

ENVIRONNEMENT CLIENT/SERVEUR

Cours 8

Modèle MVC

JavaServer Pages (2)

kn@lri.fr



- 1 Rappels sur les systèmes d'exploitations / Communication par mémoire partagée ✓
- 2 Réseaux généralités, IP, UDP ✓
- 3 Multi-thread en Java, TCP ✓
- 4 Broadcast UDP et étude de cas ✓
- 5 Rappels Java/Java 8/Sérialisation ✓
- 6 Rappels Java (suite)/Appel de méthode à distance ✓
- 7 JavaServer Pages (1) ✓
- 8 Modèle MVC/JavaServer Pages (2)
 - 8.1 Modèle MVC
 - 8.2 Modèle : Rappels sur JDBC
 - 8.3 Contrôleur : HttpServlet
 - 8.4 Vue : JSP et JSTL

Qu'est-ce que le modèle MVC ?



C'est un *design pattern* qui permet de modéliser des applications « interactives » :

- L'application possède un état interne
- Un « utilisateur » (ça peut être un programme externe) interagit avec le programme pour modifier l'état interne
- L'application affiche à l'utilisateur le résultat de son opération

Ces trois aspects sont représentés par trois composants :

- Le Modèle (représentation de l'état interne)
- La Vue (affichage du modèle)
- Le Contrôleur (modification du modèle)

En quoi est-ce adapté aux applications Web ?



Une application Web typique :

- Présente au client un formulaire permettant de passer des paramètres (C)
- Effectue des opérations sur une base de donnée (M) à partir des paramètres
- Affiche une page Web montrant le résultat de l'opération (V)

Avantages du Modèle MVC ?



La séparation permet d'obtenir :

Maintenance simplifiée : Le code d'une action est centralisé à un seul endroit

Séparation des privilèges : Pas besoin que la vue ai un accès à la base de donnée
par exemple

Test simplifié : Les composants peuvent être testés indépendamment



Dans une application JSP typique, le modèle MVC peut être implémenté de la manière suivante :

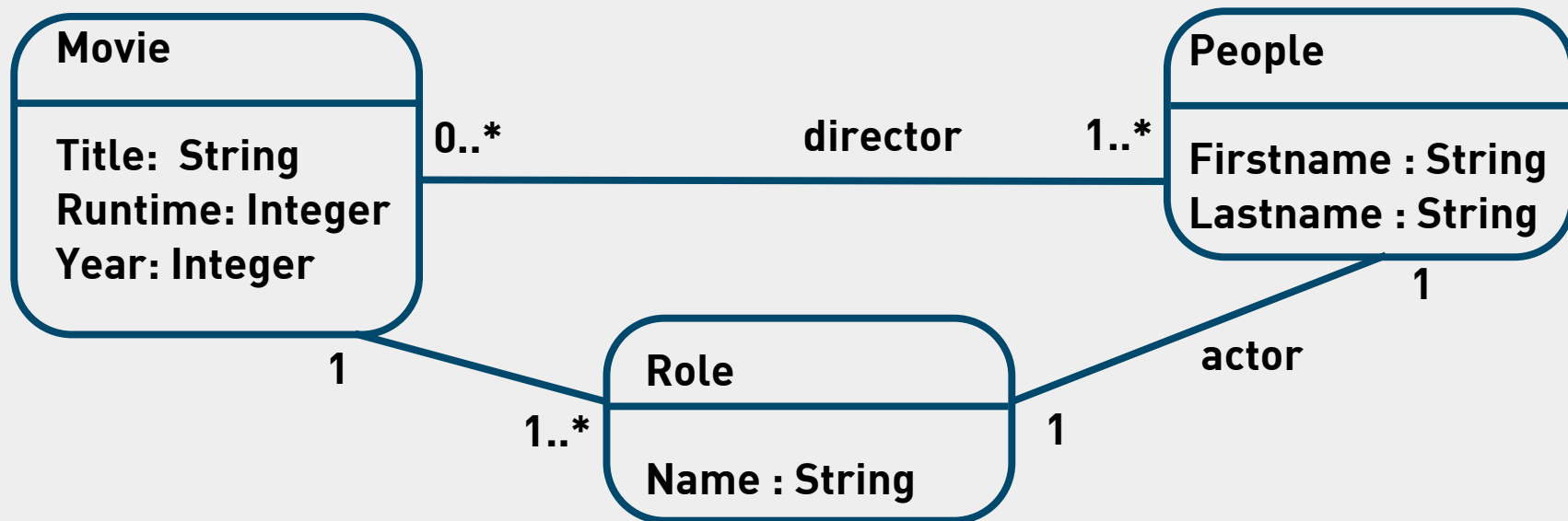
- Le modèle est représenté par des classes qui encapsulent les appels à une base de données et présentent les résultats sous forme de POJO (*Plain Old Java Objects*).
- Le contrôleur est représenté par des `HttpServlets`. Ce sont des classes Java dont le but est de : récupérer les paramètres utilisateur et les valider, appeler les opérations du Modèle avec ses paramètres, passer le résultat de l'opération à la Vue.
- La vue est un ensemble de page JSP qui affichent les résultats. Elles contiennent le moins possible de code Java et ne doivent se soucier que de l'affichage

