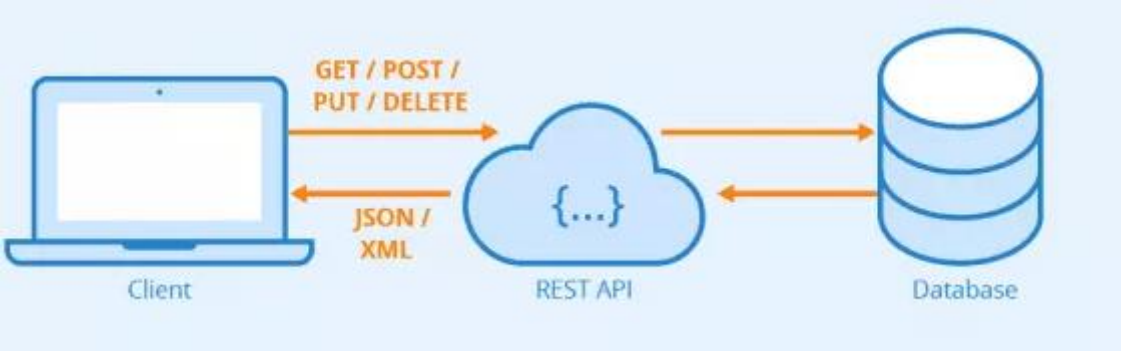
TP8 Les API REST et documentation avec Swagger

L'objectif de ce TP est de mettre en place un ensemble des API REST représentant les opérations CRUD sur les entités `Film`, `Categorie` et `Acteur`.
Un API est un ensemble de règles permettant à différents programmes de communiquer entre eux.
En particulier, un API REST (ou Restful) est une interface de programmation d'application qui fait appel à des requêtes HTTP pour obtenir (GET), placer (PUT), publier (POST) et supprimer (DELETE) des données à travers un serveur. Les API REST rend possible la communication entre Back-end implémenté sur un serveur web avec des applications Front-end web ou mobile. Cette communication à pour but de permettre l'accès aux données des BD au format JSON.



Les opérations CRUD de l'API Rest correspondent aux verbes HTTP et aux opérations SQL, sont indiqués dans le tableau ci-dessous :

Opération	**SQL**	**Méthode HTTP**
Create	INSERT	POST
Read	SELECT	GET
Update	UPDATE	PUT
Delete	DELETE	DELETE

Chaque action dans un contrôleur REST va retourner une donnée de Type Entité ou liste d'Entités qui sera convertie par Spring Boot en format JSON.

1. Création du contrôleur REST pour l'entité Film

Dans le package `controller` créer une nouvelle classe `RestFilmController`. Annoter cette classe par `@RestController` afin que les actions de ce contrôleur deviennent des API REST exposés aux clients Web à travers des requêtes HTTP.
Ajouter une annotation `@RequestMapping` qui représente l'URL commun `/api/films` à toutes les requêtes HTTP qui seront traités par les actions de ce contrôleur.

```java
@RestController
@RequestMapping("/api/films")
public class RestFilmController {

    @Autowired
    IServiceFilm iServiceFilm;

}
```

Cela veut dire que toutes les API Rest concernant les films auront un URL qui commence par : `http://localhost:8080/api/films`

Selon la méthode d'envoi de la requête et selon la suite de celle-ci, chaque action renvoi les données JSON appropriées.

Par exemple, pour renvoyer la liste des films il faut consommer l'API avec l'URL suivant en utilisant la méthode GET : `http://localhost:8080/api/films`

```java
@GetMapping("")
public List<Film> getall(){
    return iServiceFilm.findAllFilms();
}
```

Avant de tester cette API et pour éviter un accès en boucle infinie entre la liste des films d'une catégorie et un film, nous devons annoter la liste des films dans l'entité `Categorie` et l'entité `Acteur` par l'annotation `@JsonIgnore`. Cette annotation permet d'ignorer la liste des films dans ces 2 entités et par suite éviter la boucle des références.

Tester cette API dans un client comme POSTMAN.

