CHAPITRE 6

Modèle : La base de données

Le moment est venu de mettre en place notre base de données pour compléter l'architecture de notre application. Tous ce qui suit utilise MySQL mais, à quelques détails près, on peut très bien le remplacer par PotsgreSQL ou n'importe quel autre système relationnel.

Comme point de départ nous prenons une base existante, *Films*, pour laquelle je fournis un jeu de données permettant de faire des premières expérimentations. La conception de cette base (notée en UML) est brièvement présentée. Nous verrons ensuite comment y accéder avec notre application Web grâce à l'interface d'accès à une base relationnelle en Java : JDBC. À l'issue de ce chapitre vous devriez pouvoir écrire un outil de soumission de requêtes SQL. Et vous devriez surtout être en mesure d'adopter une position critique sur l'utilisation d'une API d'assez bas niveau comme JDBC dans le cadre d'une application basée sur le motif MVC.

6.1 S1 : Installations et configurations

Supports complémentaires :
– Diapos pour la session "S1 : Installations et configurations"
– Vidéo associée : https ://avc.cnam.fr/univ-r_av/avc/courseaccess ?id=1940
Nous avons besoin de quelques installations complémentaires :
– MySQL ;
– un outil d'administration et de gestion de nos bases, phpMyAdmin ;
– les librairies de connexion Java/MySQL ;
– et une première base pour commencer sans attendre.

6.1.1 MySQL et phpMyAdmin

Pour MySQL, s'il n'est pas déjà installé dans votre environnement, je vous laisse procéder sans instruction complémentaire : vous trouverez des dizaines de ressources sur le Web pour le faire. En ce qui concerne phpMyAdmin, il vous faut une suite Apache/PHP/MySQL, soit un des environnements de développement les plus courants à l'heure actuelle. Là encore je vous laisse trouver sur le Web les instructions pour vous aider. Pour faire bref, vous devriez disposer grâce à cette installation d'un serveur Web Apache en écoute sur le port 80 (il ne sera donc pas en conflit avec Tomcat), doté d'une intégration au moteur PHP et sur lequel est installée l'application phpMyAdmin.

6.1.2 Première base de données

Dernière étape préparatoire : créer une base et lui donner un contenu initial. Nous vous proposons application de gestion de films, avec notations et réservation, comme exemple. La base de données représente des films avec leurs acteurs et metteurs en scène, et des internautes qui donnent des notes à ces films. L'application permet entre autres

d'effectuer des recommandations en fonction des notes données. Le schéma (conçu pour illustrer les principaux cas de figure des entités et associations) est donné en notation UML dans la figure *Schéma (partiel) de la base Films*.

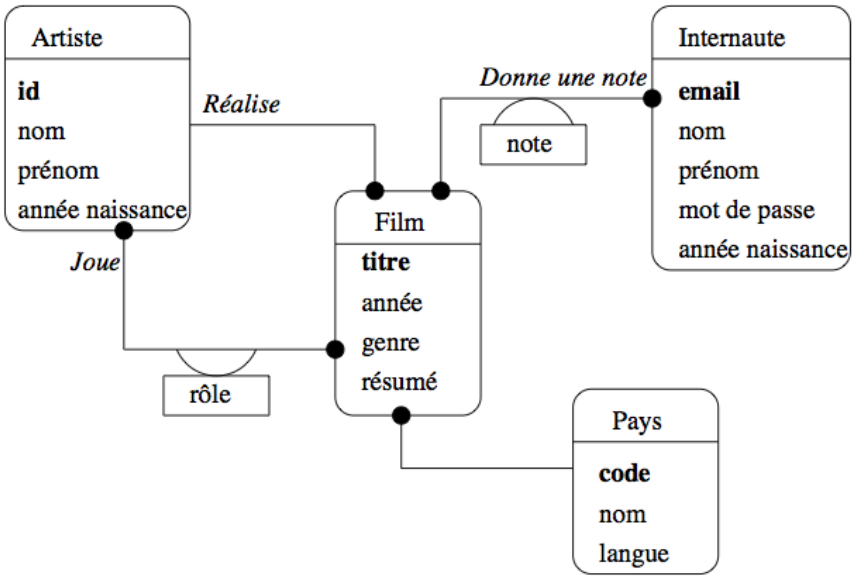

Fig. 6.1 – Schéma (partiel) de la base *Films*

Je suppose que vous avez les notions suffisantes pour interpréter ce schéma correctement. Quelques choix de simplification ont été effectués. On voit par exemple qu'un film n'a qu'un seul metteur en scène, qu'un acteur ne peut pas jouer deux rôles différents dans le même film. Les identifiants sont en général des séquences, à l'exception des internautes idéntifiés par leur *email* (ce qui n'est pas un bon choix mais va nous permettre d'étudier aussi cette situation).