- 1 Rappels sur les systèmes d'exploitations / Communication par mémoire partagée ✓
- 2 Réseaux généralités, IP, UDP ✓
- 3 Multi-thread en Java, TCP ✓
- 4 Broadcast UDP et étude de cas ✓
- 5 Rappels Java/Java 8/Sérialisation ✓
- 6 Rappels Java (suite)/Appel de méthode à distance ✓
- 7 JavaServer Pages (1) ✓
- 8 Modèle MVC/JavaServer Pages (2)
 - 8.1 Modèle MVC ✓
 - 8.2 Modèle : Rappels sur JDBC
 - 8.3 Contrôleur : HttpServlet
 - 8.4 Vue : JSP et JSTL

Une base de données de films



On se place dans le cadre d'une application permettant d'accéder à une base de données de films. Le schéma *logique* de la base est représenté par le diagramme UML suivant :



- Un film peut avoir plusieurs réalisateurs, et au moins un;
- Un film a au moins un « acteur ». Un acteur est composé d'un rôle *pour ce film* et d'une personne. Attention, le *Role* de Luke Skywalker est différent pour les films Starwars IV, V, VI (même si c'est le même acteur et le même nom de rôle).

Encodage « naturel » en SQL (1/2)



```
CREATE TABLE PEOPLE (pid INTEGER, firstname VARCHAR(30),  
                        lastname VARCHAR(30),  
                        PRIMARY KEY(pid));
```

```
CREATE TABLE MOVIE (mid INTEGER, title VARCHAR(90) NOT NULL,  
                      year INTEGER NOT NULL,  
                      runtime INTEGER NOT NULL, rank INTEGER NOT NULL,  
                      PRIMARY KEY (mid));
```

```
CREATE TABLE ROLE (mid INTEGER, pid INTEGER, name VARCHAR(70),  
                     PRIMARY KEY(mid, pid, name),  
                     FOREIGN KEY (mid) REFERENCES MOVIE,  
                     FOREIGN KEY (pid) REFERENCES PEOPLE);
```

```
CREATE TABLE DIRECTOR (mid INTEGER, pid INTEGER, PRIMARY KEY (mid, pid),  
                          FOREIGN KEY (mid) REFERENCES MOVIE,  
                          FOREIGN KEY (pid) REFERENCES  
                          PEOPLE);
```

Encodage « naturel » en SQL (2/2)



- entités $\Rightarrow$ tables de données
- relations $\Rightarrow$ tables de jointure & contraintes de clé
- (économie d'une table de données pour **ROLE.name**)

On utilisera ce schema dans le TP d'aujourd'hui et le TP noté de la séance 10.

