# Projet ENSTA

## Contexte du projet

On souhaite mettre en place une application MyFilms qui va permettre de gérer des listes de films.
Ci-dessous les principales règles fonctionnelles auxquelles le projet doit répondre :

L'application MyFilms permet à des utilisateurs de se connecter.
Un utilisateur peut se connecter à l'application.
Un utilisateur peut voir la liste des films disponibles.
Un utilisateur peut cliquer sur un film pour voir son détail et son réalisateur, ainsi que si celui-ci est célèbre.
Un réalisateur devient célèbre s'il a réalisé au moins 3 films.
Un utilisateur peut donner une note à un film entre 0 et 20.
Un utilisateur peut visualiser la moyenne des notes d'un film.
Un utilisateur peut ajouter un film à la liste de ses films favoris.
Un utilisateur peut consulter la liste de ses films favoris.
Un utilisateur peut retirer un film de ses favoris.
Un administrateur peut créer un nouveau réalisateur.
Un administrateur peut créer un nouveau film.
Un film doit avoir un titre, une durée strictement positive et un réalisateur associé.
Un réalisateur doit avec un nom, un prénom et une date de naissance.

## Environnement technique du projet

L'objectif de ce projet est de construire une mini application de bout en bout qui permettra de gérer des films. Elle sera donc constituée de :

- Une couche de Persistance pour accéder à une base de données
- Une couche de Service pour effectuer les traitements métiers
- Une couche Controller pour exposer les services via des web services
- Une couche Présentation pour afficher et traiter les données

D'un point de vue technique, nous utiliserons :

- Le langage Java
- Une base de données H2 embarquée
- Le framework Spring
- Le framework Swagger
- Le framework Angular
- Le gestionnaire de build de projet Maven

L'archive fournie, `myfilmlist.zip`, contient un squelette du projet généré grâce à Spring Boot (https://start.spring.io/).
Dans le projet `myfilmlist`, on trouve principalement le contenu suivant :

| | |
|---|---|
| src/main/java | Contient les sources .java |
| com.ensta.myfilmlist.<br>MyfilmlistApplication.java | Classe principale de lancement du programme Java : lance le serveur d'application |
| com.ensta.myfilmlist.<br>MyfilmlistMain.java | Classe principale de lancement du programme Java : exécute un traitement et s'arrête |
| com.ensta.myfilmlist.dto | Package contenant les DTO en retour des services |
| com.ensta.myfilmlist.exception | Package contenant les exceptions |
| com.ensta.myfilmlist.form | Package contenant les données de formulaires |
| com.ensta.myfilmlist.mapper | Package contenant les mappers permettant de convertir les objets entre les couches |
| com.ensta.myfilmlist.model | Package contenant les objets du modèle |
| com.ensta.myfilmlist.persistence | Contient des utilitaires pour se connecter à la base de données |
| src/main/resources | Contient les ressources du projet (Configuration, …) |
| application.properties | Fichier de configuration de Spring Boot |
| data.sql | SQL pour construire et initialiser la base de données |
| src/test/java | Classes de test de l'application |
| src/test/resources | Contient les ressources pour les tests |
| mvnw et mvnw.cmd | Exécutable Maven pour builder le projet |
| pom.xml | Fichier de configuration du build Maven |

Le projet peut se lancer depuis un IDE ou en ligne de commandes :

```
mvnw spring-boot:run
```

- Base de données H2 embarquée

On utilise une base de données H2, elle a l'avantage d'être intégrée, donc il n'y a rien à installer dans le projet. On peut modifier l'initialisation de la base dans le fichier : `src/main/ressources/data.sql`.
Une fois le projet lancé, on pourra se connecter à la base via l'URL :

```
http://localhost:8080/h2-console/
```

```
Driver class : org.h2.Driver
JDBC URL : jdbc:h2:mem:film
User Name : sa
Password :
```

Ce n'est évidemment pas sécurisé, on peut le modifier avec le fichier `application.properties`, mais pour notre projet cela sera suffisant.
La configuration de cette base de données se fait avec le fichier `src/main/ressources/application.properties`.

- Swagger

Swagger est un framework permettant de tester les web services sans avoir réalisé le front. On peut y accéder via l'URL :

```
http://localhost:8080/swagger-ui/
```

Remarque : Les notions ci-dessus seront vues au fur et à mesure du projet. Dans un premier temps, nous allons démarrer le projet comme une application Java standard, via un IDE.

Précisions à propos du TP

Il n'y a pas qu'une seule façon de réaliser ce TP, mais dans un premier temps, on va fixer les noms des classes et les signatures des méthodes afin de faciliter les tests par la suite. Il vaut donc mieux suivre les conventions proposées tout au long des TP. Si aucune convention n'est proposée, alors l'implémentation sera à votre choix.

Dans une consigne, si on précise qu'un paramètre d'une méthode est non null, alors on considère qu'il n'est pas nécessaire de le tester dans la méthode en elle-même : c'est à celui qui appelle la méthode de s'assurer que le paramètre n'est pas null.

## Partie 1 : Mise en place des entités et services

Dans cette partie, nous allons créer les principales entités de l'application, et initialiser les services effectuant les traitements métiers.
Pour tester, nous allons utiliser la classe `MyfilmlistMain.java`, qui servira de point d'entrée au lancement de notre application.

### 1. Création des modèles

1.1. Dans le package `com.ensta.myfilmlist.model`, créer une classe `Realisateur.java`.

Un Réalisateur possède les attributs suivants :
- id (long)
- nom (String)
- prenom (String)
- dateNaissance (LocalDate)
- filmRealises (List<Film>)
- celebre (boolean)

1.2. Créer les attributs de la classe `Realisateur`, ainsi que les getter/setter associés.

1.3. Ajouter l'attribut realisateur (Realisateur) à la classe Film, ainsi que les getter/setter associés.

1.4. Dans le même package, créer une classe `Utilisateur.java`.

Un Utilisateur possède les attributs suivants :
- id (long)
- nom (String)
- prenom (String)

1.5. Créer les attributs de la classe `Utilisateur`, ainsi que les getter/setter associés.

## 2. Création du service

2.1. Créer un nouveau package `com.ensta.myfilmlist.service`.

2.2. A l'intérieur de ce package, créer une interface `MyFilmsService.java`.

2.3. Créer un nouveau package `com.ensta.myfilmlist.service.impl`.

2.4. A l'intérieur de ce package, créer une classe `MyFilmsServiceImpl.java` qui implémente l'interface `MyFilmsService.java`.

2.5. Dans cette classe, créer une nouvelle méthode `updateRealisateurCelebre` avec les caractéristiques suivantes :

- La méthode prend en paramètre un `Realisateur` non null contenant la liste non nulle des films qu'il a réalisés.
- La méthode met à jour le statut « celebre » de l'utilisateur en fonction de la liste des films qu'il a réalisés. Si un réalisateur a réalisé au moins 3 films, son statut « celebre » passe à « true ». Sinon, son statut « celebre » passe à « false ».
- La méthode renvoie le `Realisateur` mis à jour avec son statut « celebre ».
- La méthode renvoie une exception de type `ServiceException` en cas d'erreur.

2.6. Créer et utiliser une constante `NB_FILMS_MIN_REALISATEUR_CELEBRE` pour le chiffre « 3 ».
On pourra la laisser dans la classe `MyFilmsServiceImpl.java`.

2.7. Déclarer la méthode dans l'interface et faire la Javadoc.

2.8. Dans la classe `MyFilmsServiceImpl.java`, ajouter l'annotation `@Override` sur la méthode qui implémente celle de l'interface.

Rappel : cette annotation permet d'indiquer que la méthode implémente ou redéfinie le comportement de la méthode héritée d'un type parent. On l'ajoute donc sur chaque méthode qui possède cette caractéristique.