Voici le schéma relationnel obtenu à partir de cette conception. Les clés primaires sont notées en **gras**, les clés étrangères en *italiques*.
– Film (**id**, titre, année, genre, résumé, *id_realisateur*, *code_pays*)
– Artiste (**id**, nom, prénom, année_naissance)
– Internaute (**email**, nom, prénom, mot_de_passe, année_naissance)
– Pays (**code**, nom, langue)
– Rôle (**id_film, id_acteur**, nom_rôle)
– Notation (**id_film**, **email**, note)
Pour créer la base, récupérez l'export SQL de la base Webscope ici : http ://orm.bdpedia.fr/files/webscope.sql. Grâce à phpMyAdmin vous pouvez importer ce fichier SQL dans votre propre système. Voici les étapes :
– à partir de la page d'accueil de phpMyAdmin, créez une nouvelle base (appelez-la *webscope* par exemple) en indiquant bien un encodage en UTF-8 ;
– cliquez sur le nom de votre base, puis allez à l'onglet *Importer* ; vous pouvez alors charger le fichier SQL `webscope.sql` que vous avez placé sur votre disque : toutes les tables (et leur contenu) seront créées.
Il faut également créer un ou plusieurs utilisateurs. Dans la fenêtre *SQL* de phpMyAdmin, entrez la commande suivante (vous êtes libres de changer le nom et le mot de passe bien sûr).

```
GRANT ALL PRIVILEGES ON webscope.* TO orm@localhost IDENTIFIED BY 'orm';
```

Après installation, vous devriez disposer avec phpMyAdmin d'une interface d'inspection et de gestion de votre base, conforme à la copie d'écran *L'interface phpMyAdmin en action sur notre base webscope*.

Je vous invite à explorer avec phpMyAdmin le schéma et les tables de la base pour vous familiariser avec eux. Faites quelques requêtes SQL pour vous préparer à les intégrer dans l'application. Par exemple :
– trouver les titres des films dirigés par Hitchcock ;
– les films parus avant 2000, avec Clint Eastwood comme acteur ;

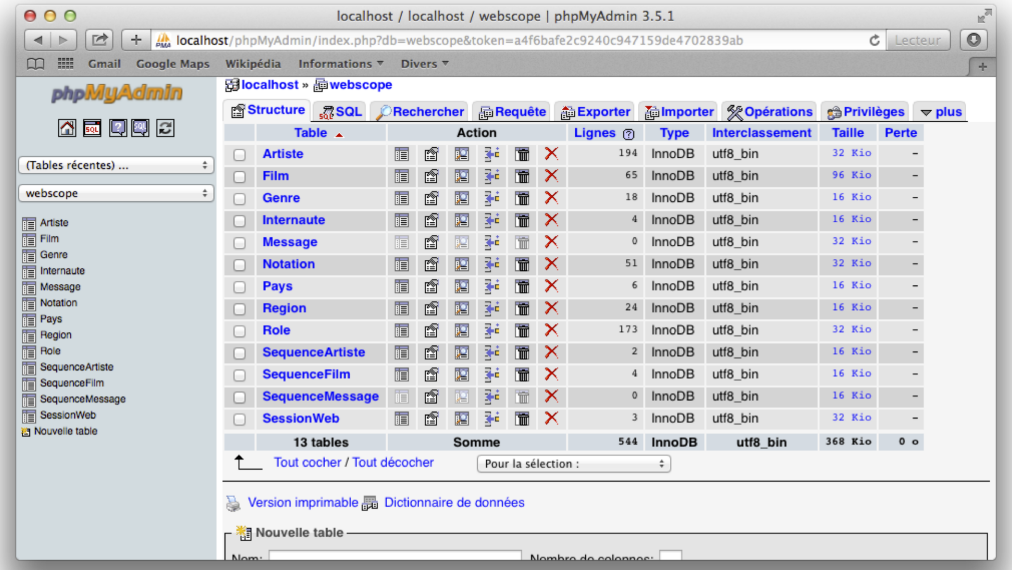

Fig. 6.2 – L'interface phpMyAdmin en action sur notre base *webscope*

– les films qui ont obtenu une note de 5.

