

Université Lille 1
Master mention Informatique – M1

Construction d'applications réparties

V. Java EE

Romain.Rouvoy@univ-lille1.fr

Introduction

Java EE : Java Enterprise Edition

Ensemble de concepts pour le développement d'applications réparties

- défini par Sun/Oracle
- basé sur Java
- un ensemble de spécifications (JSR)
voir liste : <http://java.sun.com/javaee/technologies/>
- en évolution "permanente" depuis 1996/97
J2EE 1.0 (servlet + EJB + JDBC), ... , 1.3, 1.4 (2003 Web Services), 5 (2006 Java 5)
Java EE 6 (2009 profile), Java EE 7 (2013 HTML5, websocket), Java EE 8 (à venir)
- domaines applicatifs visés
 - E-commerce (B2B & B2C)
 - systèmes d'informations
 - sites web
 - plates-formes de service (Audio-visuel, telco, ...)

Introduction

Java EE : Java Enterprise Edition

- implémentation de référence : GlassFish
- commerciales
 - WebSphere (IBM), WebLogic (Oracle), SAP, ...
 - voir java.sun.com/javaee/overview/compatibility.jsp
- *open source*
JBoss, JOnAS, GlassFish, ...

Processus de certification mis en place par Oracle

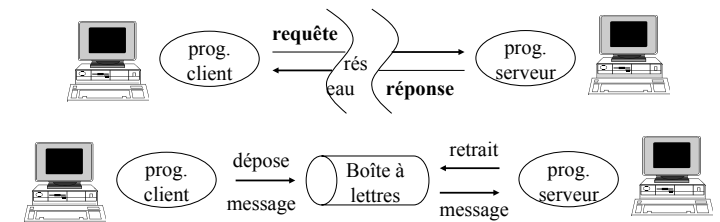
- TCK (Test Compatibility Kit)
- payant sauf pour plates-formes *open-source*
- assez lourd (~20 000 tests) à mettre en oeuvre

Introduction

Java EE : Java Enterprise Edition

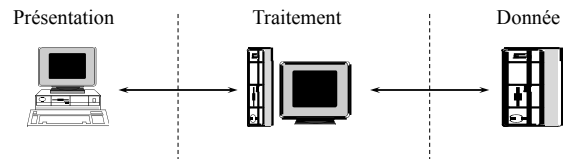
Un ensemble de technologies *middleware* pour la construction d'applications réparties

- communications distantes
 - RMI-IIOP : requête/réponse (TCP + IIOP + sérialisation Java)
 - JAX : requête/réponse Web Service (SOAP, REST)
 - JMS : MOM (*message oriented middleware*) : message + boîte à lettres



Introduction

Architecture client/serveur 3 tiers



Introduction

Architecture client/serveur 3 tiers

Présentation

- applications Java
- applications Web à base de JSP, servlet

Traitement

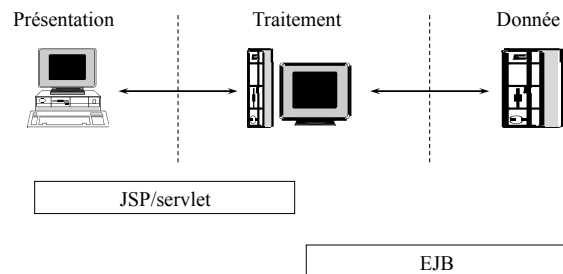
- applications à base de
 - composants EJB : classes Java conformes au modèle EJB
 - interfaces accès distant REST, SOAP, RMI

Donnée

- stockage pour les données des applications
- accessible via JDBC
- SGBD relationnel (Oracle, SQL Server, PostgreSQL, MySQL, ...)

Introduction

Architecture client/serveur 3 tiers



Introduction

Modèle/Vue/Contrôleur

Mise en oeuvre courante des applications Java EE

- modèle : les données de l'application
- vue : la présentation
- contrôleur : la gestion des interactions avec l'utilisateur

Nombreux frameworks de programmation existants

- Spring, Struts, JSF

Introduction

Vocabulaire

- *serveur d'applications*
 - le logiciel qui héberge les applications Java EE
- *conteneur*
 - l'environnement qui exécute les JSP/servlets, les EJBs
- séparation forte entre *code métier* et *code technique/non fonctionnel*
- code organisé sous forme de *services*
 - *services (code) techniques*
 - fournit par le serveur d'applications
 - nommage, persistance, transaction, sécurité

Ressources

Ressources

Documentation Java EE

<http://www.oracle.com/technetwork/java/javasee/documentation/index.html>

Livres

Java EE - Les cahiers du programmeur. Antonio Goncalves. Eyrolles.

Mastering Enterprise Java Beans. E. Roman, R. Sriganesh, G. Brose.

<http://www.theserverside.com/tt/books/wiley/masteringEJB/index.tss>

Java Server Pages (JSP)

Romain Rouvoy

Université Lille 1

Romain.Rouvoy@univ-lille1.fr

Plan

1. Principe
2. Développement
3. Fonctionnalités

1. Principe

Java Server Pages (JSP)

Programme Java s'exécutant **côté serveur Web**

servlet prog. "autonome" stockés dans un fichier `.class` sur le serveur
JSP prog. **source** Java embarqué dans une page `.html`

	côté client	côté serveur
<code>.class</code> autonome	applet	servlet
embarqué dans <code>.html</code>	JavaScript	JSP

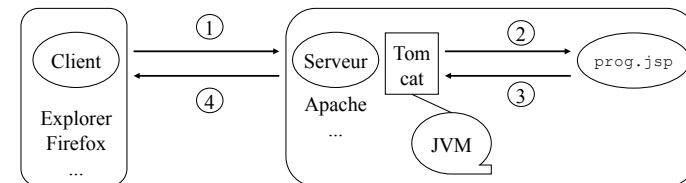
Servlet et JSP

- exécutable avec tous les serveurs Web (Apache, ...)
- auxquels on a ajouté un "moteur" de servlet/JSP (le plus connu : **Tomcat**)
- JSP compilées automatiquement en servlet par le moteur

1. Principe

Java Server Pages (JSP)

- du code Java **embarqué** dans une page HTML entre les balises `<% et %>`
- extension `.jsp` pour les pages JSP
- les fichiers `.jsp` sont stockés sur le serveur (comme des docs)
- ils sont désignés par une URL `http://fil.univ-lille1.fr/prog.jsp`
- le chargement de l'URL provoque l'exécution de la JSP côté serveur



2. Développement

Illustration du fonctionnement

```
<HTML> <BODY>
<H1>Table des factorielles</H1>
<%
  int i,fact;
  for ( i=1,fact=1 ; i<4 ; i++, fact*=i ) {
    out.print( i + "!" = " + fact + "<BR>" );
  }
%>
</BODY> </HTML>
```

invocation
⇒
exécution
côté serveur



2. Développement

Principe de fonctionnement

```
<HTML> <BODY>
<H1>Table des factorielles</H1>
<%
  int i,fact;
  for ( i=1,fact=1 ; i<4 ; i++, fact*=i ) {
    out.print( i + "!" = " + fact + "<BR>" );
  }
%>
</BODY> </HTML>
```

du code Java

résultat = HTML
généralisé via l'objet
prédéfini out

ce qui est
renvoyé
au client

```
<HTML> <BODY>
<H1>Table des factorielles</H1>
1! = 1<BR>
2! = 2<BR>
3! = 6<BR>
</BODY> </HTML>
```

2. Développement

Mécanismes mis en œuvre

- **plusieurs** zones `<% ... %>` peuvent cohabiter dans une même JSP
 - lors du premier chargement d'une JSP (ou après modification), le **moteur**
 - rassemble **tous** les fragments `<% ... %>` de la JSP dans une classe
 - la **compile**
 - l'**instancie**⇒ JSP = objet Java présent dans le moteur
 - puis, ou lors des chargements suivants, le **moteur**
 - exécute le code dans un **thread**
- ⇒ délai d'attente lors de la 1ère invocation dû à la compilation
- ⇒ en cas d'erreur de syntaxe dans le code Java de la JSP
message récupéré dans le navigateur

2. Développement

Directive `<%= ... %>`

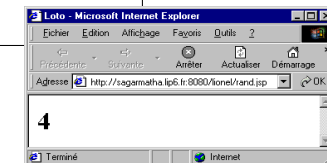
La directive `<%= expr %>` génère l'affichage d'une valeur de l'expression `expr`
⇒ `<%= expr %>` raccourci pour `<% out.print(expr); %>`

```
<HTML> <BODY>

<% int aleat = (int) (Math.random() * 5); %>

<H1> <%= aleat %> </H1>

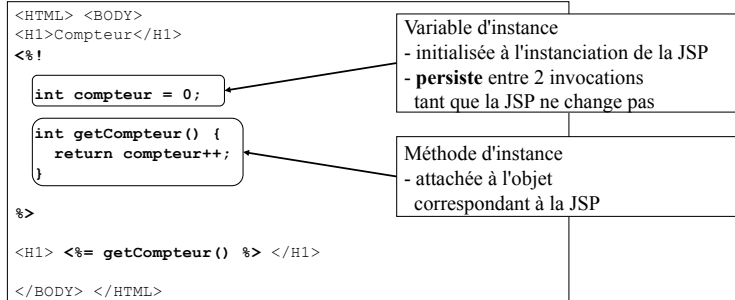
</BODY> </HTML>
```



2. Développement

Méthodes et variables d'instance

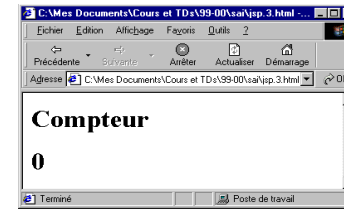
Des **méthodes** et des **variables** d'instance peuvent être associées à une JSP entre les directives `<%!` et `%>`



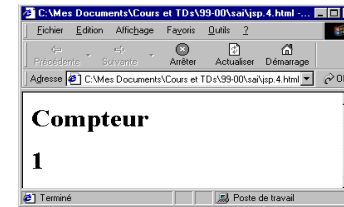
2. Développement

Exemple

1ère invocation



2ème invocation



2. Développement

Variables d'instance

Attention !!

