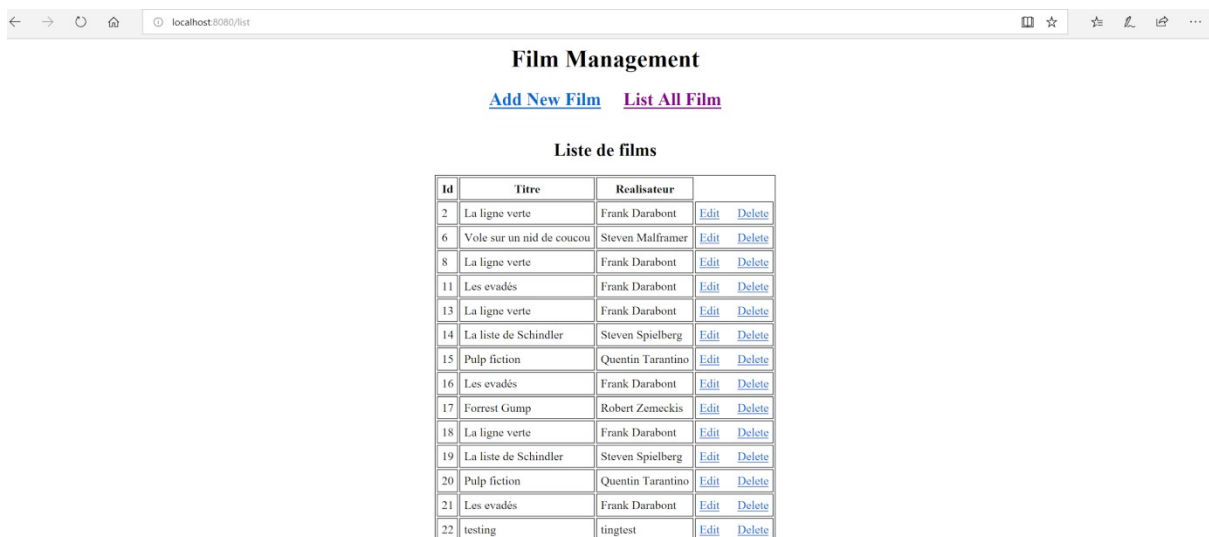


TP 3 - Java EE

1. Présentation du TP

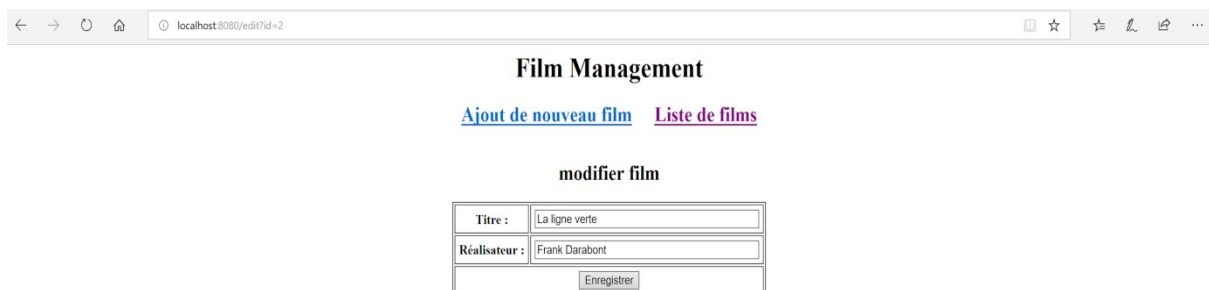
Le but du TP est de faire la gestion d'une bibliothèque de films, c'est-à-dire les stocker, les modifier, les afficher, les supprimer sur une page web.

L'objectif est d'obtenir le fonctionnement suivant :
Lorsque l'on arrive sur l'URI **/list**, on obtient la liste de tous les films.



