un moyen d'obliger les users à saisir les données attendues/conformes :

→ Ajouter la dépendance de gestion de validation

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

→ Dans les entités Jpa, ajouter les annotations nécessaires

```
@NotNull
@Size(min=10,max=50)
private String description;

@DecimalMin("50")
private double price;
```

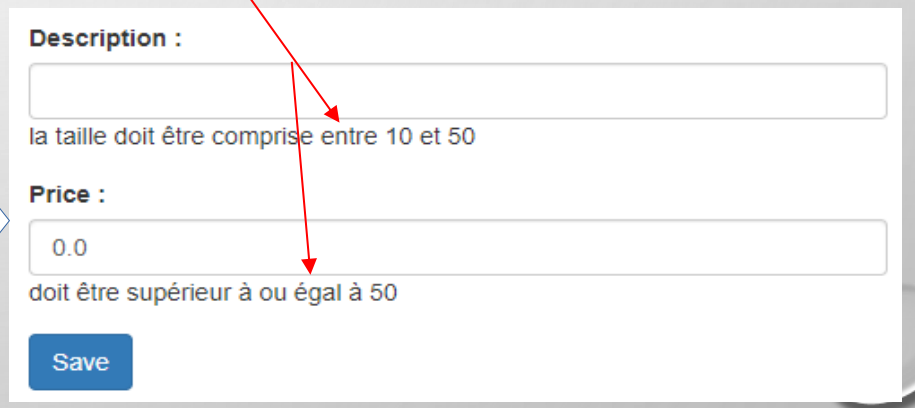
→ Dans la méthode de validation, ajouter les mécanismes qui permettront à Spring de vérifier les saisies et le cas échéant, les erreurs

```
@PostMapping("/save")
public String save(Model model, @Valid Article article , BindingResult bindingResult) {
    if(bindingResult.hasErrors())   return "article";
    // s'il n'y a pas de saisie d'un champ selon certains critères, pas d'insertion en base
    articleRepository.save(article);
    return "article";
}
```

→ Indiquer dans la vue les erreurs de saisie

```
<div class="form-group">
    <label class="control-label"> Description : </l
    <input class="form-control" type="text" name="d
    <span th:errors="${article.description}"></span
</div>
<div class="form-group">
    <label class="control-label"> Price : </label>
    <input class="form-control" type="text" name="p
    <span th:errors="${article.price}"></span>
</div>
```

Résultat final



BindingResult

Step by step

L'objet **BindingResult** est utilisé dans Spring MVC pour stocker les résultats de la liaison entre les données saisies par l'utilisateur dans un formulaire et les objets Java correspondants. Voici comment il fonctionne dans notre contexte :

1/ Lorsqu'un formulaire est soumis au contrôleur via une méthode en **@PostMapping**, Spring MVC associe automatiquement les données saisies par l'utilisateur aux propriétés de l'objet **Article** en fonction des noms des champs du formulaire et des noms des propriétés de l'objet.

2/ Après la liaison des données, Spring MVC exécute la validation sur l'objet **Article** en utilisant les annotations de validation définies sur ses propriétés (par exemple, **@NotNull**, **@Size**…)

3/ Si des erreurs de validation sont détectées pendant ce processus, elles sont stockées dans l'objet **BindingResult** associé à l'objet **Article**. Chaque erreur est associée à un champ spécifique du formulaire.

4/ Dans votre le contrôleur, vous pouvez vérifier si des erreurs de validation se sont produites en appelant la méthode **hasErrors()** de l'objet **BindingResult**. Si des erreurs sont présentes, cela signifie que des données invalides ont été soumises dans le formulaire.

5/ Renvoi à nouveau du formulaire avec les erreurs affichées pour que l'utilisateur les corrige.
if(bindingResult.hasErrors()) teste si des erreurs de validation sont présentes. Si c'est le cas, le contrôleur renvoie à nouveau la vue "article" avec les erreurs de validation affichées.

Enfin Ajouter un article vide dans le model pour insérer des données par défaut dans les champs de saisie

```java
@GetMapping("/article")
public String article(Model model) {
    model.addAttribute("article" , new Article());      //injection d'un article par défaut dans le formulaire de la vue article
    return "article";
}

@PostMapping("/save")
public String save(@Valid Article article , BindingResult bindingResult) {
    if(bindingResult.hasErrors())   return "article";
    // s'il n'y a pas de saisie d'un champ selon certains critères, pas d'insertion en base
    articleRepository.save(article);
    return "redirect:/index";
}
```

Indispensable car nous essayons d'accéder aux attributs de l'objet côté vue

Grâce à la fonctionnalité de liaison de données de spring Mvc permettant de mapper automatiquement les données provenant de la requêtes http aux attributs de l'objet Article, on peut l'exploiter ici !

Une fois validé le formulaire, SpringMvc vérifie que l'objet article correspond aux conditions prévues via les mécanismes d'annotations, si ce n'est pas le cas l'objet bindingResult comprend des erreurs récupérées ici par thymeleaf pour informer les utilisateurs

```html
<div layout:fragment="content">
    <div class="col-md-4 col-xs-8">
        <div class="panel panel-primary">
            <div class="panel-heading">Saisir nouvel article</div>
            <div class="panel-body">
                <form th:action="@{/save}" method="post">
                        <div class="form-group">
                            <label class="control-label"> Description : </label>
                            <input class="form-control" type="text" name="description" th:value="${article.description}">
                            <span th:errors="${article.description}" class="text-danger"></span>
                        </div>
                        <div class="form-group">
                            <label class="control-label"> Price : </label>
                            <input class="form-control" type="text" name="price" th:value="${article.price}">
                            <span th:errors="${article.price}" class="text-danger"></span>
                        </div>
                        <button type="submit" class="btn btn-primary">Save</button>
                </form>
            </div>
        </div>
    </div>
</div>
```

SYNTHESE SPRING MVC

- Possible de mettre en œuvre plusieurs contrôleurs : Article, Catégorie, Cart, Login…
- Et de nombreuses pages Html associés : articles.html, cart.html, order.html…

- Chaque contrôleur, une fois sollicité, peut facilement injecter des données dans le model(après avoir sollicité Spring Data) et renvoyer vers la vue qui n'a plus qu'à se servir.

- Nous avons utilisé Thymeleaf pour 2 raisons (il existe d'autre solution) :
  - Gestion de l'utilisation de la Spring Expression Language
  - Exploiter son template engine ou moteur de rendu de document
    - Au passage, il en existe d'autre : FreeMarker, Groovy Template, Mustache

- Le process de mise en œuvre d'une feature est simple :
  - Réalisation de la maquette puis de la vue
  - Suivi du contrôleur qui utilise spring data ici avant de renvoyer la vue au DS
  - Possibilité d'intégrer un framework Css
  - Mise en œuvre de la pagination ou formulaire simplifié avec option Validation

- Reste la mise en œuvre de la couche sécurité

Angular & Spring