`<%! int cpt = 0; %>` \neq `<% int cpt = 0; %>`

- variable **d'instance** de la JSP (**persiste**)
- variable **locale** à la JSP (**réinitialisée** à chaque invocation de la JSP)

2. Développement

La directive `<%@ page ... %>`

Donne des informations sur la JSP (non obligatoire, valeurs par défaut)

`<%@ page import="..." %>` (ex. `<%@ page import="java.io.*"%>`)
les "import" nécessaires au code Java de la JSP

`<%@ page errorPage="..." %>` (ex. `<%@ page errorPage="err.jsp"%>`)
fournit l'URL de la JSP à charger en cas d'erreur

`<%@ page contentType="..." %>` (ex. `<%@ page contentType="text/html"%>`)
le type MIME du contenu retourné par la JSP

`<%@ page isThreadSafe="..." %>` true ou false
true la JSP peut être exécutée par +ieurs clients à la fois (valeur par défaut)

`<%@ page isErrorPage="..." %>` true ou false
true la JSP est une page invoquée en cas d'erreur

2. Développement

Les objets implicites

Objets prédéclarés utilisables dans le code Java des JSPs

out	le flux de sortie pour générer le code HTML
request	la requête qui a provoqué le chargement de la JSP
response	la réponse à la requête de chargement de la JSP
page	l'instance de servlet associée à la JSP courante (= this)
exception	l'exception générée en cas d'erreur sur une page
session	suivi de session pour un même client
application	espace de données partagé entre toutes les JSP

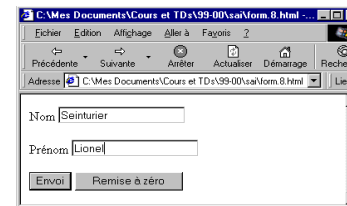
2. Développement

Récupération des données d'un formulaire

Méthode `String getParameter(String)` de l'objet prédéfini `request`

⇒ retourne le texte saisi

⇒ ou `null` si le nom de paramètre n'existe pas



```
<HTML> <BODY>
<FORM ACTION="http://..."
  METHOD=POST>
  Nom <INPUT NAME="nom"> <P>
  Prénom <INPUT NAME="prenom"> <P>
  <INPUT TYPE=SUBMIT VALUE="Envoyer">
  <INPUT TYPE=RESET
    VALUE="Remise à zéro">
  </FORM>
</BODY> </HTML>
```

2. Développement

Récupération des données d'un formulaire

```
<HTML> <BODY>
<H1>Exemple de résultat</H1>
Bonjour
<%= request.getParameter("prenom") %>
<%= request.getParameter("nom") %>
</BODY> </HTML>
```



3. Fonctionnalités

3. Fonctionnalités

3.1 Session

3.2 Chaînage

3.3 Gestion des erreurs

3.4 Données globales

3.1 Session

Suivi de session

- HTTP protocole non connecté
- pour le serveur, 2 requêtes successives d'un même client sont **indépendantes**

Objectif : être capable de "suivre" l'activité du client sur +ieurs pages

Notion de session

- ⇒ les **requêtes** provenant d'un **utilisateur** sont associées à une même session
- ⇒ les sessions ne sont pas éternelles, elles **expirent** au bout d'un délai fixé

Objet prédéfini `session` de type `HttpSession`

- ⇒ la session courante ou une nouvelle session

3.1 Session

Suivi de session

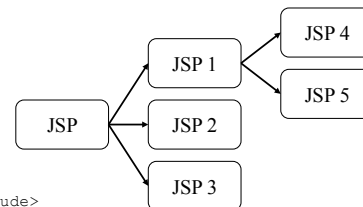
Méthodes appelables sur l'objet prédéfini `session`

- `void setAttribute(String name, Object value)`
ajoute un couple (name, value) pour cette session
- `Object getAttribute(String name)`
retourne l'objet associé à la clé name ou null
- `void removeAttribute(String name)`
enlève le couple de clé name
- `java.util.Enumeration getAttributeNames()`
retourne tous les noms d'attributs associés à la session
- `void setMaxIntervalTime(int seconds)`
spécifie la durée de vie maximum d'une session
- `long getCreationTime() / long getLastAccessedTime()`
retourne la date de création / de dernier accès de la session
en ms depuis le 1/1/1970, 00h00 GMT → `new Date(long)`

3.2 Chaînage

Inclusion de JSP

- aggrégation des résultats fournis par plusieurs JSP
- ⇒ meilleure modularité
- ⇒ meilleure réutilisation



Directives `<jsp:include>` et `</jsp:include>`

```
<HTML> <BODY>
<H1>JSP principale</H1>

<jsp:include
  page="inc.jsp"
</jsp:include>

</BODY> </HTML>
```

URL

Fichier `inc.jsp`

```
<B>JSP include</B>

<P>
<%= (int) (Math.random()*5) %>
</P>
```

Pas de `<HTML>` `<BODY>`

3.2 Chaînage

Inclusion de JSP

Résultat

```
<HTML> <BODY>
<H1>JSP principale</H1>

<B>JSP include</B>
<P>
<%= (int) (Math.random()*5) %>
</P>

</BODY> </HTML>
```



Remarque

1. directives `<jsp:include>` et `</jsp:include>` inclusion dynamique
2. directive `<%@ include file="..." %>` inclusion statique

3.2 Chaînage

Délégation de JSP

Une JSP peut déléger le traitement d'une requête à une autre JSP
⇒ prise en compte **complète** de la requête par la JSP déléguée

Directives `<jsp:forward>` et `</jsp:forward>`

```
<HTML> <BODY>
<H1>JSP principale</H1>

<jsp:forward
  page="forw.jsp"
</jsp:forward>

Ignoré !!
</BODY> </HTML>
```

URL

Fichier forw.jsp

```
<HTML> <BODY>
<H1>JSP déléguée</H1>

<P>
<%= (int) (Math.random()*5) %>
</P>
</BODY> </HTML>
```

```
<HTML> <BODY>
```

3.2 Chaînage

Délégation de JSP

Résultat

```
<HTML> <BODY>
<H1>JSP déléguée</H1>

<P>
<%= (int) (Math.random()*5) %>
</P>
</BODY> </HTML>
```



3.2 Chaînage

Délégation et inclusion de JSP

Transmission de paramètres aux **inclus** et aux **délégués**
Utilisation de couples (name, value)

Directive `<jsp:param name="..." value="..." />`

```
<HTML> <BODY>
<H1>JSP principale</H1>

<jsp:forward page="forw.jsp">
  <jsp:param name="nom" value="Seinturier" />
  <jsp:param name="prenom" value="Lionel" />
</jsp:forward>

</BODY> </HTML>
```

3.2 Chaînage

Délégation et inclusion de JSP

Récupération des paramètres
⇒ à la récupération des paramètres transmis via un formulaire

```
<HTML> <BODY>
<H1>JSP déléguée/incluse</H1>

Nom : <%= request.getParameter("nom") %>
Prénom : <%= request.getParameter("prenom") %>

</BODY> </HTML>
```

3.3 Gestion des erreurs

Gestion des erreurs

Erreur de syntaxe

- dans les directives JSP (ex. : oubli d'une directive %>)
- dans le code Java

Erreur d'exécution du code Java (ex. : NullPointerException)

⇒ dans tous les cas, erreur récupérée dans le **navigateur** client

2 possibilités

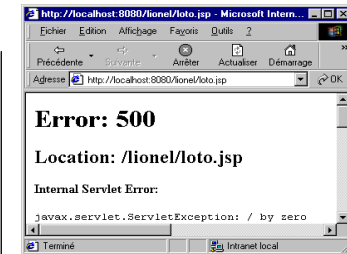
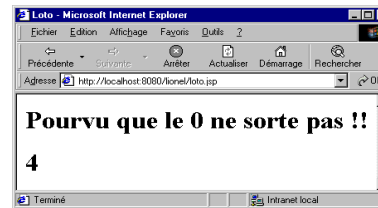
- conserver la **page par défaut** construite par le moteur
- en concevoir une adaptée aux besoins particuliers de l'application
 - ⇒ utilisation des directives `<%@ page errorPage="..." %>` et `<%@ page isErrorPage="..." %>`

3.3 Gestion des erreurs

Exemple de gestion d'erreur

```
<HTML> <BODY>
<H1>Pourvu ... !!</H1>
<% int hasard =
    (int) ( Math.random() * 5 );
%>
<H1> <%= 12 / hasard %> </H1>
</BODY> </HTML>
```

Si hasard = 0
page d'erreur par défaut



3.4 Gestion des erreurs

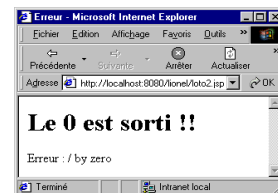
Exemple de gestion d'erreur

```
<HTML> <BODY>
<H1>Pourvu ... !!</H1>
<%@ page
    errorPage="err.jsp" %>
<% int hasard = ... %>
<H1> <%= 12 / hasard %> </H1>
</BODY> </HTML>
```

```
<HTML> <BODY>
<%@ page isErrorPage="true" %>
<h1>Le 0 est sorti !!</h1>
Erreur :
<%= exception.getMessage() %>
</BODY> </HTML>
```

Si hasard = 0
page d'erreur **err.jsp**

Récupération de l'erreur via
l'objet prédéfini **exception**



3.4 Données globales

Partage de données entre JSP

Notion de contexte d'exécution

= ensemble de couples (name, value) partagées par toutes les JSP instanciées

⇒ objet prédéfini application

Méthodes appelables sur l'objet prédéfini application

- void setAttribute(String name, Object value)
ajoute un couple (name, value) dans le contexte
- Object getAttribute(String name)
retourne l'objet associé à la clé name ou null
- void removeAttribute(String name)
enlève le couple de clé name
- java.util Enumeration getAttributeNames()
retourne tous les noms d'attributs associés au contexte

