Leçon 1 : Introduction à Spring Boot

Présentation de Spring Boot :

Le Framework Spring Boot est une extension du Framework Spring qui a pour but de mettre en place rapidement des applications Java (Web et/ou API Rest). Grâce à son système modulaire de dépendances et son principe de configuration automatique, il permet de disposer d'une structure de projet complète et immédiatement opérationnelle.

Les principaux avantages du Framework Spring Boot (par rapport au Framework Spring) :

- Un site starter : (`https://start.spring.io/`) qui permet de générer rapidement la structure du projet en y incluant toutes les dépendances (bibliothèques) nécessaires à l'application. De ce fait, toutes les dépendances sont regroupées dans un même dépôt afin de faciliter la gestion de celles-ci et assurer la compatibilité entre eux.
- Déploiement simple : un conteneur embarqué (ex : Tomcat) dans le projet afin de faciliter son déploiement. On a plus besoin d'un serveur externe pour le déploiement comme en Java EE.
- L'autoconfiguration : applique une configuration par défaut au démarrage de l'application pour toutes dépendances présentes dans celle-ci.

Principe de l'injection des dépendances : (Couplage faible)

Le noyau du Framework Spring et de Spring Boot Spring Core se base sur un patron de conception appelé *Inversion of Control (IoC)*.

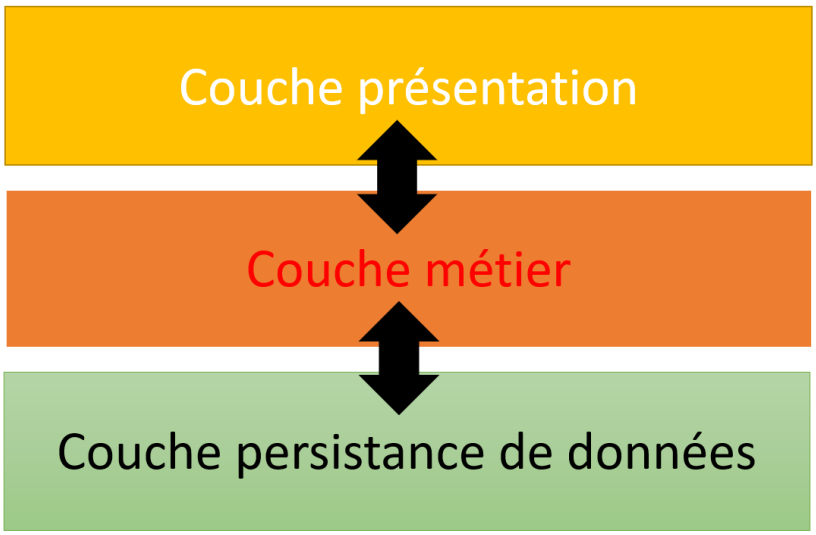
L'injection des dépendances implémente le principe de l'IoC. Elle permet **d'instancier** des objets et de créer des dépendances nécessaires entre elles, sans avoir besoin de coder cela par les développeurs (en utilisant l'instanciation statique avec l'opérateur new). C'est au Framework de gérer ces dépendances dynamiquement au moment du déploiement de l'application Spring Boot.

Ceci a pour but de diminuer le couplage entre les objets (appliquer le principe de couplage faible entre les objets) afin d'obtenir un code lisible et facilement extensible.

Architecture logicielle d'une application Spring Boot :

L'architecture d'un logiciel décrit la manière dont seront organisés les différents éléments d'une application et comment ils interagissent entre eux. Il existe plusieurs types d'architecture parmi eux la plus connue celle organisée en couches.

C'est une architecture hiérarchique où chaque niveau utilise les services de la couche en dessous et offre ses services à la couche en dessus comme le montre la figure suivante :



➢ La couche présentation a pour rôle de gérer les requêtes et les réponses HTTP avec le client et de traduire les paramètres des requêtes en objets. Les composants de cette couche différent selon le type de l'application Spring Boot à réaliser (Web ou API REST).

➢ La couche métier gère toute la logique métier de l'application : elle récupère les données de la couche persistance, effectue les traitements et validation sur ces données et les expose à la couche présentation comme des services.

➢ La couche persistance de données contient toute la logique de stockage et accès aux données (couche DAO). Elle effectue aussi la correspondance entre les objets entités depuis et vers des lignes dans la BD moyennant un ORM (Object Relational Mapping).

