Applications orientées données (NSY135)

6 – Modèle: La base de données

Auteurs: Raphaël Fournier-S'niehotta et Philippe Rigaux
(philippe.rigaux@cnam.fr,fournier@cnam.fr)

Département d'informatique
Conservatoire National des Arts & Métiers, Paris, France

Plan du cours

Introduction

- 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.
- Chaque exemple JDBC correspondra à une action du contrôleur,
- 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 MVC.

Servlet

```java
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);
  }
  }
```

Servlet (suite)

- 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 faudrait faire mieux (pour partager ces paramètres avec d'autres contrôleurs), mais pour l'instant cela suffira.

Contrôleur : 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>
```

Modèle : TestJdbc

- Se charge de la connexion à MySQL
- implantation des différentes actions de test des fonctionnalités JDBC

```
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");
  }
  (...)
}
```

Modèle (suite)

- 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".

Connexion à la base

```
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);
}
```

- Paramètres
  - 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 port, par défaut fixé à 3306
  - le nom de la base
  - le nom d'utilisateur MySQL (ici variable *u*)
  - le mot de passe (ici variable *p*).
- Exemple

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

- affichons maintenant la vue par défaut (`index.jsp`) et cliquons sur Connexion pour déclencher l'action qui exécute cette méthode connect

Requête : première approche

- On arrive à l'essentiel : accéder aux données de la base
- Et l'on se confronte au fameux **impedance mismatch**, l'incompatibilité de représentation des données entre une base relationnelle et une application orientée objet (OO)
- C'est cette incompatibilité qui nécessite des conversions répétitives et rend leur intégration dans une architecture générale (type MVC) laborieuse

Mécanisme d'interrogation

- Mécanisme assez simple (avec des requêtes "non préparées"):
  - 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.
- Ajout de "chercheFilmA" dans "TestJdbc.java"

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

- Peu de choses sont effectuées par cette méthode
- Elle retourne l'objet "ResultSet" créé par l'exécution de "SELECT * FROM Film"

Mécanisme d'interrogation

- Retourner ce ResultSet est mauvais
- Cela implique que le travail est délégué aux autres composants dont ce n'est pas le rôle
- Dans le contrôleur cette méthode est appelée par l'action "requeteA" :

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

- impose donc d'importer les classes JDBC "java.sql.*" dans le contrôleur (propagation de dépendances dans diverses couches)

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>
```

Vue (suite)

- Nous avons été obligés d'introduire de la programmation Java dans notre vue !
- Pire, 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 très sérieusement la maintenance du code à court ou moyen terme.
- On va faire mieux, en restreignant strictement l'utilisation de JDBC au modèle.

Requête : deuxième approche

```java
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;
}
```

- on ne renvoie plus un objet `ResultSet` mais une structure Java
- permettant de représenter le résultat de la requête sans dépendance à JDBC
- le mécanisme de parcours d'un résultat avec JDBC est
  - on boucle sur l'appel à la méthode `next()` de l'objet `ResultSet` (itérateur pointant successivement sur les lignes du résultat)
  - Des méthodes `getInt()`,`getString()`, `getDouble()`, etc., récupèrent les valeurs des champs en fonction de leur type

Nouvelle version de la vue

- il faut mettre à jour le contrôleur

```
<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>
```

- on s'appuie cette fois sur les balises de la JSTL
- possible grâce à l'emploi de structures Java standard plutôt que "ResultSet"
- Remarquez la soimplicité du code
- et l'indépendance de la notion d'accès à la BDD

Requête : troisième approche

- limitation évidente de la méthode précédente
- nous manipulons des structures (listes et tables de hachage), pour représenter des **entités** de notre domaine applicatif
- on préférerait manipuler ces entités comme des **objets** :
  - encapsulation des propriétés
  - possibilité d'associer des méthodes
  - navigation vers d'autres objets associés (acteurs, metteur en scène)
- on va introduire une classe Film, un *bean* classique, et modifier `TestJdbc` pour renvoyer des instances de `Film`

Film

```java
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;}
}
```

JDBC modifié

```
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;
}$
```

3e approche (suite)

- On renvoie donc une liste de films
- tout ce qui concerne la base de données est confiné dans l'objet-service JDBC
- en complétant le contrôleur et la vue, on constate qu'ils changent très peu par rapport à la version précédente
- le niveau indépendance est presque total
- cela demande un peu plus de travail que la version "basique"
  - définir une classe supplémentaire
  - appliquer des fastidieux `get` et `set` pour chaque propriété pour les conversions (**incompatibilités**)
- Nous allons bientôt voir comment nous épargner ces tâches répétitives (plus d'excuses pour ne pas faire les choses proprement !)

Plan du cours

Résumé

- Dans toute application, l'accès à une base de données relationnelle se fait par l'interface fonctionnelle (API) du SGBD.
- En Java, l'interface JDBC normalise ces accès
- 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
  - on itère sur ce curseur en obtenant à chaque étape une structure (typiquement un tableau) représentant une ligne de la table
- Notre application est orientée objet
- nous avons modélisé notre domaine fonctionnel par des classes
- et les instances de ces classes (des objets) doivent être rendus persistants par stockage dans la base relationnelle
- on doit donc souvent convertir des objets en lignes, et réciproquement de convertir des lignes ramenées par des requêtes en objets

Résumé (suite)

Deux derniers points :

- 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.
- c'est le rôle d'une solution ORM, abordée dans le prochain chapitre