Ici on veut faire une mini-application simplifiée pour rechercher toutes les personnes qui contiennent une chaîne donnée dans leur nom, trié par ordre alphabétique.

On va créer deux classes Java pour le modèle :

Person : Une classe représentant une personne, avec son nom et son prénom

PersonDB : Une classe encapsulant la connexion à la base et permettant de renvoyer l'ensemble (Java) de toutes les personnes de la base ayant une certaine chaîne dans son nom.

```
public class Person {  
    private final String firstname;  
    private final String lastname;  
  
    public Person(String f, String l) {  
        this.firstname = f;  
        this.lastname = l;  
    }  
    public String getFirstname() { return firstname; }  
    public String getLastName() { return lastname; }  
}
```

Classe Person



- Doit être publique
- Doit posséder des *getter* publiques pour les attributs qu'on veut afficher dans la vue

Le *getter* pour une propriété `foo` est une méthode publique `getFoo()`

Classe PersonDB et rappels JDBC



```
public class PersonDB {  
    Connection cnx;  
    public PersonDB() {  
        Class.forName("org.postgresql.Driver");  
        cnx = DriverManager.getConnection(  
            "jdbc:postgresql://host:port/base",  
            "username", "password");  
    }  
    public Vector<Person> getPersons (String s) throws SQLException {  
        Vector<Person> res = new Vector<>();  
        Statement st = cnx.createStatement();  
        ResultSet r = st.executeQuery("SELECT * FROM PEOPLE "  
            + " WHERE LASTNAME LIKE "  
            + " '%" + s + "%'");  
        while (r.next())  
            res.add(new Person<>(r.getString("FIRSTNAME"),  
                r.getString("LASTNAME")));  
        return res;  
    }  
}
```

```
Class.forName("org.postgresql.Driver");  
connection =  
DriverManager.getConnection("jdbc:postgresql://host:port/" + base,  
    username, password);
```

- On importe dans la JVM courante le classe qui code le driver vers une base de donnée (ici Postgresql)
- Le code de cette classe doit se trouver dans un .jar ou .class accessible depuis le CLASSPATH
- La classe DriverManager maintient une Map entre chaîne de caractères ("jdbc:postgresql") et classe (org.postgresql.Driver)
- La méthode getConnection utilise le préfixe de l'URL de connexion pour savoir quel driver utiliser.

Exécution de requêtes



- On crée un objet Statement
- L'évaluation d'une requête se fait via executeQuery sur le Statement.
- Un ResultSet implémente une interface d'itérateur, initialement positionné avant la première ligne de résultats.
- La méthode next avance dans l'itérateur et renvoie vrai tant qu'on est sur un résultat.
- On accède à la colonne voulue avec getType. On doit donner le type Java correspondant au type SQL de la colonne. On peut accéder aux colonnes par numéro (à partir de 1) ou par nom.

Remarque : On devrait plutôt utiliser un PreparedStatement pour éviter l'injection de code. Ici on va utiliser le contrôleur pour valider la chaîne (exemple) mais ce n'est pas idéal

- 1 Rappels sur les systèmes d'exploitations / Communication par mémoire partagée ✓
- 2 Réseaux généralités, IP, UDP ✓
- 3 Multi-thread en Java, TCP ✓
- 4 Broadcast UDP et étude de cas ✓
- 5 Rappels Java/Java 8/Sérialisation ✓
- 6 Rappels Java (suite)/Appel de méthode à distance ✓
- 7 JavaServer Pages (1) ✓
- 8 Modèle MVC/JavaServer Pages (2)
 - 8.1 Modèle MVC ✓
 - 8.2 Modèle : Rappels sur JDBC ✓
 - 8.3 Contrôleur : **HttpServlet**
 - 8.4 Vue : JSP et JSTL

La classe `HttpServlet` permet d'implémenter le contrôleur. C'est vers cette classe que sont compilées les pages JSP, mais dans le contrôleur, on ne va faire aucun affichage, mais calculer un résultat et le stocker pour que la vue puisse l'afficher.