Correction

```
- SELECT titre FROM Film
  INNER JOIN Artiste ON Film.id_realisateur=Artiste.id
  WHERE Artiste.nom='Hitchcock'

- SELECT * FROM Film as f
  INNER JOIN Role as r ON f.id=r.id_film
  INNER JOIN Artiste as a ON a.id=r.id_acteur
  WHERE a.nom='Eastwood' AND f.annee<2000;

- SELECT * FROM Film as f
  INNER JOIN Notation as n ON f.id=n.id_film
  WHERE n.note=5;
```

6.1.3 Connecteur JDBC

L'interface *Java Database Connectivity* ou JDBC est une API intégrée à la *Java Standard Edition* pour communiquer avec des bases relationnelles. Elle est censée normaliser cette communication : en principe une application s'appuyant sur JDBC peut de manière transparente passer d'une base MySQL à PostgreSQL ou à un autre système relationnel. En pratique cela suppose une certaine rigueur pour s'en tenir à la partie normalisée de SQL et éviter les extensions particulières de chaque système.

JDBC est implanté pour chaque système relationnel sous la forme d'un "pilote" (*driver*). Pour MySQL ce pilote est le *Connector/J* que l'on peut récupérer gratuitement, par exemple sur le site principal Oracle/MySQL http ://dev.mysql.com/downloads/connector/j/. Je vous laisse procéder pour obtenir ce pilote, sous la forme typiquement d'un fichier ZIP *mysql-connector-java-xx.yy.zz.zip*. En décompressant cette archive vous obtenez, entre autres,

un fichier JAR *mysql-connector-java-xx.yy.zz-bin.jar*. *Copiez ce fichier dans le répertoire ''WEB-INF/lib'' de votre application*. L'API JDBC devient alors disponible.

Note : Pour des librairies si courantes qu'elles sont utilisées dans beaucoup d'applications, il peut être profitable de les associer directement à Tomcat en les plaçant dans *TOMCAT_HOME/lib*. Cela évite de les réinclure systématiquement dans le déploiement de chaque application.

Vous trouverez également dans le sous-répertoire *doc* une documentation complète sur le connecteur JDBC de MySQL (applicable dans une large mesure à un autre système relationnel). Mettez cette documentation de côté pour vous y reporter car nous n'allons pas tout couvrir dans ce qui suit.

6.2 S2 : introduction à JDBC

Supports complémentaires :
– Diapos pour la session "S2 : Introduction à JDBC"
– Vidéo associée : https ://avc.cnam.fr/univ-r_av/avc/courseaccess ?id=1939
Nous allons découvrir JDBC par l'intermédiaire d'un ensemble d'exemples accessibles par notre application Web dans un unique contrôleur, que nous appelons *Jdbc* pour faire simple. Ce contrôleur communiquera lui-même avec un objet-modèle `TestJdbc` qui se chargera des accès à la base. Nous restons toujours dans un cadre MVC, avec l'exigence d'une séparation claire entre les différentes couches.