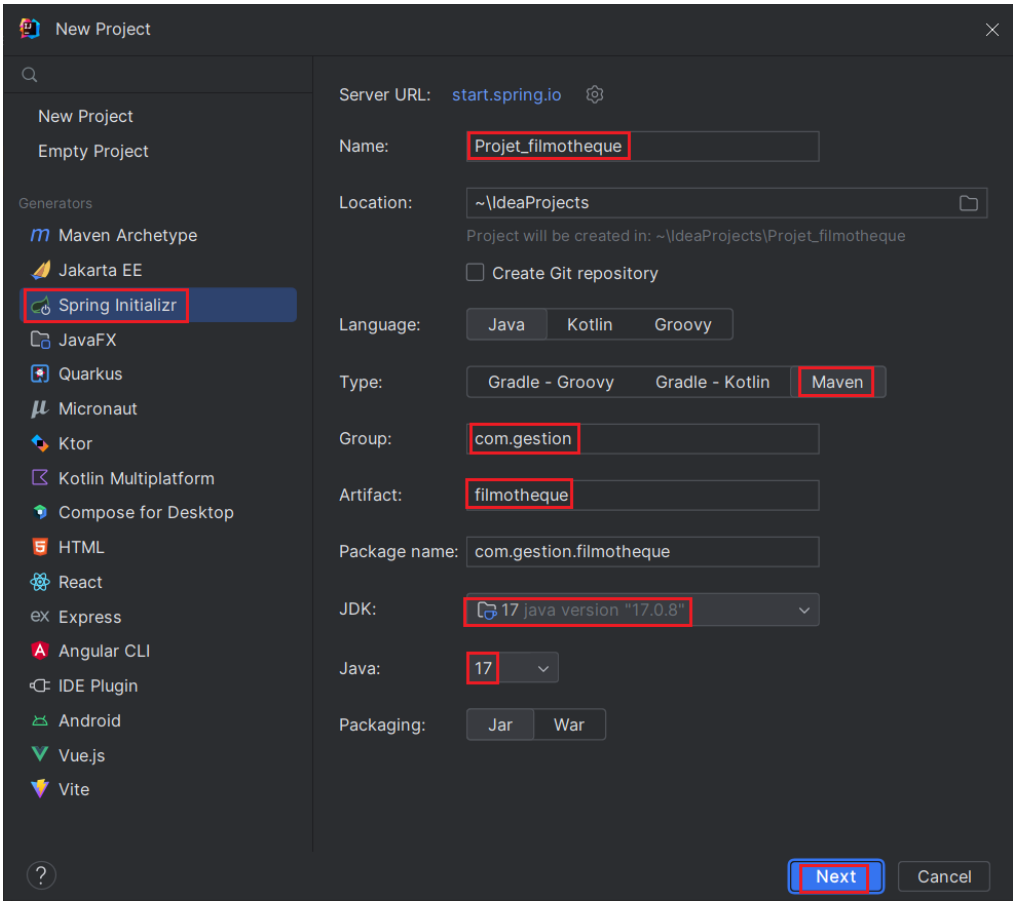
Enoncé du TP1 :

L'objectif de ce TP est de développer une application Spring web MVC comme étant un projet Spring Boot permettant la gestion (CRUD) d'une BD composée de 2 tables reliées entre elles `film` et `categorie`. La règle de gestion étant une catégorie contient plusieurs films et un ou plusieurs films appartiennent à une catégorie.

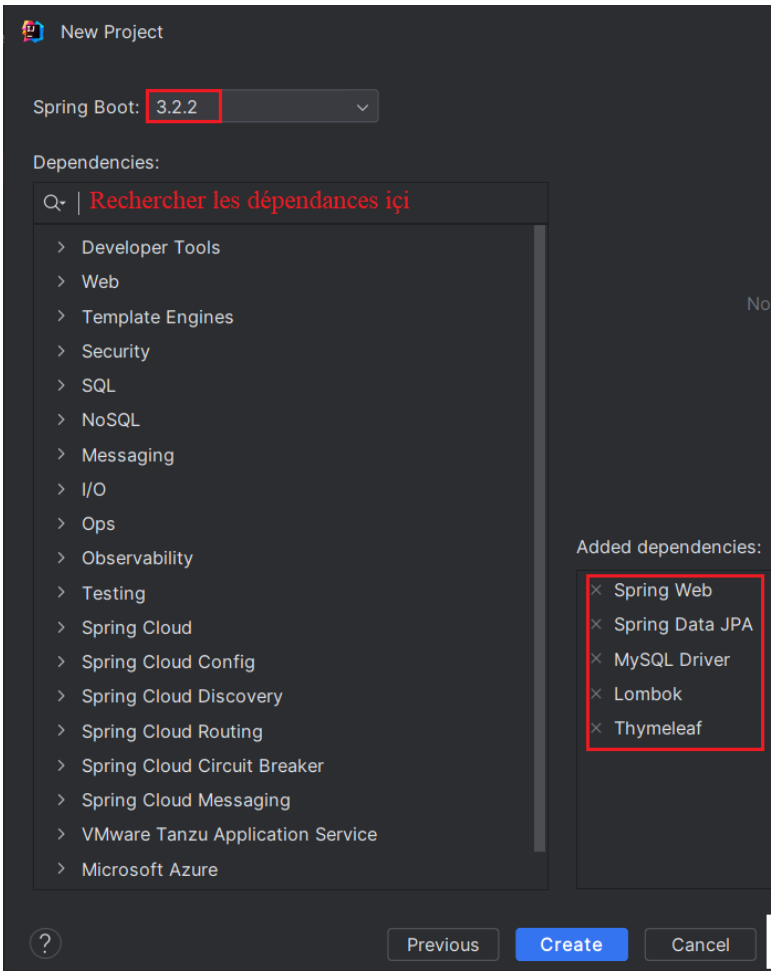
Nous allons commercer par créer un nouveau projet Spring Boot avec des dépendances gérées par Maven. Maven est un gestionnaire des dépendances des projets Java. Cette gestion s'articule sur un élément POM (Project Object Model) matérialisé par un fichier XML qui contient la description des ces dépendances du projet. Les dépendances à ajouter dans le projet sont :