Film Management			
Add New Film List All Film			
Liste de films			
Id	Titre	Réalisateur	
2	La ligne verte	Frank Darabont	Edit Delete
6	Vole sur un nid de coucou	Steven Malfamer	Edit Delete
8	La ligne verte	Frank Darabont	Edit Delete
11	Les évadés	Frank Darabont	Edit Delete
13	La ligne verte	Frank Darabont	Edit Delete
14	La liste de Schindler	Steven Spielberg	Edit Delete
15	Pulp fiction	Quentin Tarantino	Edit Delete
16	Les évadés	Frank Darabont	Edit Delete
17	Forrest Gump	Robert Zemeckis	Edit Delete
18	La ligne verte	Frank Darabont	Edit Delete
19	La liste de Schindler	Steven Spielberg	Edit Delete
20	Pulp fiction	Quentin Tarantino	Edit Delete
21	Les évadés	Frank Darabont	Edit Delete
22	testing	tingtest	Edit Delete

À côté de chaque film, faites en sorte d'avoir la possibilité d'éditer ou de supprimer ledit film.
L'édition peut se présenter ainsi :



Film Management	
Ajout de nouveau film Liste de films	
modifier film	
Titre :	<input type="text" value="La ligne verte"/>
Réalisateur :	<input type="text" value="Frank Darabont"/>
<input type="button" value="Enregistrer"/>	

Pour ce qui est de la page d'ajout de nouveaux films, elle est similaire à celle de l'édition.

Dans l'archive que vous avez téléchargée, vous avez déjà à votre disposition la **persistance**, dans le fichier **EstablishConnection.java** dans le **package utils**.

C'est quoi la persistance ?

C'est la couche qui permet entre autres de communiquer avec la base de données.

2. Projet

a. Génération

Vous allez pouvoir générer votre projet avec la commande :

```
mvn archetype :generate
-archetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-quickstart
-DarchetypeVersion=1.4
```

(sans saut à la ligne)

Entrez dans l'ordre TP3Ensta, TP3Ensta, puis entrez sans aucune valeur, puis ensta et faite entrer à nouveau.

Dans le pom.xml ajouter :

```
<packaging>war</packaging>
```

sous la version de votre projet.

Ajoutez aussi :

```
<dependency>
```

```
    <groupId>javax.servlet</groupId>
```

```
    <artifactId>javax.servlet-api</artifactId>
```

```
    <version>3.1.0</version>
```

```
    <scope>provided</scope>
```

```
</dependency>
```

```
<dependency>
```

```
    <groupId>javax.servlet.jsp</groupId>
```

```
    <artifactId>javax.servlet.jsp-api</artifactId>
```

```
    <version>2.3.1</version>
```

```
    <scope>provided</scope>
```

```
</dependency>
```

```
<dependency>
```

```
    <groupId>jstl</groupId>
```

```
    <artifactId>jstl</artifactId>
```

```
    <version>1.2</version>
```

```
</dependency>
```

```
<dependency>
```

```
    <groupId>com.h2database</groupId>
```

```
    <artifactId>h2</artifactId>
```

```
    <version>1.4.197</version> <!-- Si vous utilisez Java 6 version
```

```
1.4.191 -->
```

```
</dependency>
```

dans la balise dependencies, et enfin :

```
<plugin>
```

```
    <groupId>org.codehaus.mojo</groupId>
```

```
<artifactId>exec-maven-plugin</artifactId>
<version>1.4.0</version>
<configuration>
  <mainClass>ensta.utils.FillDatabase</mainClass>
</configuration>
</plugin>
```

dans la balise plugins . Vous pouvez maintenant extraire main.zip dans le dossier src

Dans le terminal lancez la commande :

```
mvn clean install exec:java
```

Ce qui va initialiser votre base de donnée.

Et la commande suivante va vous permettre de lancer votre serveur :

```
mvn clean install tomcat7:run
```

b. Première partie : Modèle

Nous allons commencer par nous intéresser à la création de notre **modèle**.

C'est quoi un modèle ?

On crée une classe Java qui aura les mêmes attributs que la table de la BDD à laquelle elle correspond. Cette classe sera chargée de représenter les instances de la table dans notre programme Java.