## 3. Tests du service

3.1. Se rendre dans la classe `MyfilmlistTest.java` et décommenter :
- La déclaration du service :

```
private MyFilmsService myFilmsService = new
MyFilmsServiceImpl();
```

- Le contenu de la méthode `updateRealisateurCelebreTest()`.

Effectuer les imports de packages nécessaires.

3.2. Lancer l'application Java, et vérifier dans la console le résultat :

```
James Cameron est-il celebre ? false
Peter Jackson est-il celebre ? true
```

## 4. Création d'autres méthodes du service

4.1. Dans `MyFilmsService.java`, créer une nouvelle méthode `calculerDureeTotale()` avec les caractéristiques suivantes :
- La méthode prend en paramètre une liste non nulle de films non nuls
- La méthode calcule la somme des durées de ces films
- La méthode renvoie la durée totale des films en paramètre

4.2. Se rendre dans la classe `MyfilmlistTest.java` et décommenter la méthode `calculerDureeTotaleTest()`, puis lancer le programme Java et vérifier le résultat

```
La duree totale de la trilogie "Le Seigneur des Anneaux" est de :
558 minutes
```

4.3. Dans l'interface `MyFilmsService.java`, définir une nouvelle méthode `calculerNoteMoyenne()` avec les caractéristiques suivantes :
- La méthode prend en paramètre un tableau de décimal (double) non nul
- La méthode calcule la moyenne de ces notes, et arrondit le résultat à 2 chiffres maximum après la virgule
- La méthode renvoie cette note moyenne, ou 0 par défaut

Implémenter cette méthode dans la classe `MyFilmsServiceImpl.java`.
Vous pourrez avoir besoin des utilitaires :
- `Math.round(a)` : renvoie l'entier le plus proche du nombre donné (arrondi)
- `Math.pow(a, b)` : renvoie le résultat du calcul : $a^b$.

4.4. Se rendre dans la classe `MyfilmlistTest.java` et décommenter la méthode `calculerNoteMoyenneTest()`, puis lancer le programme Java et vérifier le résultat

```
La note moyenne est : 15.17
```

4.5. Faire la Javadoc des méthodes créées dans le Service.

## 5. Utilisation des Streams

5.1. Modifier la méthode `calculerDureeTotale()` du service `MyFilmsServiceImpl.java` pour utiliser l'Api Streams.

Pour cela :

- Commencer par créer un `Stream<Film>` à partir de la liste de `Films`.
- Effectuer une `map` pour extraire des `Film` les `Integer` correspondants à leur durée (opération intermédiaire). On peut passer par un `IntStream`.
- Effectuer la somme des durées (opération terminale).

5.2. Modifier la méthode `calculerNoteMoyenne()` du service `MyFilmsServiceImpl.java` pour utiliser l'Api Streams.

Pour cela :

- Commencer par créer un `Stream` pour le tableau (`Arrays.stream(…)`).
- Calculer la moyenne des notes (`average`)
- Le résultat ressemble à un `Optional<Double>` (`OptionalDouble`), il faut donc récupérer le résultat au format `double`.

BONUS : Dans le service `MyFilmsService.java`, créer une méthode `updateRealisateurCelebres()` avec les caractéristiques suivantes :

- La méthode prend en paramètre une liste non nulle de `Realisateur` non nuls, contenant la liste non nulle des films qu'ils ont réalisés.
- La méthode met à jour le statut « celebre » de chaque `Realisateur` et renvoie une liste contenant uniquement les `Realisateur` célèbres.
- La méthode renvoie une exception de type `ServiceException` en cas d'erreur.

L'implémentation de cette méthode doit utiliser l'API Stream de Java.
Dans la classe `MyfilmlistTest.java`, écrire une méthode pour tester le fonctionnement de ce traitement.

## Partie 2 : Connexion à la base de données

Dans cette partie, on va initialiser la couche persistance de l'application afin de communiquer avec la base de données. On mettra en place les principales opérations CRUD dont on aura besoin par la suite pour les films et les réalisateurs. On mettra également en place la couche service associée.

Côté technique, dans un premier temps, on utilisera l'API standard JDBC de Java. Puis, dans un second temps, on utilisera Spring JDBC pour simplifier l'écriture des méthodes.

Pour tester les développements, la classe `ConnectionManager` se charge de créer une connexion vers une base de données H2 en mémoire, et d'initialiser les données depuis le fichier `src/main/resources/data.sql`.

## 1. Récupération des films

1.1. Créer un nouveau package `com.ensta.myfilmlist.dao`, contenant une interface `FilmDAO.java`, et un sous-package `com.ensta.myfilmlist.dao.impl`, contenant une implémentation de cette interface nommée `JdbcFilmDAO.java`.

1.2. Créer une nouvelle méthode `findAll()` dans l'interface et son implémentation avec les caractéristiques suivantes :

- La méthode ne prend pas de paramètre.
- La méthode renvoie la liste de tous les `Film` en base sous forme de `List` non nulle. Chaque film contiendra son identifiant, son titre et sa durée.
- La méthode ne lève pas de Checked Exception (si des Checked Exception sont levées, il faudra les traiter directement dans l'implémentation de la méthode).

On pourra récupérer la DataSource dans un attribut du DAO via :

```
private DataSource dataSource = ConnectionManager.getDataSource();
```

1.3. Dans `MyFilmsService.java`, créer une nouvelle méthode `findAllFilms()` qui va appeler la méthode du `FilmDAO` pour récupérer l'ensemble des films. Cette méthode a les caractéristiques suivantes :

- La méthode ne prend pas de paramètre
- La méthode récupère la liste de tous les films en base de données
- La méthode renvoie une liste non nulle des `FilmDTO` associés aux films
- La méthode peut lever une `ServiceException` en cas d'erreur

On devra par conséquent implémenter le contenu de cette méthode.
Pour cela :

- Utiliser la classe `FilmMapper` pour convertir un `Film` en `FilmDTO`.
- Dans le service, ajouter un attribut de type `FilmDAO` et l'instancier. En effet, on pourra toujours utiliser cette même instance dans toutes les méthodes du service.

1.4. Se rendre dans la classe `MyfilmlistTest.java` et décommenter le contenu de la méthode `findAllFilmsTest()`. Dans la classe `MyfilmlistMain.java`, décommenter la ligne pour initialiser la base de données :

```
// Demarrage de la base de donnees
ConnectionManager.initDatabase();
```

Lancer le programme et vérifier qu'il y a bien 4 films qui sont affichés.

## 2. Création des films

Pour pouvoir créer un film, il faut pouvoir lui affecter un réalisateur : cela fait partie des contraintes du projet. Par conséquent, on va avoir besoin de récupérer des réalisateurs en base de données. On va donc commencer par ajouter des méthodes CRUD pour les réalisateurs au niveau de la couche DAO.

A partir de maintenant, on va utiliser l'API `JdbcTemplate` fournie par Spring, qui va simplifier l'écriture des DAO. On peut récupérer cet objet dans les DAO via :

```
private JdbcTemplate jdbcTemplate =
ConnectionManager.getJdbcTemplate();
```

Les méthodes du `JdbcTemplate` ne lèvent pas de Checked Exception mais des `RuntimeException` qui héritent de `DataAccessException`. Il faudra penser à les catcher dans la couche service, et lever des `ServiceException` à la place. Remarque : on aurait aussi pu traiter ces exceptions dans la couche Controller.

2.1. Adapter la méthode `findAll()` du `FilmDAO` pour utiliser un JdbcTemplate à la place de la dataSource. On pourra utiliser la méthode `jdbcTemplate.query(String, RowMapper)`.
On pourra ensuite retirer l'attribut dataSource de la classe.