Servlet

Romain Rouvoy

Université Lille 1

Romain.Rouvoy@univ-lille1.fr

Plan

1. Principe
2. Développement
3. Fonctionnalités
4. Archivage
5. Moteurs
6. Servlet 3.0

1. Principe

Servlet

Programme Java s'exécutant **côté serveur Web**

servlet prog. "autonome" stockés dans un fichier `.class` sur le serveur
JSP prog. **source** Java embarqué dans une page `.html`

	côté client	côté serveur
<code>.class</code> autonome	applet	servlet
embarqué dans <code>.html</code>	JavaScript	JSP

Servlet et JSP

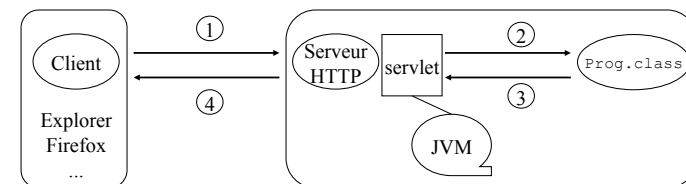
- exécutable avec tous les serveurs Web (Apache, IIS, ...)
- auxquels on a ajouté un "moteur" de servlet/JSP (le plus connu : **Tomcat**)
- JSP compilées automatiquement en servlet par le moteur

1. Principe

Servlet

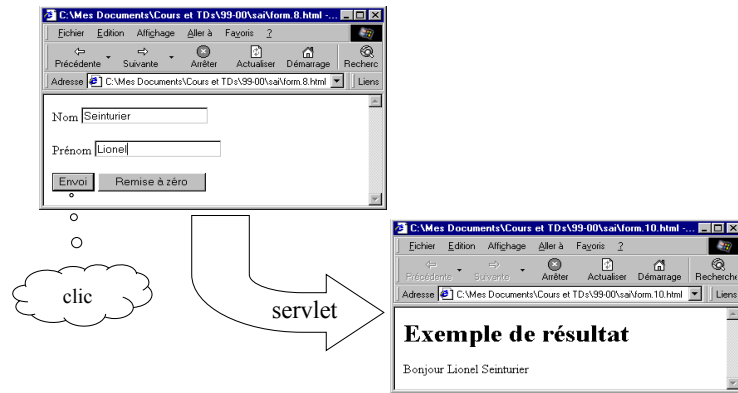
Principe

- les fichiers de *bytecode* (`.class`) sont stockés sur le serveur
- ils sont désignés par une URL ex. : `http://www.lifl.fr/servlet/Prog`
- le chargement de l'URL provoque l'exécution de la servlet
 - ⇒ servlets étendent le comportement du serveur Web
 - ⇒ sont exécutées par un "moteur" (ex. Tomcat)



1. Principe

Illustration du fonctionnement des servlets



2. Développement

Mécanismes mis en œuvre

- écriture d'une servlet = écriture d'une **classe Java**
 - lors du premier chargement d'une servlet (ou après modification), le **moteur**
 - **instancie** la servlet
 - ⇒ servlet = objet Java présent dans le moteur
 - puis, ou lors des chargements suivants, le **moteur**
 - exécute le code dans un **thread**
- ⇒ le code produit un résultat qui est envoyé au client
⇒ en cas d'erreur dans le code Java de la servlet message récupéré dans le navigateur

2. Développement

Développement

Utilisation des packages Java `javax.servlet.*` et `javax.servlet.http.*`

- extension de la classe `javax.servlet.http.HttpServlet`
- redéfinition de la méthode `service` de cette classe
 - définit le code à exécuter lorsque la servlet est invoquée
 - elle est appelée par le "moteur" de servlet

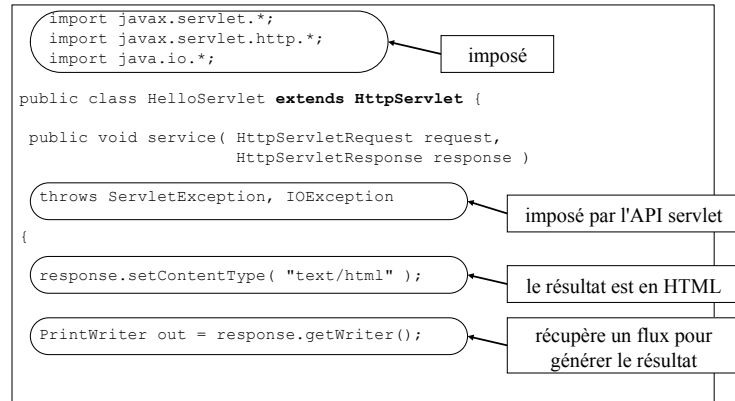
```
void service( HttpServletRequest request, HttpServletResponse response );
```

représente la **requête**
envoyée par le client
⇒ renseigné automatiquement
par le "moteur"

représente la **réponse**
retournée par la servlet
⇒ à **renseigner** dans
le code de la servlet

2. Développement

Exemple de servlet



2. Développement

Exemple de servlet (suite)

```
out.println( "<html><body>" );
out.println( "<h1>Hello depuis une servlet</h1>" );
out.println( "</body></html>" );
} }
```

génération du
code HTML

Compilation

⇒ HelloServlet.class installé
dans l'arborescence de Tomcat

Chargement via une URL de type
`http://.../servlet/HelloServlet`
⇒ exécution de HelloServlet.class



2. Développement

Exemple de servlet (code complet)

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloServlet extends HttpServlet {

    public void service( HttpServletRequest request,
                        HttpServletResponse response )
        throws ServletException, IOException {

        response.setContentType( "text/html" );
        PrintWriter out = response.getWriter();

        out.println( "<html><body>" );
        out.println( "<h1>Hello depuis une servlet</h1>" );
        out.println( "</body></html>" );

    } }
```

2. Développement

Deuxième exemple de servlet

Chaque servlet n'est instanciée **1 seule fois**

⇒ persistance de ces données entre 2 invocations

```
public class CompteurServlet extends HttpServlet {

    int compteur = 0;

    public void service( HttpServletRequest request,
                        HttpServletResponse response )
        throws ServletException, IOException {

        response.setContentType( "text/html" );
        PrintWriter out = response.getWriter();

        out.println( "<html><body>" );
        out.println( "<h1> "+ compteur++ + "</h1>" );
        out.println( "</body></html>" );

    } }
```

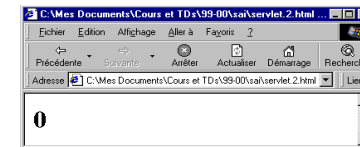
2. Développement

Deuxième exemple de servlet

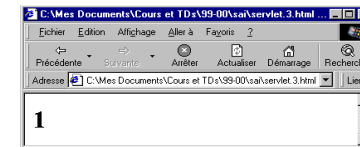
Chaque servlet n'est instanciée **1 seule fois**

⇒ persistance de ces données entre 2 invocations

1ère invocation



2ème invocation



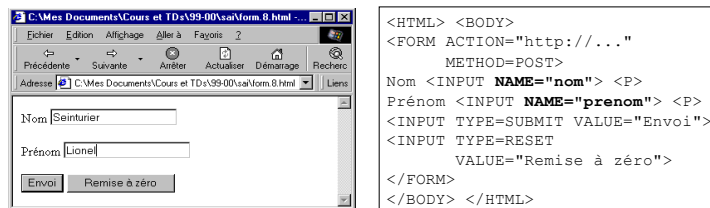
2. Développement

Récupération des données d'un formulaire

Méthode `String getParameter(String)` d'un objet `request`

⇒ retourne le texte saisi

⇒ ou null si le nom de paramètre n'existe pas



```
<HTML> <BODY>
<FORM ACTION="http://..."
      METHOD=POST>
  Nom <INPUT NAME="nom"> <P>
  Prénom <INPUT NAME="prenom"> <P>
  <INPUT TYPE=SUBMIT VALUE="Envoi">
  <INPUT TYPE=RESET
        VALUE="Remise à zéro">
</FORM>
</BODY> </HTML>
```

2. Développement

Récupération des données d'un formulaire

```
public class CompteurServlet extends HttpServlet {

    public void service( HttpServletRequest request,
                        HttpServletResponse response )
        throws ServletException, IOException {

        response.setContentType( "text/html" );
        PrintWriter out = response.getWriter();

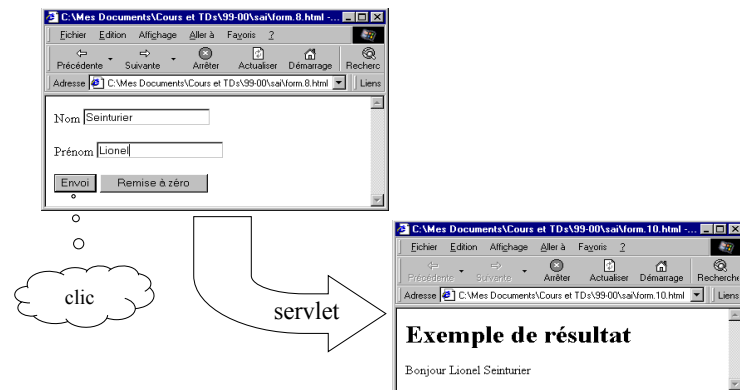
        String nom = request.getParameter( "nom" );
        String prenom = request.getParameter( "prenom" );

        out.println( "<html><body>" );
        out.println( "<h1>Exemple de résultat</h1>" );
        out.println( "Bonjour " + prenom + " " + nom );
        out.println( "</body></html>" );

    }
}
```

2. Développement

Récupération des données d'un formulaire



2. Développement

Différenciation des méthodes HTTP

- `service()` traite toutes les requêtes HTTP
- possibilité de différencier les traitements en fonction de la commande HTTP
POST, GET, ... `doGet()` `doHead()` `doPost()` `doPut()` `doDelete()` `doTrace()`
- les méthodes `doXXX()` ont le même profil/fonctionnement que `service()`

