



Une entreprise ordinaire à l'agilité extraordinaire

FORMATION JAVA SPRING ANGULAR

SPRING MVC

mohamed.el-babili@fms-ea.com

33+ 628 111 476

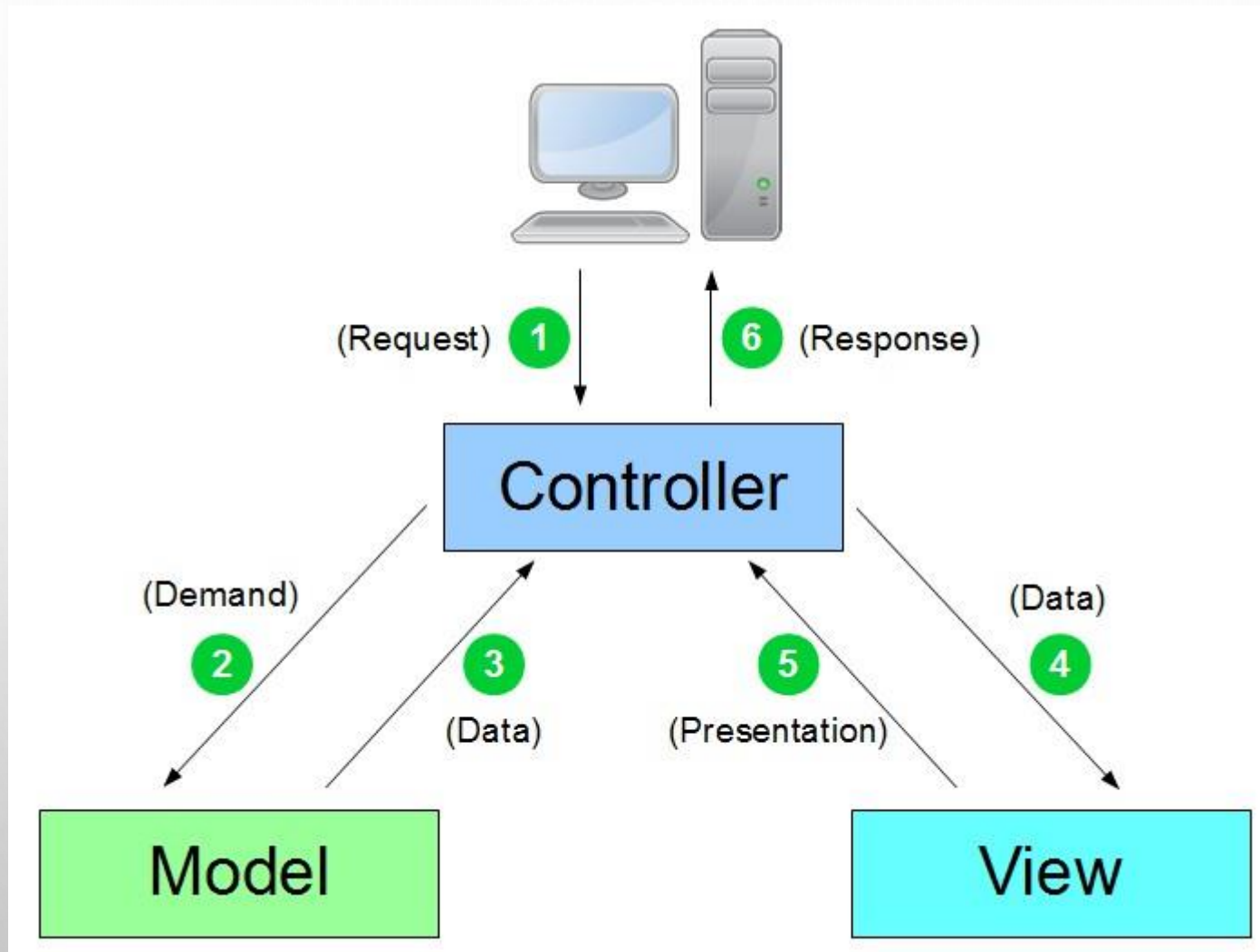
Version : 2.0

DMAJ : 22/05/23

SOMMAIRE