- `Spring Web` : définit une application web MVC RestFull avec Tomcat comme serveur intégré
- `Spring Data JPA` : c'est la dépendance qui permet de persister les données dans une BD en utilisant le Framework Hibernate
- `MySQL Driver` : le pilote pour se connecter à des BD MySQL
- `Lombok` : dépendance pour réduire le code des classes entités
- `Thymeleaf` : dépendance pour utiliser le moteur de Template Thymeleaf dans les vues.

Lancer le IDE Intellij, puis créer un nouveau projet Spring Initializr.



En cliquant sur Next pour définir la version de Spring Boot et choisir les dépendances :



En cliquant sur Create, Maven va télécharger les dépendances depuis le dépôt central vers le dépôt local (le dossier `.m2` situé sous `C:\users\votre_session`). Pour la première création d'un projet, le téléchargement peut prendre quelques minutes selon le débit de la connexion.

Le package principale du projet `com.gestion.filmotheque` contient la classe exécutable `FilmothequeApplication.java`; qui représente le point de démarrage de l'application Spring Boot (on l'exécute comme une Application Java).

En respectant l'architecture d'une application Spring Boot (voir plus haut), Vous devez créer au fur et à mesure les packages suivants :

- `com.gestion.filmotheque.entities` contenant la définition des entités `Film` et `Categorie`.
- `com.gestion.filmotheque.repository` contenant 2 interfaces `FilmRepository` et `CategorieRepository` héritant chacune de l'interface générique `JpaRepository<T,ID>`
- Les paramètres d'accès à la BD sont définis dans le fichier `application.properties` sous `src/main/resources`.

Ces 3 composants représentent la couche persistance dans l'architecture de l'application.

- `com.gestion.filmotheque.service` contenant 2 interfaces qui contiennent chacune la signature des méthodes métier sur les entités `Film` et `Categorie` et 2 classes (annotées par `@Service`) implémentant chacune les méthodes métier déclarées dans chacune des interfaces.

Le contenu de ce package représente la couche métier de l'application.

- `com.gestion.filmotheque.controller` contenant 2 classes dont chacune joue le rôle d'un contrôleur web (annoté par `@Controller`) et contiendra les actions traitant les requêtes HTTP. Chaque action retournera soit le nom de la vue, soit une redirection vers un URL d'une autre action. Le contenu de ce package représente la couche présentation de l'application.

Etape 1 : Définition des entités

Lombok est une dépendance qui a pour but de réduire le code des classes entités en remplaçant la définition des setters, getters et constructeurs par des annotations.

L'annotation `@Data` permet de remplacer tous les getters et setters des attributs.

L'annotation `@NoArgsConstructor` permet de remplacer le constructeur par défaut

L'annotation `@AllArgsConstructor` permet de remplacer le constructeur avec paramètres

Créer le package `com.gestion.filmotheque.entities` puis commencer par créer l'entité `Film`

```java
package com.gestion.filmotheque.entities;


@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Film {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String titre;
    private String description;
    private int anneeparution;
    @ManyToOne
    private Categorie categorie;

}
```

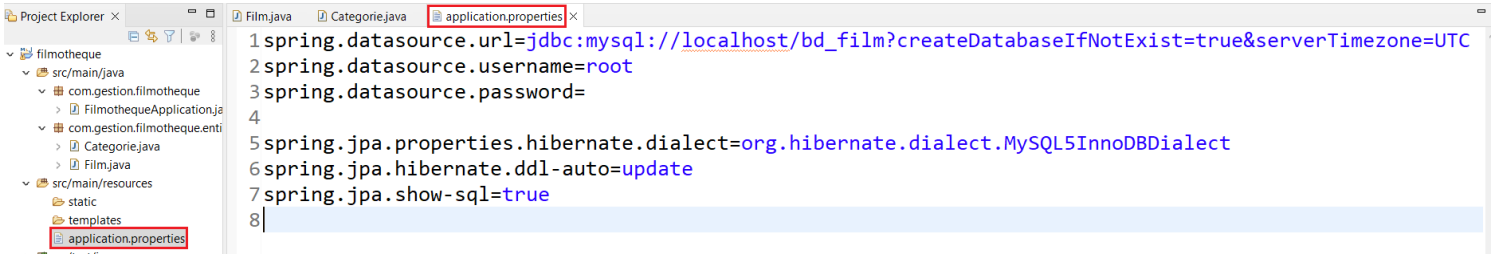
Créer l'entité `Categorie` comme suit :

```java
package com.gestion.filmotheque.entities;


@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Categorie {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String nom;
    @OneToMany(mappedBy = "categorie", cascade = CascadeType.ALL)
    List<Film> films;
}
```