➢ Définissez dans le même contrôleur une action `getparid` qui renvoi une entité `Film` selon son id (par ex : 2) récupéré comme variable de l'URL (path) de la requête suivante envoyée par la méthode GET :

`http://localhost:8080/api/films/2`

➢ Définissez une autre action `add` qui permet d'ajouter une nouvelle instance de l'entité `Film` à partir des valeurs récupérées du corps de la requête HTTP envoyée par la méthode POST :

`http://localhost:8080/api/films/add`

Dans POSTMAN, définissez dans Body →raw (type JSON) un objet JSON contenant les données du film à ajouter (sans id). La réponse à cette requête sera l'instance du Film ajoutée avec l'id attribué.

➢ Définissez une action `delete` qui reçoit un id d'une entité film comme variable dans l'URL dans une requête HTTP envoyée par la méthode DELETE et qui supprime l'entité et retourne un message de confirmation.

`http://localhost:8080/api/films/2`

➢ Définissez une action update qui permet de modifier une instance existante d'un Film selon les valeurs récupérées du corps de la requête HTTP envoyée par la méthode PUT :

`http://localhost:8080/api/films/update`

Dans POSTMAN, définissez le Body →raw (type JSON) un objet JSON contenant les données du film à modifier (avec id existant). La réponse à cette requête sera l'instance du Film modifié.

Créer de la même manière deux autres contrôleurs `RestCategorieController` et `RestActeurController`.

Traitement des exceptions personnalisées

Lors de la recherche, la suppression ou la modification d'un film avec id non existant, Spring Boot génère une exception par défaut (longue et désagréable pour l'utilisateur). On souhaite personnaliser cette exception avec un message adéquat.

Pour cela il faut définir certaines classes d'exception utilisant les annotations suivantes :

`@ControllerAdvice` est l'annotation pour manipuler généralement les exceptions

`@ExceptionHandler` est l'annotation qui traite et retourne le message personnaliser pour une exception.

Dans un package exception commençons par créer la classe qui va traiter généralement les exceptions

```
package com.gestion.filmotheque.exception;

import org.springframework.web.bind.annotation.ControllerAdvice;

@ControllerAdvice
public class FilmExceptionController {

}
```

Dans le même package créer une deuxième classe qui se déclare pour traiter les exceptions de no existence des films et qui hérite de la classe `RuntimeException` :

```
public class FilmNotfoundException extends RuntimeException {

    private static final long serialVersionUID = 1L;
}
```

Maintenant dans la première classe, on ajoute une exception qui traite les exceptions de la deuxième classe en indiquant le message à retourner en cas d'exception :

```
@ControllerAdvice
public class FilmExceptionController {

    @ExceptionHandler(value = FilmNotfoundException.class)

    public ResponseEntity<Object> exception(FilmNotfoundException exception) {
        return new ResponseEntity<>("Film not found", HttpStatus.NOT_FOUND);
    }

}
```

Dans les actions de recherche, suppression et modification de `RestFilmController`, il suffit d'ajouter une condition sur la non-existence d'un film par son id pour déclencher l'exception. Par exemple action de recherche par id devient :

```
@GetMapping("/{id}")
public Film getParId(@PathVariable int id){
    if(!iServiceFilm.filmExist(id))throw new FilmNotfoundException();
    return iServiceFilm.findFilmById(id);
}
```

Sans oublier d'implémenter dans la classe Service une méthode `filmExist(id)` qui retourne `false` s'il n'existe pas un film avec le id en paramètre, `true` sinon :

```
@Override
public Boolean filmExist(int id) {
    return filmRepository.existsById(id);
}
```

Modifier les actions `delete` et `update` dans le même contrôleur, Puis définissez et appliquez des classes d'exception personnalisées pour les entités `Categorie` et `Acteur`.