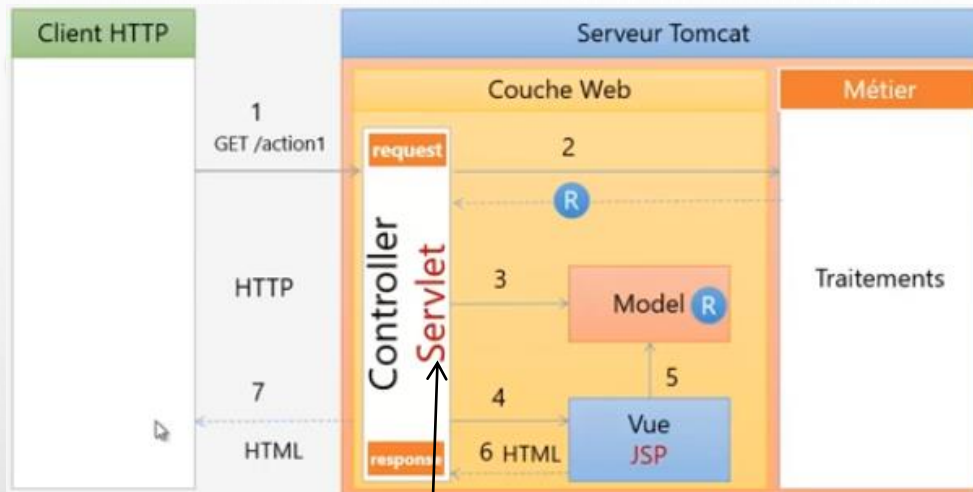
- MVC
- ARCHITECTURE JAVA EE / SPRING MVC
- SPÉCIFICATIONS DE L'APPLI À RÉALISER
- MAQUETTE/APPLI
- ARCHITECTURE DE L'APPLI
- RÉALISATION DE LA 1ÈRE PHASE DE L'APPLI
- RÉALISATION DE LA COUCHE WEB/MVC
- SYNTHÈSE SPRING MVC