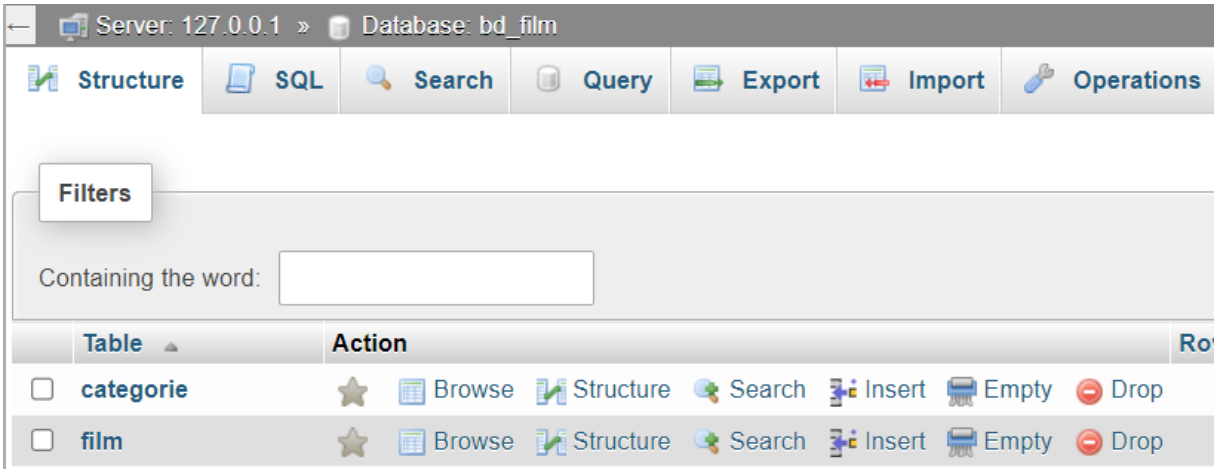
Les 2 entités `Film` et `Categorie` sont reliés respectivement par 2 relations `ManyToOne` et `OneToMany` en choisissant l'entité Categorie comme esclave dans cette relation (`mappedBy`). L'option cascade prend la valeur All ce qui signifie entre autres la suppression d'une catégorie entraine la suppression des films qui lui appartient.

Etape 2 : Définir les paramètres d'accès à la BD

Ajouter dans le fichier `application.properties` les paramètres suivants :

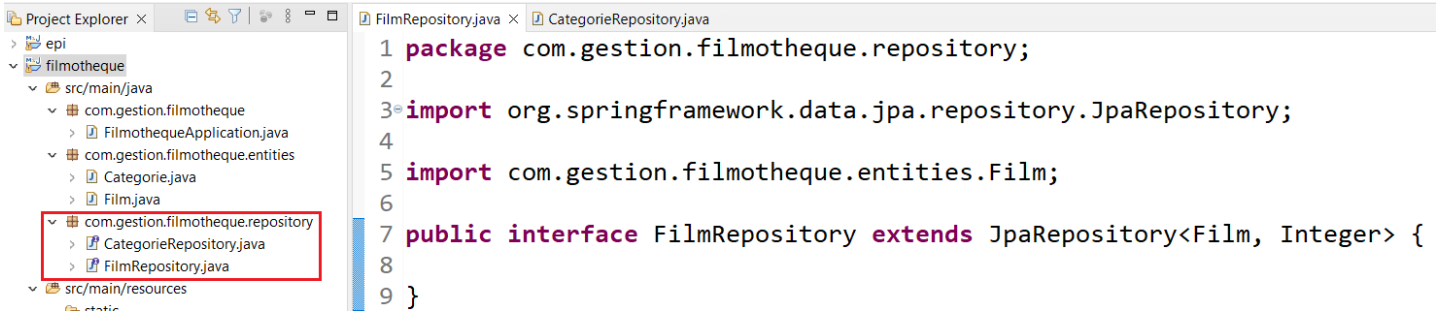


Pour exécuter votre application afin de générer les tables dans la BD `bd_film`, faites clic-droit sur le projet puis choisir Run as → Spring Boot App. Normalement la BD sera crée contenant les 2 tables liées `film` et `categorie` :



Etape 3 : Création des Repository

Créer l'interface `FilmRepository` Comme suit.



