# Spring 3.x

*Sujet de Travaux Pratiques*

# Table des matières

# 1. Introduction

Le gérant d'un magasin de vente de DVD désire réaliser une application permettant de gérer à distance (Extranet) son stock de DVD.

Cette application dispose d'une console Web d'administration qui permet à un administrateur de gérer :

- le stock de films
- les informations concernant chaque film proposé

Un film étant représenté par un JavaBean ayant pour propriétés :

- Un titre
- Un nombre d'exemplaires de DVD disponibles
- Un genre
- Un acteur principal du film
- Éventuellement les autres acteurs du film

# 2. Architectures d'entreprise

## 2.1. TP n°1 (Assisté)

Avant de réaliser concrètement notre application d'entreprise, nous allons expérimenter l'architecture cible autour de l'entité métier « Film ».

Nous créerons cette classe avec Eclipse Luna ou Netbeans 8 dans une Projet Java nommé `core`. Ceci va nécessiter Java 8.

Nous compilerons cependant nos projets au niveau 1.6 seulement.

Les projets seront proposés au format Maven mais vous trouverez également des librairies compatibles dans les ressources fournies.

L'entité métier « Film » sera matérialisée par une classe java `[com.mycompany].dvdstore.core.entity.Film` au format JavaBean ayant pour l'instant des propriétés simples :

- `titre (String)`
- `genre (String)`
- `nbExemplaires (int)`

Nous allons ensuite créer dans ce même projet une ébauche de Service Métier par la classe `com.mycompany.dvdstore.core.service.FilmService` disposant d'une méthode :

- `public void registerFilm(Film f)`

La méthode consistera en un simple affichage des propriétés du film dans la console (`System.out`)

Nous allons enfin créer un client par le biais d'une classe `Runner` exécutable (donc avec un `main`) qui permettra d'exécuter le service de la manière suivante :

1. Instancier un `Film` en lui affectant titre, genre et nombre d'exemplaires
2. Instancier `FilmService`
3. Exécuter la méthode `registerFilm` en lui donnant en paramètre le film précédemment créé

## 2.2. TP n°2

Ajoutez un composant de type `com.mycompany.dvdstore.core.controller.DefaultFilmController` avec une méthode `registerFilmFromConsoleInput` qui permette par le biais de l'entrée console (`System.in`) de demander à l'utilisateur le titre et le nombre d'exemplaires du film à afficher.

```
Scanner sc=new Scanner(System.in);

System.out.println("Quel est le titre du film ?");

String titre=sc.nextLine();
```

L'affichage en lui-même sera toujours produit grâce au service du TP n°1.

## 2.3. TP n°3

Créez un composant de type : `com.mycompany.dvdstore.core.repository.FileFilmDao` qui

dispose d'une méthode `save` qui soit en charge d'écrire les informations concernant le nouveau film dans un fichier (une ligne par Film, propriétés séparées par des « ; »).

```
FileWriter writer;
try{
	writer=new FileWriter("C:\\temp\\films.txt",true);
	writer.write("blabla\n");
	writer.close();
}catch (IOException e){
	e.printStackTrace();
}
```

Modifiez le composant de service afin d'exploiter ce nouveau DAO.

En résumé :

- D'un point de vue architectural, cette application est composé d'un seul tier physique, votre localhost.
- Selon la perspective logique, vous disposez
  - d'un tier client (la classe `Runner`)
  - d'un tier donnée (le fichier texte)
  - d'un tier applicatif (le reste)
- D'un point de vue développeur, votre le tier applicatif est structuré en 3 couches :
  - La couche controller
  - La couche service
  - La couche repository

# 3. Chapitre 2 : Conteneur léger et IOC

## 3.1. TP n°1

Nous allons faire fonctionner `Runner` (création de Film) par l'intermédiaire de Spring.

Les classes DAO, Service et Contrôleur ne seront pas instanciées manuellement. Ceci sera laissé à la charge du conteneur léger, vous aurez cependant un fichier XML de configuration.

L'application utilisera dans un premier temps le DAO de type Fichier pour obtenir le résultat (rappel : ce DAO ne tient pas compte des Acteurs), dans un second temps nous l'interchangerons avec un DAO qui exploite une base de données.

Ajoutez une interface au service et au DAO pour définir le contrat exposé au couches supérieures.

Il ne sera cependant pas nécessaire dans cette formation d'ajouter une interface pour le contrôleur.

Vous pourriez nommer ce fichier comme bon vous semble mais prenez la bonne habitude dès maintenant de le nommer `applicationContext.xml`.

- Utilisez le modèle de fichier de configuration vierge ci-dessous, complétez-le et placez-le dans les sources de l'application :

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
...
</beans>
```

- Vous utiliserez un `BeanFactory` pour récupérer le contrôleur.

- Compilez et exécutez à l'aide des librairies fournies :

- org.springframework.core*.jar
- org.springframework.beans*.jar
- commons-logging*.jar

Ou les dépendances Maven suivantes :

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-core</artifactId>
   <version>3.2.6.RELEASE</version>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
```

```
    <artifactId>spring-beans</artifactId>
    <version>3.2.6.RELEASE</version>
</dependency>
```

## 3.2. TP n°2

- Reprenez le code du TP écrit précédemment en faisant appel à un `ClassPathXmlApplicationContext` en lieu et place du `BeanFactory`.