2.2. Comme pour `FilmDAO`, créer une interface `RealisateurDAO.java` et son implémentation `JdbcRealisateurDAO.java`.

2.3. Définir les 3 méthodes suivantes dans l'interface puis leurs implémentations :
- `List<Realisateur> findAll();`
- `Realisateur findByNomAndPrenom(String nom, String prenom);`
    - Cette méthode renvoie null si aucun utilisateur avec le nom et le prénom donné n'a été trouvé (Exception à catcher : `EmptyResultDataAccessException`).
    - Si plusieurs réalisateurs sont trouvés, on ne fera aucun traitement spécifique dans le DAO (une `RuntimeException` sera levée par le `jdbcTemplate`)
- `Optional<Realisateur> findById(long id);`
    - Cette méthode renvoie un `Optional` vide si aucun `Realisateur` avec le nom et le prénom donné n'a été trouvé.

Remarque : Pour convertir une `java.sql.Date` en `java.time.LocalDate`, utiliser la méthode `toLocalDate()`.

2.4. Dans `FilmDAO.java`, définir la méthode `save()` avec les caractéristiques suivantes :
- La méthode prend en paramètre un `Film`

- La méthode ajoute le film en base de données
- La méthode renvoie le `Film` créé avec son identifiant

Pour récupérer l'identifiant du film, on peut utiliser le code suivant :

```
KeyHolder keyHolder = new GeneratedKeyHolder();
PreparedStatementCreator creator = conn -> {
     PreparedStatement statement =
conn.prepareStatement(CREATE_FILM_QUERY,
Statement.RETURN_GENERATED_KEYS);
     // TODO : set des parametres
     return statement;
};
jdbcTemplate.update(creator, keyHolder);
film.setId(keyHolder.getKey().longValue());
```

On va maintenant ajouter le nécessaire au niveau de la couche service pour pouvoir créer un `Film`. Au niveau de la méthode du service qui effectuera ce traitement, on prendra en entrée un objet de type `FilmForm` qui contiendra les données pour créer le nouveau film. La méthode effectuera le traitement, et renverra un `FilmDTO` contenant les données du film créé (notamment son identifiant qui aura été généré au niveau de la base, et les données de son `Realisateur` associé dans un objet de type `RealisateurDTO`).
On va donc commencer par mettre à jour les DTO pour prendre en compte ce besoin.

2.5. Dans le package `com.ensta.myfilmlist.dto`, créer le `RealisateurDTO.java` en reprenant les mêmes attributs que la classe `Realisateur.java`.
Remarque : L'attribut `filmRealises` utilisera des `FilmDTO`.
On peut également écrire une méthode `toString()`.

2.6. Dans le package `com.ensta.myfilmlist.mapper`, créer la classe `RealisateurMapper.java`, en s'inspirant de ce qui est fait dans `FilmMapper.java`.
Remarque : On ne fera pas le mapping de l'attribut `filmRealises`. On pourra toujours utiliser le setter pour ajouter ce champ par la suite si besoin.

2.7. Dans `FilmDTO.java`, ajouter le champ `realisateur` de type `RealisateurDTO`. Puis, dans `FilmMapper.java`, ajouter le mapping de ce nouveau champ dans les méthodes `convertFilmToFilmDTO()` et `convertFilmDTOToFilm()`.
Attention : ce champ peut être null.

2.8. Dans `MyFilmsService.java`, créer une nouvelle méthode `createFilm()` avec les caractéristiques suivantes :

- La méthode prend en paramètre un `FilmForm`.
- La méthode effectue la création du film associé uniquement s'il est bien rattaché à un réalisateur existant. Sinon, il lève une `ServiceException`.
- La méthode renvoie le `FilmDTO` correspondant au film créé avec son id.

2.9. Dans `MyFilmsService.java`, créer les méthodes suivantes :

- `List<RealisateurDTO> findAllRealisateurs() throws ServiceException;`
  - Cette méthode renvoie la liste de tous les réalisateurs, ou une liste vide si aucun n'a été trouvé.
- `RealisateurDTO findRealisateurByNomAndPrenom(String nom, String prenom) throws ServiceException;`
  - Cette méthode renvoie un réalisateur avec le nom et le prénom donné, ou null si aucun réalisateur n'a été trouvé.

2.10. Se rendre dans la classe `MyfilmlistTest.java` et décommenter le contenu de la méthode `createFilmTest()`. Lancer le programme et vérifier qu'il y a bien 5 films qui sont affichés.

```
Le nouveau film 'Titanic' possede l'id : 5
Combien y a-t-il de films ? 5
Le realisateur du film : 'avatar' est : null
Le realisateur du film : 'La communauté de l'anneau' est : null
Le realisateur du film : 'Les deux tours' est : null
Le realisateur du film : 'Le retour du roi' est : null
Le realisateur du film : 'Titanic' est : null
```

On voit également que les réalisateurs ne sont pas affichés (`null`). C'est normal, car dans la méthode `findAll()` de `FilmDAO`, on ne remonte pas le réalisateur.

2.11. Effectuer la jointure nécessaire pour remonter le réalisateur pour chaque film au niveau du DAO. Relancer le programme, et vérifier que le réalisateur est bien renseigné.

```
Le realisateur du film : 'avatar' est : RealisateurDTO [id=1,
nom=Cameron, prenom=James, dateNaissance=1954-08-16,
celebre=false]
Le realisateur du film : 'La communauté de l'anneau' est :
RealisateurDTO [id=2, nom=Jackson, prenom=Peter,
dateNaissance=1961-10-31, celebre=true]
Le realisateur du film : 'Les deux tours' est : RealisateurDTO
[id=2, nom=Jackson, prenom=Peter, dateNaissance=1961-10-31,
celebre=true]
```

```
Le realisateur du film : 'Le retour du roi' est : RealisateurDTO
[id=2, nom=Jackson, prenom=Peter, dateNaissance=1961-10-31,
celebre=true]
Le realisateur du film : 'Titanic' est : RealisateurDTO [id=1,
nom=Cameron, prenom=James, dateNaissance=1954-08-16,
celebre=false]
```

## 3. Autres méthodes CRUD des films

On va compléter les DAO avec d'autres méthodes du CRUD qui serviront par la suite dans l'application, notamment :

- Récupérer un film par son identifiant
- Supprimer un film
- Renvoyer la liste des films réalisés par un réalisateur

3.1. Ajouter les méthodes suivantes dans `FilmDAO.java` :

- `Optional<Film> findById(long id);`
  - Cette méthode renvoie un Optional contenant le film ayant l'id donné s'il a été trouvé
- `void delete(Film film);`
  - Cette méthode prend en paramètre un film non null
- `List<Film> findByRealisateurId(long realisateurId);`
  - Cette méthode renvoie la liste non nulle des films réalisés par un `Realisateur`.

3.2 Ajouter les méthodes suivantes dans `MyFilmsService.java` :

- `FilmDTO findFilmById(long id) throws ServiceException;`
  - Cette méthode renvoie null si le film n'a pas été trouvé.
- `void deleteFilm(long id) throws ServiceException;`

Remarque : pour le moment, nous n'avons pas besoin d'un service pour remonter les films réalisés par un réalisateur, mais il est possible de l'écrire. La méthode du DAO sera utilisée dans la partie suivante.

3.3. On peut tester le résultat avec les méthodes `findFilmByIdTest()` et `deleteFilmByIdTest()` de la classe `MyfilmlistTest.java`.

```
--------------------
Le film avec l'identifiant 1 est : FilmDTO [id=1, titre=avatar, duree=162]
--------------------
Le film avec l'identifiant 5 est : FilmDTO [id=5, titre=Titanic, duree=195]
Suppression du film avec l'identifiant 5...
Le film avec l'identifiant 5 est : null
```

## 4. Mise à jour du statut « célèbre » d'un réalisateur

Pour rappel, lorsqu'on crée ou qu'on supprime un nouveau film, cela affecte le nombre de films réalisés par son réalisateur. Ainsi, son statut « célèbre » peut changer. On va désormais implémenter cette règle.