C'est le modèle. **(cf. le cours)**

Pour ce faire, vous allez tout d'abord créer un **package** que vous nommerez **model**. Dans ce package, créez une classe que vous nommerez **Film**.

Chose à savoir avant de vous lancer dans l'écriture de la classe Film

Pour ce TP, nous vous fournissons la table film :

```
CREATE TABLE IF NOT EXISTS film (
  id INT PRIMARY KEY AUTO_INCREMENT,
  titre VARCHAR(100),
  realisateur VARCHAR(50)
);
```

Tout est déjà géré à ce niveau là, ne vous en préoccupez pas ! Faites simplement en sorte de ne pas oublier d'exécuter FillDatabase.java

une fois !

La classe Film n'est qu'un reflet de ce qu'est la table film. Faites donc en sorte d'avoir dans cette classe les champs id, titre et realisateur ainsi que les mutateurs/accesseurs associés.

Ajouter à cette classe au moins un constructeur qui instancie tous les champs.

c. Deuxième partie : DAO (data access object) et Service

Nous allons maintenant nous attaquer au **DAO**.

C'est quoi un DAO ?

Il permet de gérer l'accès aux données. Les DAO...

- Encapsulent la logique liée à la base de données ;
- Respectent le CRUD (**C**reate, **R**ead, **U**ppdate, **D**eleter) ;
- Suivent généralement le design pattern « Singleton ».

(cf. le cours)

Tout d'abord commencé par créer un **package** que vous nommerez **dao**.

Dans ce package vous créerez une classe **FilmDao**.

Vous commencerez par écrire les requêtes qui seront utilisées par les méthodes de votre DAO.

Pour ce faire, prenons pour exemple ce qui à été fait dans la classe FillDatabase :

```
String InsertQuery = "INSERT INTO film (titre, realisateur) values  
(?,?)";
```

Dans la classe FillDatabase, la variable `InsertQuery` contient la requête sql qui est utilisée pour insérer un enregistrement dans la table film. Vous pouvez voir qu'à la fin, dans cette requête, nous avons `(?,?)`.

Ces deux `?` nous permettent d'appliquer une méthode sur la String pour les changer en utilisant un PreparedStatement, ainsi nous pouvons utiliser une seule String (`InsertQuery`) pour faire toutes les insertions dont on a besoin dans la table film. Voici la création du PreparedStatement :

```
Connection connection = EstablishConnection.getConnection();  
PreparedStatement insertPreparedStatement = null;
```

puis le remplacement des `?` :

```
insertPreparedStatement = connection.prepareStatement(InsertQuery);  
insertPreparedStatement.setString(1, "Forrest Gump"); // on change le  
premier ?  
insertPreparedStatement.setString(2, "Zemeckis"); // on change le  
deuxième ?
```

Fort de cette explication, vous pouvez maintenant écrire les cinq requêtes (qui

sont de simples chaîne de caractères à l'image de `InsertQuery`) dont vous aurez besoin dans les méthodes du DAO :

- **get** : Permet de récupérer un film depuis la base de données à partir d'un id donné en argument.
- **getAll** : Permet de récupérer tous les films.
- **create** : Permet de créer un film à partir d'un objet film donné en argument.
- **update** : Permet de mettre à jour un film à partir d'un objet film donné en argument.
- **delete** : Permet de supprimer un film à partir d'un id donné en argument.

Après avoir fini d'écrire le DAO, créez un nouveau package que vous nommerez **service**.

C'est quoi un Service ?

Il gère la logique de l'application et les traitements à appliquer aux données.
(cf. le cours)

Dans ce package, créez une classe `FilmService` qui ne fera qu'instancier un objet `FilmDao` et l'utiliser. En temps normal, un Service fournit une couche d'abstraction au DAO qui permet par exemple d'ajouter une étape de vérification avant de faire appel à la méthode `create` du DAO.

Dans notre cas, le Service ne fait qu'appeler les méthodes (CRUD) de notre DAO, mais il est tout de même préférable de prendre de bonnes habitudes.

d. Troisième partie : Servlet et JSP

Maintenant créer un **package ui** dans lequel vous écrirez votre classe **Servlet**.