- Conservez le fichier de configuration précédent

- Compilez et exécutez à l'aide des librairies supplémentaires suivantes :

- org.springframework.context*.jar
- org.springframework.asm*.jar
- org.springframework.expression*.jar

Ou les dépendances Maven suivantes :

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.2.6.RELEASE</version> < !-- ou 3.x supérieur -->
</dependency>
```

## 3.3. TP n°3

- Reprenez le code précédemment écrit en utilisant à nouveau l'implémentation «Fichier» du DAO.

- Faites en sorte que l'emplacement du fichier de stockage des films provienne du fichier de configuration Spring. Vous devrez ajouter une propriété de type String au DAO : *`fileLocation`*

## 3.4. TP n°4

- Reprenez le code précédemment écrit.

- Ajoutez dans la configuration un 2ème service du même type faisant également référence au même DAO.

- Ajoutez dans le constructeur de la classe de DAO un affichage indiquant que le constructeur a été invoqué.

- Exécutez la classe contenant le `main`, vous devez constater 1 appel au constructeur.

- Configurez tous les beans de type DAO en prototype et exécuter à nouveau, vous devez constater 2 appels au constructeur.

## 3.5. TP n°5

- Reprenez le code précédemment écrit et repasser le bean de type DAO en singleton.

- Modifiez le code du DAO afin que l'on puisse indiquer la propriété *`fileLocation`* par le biais d'un constructeur adapté.

- Dans la configuration, faites en sorte que le DAO soit construit par ce nouveau constructeur.

- Ajoutez une trace dans ce constructeur et exécutez à nouveau.

- Ajoutez ensuite un constructeur à `FilmService` qui prenne en paramètres le DAO exploité.

- Modifiez le fichier de configuration afin que le DAO soit fourni au Service par le biais de ce nouveau constructeur.

- Exécutez à nouveau.

- Vous pouvez maintenant supprimer le service supplémentaire ajouté au TP précédent.

## 3.6. TP n°6

- Instancier le Dao par le biais d'une méthode `getDefaultFileDao()` d'une classe `FilmDaoFactory`.

- Cette méthode retourne un Dao qui exploitent un fichier [USER_HOME]/films.txt :

Note : Pour récupérer USER_HOME, utilisez `System.getProperty("user.home")`.

## 3.7. TP n°7

- Reprenez le code précédemment écrit.

- Permettez un autowiring « byName » dans le service afin qu'il y soit automatiquement injecté le Dao. Ne modifiez pas le nom du bean de type Dao, donner lui simplement un `name` complémentaire si nécessaire.

## 3.8. TP n°7 Bis

- Obtenez le même résultat grâce à un autowiring byType.

## 3.9. TP n°8

- Reprenez le code précédemment écrit.

- Modifiez `FilmService` afin de pouvoir bénéficier de plusieurs DAO de type `FilmDAOInterface`.

- Ces DAO seront injectés dans une nouvelle propriété de type `ArrayList`.

- Nous allons déclarer 2 Dao de type fichier dans le fichier de configuration :

  - Le Dao déjà instancié par le biais du Factory
  - Un Dao permettant d'obtenir le même résultat dans le fichier « C:\temp\films.txt », ce Dao sera instancié **pour l'occasion** au moment de l'ajout dans la liste.

- Le service devra au travers d'une nouvelle méthode exploiter successivement la méthode `save` de tous les DAO.

- Modifier le contrôleur afin d'exploiter cette nouvelle méthode de service

- Exécutez à nouveau

## 3.10. TP n°9

- Revenez à l'étape 7 en modifiant à nouveau le contrôleur afin qu'il utilise la méthode de service exploitant un seul Dao.

- Annotez la propriété de type `FilmDAOInterface` dans la classe `FilmService`.

- Faites-en sorte que l'application continue à fonctionner de manière attendue.

## 3.11. TP n°10

- Faites en sorte que la classe `FilmService` puisse être détectée en tant que « Service » de l'application et que la classe DefaultFilmController puisse être détectée en tant que « Controller » de l'application.

- N'annotez pas pour l'instant le Dao même s'il aurait vocation à être déclaré sous forme de « Repository »

- Commentez les beans dans le fichier XML

- Modifier le main si nécessaire et exécutez à nouveau.

## 3.12. TP n°11

- Annotez désormais le Dao et faites en sorte qu'il soit auto-détecté par Spring. Faites en sorte que la propriété *`fileLocation`* provienne d'une propriété externalisée au sein d'un fichier conf.properties (situé à la racine du classpath).

*Note : Si vous souhaitez faire référence à une variable d'environnement*

- *dans `@Value`, il suffit d'utiliser là encore ${} : `@Value("${user.home}"....`*
- *dans le fichier properties cela fonctionne de la même manière*

- Supprimez toute référence explicite au Dao où à son Factory dans la configuration XML.

## 3.13. TP n°12

- Reprenez le code précédemment écrit.

- Séparez la configuration des beans en 2 fichiers :

  - Un nouveau fichier controllers.xml qui s'occupe uniquement de ce qui concerne les contrôleurs
  - Le fichier applicationContext.xml pour le reste

- Essayez les 2 méthodes présentées dans le cours pour utiliser l'ensemble des fichiers de configuration et conservez au final celle qui exploite la balise `<import>`.

## 3.14. TP n°13

- Reprenez le code précédemment écrit.

- Créez les fichiers de ressources internationalisées questions.properties et questions_en.properties

- Modifiez le contrôleur afin que l'utilisateur puisse au travers d'une question préalable choisir sa langue de prédilection (FR ou EN).

  - Attention, le contrôleur devra avoir un accès direct à l'ApplicationContext.
  - Truc et astuce : pour cela il suffit de déclarer un attribut de type ApplicationContext annoté @Resource ou équivalent.

- Faites en sorte que les questions suivantes posées à l'utilisateur (la première au moins) proviennent de la traduction française (Locale.FRENCH) ou de la traduction anglaise (Locale.ENGLISH)

## 3.15. TP n°14

- Reprenez le code précédemment écrit.

- Demandez à l'utilisateur la date de sortie du Film au format « dd/MM/yyyy ». Cette information ne sera pas

ajoutée dans le fichier films.txt.

- Dans le cas où cette date à < 10/09/1950, refusez l'insertion en base.

  - Vous ajouterez dans le contrôleur une propriété de type Date qui définit cette date minimale.

- Vous devrez ajouter la librairie :

  - joda-time-*.jar

Ou la dépendance Maven :