Le contrôleur (1)



```
//Cette annotation permet de dire que le Servlet sera
//associée à l'URL /APPNAME/PersonListServlet
@WebServlet("/PersonListServlet")
public class PersonListServlet extends HttpServlet {

    //Tomcat se sert de l'ID Pour savoir qu'un servlet a été
    //modifié et donc que la version en cache doit être invalidée
    private static final long serialVersionUID = 1234L;

    //La méthode appelée si la requête est POST
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
                          throws ServletException, IOException {
        doGet(request, response);
    }
    //La méthode appelée si la requête est GET
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
                          throws ServletException, IOException {
        ...
    }
}
```

Le contrôleur (2)



- Comme la classe Java ne correspond pas à un fichier JSP, il faut donner (via une **annotation**) un nom de fichier/URL virtuel pour appeler cette classe
- Pour chaque **type** de requête HTTP (POST, GET, ...) redéfinit une méthode **doXXX** (où XXX vaut Get, Post, ...)
- Les requêtes ne peuvent lever que des **ServletException** ou **IOException**
- Ces méthodes prennent en argument une **request** (la requête HTTP que nous a amené sur ce servlet) et un **response** (qui sera envoyé au client)

Dans notre exemple on dit que si on est appelé en **POST** alors on fait la même chose qu'en **GET**.

Le contrôleur (3) : la méthode doGet



```
try {
    //On crée un Modèle et on le stocke dans la session
    PersonDB db = (PersonDB) request.getSession().getAttribute("db");
    if (db == null) {
        db = new PersonDB();
        request.getSession().setAttribute("db", db);
    }

    //Récupération du paramètre GET comme dans un JSP
    String s = request.getParameter("s");

    //Netoyage de la chaîne
    String ss = s.replaceAll("([%_0-9;,]|--)+", "");

    Vector<Person> v = db.getPersons(ss);
    request.setAttribute("people", v);

    RequestDispatcher rd =
        request.getRequestDispatcher("/person_list.jsp");

    rd.forward(request, response);
} catch (Exception e) { throw new ServletException(e); }
```

Le contrôleur (4)



- L'objet request permet d'accéder à la session et à l'application
- On stocke le modèle (qui contient une connexion à la base dans la session)
- On récupère le modèle et on l'appelle
- On stocke les résultats dans un attribut de requête (pour que la page JSP de la vue puisse y accéder)
- On récupère un `RequestDispatcher` qui permet de charger une pages JSP (donc une vue) particulière à la fin de la requête GET, avec la méthode `forward`
- On encapsule toute exception éventuelle dans un `ServletException`

Mécanisme général



1. Dans un servlet S1 on effectue un traitement, puis on appelle
`request.getRequestDispatcher("S2").forward (request,response)`
2. Dans S2 on effectue un traitement, puis on appelle
`request.getRequestDispatcher("S3").forward (request,response)`
3. ...
4. Dans Sn on effectue un traitement, puis on appelle
`request.getRequestDispatcher("view.jsp").forward (request,response)`

On peut ainsi enchaîner les servlets. Les Si peuvent travailler sur l'objet request ainsi que sur les *headers* de la réponse via l'objet response (.addCookie, addHeader, ...).

Les Si ne doivent pas écrire le contenu de la réponse (i.e. pas de .getWriter() dans les servlets), sinon Si+1 renverra une erreur.