MVC



ARCHITECTURE JAVA EE / SPRING MVC

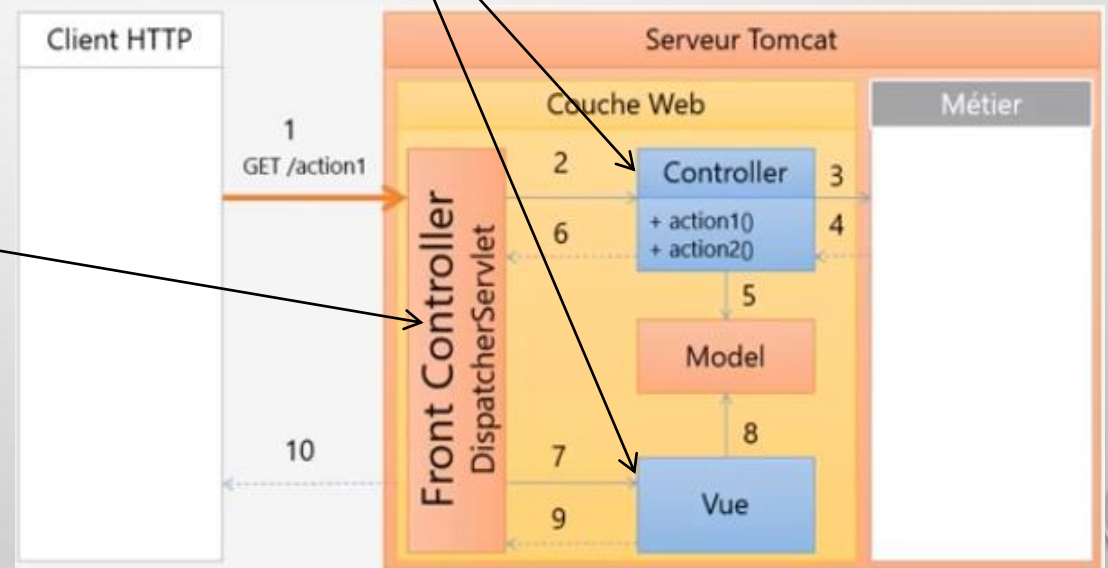
Architecture Java EE



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). Celui ci 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 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 / Spring Data Jpa Hibernate / Spring Mvc / Thymeleaf / Lombok
- Java 8 / Eclipse / SGBD MariaDB / BootStrap

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

Mes articles

localhost:8080/index

Mon site demo

Accueil

Article

Catégorie

LogOut

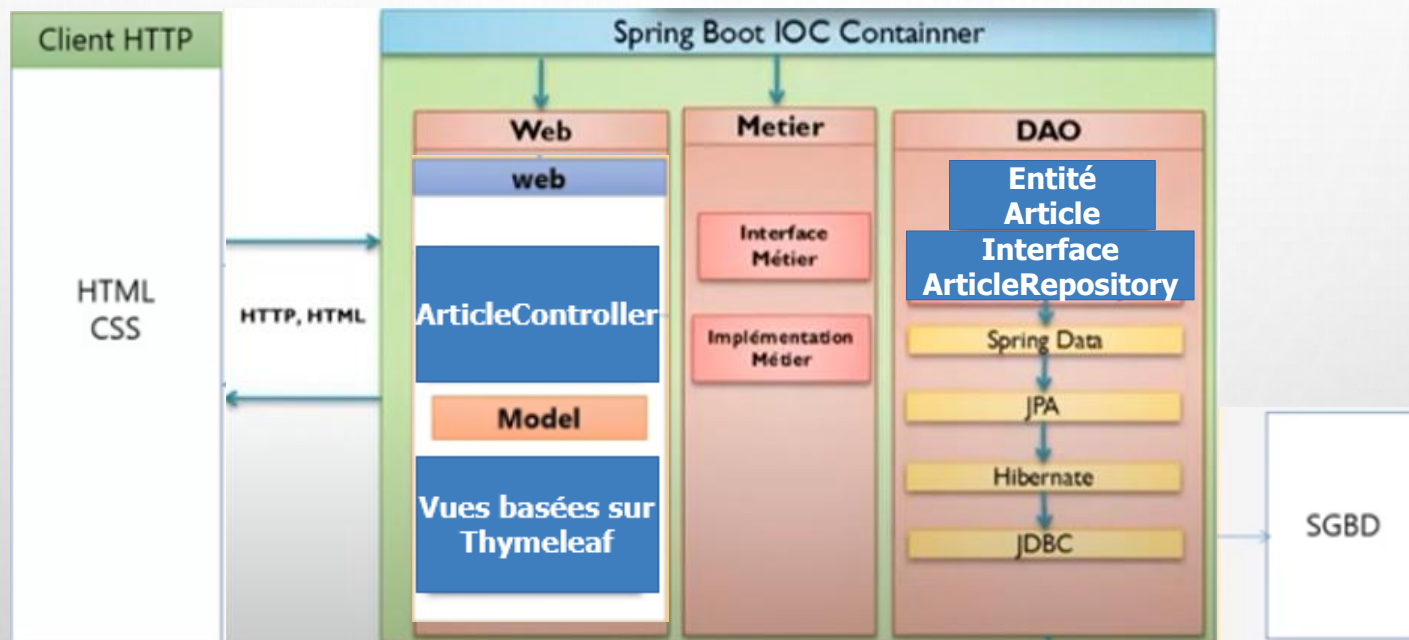
List articles

keyword

Id	Description	Price		
1	Samsung S8	250.0	<input type="button" value="Delete"/>	<input type="button" value="Edit"/>
2	Samsung S9	300.0	<input type="button" value="Delete"/>	<input type="button" value="Edit"/>
3	Iphone 10	500.0	<input type="button" value="Delete"/>	<input type="button" value="Edit"/>
4	Xiaomi MI11	100.0	<input type="button" value="Delete"/>	<input type="button" value="Edit"/>
5	OnePlus 9 Pro	200.0	<input type="button" value="Delete"/>	<input type="button" value="Edit"/>