```
<dependency>
	<groupId>joda-time</groupId>
	<artifactId>joda-time</artifactId>
	<version>2.0</version>
</dependency>
```

Attention : Un bug des versions <3.2M2 de Spring empêche la concomitance des annotations `@DateTimeFormat` et `@Value`, `@DateTimeFormat` est alors tout bonnement ignoré.

Groupe en avance : Faites en sorte que cette date minimale soit modifiable par le biais de conf.properties.

## 3.16. TP n°15

- Ajouter une méthode `list()` au Dao et son interface permettant de retourner l'ensemble des lignes du fichier texte sous forme d'une `List<Film>` (Sans acteurs).

- Ajouter au service et à son interface une méthode `getFilmLightList()` retournant par l'intermédiaire du Dao la liste des Films (Sans acteurs).

- Ajouter au contrôleur une nouvelle méthode affichant la liste des films obtenus par le service (Titre, Genre et nombre d'exemplaires).

- Créer une classe `Runner2` exploitant cette nouvelle méthode de contrôleur.

# 4. Chapitre 3 : Spring pour le Web, SpringMVC

## 4.1. TP n°1

- Créez une application Web Dynamique supportant la norme Servlet 2.5 (Java EE 5).

- Ajoutez au web.xml la référence au listener du contexte d'application Spring

- Ajoutez les librairies déjà utilisées dans WEB-INF/lib et également la librairie :

  - org.springfrawework.web*.jar

Avec Maven, il faudra ajouter une dépendance vers le projet core et vers :

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-web</artifactId>
	<version>3.2.6.RELEASE</version>
</dependency>
```

- Si vous n'utilisez pas Maven, copiez l'ensemble des sources existantes (TP 15)

- Créez une servlet `ListServlet` accessible via l'uri /ListServlet

- Reportez l'intégralité du code du `main` dans le servlet et affichez le résultat sous forme HTML.

  - Vous récupérerez le context Spring via `WebApplicationContextUtils`
  - Vous récupérerez la liste des films auprès du service
  - Vous ferez afficher la liste (Titre uniquement) directement via la servlet (`PrintWriter`)

- Reportez l'intégralité de la configuration Spring dans un fichier WEB-INF/applicationContext.xml ou indiquez l'emplacement du fichier explicitement dans une variable de contexte dans le web.xml

- Déployez l'ensemble de l'application dans le conteneur de servlet tomcat via Eclipse ou Netbeans

- Demandez la servlet via votre navigateur web.

- Attention : Certaines versions de Tomcat associées à certaines versions de Netbeans engendrent un souci dans le fichier catalina.bat / sh, ci-dessous le correctif :

http://stackoverflow.com/questions/26485487/error-starting-tomcat-from-netbeans-127-0-0-1-is-not-recognized-as-an-inter

## 4.2. TP n°2

- Ajoutez la `DispatcherServlet` sous le nom “actions” mappé sous l'uri *.do

- Crééz le fichier de configuration Spring WEB-INF/actions-servlet.xml (Rappel : ce fichier n'est pas à indiquer dans le web.xml)

- Créez une page /pages/hello.jsp affichant le message « Bonjour de Spring MVC ! ». Le message doit être ajouté au `ModelAndView` sous le nom « message ».

- Créez un contrôleur `HelloController` (hérité de `AbstractController`) qui va donc ajouter le message dans le `ModelAndView` retourner la vue qui correspond à hello.jsp

- Ajoutez dans actions-servlet.xml le handlerMapping sous forme de `BeanNameUrlHandlerMapping`

- Ajoutez dans actions-servlet.xml l'uri sayhello.do pointant vers le contrôleur

- Ajoutez dans actions-servlet.xml le `viewResolver` de type `UrlBasedViewResolver` ayant un `viewClass` de type `JstlView`, un prefix `/pages` et un suffix `.jsp`

- Ajoutez également les librairies :

- org.springfrawework.web.servlet*.jar
- jstl.jar
- standard.jar

Ou avec Maven :