6.2.1 Contrôleur, vues et modèle

Chaque exemple JDBC correspondra à une action du contrôleur, et en l'absence d'action nous afficherons une simple page HTML présentant le menu des actions possibles. Nous suivons les règles définies à la fin du chapitre *Modèle-Vue-Contrôleur (MVC)*. Voici l'essentiel du code de cette *servlet*.

```
package controleurs;

import java.sql.*;

    /**
    * Servlet implementation class Jdbc
    */
    @WebServlet("/jdbc")
    public class Jdbc extends HttpServlet {

      private static final String SERVER="localhost", BD="webscope",
                    LOGIN="orm", PASSWORD="orm", VUES="/vues/jdbc/";

      protected void doGet(HttpServletRequest request,
                    HttpServletResponse response) throws ServletException, IOException {

            // On devrait récupérer l'action requise par l'utilisateur
            String action = request.getParameter("action");
            // Notre objet modèle: accès à MySQL
            TestsJdbc jdbc;
            // La vue par défaut
            String maVue = VUES + "index.jsp";

            try {
                    jdbc = new TestsJdbc();

                    if (action == null) {
```

```
                        // Rien à faire
                } else if (action.equals("connexion")) {
                        // Action + vue test de connexion
                        jdbc.connect(SERVER, BD, LOGIN, PASSWORD);
                        maVue = VUES + "connexion.jsp";
                } else if (action.equals("requeteA")) {
                    // Etc...
                }
            } catch (Exception e) {
                    maVue = VUES + "exception.jsp";
                    request.setAttribute("message", e.getMessage());
            }

            // On transmet à la vue
            RequestDispatcher dispatcher = getServletContext()
                            .getRequestDispatcher(maVue);
            dispatcher.forward(request, response);
        }
}
```

Notez l'import de `java.sql.*` : toutes les classes de JDBC. L'ensemble des instructions contenant du code JDBC (encapsulé dans `TestJdbc`) est inclus dans un bloc `try` pour capturer les exceptions éventuelles.

Nous avons également déclaré des variables statiques pour les paramètres de connexion à la base. Il faudra faire mieux (pour partager ces paramètres avec d'autres contrôleurs), mais pour l'instant cela suffira.

Voici maintenant la page d'accueil du contrôleur (conventionnellement nommée `index.jsp`).

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
       pageEncoding="UTF-8"%>

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

     <html>
     <head>
     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
     <title>Menu JDBC</title>
     </head>
     <body>

       <h1>Actions JDBC</h1>

       <ul>
             <li><a
                    href="${pageContext.request.contextPath}/jdbc?action=connexion">Connexion</a>
             <li><a
                    href="${pageContext.request.contextPath}/jdbc?action=requeteA">RequêteA</a>
             </li>
             <li>...</li>
       </ul>

     </body>
</html>
```

Et pour finir, nous avons besoin d'un *modèle* qui se chargera en l'occurrence de se connecter à MySQL et d'implanter les différentes actions de test des fonctionnalités JDBC. Appelons-le `TestsJdbc`, et plaçons-le dans le *package* `modeles`. Voici le squelette de cette classe, que nous détaillerons ensuite.

```
package modeles;

import java.sql.*;

public class TestsJdbc {
  private static final Integer port = 3306;

  /**
   * Pour communiquer avec MySQL
   */
  private Connection connexion;

  /**
   * Constructeur sans connexion
   */
 public TestsJdbc() throws ClassNotFoundException {
        /* On commence par "charger" le pilote MySQL */
        Class.forName("com.mysql.jdbc.Driver");
  }
  (...)
}
```

Nous avons un objet privé, `connexion`, instance de la classe JDBC `Connection`. L'interface JDBC est abstraite : pour communiquer avec un serveur de données, il faut charger le pilote approprié. C'est ce que fait l'instruction :

```
Class.forName("com.mysql.jdbc.Driver");
```

Si vous avez bien placé le `jar` de MySQL dans le répertoire `WEB-INF/lib`, le chargement devrait s'effectuer correctement, sinon vous aurez affaire à une exception `ClassNotFoundException`. Voyons maintenant, pas à pas, comment développer des accès JDBC. Je vous invite à intégrer chacune des méthodes qui suit dans votre code, et à les tester sur le champ.

6.2.2 Connexion à la base

La première chose à faire (après le chargement du pilote) est la connexion à la base avec un compte utilisateur. Voici notre méthode `connect`.