Documentation des API REST avec Swagger

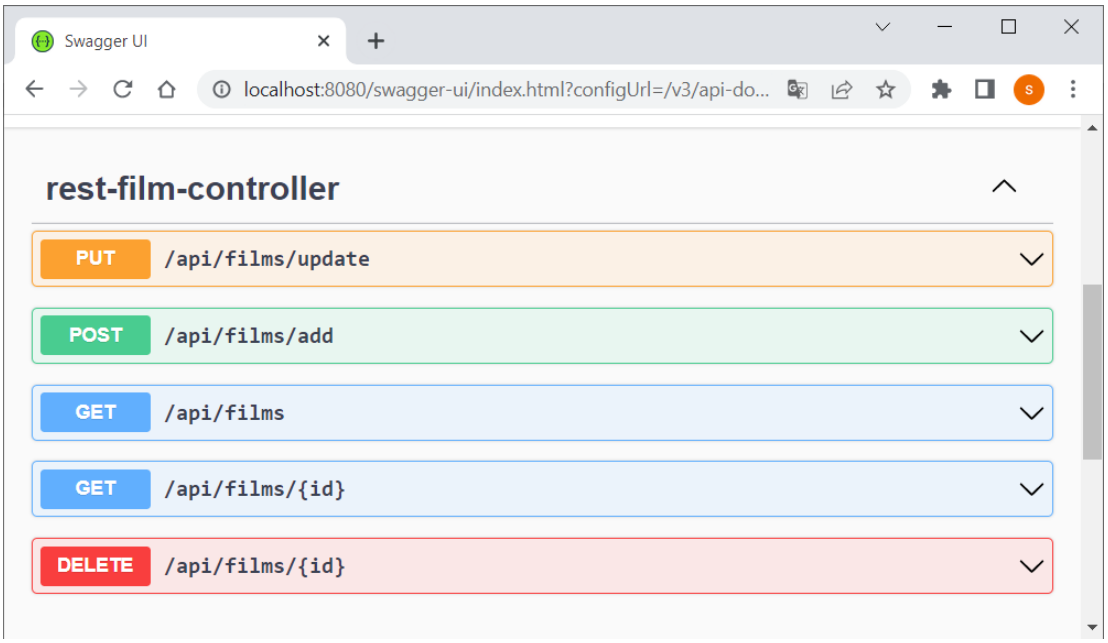
Une fois les contrôleurs REST sont définis avec leurs actions, il est fortement recommandé de documenter les API afin de pouvoir les invoquer **dans un navigateur** à travers les requêtes http (au lieu d'utiliser un client comme POSTMAN). Pour cela il suffit d'ajouter la dépendance `Swagger` qui va générer automatiquement cette documentation.

Arrêter le serveur, ajouter au fichier `pom.xml` la dépendance suivante :