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-webmvc</artifactId>
	<version>3.2.6.RELEASE</version>
</dependency>
<dependency>
	<groupId>javax.servlet</groupId>
	<artifactId>jstl</artifactId>
	<version>1.2</version>
</dependency>
```

## 4.3. TP n°2 Suite [Groupe en avance]

Annulé

## 4.4. TP n°3

- Obtenez le même résultat qu'au TP 1 mais cette fois via un contrôleur Spring MVC : `FilmController` accessible par l'url `film/list.do`

- Vous injecterez directement le service dans le contrôleur

- La liste sera affichée via une page `/pages/list.jsp`

## 4.5. TP n°4

- Obtenez le même résultat que pour le TP précédent mais par le biais de @Controller.

## 4.6. TP n°5

- Permettez à l'utilisateur de cliquer sur les titre de la liste.

- Les liens mèneront vers `FilmController` qui sera capable de présenter plus en détail un film (page description.jsp).

- Le lien fournira l'identifiant du film à afficher. Il faudra ajouter cette nouvelle propriété (int) dans la classe Film.

- Ajoutez manuellement dans le fichier texte des identifiants aux films.

- Récupérez dans le contrôleur cet identifiant via l'annotation @RequestParam et fournissez à la JSP le film à afficher via un attribut de type Model (ou – plus compliqué - l'annotation @ModelAttribute).

- Bien évidemment, transitez par une nouvelle méthode de service et une nouvelle méthode de Dao.

## 4.7. TP n°6

- Permettez la création de film (Équivalent `Runner`) via `FilmController`.

- Créez une page de formulaire permettant de saisir titre, genre et nombre d'exemplaires (/pages/ajout-form.jsp).

- Donnez l'attribut `commandName="ajoutcmd"` à la balise `form`

- Les valeurs de ce formulaire seront stockées dans une instance de `FilmCommand`, classe très proche de `Film` (Rappel :Utiliser la classe `Film` pour encapsuler les valeurs du formulaire peut être considéré comme une erreur de conception)

- Le contrôleur `FilmController` aura 2 fonctions complémentaires (donc 2 méthodes)

  - Afficher le formulaire vide
  - Après soumission du formulaire, affiche dans une page de confirmation (/pages/ajout-ok.jsp) le texte : «Le film à été créé ».
    - L'identifiant du nouveau Film sera celui du dernier du fichier incrémenté de 1, il faudra modifier la méthode `save` du Dao.

- Le contrôleur sera globalement accessible via l'URI /film.

  - Pour afficher le formulaire vide, l'URI complète sera /film/showform.do
  - Pour valider le formulaire, l'URI complète sera /film/add.do

## 4.8. TP n°7

- Reprenez l'exercice précédent et faites en sorte que, en cas d'erreur de saisie sur le formulaire d'ajout, le contrôleur affiche à nouveau la vue formulaire avec à côté des champs incriminés un message explicatif.

- Ajoutez la librairie :

  - validation-api-*.GA.jar
  - hibernate-validator-*.GA.jar
  - slf4j-api-*.jar
  - slf4j-log4j-*.jar
  - log4j*.jar

Avec Maven , ajoutez dans le pom :

```
<dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator</artifactId>
            <version>4.1.0.Final</version>
</dependency>
<dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
       <version>1.0.0.GA</version>
</dependency>
<dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.5.6</version>
</dependency>
<dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.5.6</version>
</dependency>
```

- Nous considérerons que

- le titre,et le genre sont obligatoires (NotEmpty)
- le nombre d'exemplaires est obligatoire (NotNull)
- le genre ne peut pas dépasser 10 caractères
- le nombre d'exemplaires doit être > 0
- Ne vous préoccupez pas de l'erreur de conversion qui peut exister si l'utilisateur ne saisit pas un numérique dans le champ nombre d'exemplaires

## 4.9. TP n°8 [Optionnel - Selon avancement]

- Reprenez l'exercice précédent.

- Faites en sorte que le formulaire de création de film contienne toujours les dernières valeurs saisies par l'utilisateur.

- Lors d'une première création, il doit contenir des valeurs par defaut
- Lors d'une création suivante, il doit s'agir des valeurs précédentes

Aide : Utiliser un bean de scope session.

## 4.10. TP n°9 [Optionnel - Selon avancement]

- Reprenez l'exercice précédent.

- Faites en sorte que l'URL permettant d'obtenir la liste des films, fournisse cette liste au format JSON.

- Assurez-vous d'avoir activé la « content negociation » grâce à la balise <mvc:annotation-config/>

- Annotez le type de retour avec @ResponseBody

- Ajoutez les librairies Jackson :

  - jackson-core-asl-xxx.jar
  - jackson-mapper-asl-xxx.jar

Ou :

```
<dependency>
                <groupId>org.codehaus.jackson</groupId>
                <artifactId>jackson-mapper-asl</artifactId>
                <version>1.9.5</version>
            </dependency>
            <dependency>
                <groupId>org.codehaus.jackson</groupId>
                <artifactId>jackson-core-asl</artifactId>
                <version>1.9.5</version>
            </dependency>
```

Vous pouvez également exploiter le contenu retourné grâce à la page HTML liste.html et les js associés fournis dans les ressources.

# 5. Chapitre 4 : Spring et la persistance

## 5.1. TP n°1

- Reprenez le code du dernier TP du Chapitre 3.

- Exécutez le script de la base de données VIDEO fourni en annexe pour MySQL (paramètres de connexion fournis par le formateur)

- Créez une classe `JdbcFilmDao` qui implémente la même interface que `FileFilmDao` et hérite de `JdbcDaoSupport`. Cette classe exploitera la méthode query de `JdbcTemplate` pour récupérer la liste des films. Ne tenez pas comptes des acteurs. Utilisez un `RowMapper` pour alimenter la collection.

- Déclarez le DAO dans la configuration XML. Pour le moment n'utilisez pas l'annotation @Repository.

Note : Attention le DAO fait référence à une `Datasource` pointant vers la base VIDEO.

La datasource sera publiée sous le nom jdbc/video et instanciée soit :

- Directement via Spring via commons-dbcp

- Ajoutez les librairies spécifiques Spring :

- Par Tomcat grâce au fichier META-INF/context.xml joint.
- org.springframework.transaction*.jar
- org.springframework.jdbc*.jar

Avec Maven :

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-jdbc</artifactId>
	<version>3.2.6.RELEASE</version>
</dependency>
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-tx</artifactId>
	<version>3.2.6.RELEASE</version>
</dependency>
```

- Si vous instanciez le pool manuellement via common-dbcp, ajoutez également les librairies suivantes :

- commons-pool.jar
- commons-dbcp.jar
- mysql.jar ou ojdbc.jar

Avec Maven :

```
<dependency>
```

```
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.30</version>
</dependency>
<dependency>
            <groupId>commons-pool</groupId>
            <artifactId>commons-pool</artifactId>
            <version>1.6</version>
</dependency>
<dependency>
            <groupId>commons-dbcp</groupId>
            <artifactId>commons-dbcp</artifactId>
            <version>1.4</version>
</dependency>
```

- Sinon, ajoutez dans le répertoire lib de Tomcat le driver JDBC :

- mysql.jar ou ojdbc.jar

- A l'aide de Spring, remplacer l'implémentation de Dao actuelle par celle utilisant JDBC.