Cette interface hérite toutes les méthodes CRUD dont les signatures se trouvent dans l'interface `JpaRepository` et leurs implémentations se trouvent dans une classe de Hibernate. Définissez de même l'interface `CategorieRepository`.

Etape 4 : Création des composants de la couche service

Créer le package `com.gestion.filmotheque.service` qui contient pour chacune des entités `Film` et `Categorie` :

- une interface (nommée `IServiceFilm` pour l'entité `Film`) contenant la signature des méthodes CRUD sur cette entité
- une classe (nommée `ServiceFilm` pour l'entité `Film`) qui implémente cette interface et qui est annotée par `@Service` (pour injecter sa dépendance dans les classes contrôleurs de la couche Présentation). Cette classe doit injecter une dépendance sur l'interface Repository avec l'annotation `@Autowired`.

IServiceFilm.java

```java
package com.gestion.filmotheque.service;

import java.util.List;

import com.gestion.filmotheque.entities.Film;

public interface IServiceFilm {

    public Film createFilm(Film f);
    public Film findFilmById(int id);
    public List<Film> findAllFilms();
    public Film updateFilm(Film f);
    public void deleteFilm(int id);

}
```

erviceFilm.java *ServiceFilm.java

```java
package com.gestion.filmotheque.service;

import java.util.List;
@Service
public class ServiceFilm implements IServiceFilm{

    @Autowired
    FilmRepository filmRepository;

    @Override
    public Film createFilm(Film f) {
        return filmRepository.save(f);
    }

    @Override
    public Film findFilmById(int id) {
        return filmRepository.findById(id).get();
    }

    @Override
    public List<Film> findAllFilms() {
        return filmRepository.findAll();
    }

    @Override
    public Film updateFilm(Film f) {
        return filmRepository.save(f);
    }

    @Override
    public void deleteFilm(int id) {
        filmRepository.deleteById(id);
    }

}
```

On remarque que l'implémentation de `createFilm` et `updateFilm` est la même (appel de la méthode `save(f)`. Ceci s'explique par le fait que la méthode `save` ajoute un film f lorsque celui-ci ne possède pas de id, mais elle effectue la modification si le film f contient un id existant.

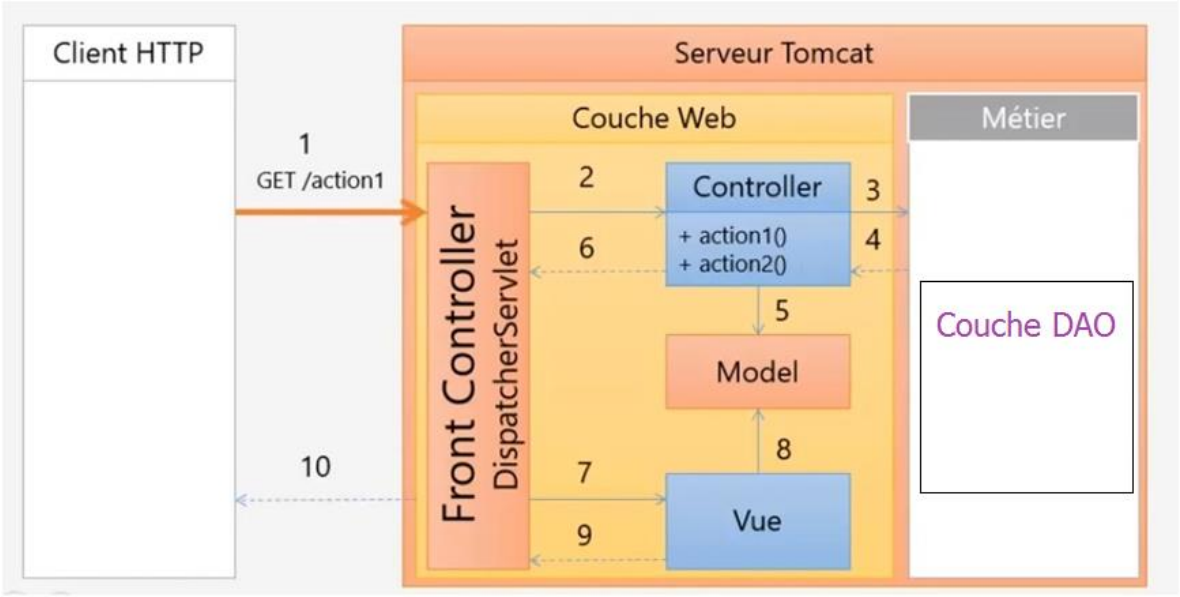
Définissez de la même façon l'interface `IServiceCategorie` et la classe `ServiceCategorie`.