4.1. Ajouter les méthodes suivantes dans `RealisateurDAO.java` :

- `Realisateur update(Realisateur realisateur);`
    - Cette méthode met à jour tous les champs du réalisateur, sauf son identifiant.
    - Cette méthode renvoie un réalisateur mis à jour.

4.2. Lorsqu'on crée ou qu'on supprime un film, il faut mettre à jour le statut « célèbre » d'un réalisateur. Dans le service `MyFilmsServiceImpl.java`, utiliser (et mettre à jour si besoin) la méthode `updateRealisateurCelebre()` pour mettre à jour le `Realisateur` avec son nouveau statut en base de données. On peut tester le résultat avec la méthode `updateRealisateurCelebre()` de la classe `MyfilmlistTest.java`.

```
James Cameron est-il celebre ? false
James Cameron a realise deux nouveaux films
James Cameron est-il celebre ? true
Ce n'est pas James Cameron qui a realise le Hobbit, suppression du
film !
James Cameron est-il celebre ? false
```

## BONUS : Création et récupération des réalisateurs

Dans le DAO des réalisateurs, ajouter la méthode `save` pour enregistrer un nouveau réalisateur, comme pour les films. Dans la couche service, créer les services associés aux méthodes de DAO des réalisateurs `save` et `findById`.

## Partie 3 : Mise en place de l'API REST

### 1. Injection de Dépendances

Les Services et DAO sont initialisés via l'instruction `new` : à chaque fois qu'on crée un nouveau service ou un nouveau contrôleur, il faudra recréer des instances de ces classes. Or, ces classes sont « sans état », elles ne font qu'exécuter des traitements. On pourrait donc avoir une unique instance qu'on utilise à chaque fois qu'on en a besoin. C'est ce qu'on appelle un « Singleton ».
Avec Spring, on peut facilement créer un singleton en utilisant l'Injection de Dépendances via l'utilisation d'un contexte Spring, et les annotations `@Component`, `@Service`, `@Repository`, `@Autowired`, …

1.1. Dans les implémentations des interfaces DAO, ajouter une annotation `@Repository` au-dessus des classes.

```
@Repository
public class JdbcFilmDAO implements FilmDAO {
     …
}
```

1.2. Dans l'implémentations de l'interface Service, ajouter une annotation `@Service` au-dessus de la classe. Remplacer l'instanciation des DAO par l'utilisation de l'annotation `@Autowired`.

```
@Autowired
private FilmDAO filmDAO;
```

Cette annotation indique à Spring qu'il va devoir lui-même trouver l'instance à partir du type de l'objet. Comme il n'y a qu'une seule implémentation du DAO pour le moment, Spring va automatiquement injecter celle-ci.

1.3. Dans la classe `MyfilmlistTests.java`, ajouter une annotation `@Component` au-dessus de la classe. Utiliser à nouveau `@Autowired` pour injecter le service.

1.4. Dans la classe `MyfilmlistMain.java`, supprimer l'initialisation de la variable `myFilmListTests` et décommenter les lignes Spring pour créer le contexte.

```
// MyfilmlistTests myFilmListTests = new MyfilmlistTests();

// Initialisation du Contexte Spring
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
```

```
context.register(MyfilmlistTests.class);
context.scan("com.ensta.myfilmlist.*");
context.refresh();
MyfilmlistTests myFilmListTests =
context.getBean(MyfilmlistTests.class);
```

Ce code initialise un contexte Spring qui va parcourir les annotations du package à scanner, et créer les Bean associés. Il va également réaliser les injections nécessaires à l'initialisation de ces Beans, c'est-à-dire là où il trouvera des annotations `@Autowired`. Enfin, il va fournir l'unique instance de la classe `MyfilmlistTests.java` pour initialiser la variable `myFilmListTests`.

1.5. Relancer le programme Java, et vérifier qu'il n'y a pas d'erreur et que les tests donnent toujours les bons résultats.

Remarque : On peut utiliser un try-with-resources pour initialiser le contexte, et le fermer à la fin des tests.

Désormais, on utilise bien des Singletons, c'est-à-dire des instances uniques pour les Services et les DAO.

## 2. API pour la récupération des films

Désormais, nous n'allons plus démarrer le programme via la méthode `main` de la classe `MyfilmlistMain.java`, mais via celle de la classe `MyfilmlistApplication.java`. Celle-ci utilise Spring Boot et, via la configuration contenue dans le fichier `application.properties`, va automatiquement :

- Créer le contexte Spring à partir des annotations
- Initialiser la DataSource pour se connecter à la base de données H2 embarquée
- Initialiser le contenu de la base de données en exécutant le fichier `data.sql`
- Initialiser un client H2 pour la base de données accessible à l'URL : `http://localhost:8080/h2-console`
- Démarrer un serveur Tomcat embarqué qui va écouter sur le port http 8080.

Remarque : Le projet peut également se lancer depuis la ligne de commandes avec Maven, mais on continuera d'utiliser l'IDE qui reste plus simple pour les développements :

```
mvnw spring-boot:run
```

2.1. Spring Boot est capable d'initialiser la DataSource et le JdbcTemplate pour se connecter à la base de données grâce à la configuration contenue dans le fichier `application.properties`. Il n'y a donc plus besoin d'utiliser le `ConnectionManager` dans les DAO : on peut directement injecter le bean Spring avec l'annotation @Autowired. Effectuer cette correction.

```
@Autowired
private JdbcTemplate jdbcTemplate;
```

Pour le moment, il n'y a aucune interface Web aux services, donc on ne peut accéder à aucune ressource du serveur. Dans cette partie, nous allons mettre en place la couche Controller de l'application, qui va fournir des Web Services REST permettant d'accéder aux services de l'application, et de répondre avec des données au format JSON.

2.2. Créer un nouveau package `com.ensta.myfilmlist.persistence.controller`, et une interface `FilmResource.java` contenant la déclaration de la méthode suivante :

- `ResponseEntity<List<FilmDTO>> getAllFilms() throws ControllerException`
  - Renvoie la liste non nulle de tous les films disponibles, ainsi que leur Realisateur associé. En cas d'erreur de traitement, on renvoie une `ControllerException`.

2.3. Créer un sous-package `com.ensta.myfilmlist.persistence.controller.impl`, et une classe `FilmResourceImpl.java` qui va implémenter l'interface `FilmResource`.

2.4. Annoter cette implémentation avec `@RestController` et `@RequestMapping("/film")` pour que Spring puisse créer un Bean associé à cette classe, et se préparer à faire le mapping des URL en "/film" vers ce Controller.

Remarque : on pourrait ajouter l'annotation `@RequestMapping` sur l'interface, mais on le fait plutôt sur l'implémentation, pour le cas où une nouvelle version de notre API verrait le jour, et nécessiterait une nouvelle implémentation de `RestController`, avec donc un mapping spécifique pour cette nouvelle version (ex : "`/v2/film`")

2.5. Injecter le Bean du service `MyFilmsService` en tant qu'attribut de `FilmResourceImpl.java`.

2.6. Ajouter l'annotation nécessaire à la méthode `FilmResource.getAllFilms()`, afin qu'elle réponde aux requêtes HTTP de type GET sur l'URI "`/film`".
Pour rappel, comme le contrôleur hérite d'une interface, les annotations nécessaires sur les méthodes sont à placer dans cette interface.

2.7. Ecrire l'implémentation de cette méthode. Pour cela, on va « wrapper » la réponse dans un objet `ResponseEntity` qui va représenter la réponse http à la requête. Avec cet objet, on va pouvoir indiquer le code retour http (ex 200, 404, …).
Ci-dessous un exemple d'utilisation du `ResponseEntity` :

```
// On renvoie le code 200 (HttpStatus.OK) avec le contenu du film
dans la réponse
FilmDTO unFilm = new FilmDTO();
unFilm.setTitre("Avatar")
ResponseEntity.status(HttpStatus.OK).body(unFilm)
```

Remarque : vous pouvez regarder le contenu de la classe `HttpStatus` de Spring pour voir les autres codes de retour possibles.