- Exécuter à nouveau la fonction film/list.do dans votre navigateur

## 5.2. TP n°2

Obtenez le même résultat en utilisant l'annotation `@Repository`.

- Si les 2 implémentations de DAO existantes sont dans le même package, le `<context:component-scan>` va être amené à instancier 2 implémentations compatibles avec le Service.

- Dans ce cas, répartissez vos DAO dans des sous-packages et scanner uniquement le package adéquat.

## 5.3. TP n°3 [Groupe en avance]

Réutilisez le formulaire de création de film. Il doit maintenant être capable de s'interfacer avec la base de données.

## 5.4. TP n°4

L'objectif sera de remplacer les accès à la base du chapitre 4 effectués en JDBC par des accès Hibernate.

- Ajouter au TP l'ensemble des librairies nécessaires à l'intégration du framework Hibernate :

- org.springframework.orm*.jar

- commons-collections*.jar
- hibernate3.jar
- hibernate-jpaXXX.jar
- javassist-XXX.jar
- jta.jar
- antlr*.jar
- dom4j-*.jar
- cglib-nodep-*.jar

Ou avec Maven :

```
<dependency>

	<groupId>org.springframework</groupId>

	<artifactId>spring-orm</artifactId>

	<version>3.2.6.RELEASE</version>

</dependency>

<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-core</artifactId>
	<version>3.5.6-Final</version>
</dependency>

<dependency>
	<groupId>javassist</groupId>
	<artifactId>javassist</artifactId>
	<version>3.9.0.GA</version>
	<scope>runtime</scope>
</dependency>
```

- Créer une classe `HibernateFilmDao` sur le modèle des autres Dao.

- Créer et compléter le fichier Film.hbm.xml (Acteur.hbm.xml est donné en exemple) que vous placerez sous la racine des sources Java

- Modifier en conséquence la configuration Spring (ajouter notamment le sessionFactory)

- Utiliser le dialecte `org.hibernate.dialect.MySQLInnoDBDialect` ou `org.hibernate.dialect.Oracle10gDialect`

- Exécuter la fonction film/list.do dans votre navigateur

- Ceux qui le souhaitent peuvent essayer d'ajouter une propriété `acteurPrincipal` de type Acteur dans la classe Film. Ceci nécessite de modifier mappings hibernate et Dao.

- Essayez d'afficher dans la page de description de film le nom et le prénom de l'acteur principal. A quoi faut il faire attention pour que cela fonctionne ?

## 5.5. TP n°5 [Groupe en avance]

Réutiliser le formulaire de création de film. Il doit maintenant être capable de s'interfacer avec Hibernate.

# 6. Chapitre 6 : Spring et la POA

## 6.1. TP n°1

L'objectif est de mettre en oeuvre via l'API Spring AOP une instance d'Advice qui interviendrait avant et après toutes les méthodes du service.

- Ajoutez au TP l'ensemble des librairies nécessaires à l'intégration de Spring AOP :

- org.springframework.aop*.jar
- aopalliance-*.jar

Ou avec Maven, les librairies étant déjà présentes par dépendance :

```
<dependency>
	<groupId>aopalliance</groupId>
	<artifactId>aopalliance</artifactId>
	<version>1.0</version>
</dependency>
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-aop</artifactId>
	<version>3.2.6.RELEASE</version>
</dependency>
```

- Créez un Advice nommé `DebugAdvice` qui implémente `MethodBeforeAdvice` et `AfterReturningAdvice`

- Complétez les méthode `before` et `afterReturning`

- La première affiche dans la console “La méthode XXX va être invoquée”
- La seconde affiche dans la console “La méthode XXX s'est terminée”

Note : XXX s'obtient en recherchant le nom de la méthode reçue en paramètre de `before` et `afterReturning`

- Modifiez la méthode mappée sous film/list.do afin qu'elle fasse appel à `getFilmLightList` sur le proxy de `FilmService`.

- Relancez la page film/list.do et constater les traces dans la console.

## 6.2. TP n°2

L'objectif est d'obtenir un résultat similaire au TP n°1 mais avec AspectJ

Attention : Assurez-vous tout d'abord que les services annotés ne soient pas détectés via un `component-scan` par Spring MVC. Le plus simple est de placer les contrôleurs dans un package spécifique, seul package dont on demande la détection dans le fichier `actions-servlet.xml`. Dans le cas contraire, le service serait injecté dans le contrôleur avant d'être avis, ce n'est donc pas le proxy qui serait injecté et vous ne bénéficieriez donc pas des aspects.

- Ajoutez au TP l'ensemble des librairies nécessaires à l'intégration de AspectJ :

- aspectjweaver.jar

Ou avec Maven :

```
<dependency>
	<groupId>org.aspectj</groupId>
	<artifactId>aspectjweaver</artifactId>
	<version>1.6.12</version>
</dependency>
```

- Supprimez les implémentations de `DebugAdvice` et remplacer les méthode `before` et `afterReturning` par

- `void doBefore()`
- `void doAfter()`

- Ajoutez à la configuration Spring un Aspect dont l'advice pointe sur la méthode `getFilmLightList` de `FilmService` et pour lequel il existe un pointcut avant exécution pointant vers `doBefore()` et un pointcut après exécution pointant vers `doAfter()`.

Obtenir le nom de la méthode invoquée n'est ici pas possible. Puisque nous ciblons `getFilmLightList` dans le pointcut, vous allez donc simple écrire :

- La première affiche dans la console “La méthode `getFilmLightList` va être invoquée”
- La seconde affiche dans la console “La méthode `getFilmLightList` s'est terminée”

Nous verrons tout à l'heure comment obtenir un comportement générique identique au TP 1.