Etape 5 : Création des composants de la couche présentation

Un contrôleur est une classe java annotée par `@Controller`. C'est un composant du modèle MVC. Il contient des méthodes (appelées actions) qui vont traiter les requêtes HTTP envoyés par le client. Chaque action doit retourner comme réponse à la requête :

- soit le nom d'une vue à afficher
- soit une redirection vers un URL d'une requête

Voici le chemin abordé d'une requête HTTP dans une application Spring Web MVC jusqu'à retourner la réponse :



1. Une requête HTTP (ex : get) est envoyée par un client Web
2. Le contrôleur frontal (Dispatcher Servlet) détermine quel contrôleur qui contient l'action à exécuter.
3. L'exécution de l'action entraine l'appel d'une méthode de la couche métier qui fait appel à la couche persistance (DAO) pour accéder aux données.
4. La couche métier renvoi sa réponse (les données) à l'action
5. Pour transmettre les données à la vue, l'action les ajoute comme attributs dans un objet Model.
6. L'action renvoi au contrôleur frontal le nom de la vue à afficher
7. Le contrôleur frontal ordonne la vue à s'exécuter
8. La vue récupère les données à afficher de l'objet Model et réalise l'affichage.
9. La vue renvoi l'affichage réalisé au contrôleur frontal
10. Ce dernier retourne cet affichage comme réponse à la requête HTTP.

Les annotations utilisées dans un contrôleur :

`@RequestMapping` : est l'une des annotations les plus couramment utilisées dans les applications Web Spring. Cette annotation 'mappe' les requêtes HTTP (get et post) aux actions des contrôleurs MVC. Elle possède son équivalant pour une requête spécifique :

- `@GetMapping` : pour une requête get. Exemple : `@GetMapping("all")`
- `@PostMapping` : pour une requête post. Exemple : `@PostMapping("save")`

`@PathVariable` : est une annotation qui indique que l'action possède un paramètre qui devrait être associé à une valeur dans l'URL traité. Exemple :

```
@GetMapping("delete/{id}")
public String delete(@PathVariable Long id) {
            ...
}
```

`@RequestParam` : utilisé pour lier un paramètre de la requête HTTP au paramètre de l'action. Exemple :

```
@PostMapping("/find")
public String find (@RequestParam String mot) {
            ...
}
```

Créer dans le package `com.gestion.filmotheque.controller` une classe `FilmController` qui sera le contrôleur web pour l'entité Film. Ce contrôleur traitera toutes les URLs qui commencent par `/film` et injectera une dépendance sur la classe `ServiceFilm` à travers l'interface implémentée par cette classe.

FilmController.java

```
1  package com.gestion.filmotheque.controller;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.stereotype.Controller;
5  import org.springframework.web.bind.annotation.RequestMapping;
6
7  import com.gestion.filmotheque.service.IServiceFilm;
8
9  @Controller
10 @RequestMapping("/film/")
11 public class FilmController {
12
13     @Autowired
14     IServiceFilm iServiceFilm;
15
16 }
```

Commençons par ajouter dans ce contrôleur une première action appelée `listeFilms` traitant une requête HTTP envoyée par la méthode `get` avec l'URL `/film/all` et qui va récupérer la liste des films puis les ajouter dans le model afin de les transmettre à une vue appelée `affiche` :

```java
@GetMapping("all")
public String listeFilms(Model model) {
    model.addAttribute("films", iServiceFilm.findAllFilms());
    return "affiche";
}
```

Le moteur de Template Thymeleaf est utilisé pour générer des vues HTML dans une application web MVC. Il se base sur un ensemble d'attributs ajoutés dans les balises HTML de la vue dont le moteur de Template va les interpréter et les traduire en code HTML.

Les vues Thymeleaf ont l'extension `.html` et doivent être placées dans le dossier `templates` sous `src/main/resources`.

Pour utiliser Thymeleaf dans une vue, on commence par déclarer l'utilisation de namespace Thymeleaf dans la balise HTML de la page :

```
<html xmlns:th="http://www.thymeleaf.org">
```

Ensuite, selon notre besoin on utilise les attributs spécifiques.