```
public class CompteurServlet extends HttpServlet {

    public void doGet( HttpServletRequest req, HttpServletResponse resp )
        throws ServletException, IOException {
        // ... le traitement lorsque la servlet est invoquée avec GET
    }

    public void doPost( HttpServletRequest req, HttpServletResponse resp )
        throws ServletException, IOException { /* idem POST */ }

    // ... éventuellement autres méthodes doXXX
}
```

2. Développement

Cycle de vie d'une servlet

Une servlet peut définir les méthodes `init()` et `destroy()`

- `void init(ServletConfig conf)`
méthode appelée par le moteur au **démarrage** de la servlet
 - ⇒ peut contenir le code d'initialisation de la servlet
 - ⇒ ≈ constructeur pour la servlet
 - ⇒ méthode appelée par le moteur lors de l'installation de la servlet
- `void destroy()`
méthode appelée lors de la **destruction** de la servlet
 - lors de l'arrêt du moteur
 - ou lors du déchargement de la servlet
 - peut-être appelée pour arrêter la servlet

2. Développement

Types de contenu générables par une servlet

- 80% du temps, HTML
- mais peut être n'importe quel type de contenu : GIF, PDF, DOC, ...

- type MIME du contenu indiqué par `resp.setContentType("...")`
- quelques types MIME courants

- text/html
- image/gif
- video/mpeg
- audio/mp3
- application/pdf

- application/octet-stream : un fichier binaire quelconque

2. Développement

Exemple de servlet retournant le contenu d'un fichier binaire

```
public class CompteurServlet extends HttpServlet {

    public void service( HttpServletRequest req, HttpServletResponse resp )
        throws ServletException, IOException {

        resp.setContentType("application/octet-stream");
        resp.setHeader(                               // facultatif
            "Content-Disposition",                    // fournit le nom du fichier
            "attachment;filename=monfichier.ext");     // au navigateur

        OutputStream os = resp.getOutputStream();
        File f = new File("monfichier.ext");
        byte [] content = new byte[f.length()];
        FileInputStream fis = new FileInputStream(f);
        fis.read(content);
        fis.close();
        os.write(content);

    } }
```

3. Fonctionnalités

3. Fonctionnalités

3.1 Session

3.2 Cookies

3.3 Upload

3.4 Chaînage

3.5 Concurrency

3.6 Données globales

3.1 Session

Suivi de session

- HTTP protocole non connecté
- pour le serveur, 2 requêtes successives d'un même client sont **indépendantes**

Objectif : être capable de "suivre" l'activité du client sur +ieurs pages

Notion de session

- ⇒ les **requêtes** provenant d'un **utilisateur** sont associées à une même session
- ⇒ les sessions ne sont pas éternelles, elles **expirent** au bout d'un délai fixé

Sur un objet request

- HttpSession session = request.getSession(true)
retourne la session courante pour cet utilisateur ou une nouvelle session
- HttpSession session = request.getSession(false)
retourne la session courante pour cet utilisateur ou null

3.1 Session

Méthodes d'un objet de type HttpSession

- void setAttribute(String name, Object value)
ajoute un couple (name, value) pour cette session
- Object getAttribute(String name)
retourne l'objet associé à la clé name ou null
- void removeAttribute(String name)
enlève le couple de clé name
- java.util Enumeration getAttributeNames()
retourne tous les noms d'attributs associés à la session
- void setMaxIntervalTime(int seconds)
spécifie la durée de vie maximum d'une session
- long getCreationTime() / long getLastAccessedTime()
retourne la date de création / de dernier accès de la session
en ms depuis le 1/1/1970, 00h00 GMT → new Date(long)

3.2 Cookies

Cookies

Permettent à un serveur Web de stocker de l'information chez un client

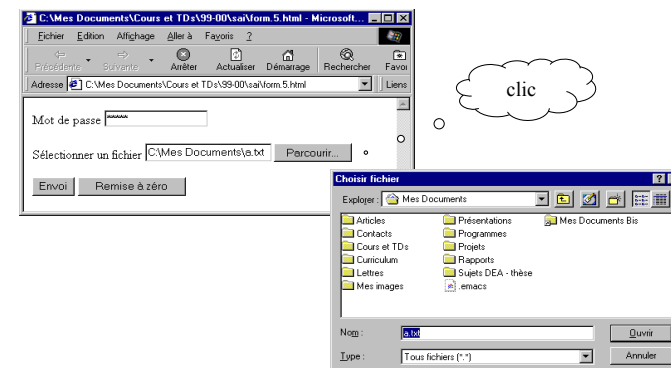
- ⇒ moyen pour savoir "par où passe" un client, quand, en venant d'où, ...
- ⇒ l'utilisateur a la possibilité d'interdire leur dépôt dans son navigateur

- définis dans la classe javax.servlet.http.Cookie
- on les crée en donnant un nom (String) et une valeur (String)
Cookie uneCookie = new Cookie("sonNom", "saValeur");
- on les positionne via un objet response
response.addCookie(uneCookie);
- on les récupère via un objet request
Cookie[] desCookies = request.getCookies();

Quelques méthodes : String getName() / String getValue()

3.3 Upload

Récupération des fichiers uploadés à partir d'un formulaire



3.3 Upload

Définition des formulaires avec *upload*

```
<HTML> <BODY>

<FORM ACTION="url servlet traitement du formulaire" METHOD=POST
      ENCTYPE="multipart/form-data">

Sélectionner un fichier <INPUT TYPE=FILE NAME="fichier">

<INPUT TYPE=SUBMIT VALUE="Envoi">
<INPUT TYPE=RESET VALUE="Remise à zéro"> <P>

</FORM>

</BODY> </HTML>
```

3.3 Upload

Encodage fichiers joints

```
-----7d225420d803c8
Content-Disposition: form-data; name="fichier"; filename="..."
Content-Type: image/gif
```

```
GIF89a& ... contenu binaire du fichier ...
-----7d225420d803c8--
```

- séparateur déterminé aléatoirement à chaque upload par le navigateur
- + dans les en-têtes HTTP de la requête
Content-Type: multipart/form-data;
boundary=-----7d225420d803c8

- format défini par la RFC 1867 de l'IETF
voir <http://www.ietf.org/rfc/rfc1867.txt>

3.3 Upload

Récupération des fichiers *uploadés* à partir d'un formulaire

- récupération du flux binaire, programmer le décodage ☹
- librairie existante Commons FileUpload
<http://jakarta.apache.org/commons/fileupload/>

Utilisation (version 1.0)

```
import org.apache.commons.fileupload.*;

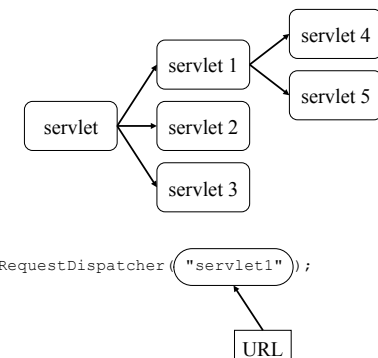
DiskFileUpload dfu = new DiskFileUpload();
List files = dfu.parseRequest(request);
for( Iterator i = files.iterator() ; i.hasNext() ; ) {
    FileItem fi = (FileItem) i.next();
    File monFichier = new File(fi.getName());
    fi.write(monFichier);
}
```

⇒ dans méthode `service` d'une servlet

3.4 Chaînage

Chaînage des servlets

- aggrégation des résultats fournis par plusieurs servlets
⇒ meilleure modularité
⇒ meilleure réutilisation



Utilisation d'un *RequestDispatcher*

- obtenu via un objet `request`

```
RequestDispatcher rd = request.getRequestDispatcher("servlet1");
```

Inclusion du résultat d'une autre servlet

```
rd.include(request, response);
```

Délégation du traitement à une autre servlet

```
rd.forward(request, response);
```

3.5 Concurrency

Gestion de la concurrence

Par défaut les servlets sont exécutées de façon *multi-threadée*

Si une servlet doit être exécutée en exclusion mutuelle
(ex. : accès à des ressources partagées critiques)
implantation de l'interface **marqueur** `SingleThreadModel`

```
public class CompteurServlet
    extends HttpServlet
    implements SingleThreadModel {

    public void service( ServletRequest request,
                        ServletResponse response )
        throws ServletException, IOException {
        /** Du code en exclusion mutuelle avec lui-même */
    }
}
```

Autre solution : définir du code `synchronized` dans la servlet

3.6 Données globales

Partage de données entre servlets

Notion de contexte d'exécution

= ensemble de couples (name, value) partagées par toutes les servlets **instanciées**
⇒ partage de données entre tous les clients

```
ServletContext ctx = getServletContext() (héritée de GenericServlet)
```

Méthodes appelables sur un objet de type `ServletContext`

- void `setAttribute(String name, Object value)`
ajoute un couple (name, value) dans le contexte
- Object `getAttribute(String name)`
retourne l'objet associé à la clé name ou null
- void `removeAttribute(String name)`
enlève le couple de clé name
- java.util Enumeration `getAttributeNames()`
retourne tous les noms d'attributs associés au contexte

4. Archivage

Format de fichier .war

Problématique : comment diffuser une application à base de servlets ?

- souvent plusieurs servlet (fichiers .class)
- des ressources additionnelles (.gif, .jpeg, .html, .xml, ...)

Solution

- monde Java : archive .jar pour la diffusion de programmes
- fichier .war = .jar pour les servlets
⇒ diffusion d'un seul fichier prêt à l'emploi
- fichiers .war se manipulent (création, extraction, ...) avec la commande jar
ex. :

```
jar cf app.war index.html WEB-INF/classes/*  création
jar tf app.war                               affiche le contenu
jar xf app.war                               extraction
```

4. Archivage

Descripteur de déploiement web.xml

Chaque archive .war doit être accompagnée d'un fichier web.xml
décrivant les servlets incluses dans l'archive

- 2 balises principales : `<servlet>` et `<servlet-mapping>`