Note : Ne pas oublier les namespaces en en-tête du fichier de configuration

- Supprimez l'appel au `ProxyFactory` dans `getFilmLightList`.

- Relancez la page film/list.do et constater les traces dans la console

## 6.3. TP n°3

L'objectif est d'obtenir un résultat similaire au TP n°1 mais avec un join point aop:around.

## 6.4. TP n°3 Suite (Groupe en avance)

Faites en sorte que, grâce à l'AOP, la liste affichée à l'utilisateur comporte un film supplémentaire qui ne provient pas de la base de données : District 9.

## 6.5. TP n°4

L'objectif est d'obtenir un résultat similaire au TP n°1 mais avec les annotations

- Ajoutez les annotations `@Aspect, @Before et @AfterReturning` (ou `@Around`) dans la classe `DebugAdvice`

- Ajoutez dans la configuration Spring la balise `<aop:aspectj-autoproxy>`

- Commentez la partie `aop:config`

- Ne supprimez pas la déclaration de l'advice dans la configuration Spring, sauf si vous l'annotez `@Component` et utilisez la balise `<component-scan>`.

- Relancez la page liste.do et constatez les traces dans la console

# 7. Chapitre 7 : Spring et les transactions

## 7.1. TP n°1

- Conservez les accès SGBD au travers de Hibernate

Nous allons mettre en œuvre un cas fonctionnel qui n'existe probablement jamais dans la réalité mais qui permet d'appliquer un aspect transactionnel à notre méthode de service et d'en constater le résultat.

- Créez dans le DAO `HibernateFilmDao` une méthode `deleteFilm` prenant en paramètre un id. Cette méthode doit supprimer le film dont l'id est passé en paramètre. Cette méthode va devoir

1. Retrouver le film à supprimer via son id (méthode `get`)
2. Supprimer l'entité retrouvée précédemment (méthode `delete`)
3. Retourner un booléen indiquant si la suppression à eue lieu ou non (pour l'instant retournez true dans tous les cas)

- Ajoutez dans `FilmService` une méthode `doDeleteAFilm` qui appelle `deleteFilm` du DAO.

- Dans `FilmService`, déplacez l'appel au DAO qui est actuellement effectué dans `getFilmLightList` vers une nouvelle méthode `doFindAllFilms`

- La méthode `getFilmLightList` devra désormais être transactionnelle (par la méthode programmatique) et appeler successivement la méthode `doDeleteAFilm` **deux fois avec 2 id différents** puis la méthode `doFindAllFilms`.

1. Effectuez une première exécution de list.do en faisant en sorte que `doDeleteAFilm` passe 2 fois l'id d'un film lié à aucun acteur secondaire (films id 3 et 4 par exemple), la liste doit s'afficher sans les films supprimés car les 2 suppressions ont été effectuées.
2. Repasser le script insert.sql après avoir vidé l'ensemble des tables
3. Effectuez une deuxième exécution de list.do en faisant en sorte que `doDeleteAFilm` passe la 2ème fois l'id d'un film qui bénéficie d'un acteur secondaire (film id 2 exemple). La deuxième suppression est alors impossible (clé étrangère non nulle). Une erreur SQL (RuntimeException) doit survenir. La page ne doit pas s'afficher (ou s'afficher vide selon ce que vous aurez écrit), vérifier en base qu'aucune des 2 suppressions n'a eue lieu.

## 7.2. TP n°2

- Reprenez l'exercice précédent avec la méthode déclarative. Pour cela, déclarez dans le fichier de configuration un pointcut sur *`execution(* formation.FilmService.getFilmLightList(..))`* ou sur *`execution(* formation.FilmService.get*(..))`* mais surtout pas sur *`execution(* formation.FilmService.do*(..))`*

1. Effectuez à nouveau l'exécution de list.do en faisant en sorte que doDeleteAFilm passe la 2ème fois l'id d'un film qui joue dans un film qui bénéficie d'un acteur secondaire (film id 2 exemple). La deuxième suppression est alors impossible (clé étrangère non nulle). Une erreur SQL (RuntimeException) doit survenir. La page ne doit pas s'afficher (ou s'afficher vide selon ce que vous aurez écrit), vérifier en base qu'aucune des 2 suppressions n'a eue lieu.
2. Effectuez ensuite à nouveau l'exécution de list.do en faisant en sorte que `doDeleteAFilm` passe 2 fois l'id d'un film lié à aucun secondaire (films id 3 et 4 par exemple), la liste doit s'afficher sans les films supprimés car les 2 suppressions ont été effectuées.

## 7.3. TP n°3

- Repasser le script insert.sql après avoir vidé l'ensemble des tables

- Reprenez le TP n°2 en ajoutant l'annotation `@Transactional` à la méthode `getFilmLightList`()

- Pour le cas d'échec, réutiliser les id 2 et 3.

- Pour le cas normal vous pouvez réutiliser les id 3 et 4.

# 8. Chapitre 8 : Spring et JUnit

## 8.1. TP n°1

- Faites à nouveau en sorte que `getFilmLightList()` récupère l'ensemble des films sans suppression.

- Ce TP va permettre d'effectuer des tests unitaires de la classe `FilmService`.

- Créez un stub object `StubFilmDao` qui va remplacer `HibernateFilmDao` et qui initialise la collection de films avec 2 films : Oblivion et Black Swan.

- Créez un fichier de configuration de test à l'image du fichier de configuration initial mais dans lequel le service précédemment créé utilise le stub object DAO

- Créez un cas de test `TestFilmService`

- Ajoutez y les méthodes `testListFilms` et `testDeleteFilm` destinées à tester les méthodes du service

- Le test `testListFilm` doit vérifier que la méthode `getFilmLightList` de `FilmService` ne renvoie ni null ni une collection de taille 0.