C'est quoi une Servlet ?

Code qui s'exécute côté serveur.

- Elle est le centre névralgique de l'application.
- Elle reçoit une requête du client, elle effectue des traitements et renvoie le résultat.
- Une servlet peut être invoquée plusieurs fois en même temps pour répondre à plusieurs requêtes simultanées.

(cf. le cours)

Lors de la création de votre classe n'oubliez de la faire hériter de **HttpServlet**, de plus n'oubliez d'y ajouter les méthodes **doGet** et **doPost** pour traiter les requêtes HTTP.

```
@WebServlet("/")
public class Servlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        // TODO
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

```
}  
}
```

C'est quoi `HttpServletRequest` et `HttpServletResponse` ?

Lorsqu'un serveur HTTP reçoit une requête, il crée deux objets qu'il transmet au conteneur de Servlets :

- ***`HttpServletRequest`*** : Contient toutes les informations relatives à la requête HTTP envoyée par le client.
- ***`HttpServletResponse`*** : Correspond à la réponse HTTP qui sera renvoyée au client par le serveur. L'objet est initialisé, mais pourra être rempli/complété lors de son passage dans la servlet avant d'être renvoyé par le serveur.

(cf. le cours)

N'oubliez pas d'y ajouter un constructeur qui initialise votre Service, c'est le conteneur web (Tomcat) qui se chargera d'y faire appel.

On va maintenant pouvoir nous occuper de notre Servlet, pour ce faire, je vous propose de séparer votre code dans des méthodes afin de ne pas avoir une méthode qui fait 100 lignes.

Commencez par coller ce code dans la méthode **doGet** :

```
String action = request.getServletPath();

switch (action) {
    case "/new":
        showAddForm(request, response);
        break;
    case "/insert":
        insert(request, response);
        break;
    case "/delete":
        delete(request, response);
        break;
    case "/edit":
        showEditForm(request, response);
        break;
    case "/update":
        update(request, response);
        break;
    case "/list":
        showAllFilm(request, response);
        break;
}
```

doGet récupère les requêtes envoyées par le client, ce code permet de faire appel à une méthode précise lorsque vous visiterez une URL :

- **insert** : Elle récupère des données écrites dans un formulaire (**FilmForm.jsp**) pour insérer un nouveau film dans la base de données.
- **update** : De la même manière que **insert** mais cette fois-ci pour modifier un film.
- **delete** : Permet de supprimer un film en récupérant son identifiant.
- **showEditForm** : Affiche la JSP qui permet de modifier un film.
- **showAddForm** : Affiche la JSP qui permet d'ajouter un film.
- **showAllFilm** : Affiche la JSP qui liste tous les films.

Vous pouvez vous lancer dans l'écriture de votre Servlet ainsi que des JSP (**prenez pour exemple la JSP fournie dans le dossier WebContent/View/**). Dans la suite de ce TP, vous trouverez des éléments de code qui vous seront utiles.

- Si vous voulez mettre en place un `forEach` dans une JSP, n'oubliez pas d'ajouter la ligne suivante en haut (avant la balise `html`) de votre JSP :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

- Récupération d'un paramètre donné dans un formulaire (par exemple dans la méthode **update**) :

```
request.getParameter("realisateur");
```

- Afficher une JSP tout en lui fournissant un attribut (un objet qu'elle peut vouloir afficher dans par exemple la méthode **showEditForm**) :

```
RequestDispatcher dispatcher =  
request.getRequestDispatcher("/WEB-INF/View/FilmForm.jsp");  
request.setAttribute("film", film);  
dispatcher.forward(request, response);
```

- Rediriger vers une JSP (par exemple pour rafraîchir une vue après une suppression dans la méthode **delete**)

```
response.sendRedirect("list");
```

- Rediriger vers une url dans un formulaire :

```
<form action="insert" method="post">  
...  
</form>
```

Les boucles dans les JSP :

```
<c:forEach var="film" items="${listFilm}">  
....  
</c:forEach>
```