```
<web-app>
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>mypackage.HelloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/version/beta/Hello</url-pattern>
  </servlet-mapping>
</web-app>
```

4. Archivage

Descripteur de déploiement web.xml

- une balise `<servlet>` par servlet
 - un nom et une classe par servlet
 - le fichier `.class` de la servlet doit être stockés dans `WEB-INF/classes`
 - éventuellement sous-répertoires correspondant aux packages

ex. : `WEB-INF/classes/mypackage/HelloServlet.class`
- une balise `<servlet-mapping>` par servlet
 - un nom (doit correspondre à une balise `<servlet>` existante)
 - une URL relative permettant d'accéder à la servlet
- plusieurs autres balises peuvent être utilisées
voir http://java.sun.com/j2ee/dtds/web-app_2_2.dtd

4. Archivage

Installation d'une archive .war dans Tomcat

- dans le répertoire `<tomcat_root>/webapps`

```
webapps
|-> myapp.war
|-> myapp
    |-> index.html
    |-> WEB-INF
        |-> web.xml
        |-> classes
            |-> mypackage
                |-> HelloServlet.class
```

- URL pour accéder à la servlet

`http://machine.com:8080/myapp/version/beta/Hello`

⇒ dépend `<url-pattern>` fournit dans `web.xml`

4. Archivage

Descripteur de déploiement web.xml

Paramètres d'initialisation

- possibilité d'inclure des paramètres d'initialisation de la servlet dans `web.xml`
 - avantage : peuvent être changés sans avoir à recompiler

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>mypackage.HelloServlet</servlet-class>
  <init-param>
    <param-name>nom</param-name>
    <param-value>valeur</param-value>
  </init-param>
</servlet>
```

Dans le code de la servlet (par ex. méthode `init`)

```
String valeur = getInitParameter("nom");
```

5. Moteurs

Moteur de servlet (*servlet engine*)

Parfois aussi appelé conteneur de servlet

- logiciel servant à exécuter des servlets
- les servlets ne sont pas des programmes autonomes (pas de `main`)
 - ⇒ doivent être pris en charge par un moteur pour être exécutées

- | | |
|----------|---|
| • Tomcat | http://tomcat.apache.org |
| • Jetty | http://jetty.mortbay.com |
| • Resin | http://www.caucho.com |

5. Moteurs

Tomcat



- le plus connu, utilisé
- logiciel écrit 100% en Java
- inclut un serveur HTTP
- par défaut port 8080
- peut s'utiliser
 - en mode *standalone* : joue le rôle du serveur HTTP + moteur servlet
 - couplé avec serveur Web Apache
- diffuser par le consortium Apache
- peut exécuter aussi des JSP

- souvent inclus dans d'autres logiciels
ex. : serveur Java EE (JBoss, JOnAS, ...)

6. Servlet 3.0

Servlet 3.0

- JSR 315 – 12/2009
- disponible dans Tomcat 7.x

- 4 ajouts principaux
 - annotations
 - déclaration programmatique
 - web fragments
 - exécution asynchrone

6. Servlet 3.0

Servlet 3.0 – Annotations

- faciliter l'écriture des servlets
- remplacer le fichier web.xml par des annotations

```
@WebServlet(  
    name="HelloServlet",           // le nom de la servlet  
    urlPatterns={"/version/beta/Hello"}, // préfixe(s) d'URL  
)  
public class HelloServlet extends HttpServlet { ... }
```

- possibilité de fournir les paramètres d'initialisation (si pertinent)

```
initParams={@WebInitParam(name="nom", value="valeur")}
```

6. Servlet 3.0

Servlet 3.0 – Déclaration programmatique

- possibilité de déclarer une servlet dynamiquement
- ajout de servlet à chaud sans avoir à redémarrer le conteneur

```
ServletContext context = getServletContext();  
String aName = "...";  
Servlet aServlet = ...;  
  
context.addServlet(aName,aServlet);
```

6. Servlet 3.0

Servlet 3.0 – Web Fragments

- possibilité de scinder le fichier web.xml en plusieurs fragments
- facilite la réutilisation de servlets développées indépendamment

```
<web-fragment>
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>mypackage.HelloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/version/beta/Hello</url-pattern>
  </servlet-mapping>
</web-fragment>
```

6. Servlet 3.0

Servlet 3.0 – Exécution asynchrone

- possibilité de rendre la main au client (navigateur) avant la fin de la requête
- un mécanisme de rappel (*callback*) fournit au client les informations complémentaires
- améliore la prise en charge
 - des requêtes nécessitant un long temps de traitement
 - des pages nécessitant une mise à jour périodique (ex. affichage cours boursiers, ...)

Java Database Connectivity (JDBC)

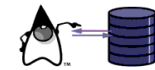
Romain Rouvoy

Université Lille 1

Romain.Rouvoy@univ-lille1.fr

JDBC

Java Database Connectivity (JDBC)



Permet à un programme Java d'interagir

- localement ou à distance
- avec une base de données relationnelle

Fonctionne selon un principe **client/serveur**

- client = le programme Java
- serveur = la base de données

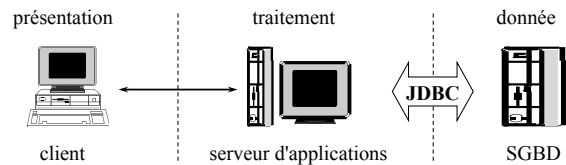
Principe

- le programme Java ouvre une connexion
- il envoie des **requêtes SQL**
- il récupère les résultats
- ...
- il ferme la connexion

JDBC

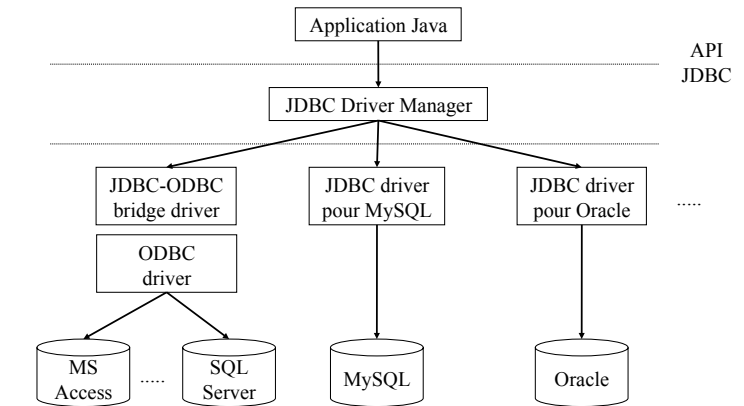
Java Database Connectivity (JDBC)

- API d'interaction avec un SGBD
- nombreuses utilisations
 - sauvegarde de données de manière sûre
 - exploration du contenu d'un SGBD
 - client/serveur 3 tiers
 - J2EE : persistance des entity beans explicite (BMP) ou transparente (CMP)



JDBC

Architecture JDBC



JDBC

Drivers JDBC

Type I

drivers pour accéder à des SGBD locaux uniquement

Type II

drivers partiellement écrits en Java
reposent sur des bibliothèques propriétaires (par ex. en C) pour accéder au SGBD

Type III

drivers 100 % Java
communiquent localement ou à distance avec le SGBD
selon un protocole réseau générique

Type IV

idem III mais utilisent un protocole propriétaire (spécifique au SGBD)

JDBC

URLs JDBC

Schéma de désignation des bases de données avec JDBC

`jdbc:driver:adresse`

La syntaxe de la partie *adresse* est spécifique au *driver*

Exemples

- MySQL : `jdbc:mysql://elios.lip6.fr/employees`
- hsqldb : `jdbc:hsqldb:hsqldb://localhost:9001/db_jonas`
- Java DB inclus dans JDK 6 (basé sur Apache DB)
 - *embedded* : `jdbc:derby:foo;create=true` (rq: foo = répertoire)
 - *network* : `jdbc:derby:../myRep/foo`
- Oracle : `jdbc:oracle:thin:@elios.lip6.fr:employees`

JDBC

Utilisation de JDBC

L'ensemble de l'API JDBC est définie dans le package `java.sql`

1. Chargement du driver (= chargement de la classe du driver dans la JVM)

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

2. Ouverture d'une connexion avec la base ages

```
Connection cx =  
    DriverManager.getConnection("jdbc:derby:ages", "login", "passwd");
```

3. Création d'une requête SQL en utilisant un objet de la classe Connection

```
Statement st = cx.createStatement();
```

4. Envoi d'une requête SELECT

```
ResultSet rs = st.executeQuery("SELECT * FROM ages");
```

Envoi d'une requête CREATE, INSERT ou UPDATE

```
int res = st.executeUpdate("INSERT INTO ages VALUES ('toto',12)");
```

JDBC

Utilisation de JDBC (suite)

5. Récupération du résultat

`rs.next()` retourne vrai tant qu'il reste des enregistrements dans le résultat
et positionne le curseur sur l'enregistrement suivant

`rs.getString(String attribut)` (ex.: `rs.getString("nom")`)
retourne la valeur de l'attribut `attribut` de type `String` dans le résultat

`rs.getInt(String attribut)`, `getBoolean`, `getBytes`, `getDouble`, `getFloat`
idem pour des attributs de type `int`, `boolean`, `byte`, `double` ou `float`

```
while( rs.next() ) {  
    String nom = rs.getString("nom");  
    int age = rs.getInt("age");  
    System.out.println( nom + " a " + age + " ans" );  
}
```

JDBC

Utilisation de JDBC (code complet)

```
import java.sql.*;
public class TestJDBC {
    public static void main( String[] args ) throws Exception {
        Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
        Connection cx =
            DriverManager.getConnection( "jdbc:derby:ages", "", "" );
        Statement st = cx.createStatement();
        ResultSet rs = st.executeQuery( "SELECT * FROM ages" );
        while (rs.next()) {
            String nom = rs.getString("nom");
            int age = rs.getInt("age");
            System.out.println( nom + " a " + age + " ans" );
        }
        /** Fermetures */
        rs.close(); st.close(); cx.close();
    }
}
```