1 2

ARCHITECTURE DE L'APPLI



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)

New Spring Starter Project

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

Package:

Working sets

☐ Add project to working sets

Working sets:

New Spring Starter Project Dependencies

Spring Boot Version:

Frequently Used:

☒ Lombok ☒ MariaDB Driver ☒ Spring Data JPA

☒ Spring Web ☒ Thymeleaf

Available:

- ▶ Developer Tools
- ▶ Google Cloud Platform
- ▶ I/O
- ▶ Messaging
- ▶ Microsoft Azure
- ▶ NoSQL
- ▶ Observability
- ▶ Ops
- ▶ SQL
- ▶ Security
- ▶ Spring Cloud
- ▶ Spring Cloud Circuit Breaker
- ▶ Spring Cloud Config
- ▶ Spring Cloud Discovery
- ▶ Spring Cloud Messaging
- ▶ Spring Cloud Routing

Selected:

- X Lombok
- X Spring Data JPA
- X MariaDB Driver
- X Thymeleaf
- X Spring Web

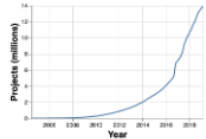
SOLUTION AVEC PROJET EXISTANT EN UTILISANT MAVEN

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

← → ↻ mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web/2.7.0

MVNREPOSITORY Search for groups, artifacts, categories

Indexed Artifacts (28.4M)



Popular Categories

- Aspect Oriented
- Actor Frameworks
- Application Metrics
- Build Tools
- Bytecode Libraries
- Command Line Parsers
- Cache Implementations
- Cloud Computing
- Code Analyzers
- Collections
- Configuration Libraries
- Core Utilities
- Date and Time Utilities
- Dependency Injection
- Embedded SQL Databases
- HTML Parsers
- HTTP Clients
- I/O Utilities
- JDBC Extensions
- JDBC Pools
- JPA Implementations
- JSON Libraries
- JVM Languages
- Logging Frameworks

Home » org.springframework.boot » spring-boot-starter-web » 2.7.0

Spring Boot Starter Web » 2.7.0

Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container

License	Apache 2.0
Categories	Web Frameworks
Organization	Pivotal Software, Inc.
HomePage	https://spring.io/projects/spring-boot
Date	(May 19, 2022)
Files	pom (2 KB) jar (4 KB) View All
Repositories	Central
Ranking	#51 in MvnRepository (See Top Artifacts)
Used By	9,076 artifacts

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.7.0</version>
</dependency>
```

☒ Include comment with link to declaration

Compile Dependencies (5)

Category / License	Group / Artifact
Web Framework Apache 2.0	org.springframework » spring-web
Web Framework Apache 2.0	org.springframework » spring-webmvc
App Server Apache 2.0	org.springframework.boot » spring-boot-starter

Lombok

```
@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)
```

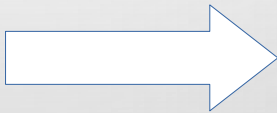
```
@SpringBootApplication
public class SpringStockMvcApplication implements CommandLineRunner {
    @Autowired
    ArticleRepository articleRepository;