```
public void connect(String server, String bd, String u, String p)
                throws SQLException {
        String url = "jdbc:mysql://" + server + ":" + port + "/" + bd;
        connexion = DriverManager.getConnection(url, u, p);
}
```

Quatre paramètres sont nécessaires (cinq si on compte le port qui est fixé par défaut à 3306).
– le nom du serveur : c'est pour nous `localhost`, sinon c'est le nom de domaine ou l'IP de la machine hébergeant le serveur MySQL ;
– le nom de la base
– le nom d'utilisateur MySQL (ici variable `u`) et le mot de passe (ici variable `p`).
Les deux premiers paramètres sont intégrés à une chaîne de connexion dont le format est lisible dans le code ci-dessus. Cette chaîne comprend notamment le serveur, le port et le nom de la base de données. Dans notre cas ce sera par exemple :

```
jdbc:mysql://locahost:3306/webscope
```

À vous de jouer. En affichant la vue par défaut (`index.jsp`) et en cliquant sur le choix `Connexion`, vous devriez déclencher l'action qui exécute cette méthode `connect`. Deux résultats possibles :

– soit tout se passe bien (c'est rarement le cas à la première tentative...) et la vue est `connexion.jsp` ;
– soit une exception est levée, et c'est la vue `exception.jsp` qui est choisie par le contrôleur ; elle devrait afficher le message d'erreur.

Prenez votre temps pour bien assimiler, si ce n'est déjà fait, les mécanismes d'interaction HTTP mis en place ici. Il est indispensable que vous familiarisiez avec les séquences de vue-contrôleur-action-modèle typiques des applications Web. Les fragments de code donnés précédemment sont presque complets, à vous des les finaliser (par exemple en créant la page `exception.jsp` qui affiche les messages d'exception).

Quand la connexion s'effectue correctement nous pouvons passer à la suite.

6.2.3 Exécution d'une requête

Nous pouvons en venir à l'essentiel, à savoir accéder aux données de la base. C'est ici que nous allons être confronté au fameux *impedance mismatch*, terme établi (quoique peu compréhensible) pour désigner l'incompatibilité de représentation des données entre une base relationnelle et une application objet. Concrètement, cette incompatibilité nécessite des conversions répétitives, et leur intégration dans une architecture générale (type MVC) s'avère laborieuse. Démonstration par l'exemple dans ce qui suit.

Le mécanisme *d'interrogation* de JDBC est assez simple :
– on instancie un objet `statement` par lequel on exécute une requête SQL ;
– l'exécution renvoie un objet `ResultSet` par lequel on peut parcourir le résultat.

C'est la méthode basique, avec des requêtes fixes dites "non préparées". En première approche, nous ajoutons une méthode `chercheFilmA` dans `TestJdbc.java`.

```
public ResultSet chercheFilmsA() throws SQLException
{
        Statement statement = connexion.createStatement();
        return statement.executeQuery( "SELECT * FROM Film");
}
```

Comme vous pouvez le voir cette méthode fait peu de choses (ce qui est mauvais signe car le travail est délégué aux autres composants dont ce n'est pas le rôle). Elle retourne l'objet `ResultSet` créé par l'exécution de `SELECT * FROM Film`. Dans le contrôleur cette méthode est appelée par l'action `requeteA` que voici.

```
if (action.equals("requeteA")) {
        ResultSet resultat = jdbc.chercheFilmsA();
        request.setAttribute("films", resultat);
        maVue = "/vues/jdbc/filmsA.jsp";
}
```

Cela impose donc d'importer les classes JDBC `java.sql.*` dans le contrôleur. Si vous avez bien assimilé les principes du MVC vous devriez commencer à ressentir un certain inconfort : nous sommes en train de propager des dépendances (la gestion d'une base de données) dans des couches qui ne devraient pas être concernées.

Le pire reste à venir : voici le vue `filmsA.jsp`.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>

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

<%@ page import="java.sql.*" %>

<html>
 <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Liste des films</title>
```

```
 </head>
 <body>

      <h1>Liste des films</h1>

      <ul>
      <%
       ResultSet films = (ResultSet) request.getAttribute("films");
        while (films.next()) {
                out.println ("<li>Titre du film: " + films.getString("titre") + "</li>");
        }
      %>
      </ul>

      <p>
              <a href="${pageContext.request.contextPath}/jdbc">Accueil</a>
      </p>

 </body>
</html>
```

Rien ne va plus ! Nous avons été obligés d'introduire de la programmation Java dans notre vue. J'espère que vous êtes conscients des forts désavantages à long terme de cette solution. Concrètement, la gestion des accès JDBC est maintenant répartie entre le modèle, la vue et le contrôleur. Tout changement dans une couche implique des changements dans les autres : nous avons des *dépendances* qui compliquent (ou compliqueront) très sérieusement la vie de tout le monde à court ou moyen terme.