JDBC

Correspondances types données SQL/Java

Type SQL	Type Java	Méthode <i>getter</i>
CHAR VARCHAR	String	getString()
INTEGER	int	getInt()
TINYINT	byte	getByte()
SMALLINT	short	getShort()
BIGINT	long	getLong()
BIT	boolean	getBoolean()
REAL	float	getFloat()
FLOAT DOUBLE	double	getDouble()
NUMERIC DECIMAL	java.math.BigDecimal	getBigDecimal()

JDBC

Correspondances types données SQL/Java

Type SQL	Type Java	Méthode <i>getter</i>
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

TIMESTAMP heure avec nanosecondes

rs.getString(String attributName) OU rs.getString(int colNum)
peuvent être utilisés pour n'importe quel type de paramètre
⇒ récupération de la valeur de l'attribut sous forme de chaîne

JDBC

Types pour les données de grande taille

BLOB *Binary Large Object*
CLOB *Character Large Object*

⇒ permettent de stocker des données de grande taille (ex. : fichier) dans une table

Type SQL	Type Java	Méthode <i>getter</i>
BLOB	java.lang.Object	getObject()
CLOB	java.lang.Object	getObject()

JDBC

Curseurs multi-directionnel (*scrollable*) & dynamique (*updatable*)

Par défaut `ResultSet` avance seule + lecture seule

```
Connexion cx = ...
Statement st = cx.createStatement( int direction, int maj );
```

direction

```
ResultSet.TYPE_FORWARD_ONLY
```

```
ResultSet.TYPE_SCROLL_INSENSITIVE
```

```
ResultSet.TYPE_SCROLL_SENSITIVE
```

maj

```
ResultSet.CONCUR_READ_ONLY
```

```
ResultSet.CONCUR_UPDATABLE
```

JDBC

Curseurs multi-directionnel (*scrollable*) & dynamique (*updatable*)

Exemple curseur multi-directionnel

```
Connexion cx = ...
Statement st = cx.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY );
ResultSet rs = st.executeQuery( "..." );
```

```
boolean rs.absolute( int row );
boolean rs.relative( int row );
int getRow();
boolean next();
boolean previous();
void beforeFirst();
void afterLast();
boolean isFirst();
boolean isLast();
```

JDBC

Curseurs multi-directionnel (*scrollable*) & dynamique (*updatable*)

Exemple curseur dynamique

```
Connexion cx = ...
Statement st = cx.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE );
ResultSet rs = st.executeQuery( "..." );
```

```
void rs.updateString( int columnIndex, String value );
void rs.updateString( String columnName, String value );
updateDouble, updateShort, updateDate, updateObject, ...
```

!! modification prise en compte seulement après !!

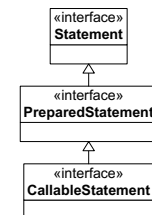
```
void rs.updateRow();
```

JDBC

Types de requêtes SQL

- "normale"
 - interprétée à chaque exécution
 - précompilée
 - paramétrable
 - préparée pour être exécutée plusieurs fois
 - gérée par le programme
 - procédure stockée
 - paramétrable
 - écrite dans le langage interne du SGBD (ex SQL Server Transac-SQL)
 - gérée par le SGBD
- + masque schéma base
 - + meilleures performances
 - + validées par rapport schéma base

- langage propriétaire (- évolution)
 - risque de mélange logiques traitement/donnée



JDBC

Requêtes SQL précompilées

1. Création en utilisant un objet de la classe `Connection`

Possibilité de définition de 1 ou +ieurs paramètres ⇨ caractères ?

```
PreparedStatement pst =  
    cx.prepareStatement("SELECT * FROM ages WHERE nom=? AND age>?");
```

2. Les paramètres sont renseignés par des appels à des méthodes *setter*

```
pst.setString( 1, "Paul" );           // 1 = 1er ?  
pst.setInt( 2, 25 );                   // 2 = 2ème ?
```

Rq : `setBoolean`, `setByte`, `setDouble`, `setFloat` pour les autres types

3. Exécution de la requête

```
ResultSet rs = pst.executeQuery();
```

Rq : `executeUpdate` pour les autres types de requêtes SQL

JDBC

Procédures stockées

Exemple de procédure stockée Transact-SQL (SQL Server)

```
CREATE PROCEDURE [pubs].[GetRange]  
    @age int  
AS  
    SELECT nom FROM ages WHERE age < @age  
GO
```

Le code JDBC d'appel de la procédure

```
CallableStatement cst = cx.prepareCall("{call pubs.GetRange(?)}");  
cst.setInt( 1, 25 );  
ResultSet rs = cst.executeQuery();
```

JDBC

Transactions

Groupes de requêtes devant être exécutés de façon **indivisible**

La transaction doit être

- validée (`commit`) ⇨ les résultats ne sont visibles qu'à partir de ce moment
- ou annulée (`rollback`)

```
cx.setAutoCommit(false);
```

déclaration du début de la transaction

```
Statement st = cx.createStatement();  
st.executeUpdate( "INSERT INTO ages VALUES ('Pierre',12)" );  
st.executeUpdate( "UPDATE ages SET age=15 WHERE nom='Paul'" );
```

```
cx.commit();
```

validation de la transaction

JDBC

Transactions

Exemple

```
CREATE TABLE comptes (  
    nom VARCHAR(30) PRIMARY KEY,  
    solde FLOAT, CHECK(solde>=0) );  
  
cx.setAutoCommit(false);  
try {  
    Statement st = cx.createStatement();  
    st.executeUpdate("UPDATE comptes SET solde=solde+montant WHERE nom='Paul'" );  
    st.executeUpdate("UPDATE comptes SET solde=solde-montant WHERE nom='Bob'" );  
    cx.commit();  
} catch( SQLException e ) {  
    cx.rollback();  
}
```

JDBC

Niveau d'isolation des transactions

Différents niveaux d'isolation possibles pour les transactions
(= à quel moment les résultats des transactions sont visibles)

Positionables par appel à `cx.setTransactionIsolation(niveau);`

`TRANSACTION_SERIALIZABLE` (niveau = 8)

- t_1 lit un ensemble d'enregistrement
- t_2 ajoute un enregistrement à l'ensemble
- si t_1 lit à nouveau l'ensemble, il ne voit pas celui ajoutée par t_2
- ⇒ les transactions apparaissent comme si elles avaient été exécutées **en séquence**

`TRANSACTION_REPEATABLE_READ` (niveau = 4)

- t_1 lit un enregistrement
- t_2 modifie cet enregistrement
- si t_1 lit à nouveau l'enregistrement, il lit la **même valeur** que précédemment

JDBC

Niveau d'isolation des transactions

`TRANSACTION_READ_COMMITTED` (niveau = 2)

- t_1 modifie un enregistrement
- t_2 ne peut pas lire cet enregistrement tant que t_1 **n'a pas été engagée** (ou annulée)

dirty reads interdits

`TRANSACTION_READ_UNCOMMITTED` (niveau = 1)

- t_1 modifie un enregistrement
- t_2 peut lire cet enregistrement

dirty reads autorisés

`TRANSACTION_NONE` (niveau = 0)

pas de support pour les transactions

Selon les couples (BD, driver JDBC),
tous ces modes ne sont pas forcément disponibles

JDBC

Traitement par lots (*batch*)

But : réduire le coût d'une série de mise à jour

1. Création d'une requête en utilisant un objet de la classe Connection

```
Statement st = cx.createStatement();
```

2. Ajout des différentes requêtes

```
st.addBatch( "INSERT INTO ages VALUES ('Pierre',45)" );
st.addBatch( "INSERT INTO ages VALUES ('Anne',45)" );
st.addBatch( "UPDATE ages SET age=15 WHERE nom='Paul'" );
```

3. Exécution des requêtes (dans l'ordre dans lequel elles ont été ajoutées)

```
int[] res = st.executeBatch();
```

Rq : possibilité d'insérer le lot dans une transaction

⇒ 0. `cx.setAutoCommit(false);`

⇒ 4. `cx.commit();`

JDBC

Méta-base

La plupart des programmes JDBC sont écrits pour des **schémas de tables connus**

But de la méta-base : découvrir à **l'exécution** le schéma des tables

Avantage : le programme peut manipuler n'importe quel schéma

1. Récupération des enregistrements d'une table

```
Statement st = cx.createStatement();
ResultSet rs = st.executeQuery( "SELECT * FROM ages" );
```

2. Récupération d'un objet de la classe ResultSetMetaData décrivant le ResultSet

```
ResultSetMetaData rsmd = rs.getMetaData();
```

3. Interrogation du ResultSetMetaData pour découvrir le schéma de la table ages

```
int columnCount = rsmd.getColumnCount(); // # de colonnes
String colLabel = rsmd.getColumnLabel(i); // nom de la colonne i
String colType = rsmd.getColumnTypeName(i); // type de la colonne i
```

JDBC

Méta-base

La méta-base contient aussi des informations sur les tables contenues dans la base

1. Récupération d'un objet de la classe DatabaseMetaData décrivant la base

```
DatabaseMetaData dbmd = cx.getMetaData();
```

2. Récupération des tables (utilisateur) de la base

```
ResultSet tables = dbmd.getTables(null,null,null,{"TABLE"});
```

3. Récupération des noms des tables (itération sur le ResultSet)

```
while ( tables.next() ) {  
    String tableName = tables.getString("TABLE_NAME");  
}
```

Rq : nombreuses autres possibilités d'interrogation du DatabaseMetaData

JDBC

Fonctionnalités avancées

JDBC 3.0 (JDK 1.4)

- récupération de valeurs auto-générées
- curseurs & transactions
- ResultSet multiples
- gestion pool de connexions
- pool de PreparedStatement
- points de reprise dans les transactions
- mode déconnecté RowSet (JDK 5)