    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



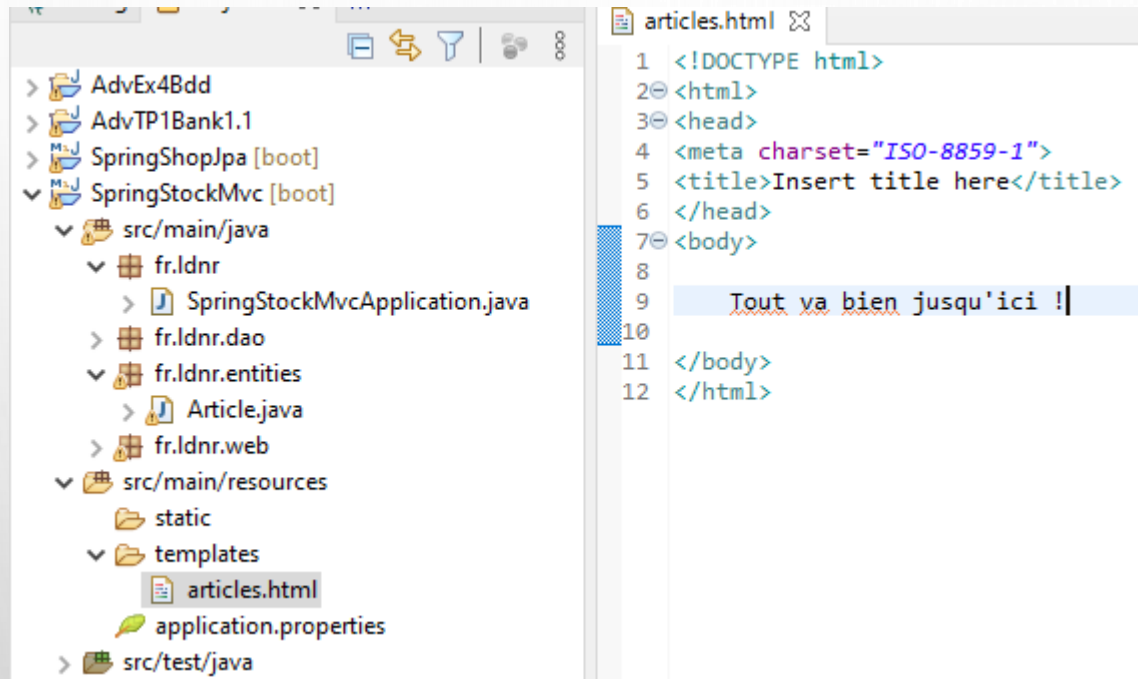
The screenshot displays an IDE interface. On the left, the 'Project Explorer' shows a project structure with the following hierarchy:

- > AdvEx4Bdd
- > AdvTP1Bank1.1
- > SpringShopJpa [boot]
- ▼ SpringStockMvc [boot]
 - ▼ src/main/java
 - ▼ fr.ldnr
 - ▼ fr.ldnr.web
 - > ArticleController.java
 - > SpringStockMvcApplication.java
 - > fr.ldnr.dao
 - ▼ fr.ldnr.entities
 - > Article.java
 - > fr.ldnr.web
 - > src/main/resources
 - > src/test/java
 - > JRE System Library [JavaSE-1.8]
 - > Maven Dependencies

On the right, the 'Editor' shows the code for `ArticleController.java`:

```
2
3+ import org.springframework.beans.factory.annotation.Autowired;
10
11 @Controller
12 public class ArticleController {
13     @Autowired
14     ArticleRepository articleRepository;
15
16     // @RequestMapping(value="/index" , method=RequestMethod.GET)
17     @GetMapping("/index")
18     public String index() {
19         return "articles"; //cette méthode retourne au dispatcherServlet une vue
20     }
21 }
22
```

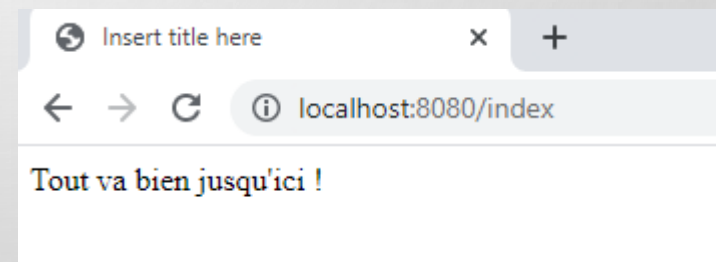
8: AJOUTER UNE VUE DANS LES RESSOURCES/TEMPLATES



Important pour la suite :