- 1 Rappels sur les systèmes d'exploitations / Communication par mémoire partagée ✓
- 2 Réseaux généralités, IP, UDP ✓
- 3 Multi-thread en Java, TCP ✓
- 4 Broadcast UDP et étude de cas ✓
- 5 Rappels Java/Java 8/Sérialisation ✓
- 6 Rappels Java (suite)/Appel de méthode à distance ✓
- 7 JavaServer Pages (1) ✓
- 8 Modèle MVC/JavaServer Pages (2)
 - 8.1 Modèle MVC ✓
 - 8.2 Modèle : Rappels sur JDBC ✓
 - 8.3 Contrôleur : HttpServlet ✓
 - 8.4 Vue : JSP et JSTL

JSP Standard Tag Library : un outil définissant des balises spéciales dans des pages JSP. Ces balises permettent de faire de la publication de données (transformer des structures de données Java en balises HTML) de manière déclarative (sans écrire de code bas niveau).

Cet outil n'est pas intégré directement à J2E mais est très utilisé. Il faut placer le .jar correspondant dans le classpath de l'application.

La vue (1)



```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
...
<body>
  <form method="get" action="PersonListServlet">
    Rechercher dans le nom : <input type="text" name="s" />
    <button type="submit">Rechercher</button>
  </form>

  <c:choose>
  <c:when test="${! empty people}">
    <ul>
      <c:forEach var="p" items="${people}">
        <li> ${p.firstname} <b>${p.lastname}</b></li>
      </c:forEach>
    </ul>
  </c:when>
  <c:otherwise> Il n'y a pas de résultats ! </c:otherwise>
</c:choose>
</body>
<html>
```

JSTL propose deux outils :

- des **balises** particulières représentant des structures de contrôle
- Un langage d'**expressions** nommé *EL expressions*

Il existe d'autres balises spécialisées que celles présentées dans la suite, par exemple pour faire du XML ou des requêtes SQL depuis la vue.

La vue (3) : balises « Core »



Ce sont les balises « de base », auxquelles on donne le préfixe `c` :

`<c:out value="..." />` : écrit dans la page le résultat de l'expression contenue dans `value`

`<c:set var="x" value="..." />` : définit la variable `x` avec le résultat de l'expression contenue dans `value`. Cette variable est réutilisable dans d'autres expressions.

`<c:forEach var="i" items="...">` : Effectue une boucle sur tous les éléments de la collection Java résultant de l'évaluation de `value`. L'indice de boucle est donné par `var`

`<c:choose>` : dénote une conditionnelle multiple. Il doit contenir un nombre arbitraire de `c:when` et éventuellement un `c:otherwise` final

`<c:case test="...">` : dénote un cas qui est choisi si l'expression contenue dans `test` est vraie.

`<c:otherwise>` : représente le cas par défaut pour un ensemble de choix

`<c:if test="...">` : permet d'effectuer un test

`<c:redirect url="...">` : permet de rediriger vers une page dont l'url est donnée.

Une expression peut apparaître n'importe où dans des attributs ou des éléments de la page. Les expressions sont délimitées par `${...}` et ne sont pas typées (il y aura une exception lors de l'exécution en cas d'erreur). Les expressions peuvent contenir :

- Des opérateurs arithmétiques : `+`, `-`, `*`, `/`, `%`
- des opérateurs de comparaison : `==`, `!=`, `<`, `<=`, `>`, `>=`
- Des opérateurs booléens : `&&`, `||`, `!`, `empty`. `empty` vaut vrai si une expression est `null` ou est une collection ou une chaîne de caractère vide.
- Des **noms de variables**. Ce sont alors des variables de requêtes
- Les variables `sessionScope` et `applicationScope` qui représentent la session et l'application
- La notation `x.y`. Si `x` est `sessionScope` ou `applicationScope` permet d'accéder à l'attribut `"y"` s'il a été défini avec `.setAttribute("y", ...)`. Si le type de `x` est une classe java publique, alors récupère une valeur via un appel au *getter* `.getY()`.

Conclusion



On ne présente ici qu'une petite partie de JSTL. Il est important en particulier de bien se concentrer sur le TP et de lire les corrigés en ligne!