2.8. Démarrer l'application en utilisant le `main` de la classe `MyfilmlistApplication.java`.
Ouvrir un navigateur et saisir l'URL : http://localhost:8080/film.
La liste des films est renvoyée au format JSON :

[{"id":1,"titre":"avatar","duree":162,"realisateur":{"id":1,"nom":"Cameron","prenom":"James","dateNaissance":"1954-08-16","filmRealises":null,"celebre":false}},{"id":2,"titre":"La communauté de l'anneau","duree":178,"realisateur":{"id":2,"nom":"Jackson","prenom":"Peter","dateNaissance":"1961-10-31","filmRealises":null,"celebre":true}},{"id":3,"titre":"Les deux tours","duree":179,"realisateur":{"id":2,"nom":"Jackson","prenom":"Peter","dateNaissance":"1961-10-31","filmRealises":null,"celebre":true}},{"id":4,"titre":"Le retour du roi","duree":201,"realisateur":{"id":2,"nom":"Jackson","prenom":"Peter","dateNaissance":"1961-10-31","filmRealises":null,"celebre":true}}]

## 3. API pour récupérer un film par son identifiant

3.1. Modifier l'interface `FilmResource.java` pour ajouter la déclaration de la méthode suivante :

- `ResponseEntity<FilmDTO> getFilmById(long id) throws ControllerException`
    - Renvoie le film ayant l'id donné ou une erreur 404 si la ressource n'a pas été trouvée. En cas d'erreur de traitement, on renvoie une `ControllerException`.

Cette méthode devra répondre aux requêtes HTTP de type GET sur l'URI "`/film/<ID_DU_FILM>`", où `<ID_DU_FILM>` correspondra à l'identifiant d'un film.

3.2. Ecrire l'implémentation de cette méthode.

3.3. Dans un navigateur, saisir l'URL `http://localhost:8080/film/1` : le film avec l'identifiant 1 doit être renvoyé. Essayer d'utiliser un identifiant qui n'existe pas en base de données, et vérifier que le statut de la réponse est bien 404.


```
{"id":1,"titre":"avatar","duree":162,"realisateur":{"id":1,"nom":"
Cameron","prenom":"James","dateNaissance":"1954-08-
16","filmRealises":null,"celebre":false}}
```

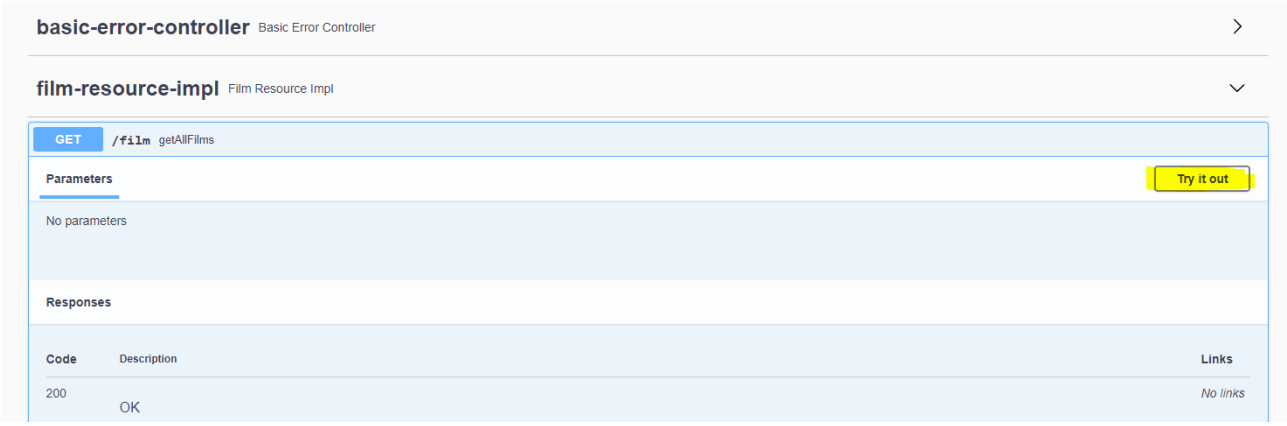
## 4. Mise en place de Swagger

Afin de documenter et pouvoir tester les web services créés, en attendant d'avoir une vraie interface client, on va utiliser le framework proposé par Swagger.

4.1. Se rendre sur l'interface de Swagger :

```
http://localhost:8080/swagger-ui/
```

4.2. On voit un service « film-resource-service ». Il contient l'ensemble des services exposés pour les films. On peut tester chaque service en cliquant sur le bouton « Try it out ». Si besoin, on peut saisir les paramètres requis.




Vérifier que les services créés renvoient bien le résultat attendu via cette interface.

4.3. On peut documenter les API Swaggers au niveau de l'interface du Controller, ce qui permet de modifier les libellés créés par défaut dans Swagger.

```
// L'API s'appelle « Film » et utilise le Tag « Film »
// Le tag « Film » contient la description de l'API
@Api(tags = "Film")
@Tag(name = "Film", description = "Opération sur les films")
public interface FilmResource {
      …
}
```

Puis au niveau des méthodes :

```
@ApiOperation(value = "Lister les films", notes = "Permet de
renvoyer la liste de tous les films.", produces =
MediaType.APPLICATION_JSON_VALUE)
@ApiResponses(value = {
          @ApiResponse(code = 200, message = "La liste des films a
été renvoyée correctement")
})
```

```
ResponseEntity<List<FilmDTO>> getAllFilms() throws
ControllerException;
```

Créer la documentation Swagger des services exposés, et tester le résultat sur l'interface. Penser à faire cette documentation pour les prochains services qui seront ajoutés.

Remarque : vous pourrez avoir besoin des imports suivants :

```
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses;
import io.swagger.v3.oas.annotations.tags.Tag;
import org.springframework.http.MediaType;
```

## 5. API pour créer un film

5.1. Modifier l'interface `FilmResource.java` pour ajouter la déclaration de la méthode suivante :

- `ResponseEntity<FilmDTO> createFilm(FilmForm filmForm) throws ControllerException`
    - Crée le film avec les paramètres donnés, et le renvoie avec son identifiant une fois créé. Le code http renvoyé est 201. En cas d'erreur de traitement, on renvoie une `ControllerException`.

Cette méthode sera annotée cette fois-ci fois avec `@PostMapping`. On va utiliser l'action HTTP POST car on envoie des données au serveur pour la création d'une nouvelle ressource. Les données seront contenues dans le corps de la requête HTTP, on va donc utiliser l'annotation `@RequestBody` pour récupérer le paramètre `FilmForm`.

Remarque : il s'agit de l'annotation `@RequestBody` de Spring : `org.springframework.web.bind.annotation.RequestBody`

5.2. Ecrire l'implémentation de cette méthode.

5.3. Tester la création d'un nouveau film en utilisant l'interface fournie par Swagger.

## 6. API pour supprimer un film

6.1. Ajouter le service `deleteFilm` qui permet de supprimer un film via son identifiant. La méthode retournera un `ResponseEntity<?>`, qui ne contiendra pas de body. Le service renverra le code 204 si la suppression s'est bien déroulée. En cas d'erreur de traitement, on renvoie une `ControllerException`.

Tester ce service avec l'interface Swagger.

## 7. Validation des données de la requête

7.1. A l'aide de l'interface Swagger, essayer de créer un nouveau film avec un id de réalisateur qui n'existe pas (ex : 10). Que se passe-t-il ?

Si vous n'avez pas prévu ce cas, vous remonterez probablement une erreur quelque part. Pour rappel, une contrainte du projet demande qu'un film ait toujours un réalisateur. De plus, un film doit toujours avoir un titre et une durée strictement positive. On va donc contrôler la validité de ces paramètres.