JDBC 4.0 (JDK 6)

- interopérabilité entre données JDBC et XML
- requêtes sous forme d'annotations

JDBC

Récupération de valeurs auto-générées

- colonne auto incrémentée à chaque ajout de tuple
- valeur unique générée automatiquement
- type auto_increment (MySQL), serial (PostgreSQL), ...

```
CREATE TABLE users ( id INT AUTO_INCREMENT, nom VARCHAR(30) )  
String req = "INSERT INTO users (nom) VALUES ('Bob')";  
st.executeStatement( req, Statement.RETURN_GENERATED_KEYS );  
ResultSet rs = st.getGeneratedKeys();  
rs.next();  
int key = rs.getInt(1);           // éventuellement +sieurs
```

0	Robert
1	Bob

JDBC

Curseurs & transactions

- curseur ouvert pendant transaction
- peut-être conservé après la fin de la transaction
- `createStatement()`, `prepareStatement()`, `prepareCall()`
`param. suppl. ResultSet.HOLD_CURSORS_OVER_COMMIT`

ResultSet multiples

- procédure stockée retournant +sieurs résultats
- consultation en // des différents résultats (en séquence < JDBC 3.0)

```
ResultSet rs1 = cst.executeQuery();  
ResultSet rs1 = cst.getMoreResults(Statement.KEEP_CURRENT_RESULT);
```

JDBC

Gestion pool de connexion

- optimiser/réguler les cx vers les SGBD
- pool gère et partage les cx de +ieurs appli

Nouveautés JDBC 3.0 : standardisation propriétés

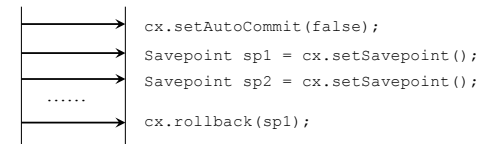
- initialPoolSize
- minPoolSize
- maxPoolSize
- maxIdleTime # secondes
- propertyCycle # secondes entre 2 exec. politique de gestion du pool
- maxStatement # max de Statement ouvert pour 1 cx du pool

Pool de PreparedStatement

JDBC

Points de reprise dans les transactions

- points de sauvegarde intermédiaires dans une transaction
- plus sûr
- retour à un point à la demande **avant fin transaction**

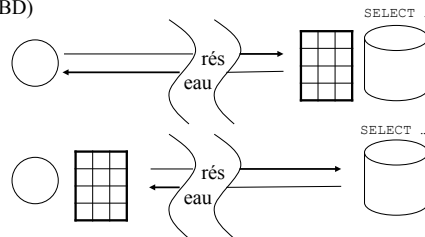


JDBC

Mode déconnecté RowSet

- par défaut JDBC c/s connecté vers SGBD
- + 1 seule copie des données (SGBD)
- + mises à jour simples

connecté vs non connecté
n messages petite taille
vs 1 message grande taille



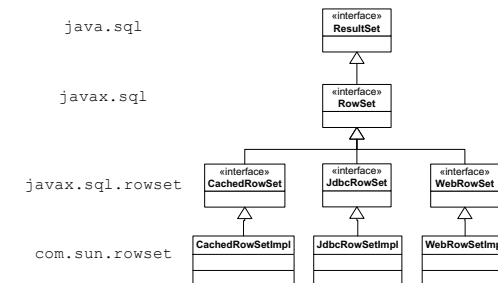
Déconnecté

- pouvoir consulter/modifier les données *off line*
- économiser les ressources réseaux (connexions moins longues)
- travailler sur des données en mémoire plutôt que directement sur un SGBD

JDBC

Mode déconnecté RowSet

- CachedRowSet : manipulation de données en mémoire
- JdbcRowSet : 1 rowset Java Bean → utilisé avec comp. graph. JTable, JTree, JList, ...
- WebRowSet : 1 rowset associé à un fichier XML

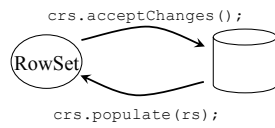


JDBC

Mode déconnecté RowSet

CachedRowSet

```
ResultSet rs = ...  
CachedRowSet crs = new CachedRowSet();  
crs.populate(rs);  
cx.close();  
// manipulation de crs
```



Enterprise Java Beans 3.1

Romain Rouvoy

Université Lille 1

Romain.Rouvoy@univ-lille1.fr

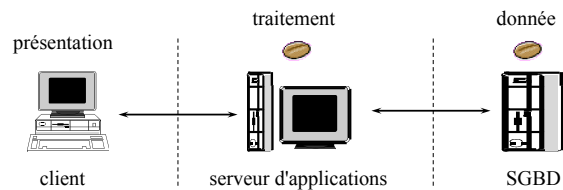
Plan

1. Composant EJB
 - 1.1 Session Bean
 - 1.2 Entity Bean
2. Fonctionnalités avancées
3. Services
 - 3.1 Transaction
 - 3.2 Sécurité
4. Design patterns EJB

1. Composant EJB

Enterprise Java Bean (EJB)

Composants applicatifs pour le développement d'applications réparties



1. Composant EJB

Enterprise Java Bean (EJB)

A server-side component that encapsulates the business logic of an application

- on se focalise sur la logique applicative
- les services systèmes sont fournis par le conteneur
- la logique de présentation est du ressort du client

Vocabulaire dans ce cours : *bean* = EJB = composant

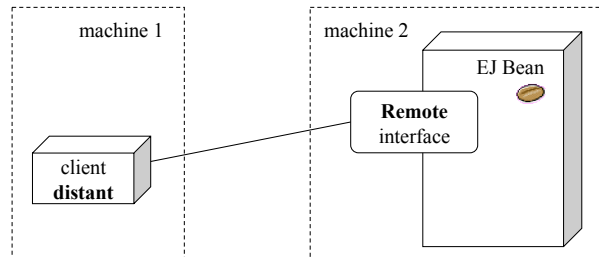
Types d'EJB

- Session : *performs a task for a client*
- Entity : *represents a business entity object that exists in persistent storage*

Plusieurs versions : actuellement EJB 3.1 (depuis 2009)

1. Composant EJB

Enterprise Java Bean

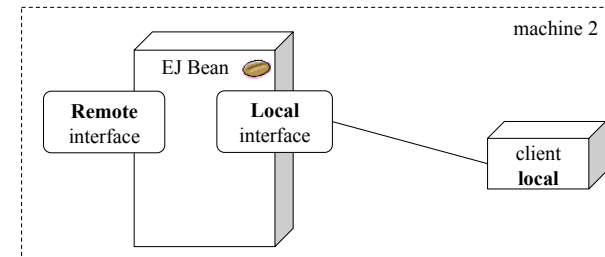


Chaque EJ Bean fournit 1 interface d'accès **distant**

- les services (méthodes) offerts par le bean à ces client

1. Composant EJB

Enterprise Java Bean



+ éventuellement 1 interface d'accès **local** (à partir EJB 2.0)

- les services offerts par le bean à ses clients locaux
- les mêmes (ou d'autres) que ceux offerts à distance
- optimisation

1.1 Session Bean

1. Définition
2. Développement
3. Client local
4. Client distant
5. Stateful session bean

1.1 Session Bean

Définition

Session Bean : représente un traitement (services fournis à un client)

1. Stateless

- sans état
- ne conserve pas d'information entre 2 appels successifs

2. Stateful

- avec un état (en mémoire)
- similaire session servlet/JSP
- 1 instance par client

3. Singleton (nouveau EJB 3.1 – Java EE 6)

- 1 seule instance pour tous les clients

1.1 Session Bean

Développement

1 interface (éventuellement 2 : Local + Remote) + 1 classe

Interface

- annotations @javax.ejb.Local OU @javax.ejb.Remote

```
import javax.ejb.Local;

@Local
public interface CalculatriceItf {
    public double add(double v1,double v2);
    public double sub(double v1,double v2);
    public double mul(double v1,double v2);
    public double div(double v1,double v2);
}
```

1.1 Session Bean

Développement

Classe

- annotations @Stateless OU @Stateful OU @Singleton

```
import javax.ejb.Stateless;

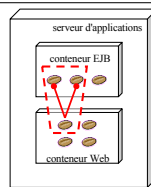
@Stateless(name="maCalcullette")
public class CalculatriceBean implements CalculatriceItf {
    public double add(double v1,double v2) {return v1+v2;}
    public double sub(double v1,double v2) {return v1-v2;}
    public double mul(double v1,double v2) {return v1*v2;}
    public double div(double v1,double v2) {return v1/v2;}
}
```

- paramètre name facultatif
- par défaut, le nom de la classe

1.1 Session Bean

Client

- typiquement un session bean ou une servlet/JSP colocalisée sur le même serveur que le bean
- mécanisme dit "injection de dépendance"
 - attribut du type de l'interface
 - annoté @EJB



```
public class ClientServlet extends HttpServlet {

    @EJB(name="maCalcullette")
    private CalculatriceItf myBean;

    public void service( HttpServletRequest req, HttpServletResponse resp ) {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        double result = myBean.add(12,4.75);
        out.println("<html><body>"+result+"</body></html>");
    }
}
```

1.1 Session Bean

Stateful Session Bean

- instance du bean reste en mémoire tant que le client est présent
- expiration au bout d'un délai d'inactivité
- similaire session JSP/servlet
- utilisation type
 - gestion d'un panier électronique sur un site de commerce en ligne
 - rapport sur l'activité d'un client

2 annotations principales

- @Stateful : déclare un bean avec état
- @Remove
 - définit la méthode de fin de session
 - la session expire à l'issue de l'exécution de cette méthode

1.1 Session Bean

Stateful Session Bean

```
@Stateful
public class CartBean implements CartItf {

    private List items = new ArrayList();
    private List quantities = new ArrayList();

    public void addItem( int ref, int qte ) { ... }
    public void removeItem( int ref ) { ... }

    @Remove
    public void confirmOrder() { ... }
}
```