➢ **La vue affiche.html**

Dans `templates` créer un nouveau fichier html (sélectionner other puis chercher html) avec le nom `affiche.html` qui va afficher la liste des produits dans un tableau HTML :

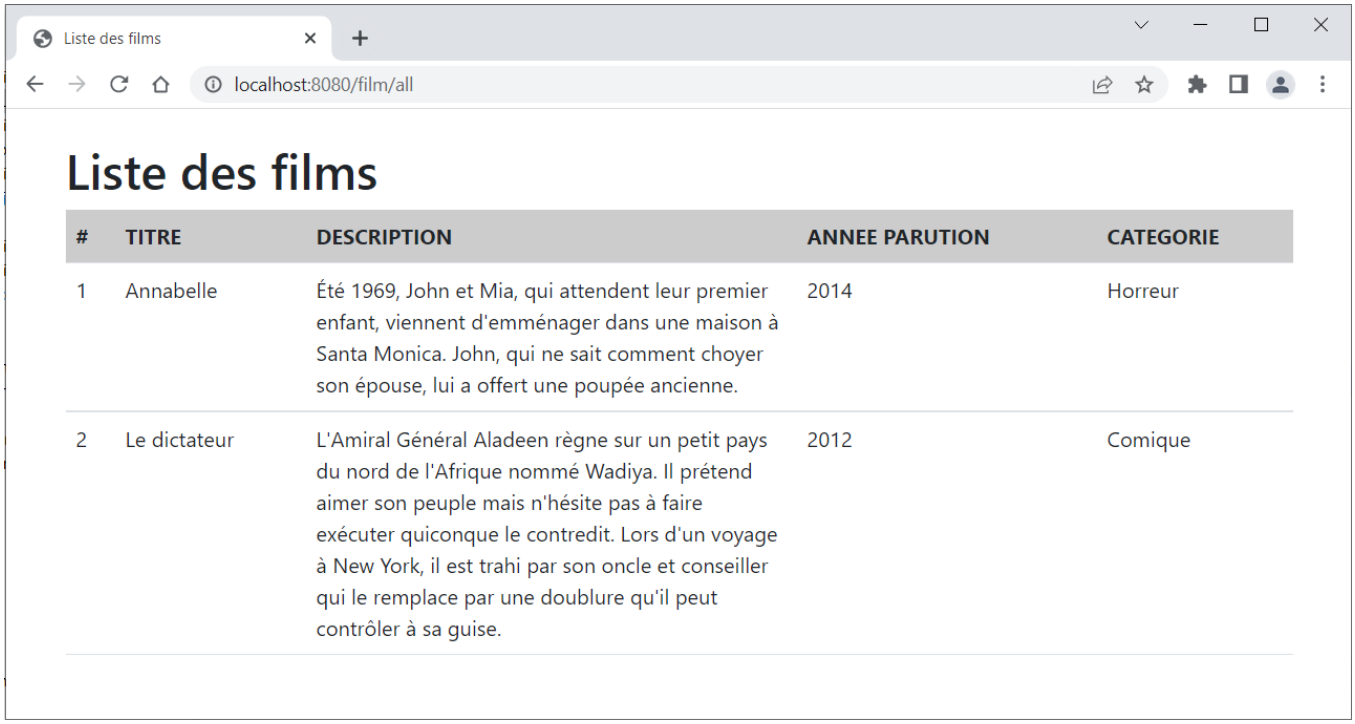
```
1  <!DOCTYPE html>
2  <html xmlns:th="http://www.thymeleaf.org">
3  <head>
4  <meta charset="utf-8">
5  <title>Liste des films</title>
6  <meta name="viewport" content="width=device-width, initial-scale=1">
7  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">
8  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"></script>
9  </head>
10 <body>
11 <div class="container mt-4">
12     <h1>Liste des films</h1>
13     <table class="table table-hover">
14         <tr style="background-color: #cccccc;">
15             <th>#</th>
16             <th>TITRE</th>
17             <th>DESCRIPTION</th>
18             <th>ANNEE PARUTION</th>
19             <th>CATEGORIE</th>
20         </tr>
21         <tr th:each="f : ${films}">
22             <td th:text="${f.id}"></td>
23             <td th:text="${f.titre}"></td>
24             <td th:text="${f.description}"></td>
25             <td th:text="${f.anneeparution}"></td>
26             <td th:text="${f.categorie.nom}"></td>
27         </tr>
28     </table>
29 </div>
30 </body>
31 </html>
```

Ce namespace indique que les attributs de Thymeleaf commencent par th

Parcours de la liste ${films} issue du model avec l'attribut Thymeleaf th:each

Affichage de l'id du film courant lu à partir de la liste avec l'attribut th:text

En exécutant l'URL `http://localhost:8080/film/all` dans le navigateur la vue suivante s'affiche :



Etape 6 : Ajout d'un nouveau Film

L'opération d'ajout se fait en 2 actions distinctes :

- Affichage du formulaire d'ajout vide à la suite à l'envoi d'une requête HTTP `/film/new` par GET.
- Remplissage et soumission du formulaire avec une requête HTTP `/film/add` envoyé par POST.

Le formulaire d'ajout d'un film doit comporter une liste déroulante qui contient les catégories (issues de la table categorie) afin de lier chaque film a sa catégorie.