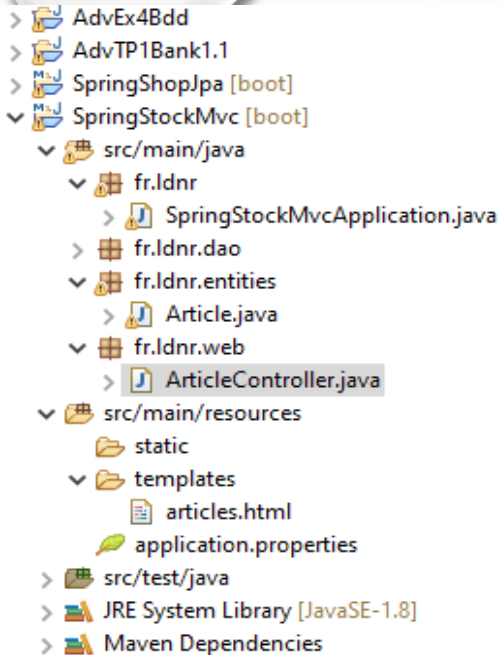
- N'oubliez pas de mettre entre commentaires vos insertions en base pour éviter les doublons (mode update)
- Privilégiez le stop and run afin d'éviter tout conflit sur le port du serveur http
- Vous pouvez changer le numéro de port dans le fichier `app.properties`, ex : `Server.port=8081`

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



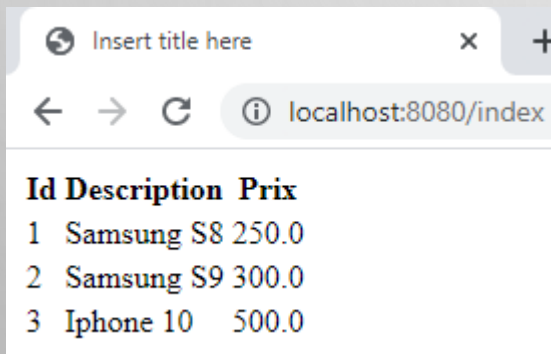
NB : contrairement au dossier `templates`, le dossier `static` contient les éléments qui ne nécessitent pas de traitement côté serveur (page html statique...)

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



```
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Controller;
7 import org.springframework.ui.Model;
8 import org.springframework.web.bind.annotation.GetMapping;
9
10 import fr.lidnr.dao.ArticleRepository;
11 import fr.lidnr.entities.Article;
12
13 @Controller
14 public class ArticleController {
15     @Autowired
16     ArticleRepository articleRepository;
17
18     // @RequestMapping(value="/index" , method=RequestMethod.GET)
19     @GetMapping("/index")
20     public String index(Model model) { //le model est fourni par spring, je peux l'utiliser comme ci
21         List<Article> articles = articleRepository.findAll(); //récupère tous les articles
22         model.addAttribute("listArticle", articles); //insertion de tous les articles dans le model
23                                     //accessible via l'attribut "listArticle"
24         return "articles"; //cette méthode retourne au dispatcherServlet la vue articles.html
25     }
26 }
```

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



```
<!DOCTYPE html>
<html xmlns:th="http://thymeleaf.org">
<head>
<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>
```

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

The screenshot shows an IDE with three panels. The left panel displays the project structure for 'SpringStockMvc [boot]':

- src/main/java
 - fr.Idnr
 - SpringStockMvcApplication.java
 - fr.Idnr.dao
 - fr.Idnr.entities
 - Article.java
 - fr.Idnr.web
 - ArticleController.java
- src/main/resources
 - static
 - css
 - bootstrap.min.css
 - templates
 - articles.html
 - application.properties

The middle panel shows the content of 'articles.html' with the following code:

```
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     </table>
24
```