Par ailleurs, on a souvent considéré dans l'énoncé qu'un paramètre serait non null lors de l'écriture d'une méthode. Mais cette vérification doit bien être faite quelque part. C'est donc dans la couche Controller qu'on va vérifier la validité des données arrivant depuis une requête avant de les envoyer à la couche Service.

7.2. Dans la classe `FilmForm.java`, rajouter les règles de validation pour les paramètres suivants :

- Un titre est non null et non vide
- La durée est un entier toujours strictement positif
- L'id du réalisateur est toujours strictement positif

Pour cela, utiliser les annotations @NotBlank et @Min ou @Positive sur les attributs à valider.

7.3. Dans l'interface `FilmResource.java`, rajouter l'annotation `@Valid` devant le paramètre `FilmForm` de la méthode.

```
public ResponseEntity<FilmDTO> createFilm(@Valid FilmForm
filmForm) throws ControllerException;
```

Cette annotation indique à Spring d'effectuer la validation de l'objet à partir des annotations qui se trouvent dans sa classe.

7.5. Annoter la classe `FilmResourceImpl.java` avec `@Validated` pour déclencher la validation des champs par Spring.

7.6. A l'aide de l'interface Swagger, vérifier que la validation fonctionne en essayant de créer un film :

- Sans titre
- Avec une durée négative
- Avec un id de réalisateur égal à 0

Une exception doit être renvoyée par Spring dans chacun des 3 cas.
Vérifier qu'on peut toujours créer un film en spécifiant des valeurs correctes.

Si ces validations permettent déjà de vérifier un certain nombre de paramètres, elles ne suffisent pas dans ce cas pour valider l'existence d'un réalisateur. Cette vérification pourra être faite dans la couche service.

7.6 Dans le service, avant de créer un film, vérifier que le réalisateur existe bien. Dans le cas contraire, renvoyer une `ServiceException` avec un message explicite. A l'aide de l'interface Swagger, essayer de créer un film avec un réalisateur qui n'existe pas, et vérifier que l'exception renvoyée est bien celle attendue.

## 8. Gestion des Exceptions

En validant les paramètres, Spring lève désormais des exceptions (ex : `BindException`). Par ailleurs, on lève également des `ControllerException` en cas d'erreur remontée depuis la couche service. Mais on ne souhaite pas renvoyer ces exceptions au client. On va donc les traiter via des handlers pour renvoyer un message dans la réponse du service web.

8.1. Créer un nouveau package `com.ensta.myfilmlist.handler`, et créer la classe `ExceptionHandlers.java` à l'intérieur.

8.2. Pour que Spring détecte que cette classe est un Bean qui va gérer les erreurs, il faut l'annoter avec `@RestControllerAdvice`.

Pour chaque exception que l'on souhaite traiter, on va créer une méthode annotée avec `@ExceptionHandler` qui prend en paramètre la classe de l'exception. Dès qu'une exception du type requis est levée, cette méthode sera donc appelée. Généralement, la méthode retourne un objet qui va contenir les informations sur l'erreur au format JSON.
Cette méthode peut également prendre en paramètre l'exception, ainsi que la requête à l'origine de celle-ci (`WebRequest`).

```
@ExceptionHandler(ControllerException.class)
public ResponseEntity<String>
handleException(ControllerException exception, WebRequest
webRequest) {
    …
}
```

8.3. Créer une méthode qui va intercepter les erreurs de type `ControllerException`. Cette méthode va renvoyer un code erreur 400, et un message indiquant qu'il y a eu une erreur.

8.4. Tester le fonctionnement de ce handler en essayant de créer un film avec un réalisateur qui n'existe pas. Swagger doit afficher le code erreur 400, ainsi que le message saisi.

8.5. Créer un autre handler qui va gérer les erreurs de validation de paramètres remontées par Spring (`BindException`). On pourra tester par exemple en essayant de créer un film avec un titre vide.

## 9. Gestion des Transactions

Si on regarde la méthode `createFilm()` du Service, on voit qu'elle effectue deux actions principales :

- Créer un nouveau film
- Mettre à jour le statut « célèbre » du réalisateur associé

Mais que se passe-t-il si on crée le film, et qu'une erreur se produit par la suite, avant que le statut « célèbre » du réalisateur ait été mis à jour ? Le réalisateur aurait alors bien réalisé 3 films, mais son statut « célèbre » serait toujours `false`. Cela va en contradiction avec le besoin fonctionnel du projet.

Pour résoudre ce problème, ces deux opérations doivent donc se dérouler de façon `atomique`, c'est-à-dire comme s'il ne s'agissait que d'une seule opération. Autrement dit, soit les deux opérations sont validées (commit), soit elles sont toutes deux annulées (rollback). Pour cela, on va les encapsuler dans une transaction.

Spring fournit un mécanisme simple pour rendre une méthode transactionnelle : on l'annote et on la configure via l'annotation `@Transactional`.

9.1. Ajouter l'annotation `@Transactional` sur la méthode `createFilm()` du service.
On peut tester son fonctionnement en générant une exception entre les deux actions, et en vérifiant avec l'interface que la création du film a bien échoué si l'exception a été levée.

9.2. Il y a au moins une autre méthode `public` qui a besoin d'être transactionnelle. Ajouter l'annotation `@Transactional` sur les méthodes du service concernées.

## BONUS : API pour les réalisateurs

Créer l'API pour les réalisateurs avec les méthodes `getAllRealisateurs()`, `getRealisateurById()`, `createRealisateur()`. Effectuer les validations des champs nécessaires, vérifier si les méthodes ont besoin d'être transactionnelles, et effectuer la documentation Swagger des services web.

## Partie 4 : Mise en place des Tests Unitaires

On va désormais tester l'application avec de vrais tests unitaires. Pour cela, il y a 3 classes de tests à déposer dans le package `com.ensta.myfilmlist` du dossier src/tests/java : `MyFilmsDAOTests.java`, `MyFilmsServiceTests.java` et `MyFilmsControllerTests.java` pour tester respectivement les DAO, Service et Controller. Ils contiennent chacun un certain nombre de méthodes annotées `@Test`, qui indique qu'elles peuvent être exécutées en tant que test unitaire avec JUnit.

Les tests dans les DAO utiliseront la base de données embarquée H2. Les données seront réinitialisées entre chaque test.
Les tests dans le Service utiliseront des mocks sur les DAO qui retourneront les données attendues.
Les tests dans le Controller utiliseront un mock sur le Service qui retournera les données attendues.

L'objectif de cette partie est que tous les tests unitaires passent correctement.

Pour la suite du TP, il faudra penser à ajouter les tests unitaires associés aux futurs développements.

1. Lancer les tests unitaires associés à ces 3 classes. Pour chaque test qui ne réussira pas, il faudra soit :
   - Comprendre ce test et corriger le code en conséquent pour qu'il s'exécute correctement
   - Rédiger ou compléter le contenu de ce test dans le cas où il contient uniquement le code :

```
Assertions.fail("Rediger le contenu de ce test : <NIVEAU>");
```

Il y a 3 NIVEAU de difficulté pour les tests : Rapide, Moyen et Difficile. Il est conseillé de commencer par les tests « Rapide » de chaque classe, avant de faire les « Moyen » ou les « Difficile ».

On commencera par les tests sur les DAO, puis les tests sur les services, et on terminera par les tests sur les Controller.

BONUS : On peut vérifier la couverture de tests de l'application (via l'IDE, ou le plugin JaCoCo avec Maven). Voir s'il est nécessaire de rajouter certains tests unitaires par rapport aux développements effectués.

# Partie 5 : Mise en place de l'interface front

## 1. Installation de React

### a. Installation de Node

Pour Linux, il faut simplement mettre à jour le système et installer node via la commande apt install.

```
sudo apt update
sudo apt install nodejs
sudo apt install npm
```

Pour Windows, il suffit de télécharger node via l'installateur sur le site https://nodejs.org/en/download/. S'assurer que l'utilitaire npm est présent dans le package lors de la phase d'installation personnalisée.