Comment faire mieux ? En restreignant strictement l'utilisation de JDBC au modèle. Après avoir vérifié que le code ci-dessus fonctionne (malgré tout), passez à la seconde approche ci-dessous.

6.2.4 Requête : deuxième approche

Voici une seconde méthode d'exécution de requête :

```
public List<Map<String, String>> chercheFilmsB() throws SQLException
{
        List<Map<String, String>> resultat = new ArrayList<Map<String, String>>();

        Statement statement = connexion.createStatement();
        ResultSet films = statement.executeQuery( "SELECT * FROM Film");
        while (films.next()) {
                Map<String, String> film = new HashMap<String, String> ();
                film.put("titre", films.getString("titre"));
                resultat.add(film);
        }
        // Et on renvoie
        return resultat;
}
```

Cette fois on ne renvoie plus un objet `ResultSet` mais une structure Java (que je vous laisse décrypter) permettant de représenter le résultat de la requête sans dépendance à JDBC. Vous voyez maintenant plus clairement le mécanisme de parcours d'un résultat avec JDBC : on boucle sur l'appel à la méthode `next()` de l'objet `ResultSet` qui se comporte comme un itérateur pointant successivement sur les lignes du résultat. Des méthodes `getInt()`, `getString()`, `getDouble()`, etc., récupèrent les valeurs des champs en fonction de leur type.

Le code du contrôleur change peu (je vous laisse l'adapter du précédent, à titre d'exercice). Voici en revanche la nouvelle version de la vue (en se restreignant au `body`) :

```
<h1>Liste des films</h1>

<ul>
  <c:forEach items="${requestScope.films}" var="film">
    <li> <c:out value="${film.titre}"/> </li>
  </c:forEach>
</ul>

<p>
        <a href="${pageContext.request.contextPath}/jdbc">Accueil</a>
</p>
```

Nous nous appuyons cette fois sur les balises de la JSTL, ce qui est permis par l'utilisation de structures Java standard en lieu et place du `ResultSet`. Le code est infiniment plus simple (j'espère que vous êtes d'accord) et surtout complètement indépendant de toute notion d'accès à une base de données. Il est possible par exemple de le tester sans recourir à MySQL.

Exercice : afficher les autres champs de la table *Film*

Vous ne devriez pas avoir de difficulté à compléter le code ci-dessus pour afficher tous les champs de la table Film : année, code du pays, etc.

Une fois l'exercice accompli, je vous propose une dernière version de cette même fonction.

6.2.5 Requête : troisième approche

La méthode qui précède est acceptable, mais elle souffre d'une limitation évidente : nous manipulons des structures (listes et tables de hachage) pour réprésenter des *entités* de notre domaine applicatif. Il serait bien plus logique et satisfaisant de représenter ces entités comme des *objets*, avec tous les avantages associés, comme l'encapsulation des propriétés, la possibilité d'associer des méthodes, la navigation vers d'autres objets associés (les acteurs, le metteur en scène), etc.

Pour cela, nous allons passer à une dernière étape : introduire une classe `Film` dans notre modèle, et modifier `TestJdbc` pour renvoyer des instances de `Film`.

Une version réduite de la classe `Film` est donnée ci-dessous. C'est un *bean* très classique.

```
package modeles;

/**
* Encapsulation de la représentation d'un film
*
 */
public class Film {

  private String titre;
  private Integer annee;

  public Film() {}

  public void setTitre(String leTitre) { titre = leTitre;}
  public void setAnnee(Integer lAnnee) { annee = lAnnee;}

  public String getTitre() {return titre;}
  public Integer getAnnee() {return annee;}
}
```

Et voici donc la méthode de notre classe JDBC.

```
public List<Film> chercheFilmsC() throws SQLException
{
        List<Film> resultat = new ArrayList<Film>();

        Statement statement = connexion.createStatement();
        ResultSet films = statement.executeQuery( "SELECT * FROM Film");
        while (films.next()) {
                Film film = new Film ();
                film.setTitre(films.getString("titre"));
                film.setAnnee(films.getInt("annee"));
                resultat.add(film);
        }
        // Et on renvoie
        return resultat;
}
```

Nous renvoyons une liste de films, et tout ce qui concerne la base de données (à part les exceptions éventuellement levées) est confiné à notre objet-service JDBC. Je vous laisse compléter le contrôleur et la vue. Ils changent très peu par rapport à la version précédente, ce qui confirme que nous sommes arrivés à un niveau d'indépendance presque total.