The right panel shows a browser preview of the application at 'localhost:8080/index'. The page title is 'Mes articles'. It displays a table with the following data:

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

Nb : lors d'une modif sur le code html, pour rafraîchir la page, il faut redémarrer l'appli car thymeleaf utilise un cache (mémoire tampon) pour optimiser le chargement des pages html, par ex la même requête recevra les infos déjà en cache. En production c'est idéal mais en phase de dev pour éviter le reboot, on peut le désactiver en ajoutant dans le fichier app.properties : `spring.thymeleaf.cache = false`

L'autre astuce consiste à ajouter la dépendance devtools (mode dev) de sorte que spring désactive tous les caches et redémarrer l'appli à la moindre modif !! à vous de voir

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")
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 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 alors vérifier si la page courante est égal à l'index de la page active
    model.addAttribute("currentPage", page);

    return "articles";
}
```

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

En résumé, pour chaque balise 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.

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

localhost:8080/index?page=1

Liste des articles		
Id	Description	Prix
6	Google Pixel 3	350.0
7	Poco F3	150.0
8	Samsung S8	250.0
9	Samsung S9	300.0
10	iPhone 10	500.0
0	1	2

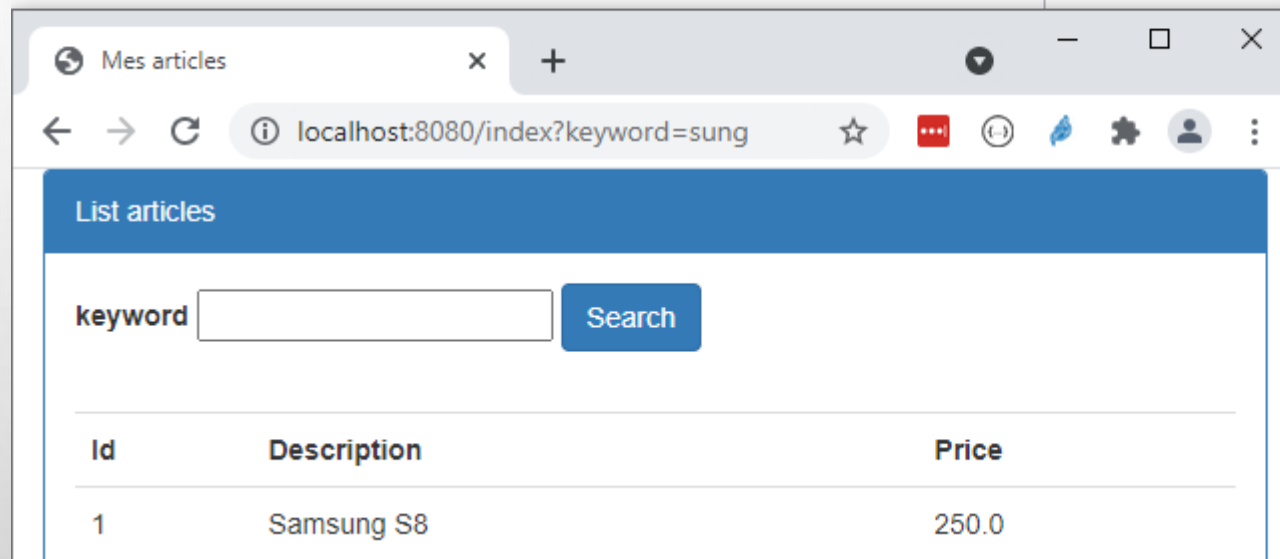
12 : UTILISATION DE SPRING DATA

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>
```



Mes articles

localhost:8080/index?keyword=sung

List articles

keyword Search

Id	Description	Price
1	Samsung S8	250.0

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

12.2

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

```
@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));
```

Veiller ici à bien choisir les imports

12.3

localhost:8080/index?keyword=sung

List articles

keyword Search

Id	Description	Price
1	Samsung S8	250.0
2	Samsung S9	300.0
8	Samsung S8	250.0
9	Samsung S9	300.0