## 2. Mise en place du projet

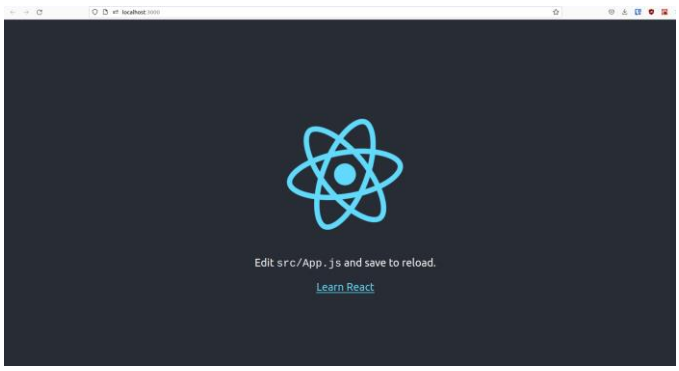
La mise en place d'un projet react est très rapide grâce à la commande très puissante `npx`. Dans un premier temps on va initialiser le projet, puis on y injectera les dépendances majeures dont on aura besoin par la suite. Il ne restera plus qu'à lancer le projet pour vérifier que tout fonctionne bien.

2.1. Création du projet

```
npx create-react-app my-film
```

Cette commande crée le projet à l'intérieur d'un répertoire my-film/ qu'elle prendra soin de créer s'il n'existe pas déjà.

2.7. Dans un navigateur, entrer l'adresse `localhost:3000`
Félicitations, le projet React a été initialisé avec succès !



2.2. Ajouter une bibliothèque de composant.

Pour ce projet on va utiliser MUI.

```
npm install @mui/material @emotion/react @emotion/styled
```

## 3. Header

L'application se présentera sous la forme suivante :

- Une page d'accueil contenant la liste des films du backend.
- Pour chaque film, un bouton qui redirige vers sa page de détails
- Une page qui comprendra les détails d'un film spécifique, ainsi que la liste des *autres* films de son réalisateur.

3.1. Le premier composant sera le header, qui se présentera sous la forme d'une barre horizontale classique. Dans src/ créer le fichier Header.js.
Dans ce fichier, créer votre premier function Component avec le contenu suivant:

```
export default function Header() {
  return (
     <h1>My Header</h1>
  )
}
```

3.2. Supprimer ce qui est retourner par `app.js`, et le remplacer par la simple balise qui suit :

```
<Header/>
```

Si on réexécute la commande `npm start`, on doit avoir un simple texte qui dit : "My header".

3.3. Dans le fichier `header.js`, remplacer le contenu par le code suivant :

```
<AppBar position="static">
 <Toolbar>
  <Typography variant="h6" component="div" sx={{ flexGrow: 1 }}>
     My Films
  </Typography>
 </Toolbar >
</AppBar>
```

Cela créera une barre de navigation qui, pour le moment, ne comporte qu'un titre.

## 4. Liste des films

4.1. Créer le composant `FilmList`

```
export default function FilmList() {
	const films= ["film1", "film2", "film3"];

	return films.map((film)=> {
		return <h1>{film}</h1>
	})
}
```

Dans App.js ajouter sous le Header

```
<FilmList />
```

Maintenant on peut voir l'affichage de film1 film2 et film3

4.5. A la place d'un simple affichage de `string`, on va plutôt créer des Cards pour afficher les films à l'intérieur.

Créer un composant FilmCard, qui prendra une prop : un objet film.

Voici un template pour votre Card:

```
<Card variant="outlined">
	<CardContent>
		<Typography variant="h5" gutterBottom>
			{props.film.titre}
		</Typography>
		<Typography variant="body1">
			{props.film.duree} minutes
		</Typography>
	</CardContent>
</Card>
```

4.6. Maintenant qu'on a le template, on va devoir remplacer la liste de String par les films. On va dans un premier temps mocker cette liste.

Créer un dossier mock et un fichier FilmMock.js

Dans ce fichier créer une liste de film dans un objet qui s'appellera mockFilms.

4.7. Dans FilmList on peut maintenant remplacer la liste de string par notre mock et remplacer la balise h1 par une FilmCard.

```
const films= mockFilms;

return films.map((film)=> {
        return <FilmCard key={film.id} film={film} />
})
```

On peut maintenant voir notre liste de films s'afficher.

## 5. Accéder au backend

On a désormais un rendu correct, mais on n'utilise pas de vraies données. Il est temps de faire appel au backend !

6.1. installer axios

```
npm install axios
```

6.2 Créer un dossier api sous src/ et un fichier FilmApi.js dedans.

```
import axios from 'axios';

const FILM_URI = 'http://localhost:8080/film'

export function getAllFilms(){
     return axios.get(FILM_URI);
}
```

La méthode getAllFilms va permettre d'interroger le backend et de récupérer la liste des films

6.3. Dans FilmList on va remplacer le contenu de notre liste par le résultat de la requête si celle-ci est un succès.

```
const [films, setFilms] = useState([]);

useEffect(() => {
  getAllFilms().then(reponse => {
     setFilms(reponse.data);
  }).catch(err => {
     console.log(err);
  })
}, []);
```

On observe que notre backend est bien interrogé et que la liste de film de la réponse est bien affichée.

## 6. Créer un nouveau film

La prochaine étape est de pouvoir créer un nouveau film.

6.1
Dans FilmApi créer les méthode d'appel api permettant de créer, éditer et supprimer un film. Elles devront s'appeler respectivement postFilm, putFilm, et deleteFilm.

6.2
Créer un nouveau composant que l'on appellera CreateFilmForm.

Ce composant sera composé de deux TextField pour le titre et la durée, d'un Select pour choisir le réalisateur, et d'un bouton. ( https://mui.com/material-ui/react-select/ )

Créer un fichier RealisateurApi dans le dossier api et créer la méthode réalisant l'appel getAllRealisateur.

De la même manière que pour récupérer la liste de film, récupérer la liste de réalisateurs.

Vous pourrez créer les valeurs dans votre Select à l'aide du fragment suivant.

```
{
realisateurs.map(realisateur  => {
return <MenuItem key={realisateur.id} value={realisateur.id}>
     {realisateur.prenom} {realisateur.nom}
</MenuItem>})
}
```

6.3
Créer une méthode qui sera appelée lors de l'appui sur le bouton pour créer un film et appeler la méthode postFilm.

6.4

Dans app.js ajouter votre composant FilmForm.

Vous êtes maintenant en mesure de créer des films, cependant ces derniers ne sont pas ajoutés à la liste. Pour le moment vous devez recharger votre page pour la mettre à jour.

6.5
Dans app.js enlever les balises FilmForm et FilmList.

Créer un composant FilmContainer qui renverra un FilmForm et une FilmList. Et ajouter la balise FilmContainer dans app.js

6.6
Modifier FilmList et FilmContainer pour que l'appel api soit fait dans FilmContainer. Pour cela vous devez faire en sorte que le composant FilmList utilise une props films qui contiendra la liste de films.

6.7
De la même manière, faite passer l'appel api pour la création du film dans le composant FilmContainer. Pour cela vous devrez faire en sorte que le composant FilmForm utilise une props onSubmit qui contiendra les actions à réaliser lors de l'appui sur le bouton.

## 7. Editer un film

7.1. Dans le composant `FilmCard`, ajouter deux boutons, qui correspondront à l'édition et la suppression d'un film.

```
<IconButton onClick={handleClickOnDeleteButton}>
  <DeleteIcon/>
</IconButton>
<IconButton onClick={handleClickOnEditButton}>
  <EditIcon/>
</IconButton>
```

et ajouter les méthodes suivantes

```
  const handleClickOnDeleteButton = () => {
     console.log("delete");
  }

  const handleClickOnEditButton = () => {
     console.log("edit");
  }
```

Vous pouvez observer que lorsque vous cliquez sur les boutons un message s'affiche dans la console de navigateur.