- Le test `testDeleteFilm` doit vérifier que la méthode `doDeleteAFilm` de `FilmService` supprime effectivement le film lorsque celui-ci existe. Pour vérifier la suppression exploitez le booléen retourné par la méthode du DAO.

- Lancez le test en exécutant la classe de test avec Eclipse (click droit → Run as → Junit Test)

- Vérifiez le résultat dans la *Vue* eclipse dédiée à Junit

Note : Si vous le souhaitez conserver dans le fichier de configuration de test l'accès à la base, la datasource devra alors être gérée par Spring (via Common dbcp : Voir chapitre 4)

## 8.2. TP n°2

- Faites en sorte que `doDeleteAFilm` renvoie une exception `FilmNotFoundException` si le film à supprimer n'existe pas (autrement dit si la méthode du DAO retourne false).

- Ajouter un test `testDeleteFilmInconnu` et faites en sorte de générer une défaillance en invoquant la méthode `doDeleteAFilm` avec un id inexistant.

- Le test sera effectué avec succès si l'exception est bel est bien renvoyée.

## 8.3. TP n°3

- Ce TP va permettre d'effectuer des tests d'intégration de la classe `FilmService`.

- Remplacez dans la configuration de test `StubFilmDAO` par `HibernateFilmDao`.

- Le test étant réalisé hors du serveur applicatif, la datasource ne sera pas instanciée par le serveur. La datasource devra alors être instanciée par Spring (via Common dbcp : Voir chapitre 4)

- Créez un nouveau cas de test `TestFilmServiceIntegration`. Il aura pour objectif de tester l'appel d'une méthode d'insertion.

- Si cela n'a pas été fait précédemment, ajoutez à `FilmService` une méthode insertion qui va appeler via le Dao.

- Ajoutez à la classe de test la méthode `testInsertFilm` qui va successivement appeler :

- `getFilmLightList` pour compter le nombre de films en base
- `doInsertAFilm` pour inserer un nouveau film : Lost Highway
- `getFilmLightList` pour recompter le nombre de films afin de s'assurer que la quantité de films dans la base à bien été augmentée de 1.

- Effectuez le test et vérifiez le résultat dans la vue Junit.

# 9. Chapitre 9 : Spring Security

## 9.1. TP n°1

- Ajoutez au TP l'ensemble des librairies nécessaires à l'intégration de Spring-security :

- spring-security-core-*.jar
- spring-security-config-*.jar
- spring-security-web-*.jar

Ou avec Maven :

```
<dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-core</artifactId>
            <version>3.2.6.RELEASE</version>
</dependency>
<dependency>
       <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>3.2.6.RELEASE</version>
</dependency>
 <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
            <version>3.2.6.RELEASE</version>
</dependency>
```

- Ajoutez un utilisateur ayant le login « formation » le mot de passe « spring ».

- Créez une page d'accueil welcome.html affichant simplement « Accueil » et contenant un lien HTML vers /film/showform.do et un lien vers /film/list.do.

- Adaptez l'application afin de bloquer l'accès à toute ressource (hormis la page d'accueil) de votre application Web aux utilisateurs non-authentifiés ou n'ayant pas le rôle ROLE_USER.

## 9.2. TP n°2

Reprenez le TP précédent.

- Protégez l'ajout de film afin que seuls les utilisateurs avec le rôle ROLE_ADMIN puissent y accéder. Créez un utilisateur admin/admin ayant le rôle ROLE_ADMIN.

- La page d'accueil reste accessible à tout le monde et la page /film/list.do accessible à ROLE_USER.

- Pour l'authentification, créez votre propre formulaire

- Ajoutez un lien permettant la déconnexion dans les pages disponibles sous la partie /film. La déconnexion doit mener l'utilisateur vers la page d'accueil.

## 9.3. TP n°3

Reprenez le TP précédent.

- Modifiez la source de données de l'authentification pour que les comptes utilisateurs soient recherchés en base de données uniquement.

- Créez une page pages/roles.jsp qui affiche l'utilisateur actuellement authentifié et la liste des droits dont il dispose. Cette page sera accessible par le biais de l'url /admin/roles.do mappé par un nouveau contrôleur `AdminController`.

- Vérifiez à nouveau que les accès aux pages /film/list.do et /film/showform.do fonctionnent.

- Vérifiez les informations des utilisateurs identifiés grâce à la page /admin/roles.do

## 9.4. TP n°4

- Modifiez l'ensemble de vos liens « déconnexion » pour qu'ils affichent entre parenthèse le nom de l'utilisateur actuellement authentifié, et ce à l'aide des taglibs Spring Security :

- spring-security-taglibs-*.jar

Avec Maven :

```
<dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-taglibs</artifactId>
            <version>3.0.5.RELEASE</version>
</dependency>
```

- Dans la page d'accueil :

- Rajoutez un lien « s'authentifier » qui pointe sur le formulaire d'authentification
- A l'aide des taglibs, faites en sorte que les liens sur lesquels un utilisateur n'a pas les droits disparaissent (/film/showform.do est accessible à un utilisateur ayant un rôle ROLE_ADMIN). Vous devrez transformer la page d'accueil en jsp.

# 10. Chapitre 10 : Spring JMX

## 10.1. TP n°1

- Reprenez le module core du TP précédent.

- Créez un service `FilmUtilsJMX` avec une méthode `increment(int id,int quantity)` capable d'incrémenter/ décrémenter de `quantity` le nombre d'exemplaires d'un film d'identifiant `id`. Vous exploiterez les mécanismes de Hibernate pour la mise à jour.

- Vous ajouterez une méthode `updateQuantity(int id, int quantity)` à votre service et une méthode `update(Film)`, il ne sera pas nécessaire d'ajouter de méthode de mise à jour dans le DAO.