0

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

- Dans le contrôleur, il faut ajouter un attribut correspondant dans le model. `model.addAttribute("keyword", kw);`

- Dans la vue, il faut afficher l'attribut et rajouter dans la pagination le mot clé 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">
    <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>
```

localhost:8080/index?page=1&keyword=sung

List articles

keyword sung Search

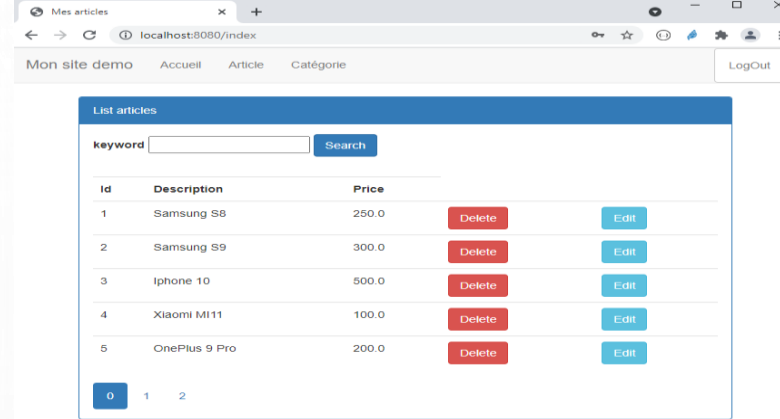
Id	Description	Price
16	Samsung S9	300.0

0 1

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>
    <a th:href="@{/delete(id=${a.id} , page=${currentPage} , keyword=${keyword})}" >Delete</a>
  </td>
</tr>
```

<!-- après une suppression, pour garder le même contexte ou page -->

Manque qqchose ici ?

12.4.2

```
@GetMapping("/delete") //on peut ne pas préciser le paramètre de la requête, 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/ une nouvelle page html et demander à la décorer avec la page template (layout:decorator)
- 7/ dans cette nouvelle page html, une section fragment qui contiendra les éléments dynamiques

List articles

keyword

Search

Id	Description	Price	
5	OnePlus 9 Pro	200.0	Delete
6	Google Pixel 5	350.0	Delete
7	Poco F3	150.0	Delete
9	Samsung S9	300.0	Delete
10	Iphone 10	500.0	Delete

0

1

2

3

pied de page

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
- 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

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

```
articles.html | article.html  
1 <!DOCTYPE html>  
2 <html xmlns:th="http://thymeleaf.org"  
3   xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"  
4   layout:decorate="layout">  
5 <head>  
6 <meta charset="utf-8">  
7 <title>Saisir un article</title>  
8 </head>  
9  
10 <body>  
11   <div layout:fragment="content">  
12     <div class="col-md-4 col-xs-8">  
13       <div class="panel panel-primary">  
14         <div class="panel-heading">Saisir nouvel article</div>  
15         <div class="panel-body">  
16           <form th:action="@{/save}" method="post">  
17             <div class="form-group">  
18               <label class="control-label">Description : </label> <input  
19                 class="form-control" type="text" name="description">  
20             </div>  
21             <div class="form-group">  
22               <label class="control-label">Price : </label> <input  
23                 class="form-control" type="text" name="price">  
24             </div>  
25             <button type="submit" class="btn btn-primary">Save</button>  
26           </form>  
27         </div>  
28       </div>  
29     </div>  
30   </div>  
31 </body>  
32  
33 </html>
```

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

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

Saisir un article x +

localhost:8080/article

Mon site demo Accueil Article Page 2 Page 3

Saisir nouvel article

Description :

Price :

Save

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 avant validation :

→ 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

→ Indiquer dans la vue les erreurs de saisie

```
@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";
}
```

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

Résultat final

Description :

la taille doit être comprise entre 10 et 50

Price :

doit être supérieur à ou égal à 50

Save

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

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

Une fois validé le formulaire, spring 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

```
<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
- 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 avant de renvoyer vers la vue
 - 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