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.5.12</version>
</dependency>
```

Enregistrer le fichier `pom.xml` pour que Maven télécharge la nouvelle dépendance, puis lancer le serveur de nouveau puis tapez l'adresse suivante dans le navigateur :

`http://localhost:8080/swagger-ui.html` la page suivante apparait :



A partir de cette page, on peut savoir le détail de chaque service REST (URL, paramètres, valeur de retour, …) avec la structure de chaque Entité.

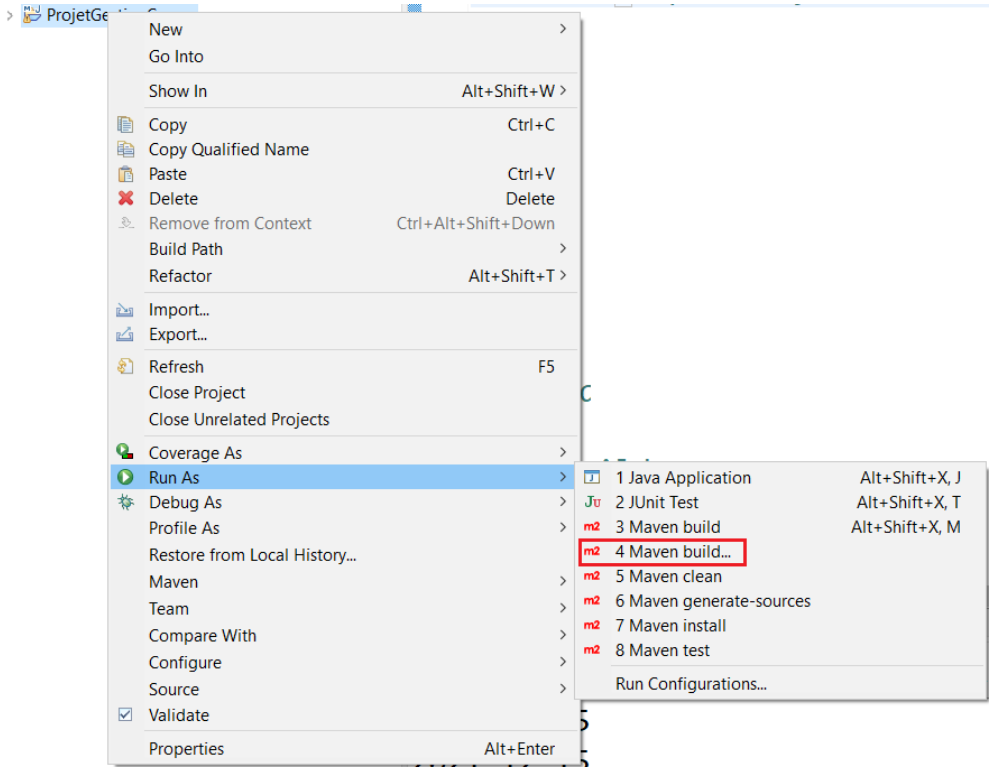
Testez les différents web services à partir de cette interface.

Exporter une application Spring Boot en fichier .jar

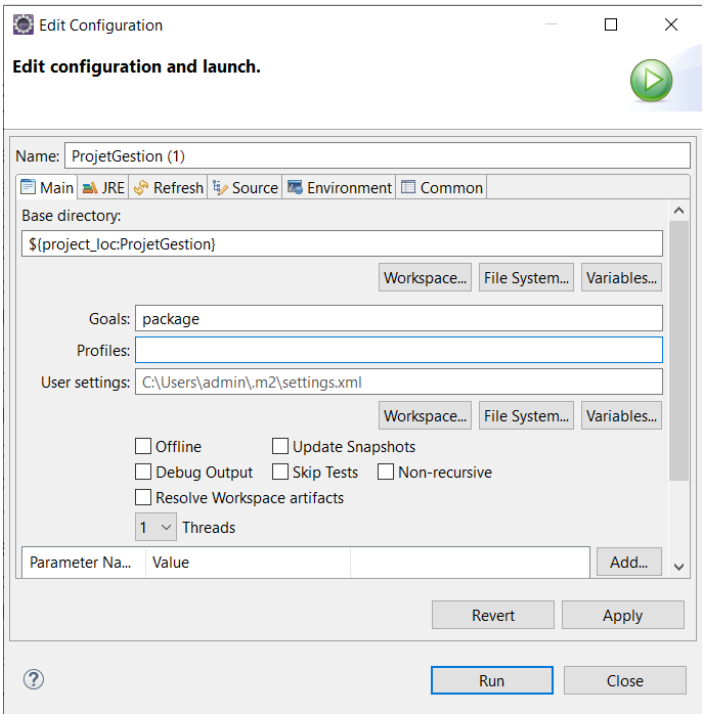
Ajouter dans le fichier pom.xml la propriété que le packaging du projet Spring Boot doit être en fichier jar :

```
<project ...>

    ...

    <packaging>jar</packaging>

    ...

</project>
```

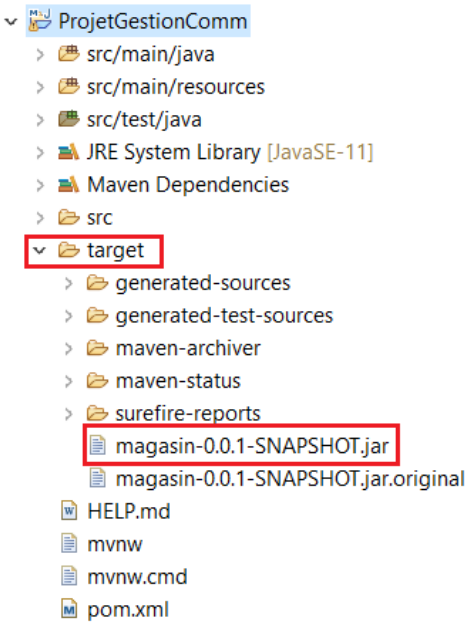
Ensuite faites un clic droit sur le projet, puis sélectionnez Exécuter en tant que > Maven build…



Ensuite spécifier la valeur `package` dans le champs `Goals` :



Maven va se charger de générer le fichier jar correspondant au délivrable de l'application qui sera placé dans le dossier target du projet :



Maintenant pour déployer notre application (en dehors de l'Eclipse) il suffit de placer ce fichier jar n'importe où sur votre machine et pour l'exécuter il suffit d'aller au répertoire contenant le jar et avec l'invite de commande tapez : **(il faut arrêter l'exécution de l'application sous Eclipse avant)** :

```
> java -jar nomfichierjar.jar
```

Ensuite tester votre application web ou API REST comme d'habitude.