Ajouter une action `afficheNewForm` dans le contrôleur `FilmController` (qui retourne la vue `ajout` qui va afficher ce formulaire d'ajout). Cette action doit envoyer à cette vue la liste des catégories :

```
@GetMapping("new")
public String afficheNewForm(Model model) {
    model.addAttribute("categories", iServiceCategorie.findAllCategories());
    return "ajout";
}
```

Le formulaire affiché dans la vue `ajout` doit contenir des champs dont les noms correspondent aux noms **des attributs** de l'entité `Film` afin de réaliser l'association (binding) entre les valeurs des champs et les attributs de l'entité :

```
<body>
<div class="container">
<h1>Saisir un nouveau Film</h1>
<form action="/film/add" method="post">
<p>
    <label for="titre" class="form-label">Titre du film : </label>
    <input type="text" name="titre" id="titre" class="form-control" required>
</p>
<p>
    <label for="description" class="form-label">Description : </label>
    <textarea rows="5" class="form-control" name="description"></textarea>
    </p>
<p>
    <label for="annee" class="form-label">Année parution : </label>
    <input type="number" min="1950" name="anneeparution" id="annee" class="form-control" required>
</p>
<p>
    <label for="categorie" class="form-label">Categorie : </label>
    <select name="categorie" class="form-select">
        <option th:each="cat : ${categories}" th:value="${cat.id}" th:text="${cat.nom}"></option>
    </select>
</p>
<p><input type="submit" class="btn btn-outline-primary" value="Ajouter"></p>
</form>
</div>
</body>
```

Cet attribut parcours la liste des categories

Cet attribut définit la valeur de chaque élement option généré

Cet attribut définit le texte de chaque élément option généré

Une fois le formulaire rempli et soumis, une requête HTTP sera envoyé avec la méthode POST ayant l'URL `/film/add`. L'action nommée `add` ci-dessous dans le contrôleur `FilmController` permettra de persister une instance de l'entité `Film` (c-a-d : l'ajouter dans la table produit) et fera une redirection vers l'URL qui affiche la liste de tous les produits :

```
@PostMapping("add")
public String add(Film f) {
    iServiceFilm.createFilm(f);
    return "redirect:/film/all";
}
```

Tester l'action de l'ajout, puis ajouter dans la vue affiche un lien hypertexte vers l'URL `/film/new`

Etape 7 : Suppression d'un Film

Pour implémenter la suppression d'un film, on va ajouter devant chaque film affiché dans la vue `affiche`, un lien hypertexte qui contient le id de ce produit à supprimer. Le clic sur ce lien va déclencher une requête HTTP (envoyée par GET) qui sera traitée dans le contrôleur par une action qui récupère ce id et effectue la suppression puis fait une redirection vers l'URL qui affiche la liste de tous les films. Voici l'aperçu de la vue `affiche` :

Voici le code a ajouter dans la vue `affiche` :

```
    <th>ACTION</th>
</tr>
<tr th:each="f : ${films}">
    <td th:text="${f.id}"></td>
    <td th:text="${f.titre}"></td>
    <td th:text="${f.description}"></td>
    <td th:text="${f.anneeparution}"></td>
    <td th:text="${f.categorie.nom}"></td>
    <td>
    <a th:href="@{/film/delete/{id} (id=${f.id})}" class="btn btn-outline-danger" >Supprimer</a>
    </td>
</tr>
```

Voici l'explication de cette ligne de code :

La balise de lien hypertexte `<a>` possède un attribut dynamique `th:href` contenant la valeur du paramètre `{id}` définit par Thymeleaf.

L'expression `@{…}` indique l' URL dynamique du lien

`{id}` indique que id est un paramètre dans cet URL (qui change d'un lien à un autre)

`(id=${f.id})` indique que La valeur de ce paramètre qui est le id du film courant.

Voici l'action à définir dans le contrôleur `FilmController` :

```
@GetMapping("delete/{id}")
public String delete(@PathVariable int id) {
    iServiceFilm.deleteFilm(id);
    return "redirect:/film/all";
}
```

L'annotation `@PathVariable` indique que l'argument id de l'action `delete` prendra sa valeur depuis la variable qui se trouve dans le path (dans l'URL).

Etant donné que la suppression est irréversible, on va ajouter un évènement JavaScript qui au clic sur le lien va afficher une boîte de confirmation (confirm) :

```
<a th:href="@{/film/delete/{id} (id=${f.id})}" class="btn btn-outline-danger"
onclick="return confirm('Voulez vous vraiment supprimer ce film?')">Supprimer</a>
```

Etape 8 : Modification un film

Tout comme l'ajout, la modification nécessite 2 étapes :

- Le choix du film à modifier en affichant un formulaire (similaire à celui de l'ajout), **rempli** par les informations du film à modifier avec un champ masqué contenant le id du film.
- la modification et la soumission du formulaire de modification.

Implémenter vous-même les 2 actions dans `FilmController` avec la vue `update` correspondante au formulaire de modification.

```
<option th:selected="${cat.id == film.categorie.id}">
```