- Annotez la classe `FilmUtilsJMX`.

- Le main doit récupérer un contexte d'application via un `ClasspathXmlApplicationContext` et doit ensuite afficher dans la sortie standard toutes les 5 secondes le nombre d'exemplaires d'un des films du fichier (Thread.sleep).

- Exécutez le main

- Via la console Jconsole ou JvisualVM (avec le Plugin MBeans), augmenter le nombre d'exemplaires du film affiché toutes les 5 secondes.

- Vous aurez besoin d'un certain nombre de librairies comme springframework.aop et ses dépendances.

# 11. Chapitre 11 : Spring JMS

## 11.1. TP n°1

Nous allons effectuer un TP simple qui va permettre avec Spring de produire et de consommer des messages JMS déposés dans un Topic.

Nous utiliserons l'implémentation Apache ActiveMQ par le biais de la librairie activemq-all-5.3.0.jar que nous déposerons dans les librairies du projet.

- Reprenez l'un des TP du chapitre 6 et ajoutez au context.xml les ressources JNDI JMS que vous trouverez dans le template de context.xml fourni. Il permettra d'instancier le ConnectionFactory et le Topic et de les placer dans l'annuaire JNDI.

- Ajoutez à un nouveau contrôleur `MessageController` une méthode de mapping /message/addToJMS.do qui ajoute dans le Topic JMS « my.topic » un message : « Message : X ». Où X correspond à un paramètre fourni dans l'url d'appel de la servlet. Vous aurez à créer une nouvelle classe héritée de `MessageCreator`. Cette méthode retourne vers une vue JSP affichant également : « Le message X a été déposé ».

- Nous allons ensuite utiliser la même application Web pour consommer les messages. Ajoutez simplement à l'applicationContext les beans nécéssaires l'initialisation d'un Listener qui se mette en écoute des messages ajoutés dans le topic et que vous référencez au sein du `DefaultMessageListenerContainer`. Ce listener affichera simplement dans la console les messages reçus.

- Ajoutez les librairies suivantes au WEB-INF/lib :

  - org.springframework.jms-*.jar
  - activemq-all-5.3.0.jar

Ou avec Maven...

- Publiez les applications dans Tomcat, ajoutez des messages par le biais de l'url /message/addToJMS.do

# 12. Chapitre 12 : Spring Remoting

## 12.1. TP n°1

- Reprenez l'application du Web du TP précédent que vous nommerez le projet Chapitre12TP1Server.

- Desactivez les sécurités placées au niveau /film/ (`filters="none"`).

- Publiez le service au moyen d'un registry RMI sur le port 1099 (Bean de type `RmiServiceExporter`). Le nom du service sera `filmService` faisant référence au bean `FilmService`.

- Placez toutes les classes nécessaires au client RMI dans une librairie `formation.jar` (`FilmServiceInterface`, et `Film, Acteur`).

- Attention, `Film` et `Acteur` doivent désormais être `Serializable`.

- Démarrer l'application.

- Créez une application non-web Chapitre12TP1Client.

- Configurez Spring en déclarant un bean `filmService` de type `RmiProxyFactoryBean`, dont la propriété `serviceUrl` pointe sur le service `filmService` du serveur local port 1099.

- Ajoutez au classpath de cette application la librairie formation.jar

- Ajoutez à l'application `Chapitre12TP1Client` une classe `Main.java` qui possède un main et qui doit invoquer la méthode distante `getFilmLightList()` de `filmService`. Le main affichera dans la console le titre de tous les films récupérés.

- Exécutez le main

## 12.2. TP n°2

Reprenez le TP précédent.

- Déactiver la partie RMI (client et serveur)

- Au niveau serveur, ajoutez au contrôleur une méthode mappée sur /film/listeTitreFilms.do qui retourne la liste des films à une vue /pages/listeTitreFilms.jsp.

- La vue /pages/listeTitreFilms.jsp fournira un contenu en plein texte text/plain. Via la JSTL, énumérez les liste des films et afficher leur titre.

- Côté client, faites appel au service et affichez dans la console le résultat obtenu.

- Pour utiliser la classe RestTemplate vous devrez inclure la librairie :

  - org.springframework.web-*.jar

## 12.3. TP n°3

Reprenez le TP précédent.

- Au niveau serveur, ajoutez au contrôleur une méthode mappée sur /film/descriptionPlain/{filmId}.do qui retourne à une vue /pages/filmPlain.jsp le film correspondant à l'ID «filmId» reçu dans le chemin de l'URL.

- La vue /pages/filmPlain.jsp fournira un contenu en plein texte text/plain. Via la JSTL, afficher le titre du film fourni par le contrôleur.

- Côté client, faites appel au service en fournissant un ID existant dans la base (ex : 2) et affichez dans la

console le résultat obtenu.

## 12.4. TP n°4

Reprenez le TP précédent.

- Nous allons maintenant retourner un film complet au client par le biais d'une sérialisation / désérialisation Objet → XML → Objet

- Ajouter un mapping *.rest. Dans la nouvelle configuration xxx-servlet.xml, ajouter le `ContentNegociatingViewResolver` conformément aux exemple du cours.

- Modifier la configuration spring de la manière indiquée dans le cours afin d'intégrer Castor en tant que solution de marshalling / unmarshalling.

- Côté serveur et client, ajoutez à la racine des sources le fichier mapping.xml joint (adaptez le si nécessaire) et référencez ce mapping dans les beans castor.

- Le main doit maintenant récupérer un objet de type Film. Affichez dans la console le titre du film reçu.

- Côté serveur et côté client, vous devrez inclure les librairies :

- org.springframework.oxm-*.jar
- castor-core.jar
- castor-xml.

Ou avec Maven...