7.2. On va procéder à l'édition d'un film via une boîte de dialogue.

```
     <Dialog onClose={handleClose} open={open}>
     <DialogTitle>Editer un film</DialogTitle>
     <DialogContent>
          //FilmForm à ajouter
     </DialogContent>
     </Dialog>
```

Réaliser les imports nécessaires

7.3
Modifier le composant FilmForm pour pouvoir lui donner un film en props et que le formulaire soit prérempi.

7.4 Dans FilmList implémenté les fonctions pour soumettre le formulaire

## 8. CRUD - Delete et finition

La dernière étape des opérations CRUD est la suppression d'un film. On clique sur le bouton de suppression et le film est supprimé de la liste.

## Partie 6 : Utilisation d'un ORM

Jusqu'à maintenant, on utilisait JDBC pour se connecter à la base de données. C'est une solution performante, mais lourde à mettre en œuvre lorsque les entités métiers grossissent et deviennent plus nombreuses.
Une solution alternative consiste à utiliser un ORM (Object Relational Mapping), qui va se charger de faire automatiquement le mapping des données en base avec les entités métiers. Il faut pour cela définir correctement ce mapping dans l'entité via des annotations JPA.

### 1. Mise en place du mapping des entités

1.1. Mettre en place le mapping JPA des entités `Film` et `Realisateur`.

### 2. Mise en place des DAO JPA

2.1. Dans le package `com.ensta.myfilmlist.dao.impl`, créer une nouvelle classe `JpaFilmDAO` qui va implémenter l'interface `FilmDAO`. C'est également un Repository Spring, donc il faut lui ajouter l'annotation correspondante.

2.2. Injecter dans cette classe le Bean `EntityManager` via l'annotation `@PersistenceContext`.

```
@PersistenceContext
private EntityManager entityManager;
```

Remarque : ici, on n'utilise pas `@Autowired`, car il y a en réalité une instance d'entityManager par Thread, donc globalement une instance par requête cliente. Comme l'objet n'est pas thread safe, les requêtes ne peuvent pas partager la même instance. L'annotation `@PersistenceContext` permet de gérer ce besoin.

2.3. Implémenter les différentes méthodes de l'interface. On utilisera :
- entityManager.persist(…) : pour sauvegarder une nouvelle entité
- entityManager.merge(...) : pour enregistrer une entité déjà existante
- entityManager.find(…) : pour renvoyer une entité par son id
- entityManager.remove(…) : pour supprimer une entité
- entityManager.createQuery(…) : pour les autres besoins

Remarque : la méthode `delete()` prend un film en paramètre. Si celui-ci correspond à une entité à l'état « détaché » (c'est-à-dire pas dans le contexte JPA), il faut la rattacher avant de la supprimer. On peut tester ce cas avec `entityManager.contains(film)`. Si cette méthode renvoie `false`, on peut

rattacher l'entité avant de la supprimer via `entityManager.merge(film)`. Il faudra également avoir vérifié auparavant que l'entité existe bien en base de données, sinon ce n'est pas la peine de faire ce traitement. Ci-dessous le code associé :

```
if (entityManager.find(Film.class, film.getId()) != null) {
    if (!entityManager.contains(film)) {
        film = entityManager.merge(film);
    }
    …
}
```

2.4. Créer une nouvelle classe `JpaRealisateurDAO`, et effectuer les mêmes opérations.

2.5. Redémarrer le serveur. Vous devriez rencontrer une erreur de type `UnsatisfiedDependencyException / NoUniqueBeanDefinitionException`.

C'est parce qu'il y a désormais 2 Bean Spring de type `FilmDAO`, et Spring ne sait donc pas lequel il doit utiliser. Il y a plusieurs façons de résoudre ce problème. Dans le cas présent, on va simplement ajouter une annotation `@Primary` au-dessus des Bean JPA, pour dire à Spring que ce sont ces Beans-là qui doivent être utilisés en priorité.

2.6. Ajouter l'annotation `@Primary` sur les classes `JpaFilmDAO` et `JpaRealisateurDAO`. Redémarrer le serveur, et tester le bon fonctionnement de l'application.

Remarque : On remarque ici que nous n'avons rien changé d'autre que l'implémentation des DAO. On peut tout à faire revenir aux implémentations JDBC en déplaçant l'annotation `@Primary` sur les DAO JDBC, et tout fonctionnera correctement. C'est donc un des grands avantages de Spring et de l'utilisation des interfaces : on peut très facilement modifier les implémentations, et donc effectuer des traitements très différents, tout en conservant le même contrat d'interface.

## Partie 7 : Améliorer le projet

Les points suivants ne sont que des axes d'amélioration d'une application déjà fonctionnelle, la priorité reste d'avoir toute la partie précédente. **N'attaquez donc pas la suite s'il vous manque des choses.**

Vous êtes libre de choisir quelle(s) amélioration(s) vous souhaitez apporter, aussi bien celles ci-dessous que d'autres auxquelles vous pourriez penser.

### 1. Choisir un réalisateur

Dans la partie 5.9., on a dans notre formulaire d'édition de film un champ pour l'ID du réalisateur. Cependant ce n'est pas très intuitif.

Modifiez votre formulaire pour faire apparaître une liste déroulante (Voir exemple du cours) afin de sélectionner directement le réalisateur.

Il faudra donc faire en sorte de récupérer la liste des réalisateurs depuis le backend.

### 2. Ajouter un genre aux films

A tout moment, un projet peut être sujet aux modifications, et les bases de données ne sont pas épargnées.

Modifiez le fichier `data.sql` du backend et ajoutez une table `Genre`, qui possédera un id et un nom. Ajoutez une colonne `genre_id` à la table `Film` et ajoutez leur valeur dans les inserts.

Modifiez votre code pour intégrer correctement ce nouvel élément dans vos films. Il doit également apparaître dans le frontend.

### 3. Opérations CRUD sur les réalisateurs

Pour avoir un gestionnaire complet d'une base de données, il faut avoir la possibilité d'agir sur sa totalité.

Modifiez votre code pour donner la possibilité d'afficher, ajouter, modifier ou supprimer un réalisateur.

Remarque : Lors de la suppression d'un réalisateur, tous les films qu'il a réalisés doivent également être supprimés.

Questions ouvertes :

- deleteRealisateur  Suppression des films associés  Transaction
- On va mettre en place un filtre sur le nom du film, et sur le nom d'un réalisateur.
- On va mettre en place un order sur l'Id ( par defaut ), sur le nom d'un film, sur sa durée.
- Mettre en place des utilisateurs. On utilisera Spring Security. Un utilisateur peut se connecter.
- Un utilisateur a une liste de films qu'il a vue. Il peut ajouter ou enlever un films de sa liste.
- Un utilisateur peut donner une note sur 20 ( note étant un entier ) pour les films.
- On affiche les notes des films, note étant calculé en faisant la moyenne des notes des utilisateur.

Question Bonus 1 :
On va faire une pagination pour les clients et les réalisateurs. C'est à dire qu'on va créer un objet Page<T> qui contient les attribues

```
private int number;
private int size;
private long total;
private List<T> data;
```

Les webservices renveront donc cet objet plutôt qu'une liste.

Question Bonus 2 :
Rajoutez une colonne genre dans la table des films. Créez une table Genre qui possède un nom. Initialisez avec des genres comme : action, thriller, policier, horreur, comédie, SF, drame, biopic. Les genres étant supposé être fixe (si modification cela sera fait par un admin ), on n'a pas besoin de faire de create et de delete.
Faire les fonction getAll, getById pour les genres. Mettre à jours les fonctions des films pour faire apparaître le genre.

Question Bonus 3 :
On va mettre en place Sonar, un outil permettant de vérifier le clean de votre code. Il vous chiffre notamment votre dette technique.