Tout cela demande un peu plus de travail que la version "basique", et un développeur peu consciencieux pourrait utiliser cet argument pour justifier de recourir au "vite fait/mal fait". Il est vrai qu'il faut définir une classe supplémentaire, et appliquer des fastidieux `get` et `set` pour chaque propriété (la fameuse *conversion* nécessitée par l'incompatibilité des représentations dont nous parlions au début de cette session). Le bénéfice n'apparaît qu'à long terme. *Sauf* que nous allons bientôt voir comment nous épargner ces tâches répétitives, et qu'il n'y aura absolument plus d'excuse à ne pas faire les choses proprement.

Exercice : comme avant, afficher les autres champs de la table *Film*

Compléter le code donné ci-dessus pour gérer complètement la table *Film*, y compris l'identifiant du réalisateur, `id_realisateur`.

Exercice : écrire une classe Artiste et des méthodes de lecture

Ecrivez une classe Artiste pour représenter les artistes. Dans la classe `testJdbc`, écrivez une méthode :

```
Artiste chercheArtiste (Integer idArtiste)
```

C'est presque comme la méthode `chercheFilmsC`, mais on renvoie un seul objet. Utilisez la connexion JDBC, construisez la requête qui recherche l'artiste dont l'id est `idArtiste`, instantiez un objet `Artiste` avec les valeurs obtenues par la requête, renvoyez-le.

Exercice : permettre d'accéder au metteur en scène du Film

Voici maintenant un exercice complémentaire très instructif : à partir d'un film, nous aimerions récupérer son metteur en scène. Concevez une extension de notre petite application qui respecte soigneusement les principes exposés jusqu'ici. Vous devriez pouvoir utiliser le code produit dans les exercices précédents, la question préalable étant : *comment représenter l'association entre un film et son réalisateur ?*.

Le dernier exercice devrait déboucher sur quelques questions importantes relatives à la représentation d'une base de données relationnelles sous forme de graphe d'objet. Cette transformation (*mapping*) objet-relationnel apparaît très séduisante en terme d'ingénierie logicielle, mais demande cependant la mise en place de bonnes pratiques, et une surveillance sourcilleuse des accès à la base effectués dans les couches profondes de notre application. À suivre !

6.3 Résumé : savoir et retenir

Dans toute application, l'accès à une base de données relationnelle se fait par l'interface fonctionnelle (API) du SGBD. Dans le cas des applications Java cette interface (JDBC) est normalisée. Mais les mécanismes restent les mêmes quel que soit le langage et le système de bases de données utilisé : on effectue une requête, on récupère un curseur sur la résultat, ou itère sur ce curseur en obtenant à chaque étape une structure (typiquement un tableau) représentant une ligne de la table.

Dans une application objet, nous avons modélisé notre domaine fonctionnel par des classes (en UML par exemple), et les instances de ces classes (des objets donc) doivent être rendus persistants par stockage dans la base relationnelle. On se retrouve donc dans la situation de convertir des objets en lignes, et réciproquement de convertir des lignes ramenées par des requêtes en objets. C'est ce que nous avons fait dans ce chapitre : à un moment donné, on se retrouve dans la situation d'avoir à copie un tableau de données issu du `ResultSet` vers un objet. Cette copie est fastidieuse à coder et très répétitive.

Vous devriez, à l'issue de ce chapitre, être sensibilisé au problème et convaincu de deux choses :
– dans une application objet, tout doit être objet ; la couche de persistance, celle qui accède à la base de données, doit donc renvoyer des objets : elle est en charge de la conversion, et c'est l'approche finale à laquelle nous sommes parvenus ;
– à l'échelle d'une application non triviale (avec des dizaines de tables et des modèles complexes), la méthode consistant à coder manuellement les conversions est très pénalisante pour le développement et la maintenance ; une méthode automatique est grandement souhaitable.

Réflechissez bien à ces deux aspects : quand ils sont intégrés vous êtes mûrs pour adopter une solution ORM, que nous abordons dans le prochain chapitre.