1.2 Entity Bean

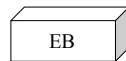
1. Définition
2. Développement
3. Gestionnaire d'entité
4. Relation
5. Autres annotations

1.2 Entity Bean

Définition

Représentation d'une donnée manipulée par l'application

- donnée typiquement stockée dans un SGBD (ou tout autre support accessible en JDBC)



Nom	Solde
John	100.00
Anne	156.00
Marcel	55.25

- correspondance objet – tuple relationnel (*mapping O/R*)
- possibilité de définir des clés, des relations, des recherches
- avantage : manipulation d'objets Java plutôt que de requêtes SQL
- mis en oeuvre à l'aide
 - d'annotations Java 5
 - de la généricité Java 5
 - de l'API JPA (Java Persistence API)

1.2 Entity Bean

Développement

- classe
- annotation `@Entity`
- variables et *getters/setters*
- chaque classe = 1 table
- chaque variable = 1 colonne de la table

1.2 Entity Bean

Développement

```
@Entity
public class Livre {
    private long id;
    private String auteur;
    private String titre;

    public Livre() {}
    public Livre(long id, String auteur, String titre) {
        this.id = id;
        this.auteur = auteur;
        this.titre = titre; }

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getAuteur() { return auteur; }
    public void setAuteur(String auteur) { this.auteur = auteur; }

    public String getTitre() { return titre; }
    public void setTitre(String titre) { this.titre = titre; } }
```

1.2 Entity Bean

Développement

- par défaut même nom de classe et de table
- sauf si annotation `@Table(name="...")`
- par défaut même nom de variable et de colonne
- sauf si annotation `@Column(name="...")`
- annotation variable ou *getter*

1.2 Entity Bean

Développement

```
@Entity
@Table(name="BOOK")
public class Livre {
    private long id;
    @Column(name="AUTHOR") private String auteur;
    @Column(name="TITLE") private String titre;

    public Livre() {}
    public Livre(long id, String auteur, String titre) {
        this.id = id;
        this.auteur = auteur;
        this.titre = titre; }

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getAuteur() { return auteur; }
    public void setAuteur(String auteur) { this.auteur = auteur; }

    public String getTitre() { return titre; }
    public void setTitre(String titre) { this.titre = titre; } }
```

1.2 Entity Bean

Développement

- annotation pour clé primaire `@Id`


```
@Id
    public long getId() { return id; }
```
- annotation pour auto-incrément `@GeneratedValue`


```
@Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public long getId() { return id; }
```
- `GenerationType.AUTO` : les numéros de séquence sont choisis automatiquement
- `GenerationType.SEQUENCE` : un générateur de numéros de séquence est à fournir

1.2 Entity Bean

Gestionnaire d'entités

Entity Manager

- assure la correspondance entre les objets Java et les tables relationnelles
- point d'entrée principal dans le service de persistance
- permet d'ajouter des enregistrements
- permet d'exécuter des requêtes
- accessible via une injection de dépendance
 - attribut de type `javax.persistence.EntityManager`
 - annoté par `@PersistenceContext`

1.2 Entity Bean

Gestionnaire d'entités

Exemple

- création de trois enregistrements dans la table des livres

```
@Stateless
public class MyBean implements MyBeanItf {

    @PersistenceContext
    private EntityManager em;

    public void init() {
        Livre l1 = new Livre(1,"Honore de Balzac","Le Pere Goriot");
        Livre l2 = new Livre(2,"Honore de Balzac","Les Chouans");
        Livre l3 = new Livre(3,"Victor Hugo","Les Miserables");

        em.persist(l1);
        em.persist(l2);
        em.persist(l3);
    }
}
```

- de façon similaire `em.remove(l2)` retire l'enregistrement de la table

1.2 Entity Bean

Gestionnaire d'entités

Recherche par clé primaire

- méthode `find` du gestionnaire d'entités

```
Livre monLivre = em.find(Livre.class,42);
```

- retourne `null` si la clé n'existe pas dans la table
- `IllegalArgumentException`
 - si 1er paramètre n'est pas une classe d'EB
 - si 2ème paramètre ne correspond pas au type de la clé primaire

1.2 Entity Bean

Gestionnaire d'entités

Recherche par requête

- requêtes SQL SELECT

```
Query q1 =
    em.createQuery("SELECT l FROM Livre l where l.auteur='Balzac'");
List<Book> list = (List<Book>) q1.getResultList();
```

- requêtes SQL SELECT paramétrées

```
Query q2 = em.createQuery("SELECT l FROM Livre l where l.id = :id");
q2.setParameter("id",42);
List<Book> list = (List<Book>) q2.getResultList();
```

1.2 Entity Bean

Gestionnaire d'entités

CRUD

```
Livre l1 = new Livre(1,"Honore de Balzac","Le Pere Goriot");
em.persist(l1); // Create

Livre l2 = find(Livre.class,2);
String titre = l2.getTitre(); // Read

l2.setTitre("Balzac");
em.persist(l2); // Update

l2.remove(); // Delete
```

1.2 Entity Bean

Relation 1-n

Entre Auteur et Livre (1 auteur, n livres)

```
@Entity
public class Auteur {
    private long id;
    private String name;
    private Collection<Livre> livres;

    public Author() { livres = new ArrayList<Livre>(); }
    public Author(String nom) { this.nom = nom; }

    @OneToMany
    public Collection<Livre> getLivres() { return livres; }

    public void setLivres( Collection<Livre> livres ) {
        this.livres = livres; }
    ...
}
```

1.2 Entity Bean

Relation 1-n

```
@Entity
public class Livre {
    private long id;
    private Auteur auteur;
    private String titre;

    public Livre() {}
    public Livre(long id, Auteur auteur, String titre) { ... }

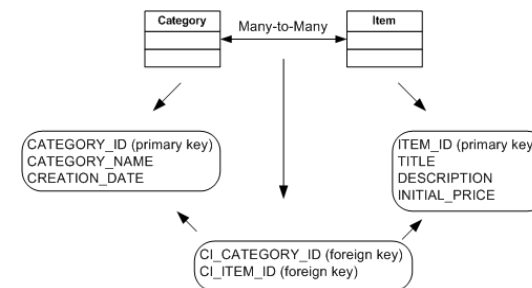
    @ManyToOne
    public Auteur getAuteur() { return auteur; }
    public void setAuteur(Auteur auteur) { this.auteur = auteur; }

    public String getTitre() { return titre; }
    public void setTitre(String titre) { this.titre = titre; }
    ...
}
```

1.2 Entity Bean

Relation n-n

- notion de table de jointure



1.2 Entity Bean

Relation n-n

```
@Entity
public class Category {
    @Id
    @Column(name="CATEGORY_ID")
    private long categoryId;

    @ManyToMany
    @JoinTable(name="CATEGORIES_ITEMS",
        joinColumns=
            @JoinColumn(
                name="CI_CATEGORY_ID",
                referencedColumnName="CATEGORY_ID"),
        inverseJoinColumns=
            @JoinColumn(
                name="CI_ITEM_ID",
                referencedColumnName="ITEM_ID"))
    private Set<Item> items;
}
```

```
@Entity
public class Item {
    @Id
    @Column(name="ITEM_ID")
    private long itemId;

    @ManyToMany
    private Set<Category> categories;
}
```

1.2 Entity Bean

Autres annotations

@Enumerated : définit une colonne avec des valeurs énumérées
EnumType : ORDINAL (valeur stockée sous forme int), STRING

```
public enum UserType {STUDENT, TEACHER, SYSADMIN};
@Enumerated(value=EnumType.ORDINAL)
protected UserType userType;
```

@Lob : données binaires

```
@Lob
protected byte[] picture;
```

@Temporal : dates
TemporalType : DATE (java.sql.Date), TIME (java.sql.Time)
TIMESTAMP (java.sql.Timestamp)

```
@Temporal(TemporalType.DATE)
protected java.util.Date creationDate;
```

1.2 Entity Bean

Autres annotations

@SecondaryTable : mapping d'un EB sur plusieurs tables

```
@Entity
@Table(name="USERS")
@SecondaryTable(name="USER_PICTURES"
    pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))
public class User { ... }
```

@Embeddable et **@Embedded** : embarque les données d'une classe dans une table

```
@Embeddable
public class Address implements Serializable {
    private String rue; private int codePostal; }

@Entity
public class User {
    private String nom;

    @Embedded
    private Address adresse;
}
```

1.2 Entity Bean

Autres annotations

@IdClass : clé composée

```
@Entity
@IdClass(PersonnePK.class)
public class Personne {
    @Id public String getName() { return name; }
    @Id public String getFirstname() { return firstname; }
}

public class PersonnePK implements Serializable {
    private String name;
    private String firstname;
    public PersonnePK( String n, String f ) { ... }
    public boolean equals(Object other) { ... }
    public int hash() { ... }
}
```

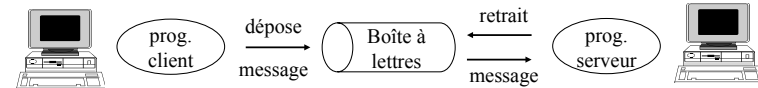
Plan

1. Composants EJB
 - 1.1 Session Bean
 - 1.2 Entity Bean
2. Fonctionnalités avancées
 - 2.1 Message Driven Bean
 - 2.2 Timer Bean
 - 2.3 Intercepteur
 - 2.4 Méthode asynchrone
3. Services
4. Design patterns EJB

2.1 Message-driven Bean

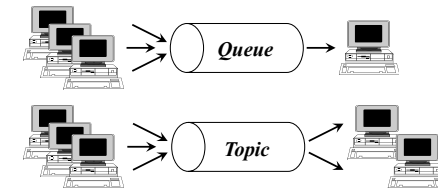
Message-driven bean (MDB)

Interaction **par envoi message asynchrone** (MOM : *Message-Oriented Middleware*)



2 modes

- n vers 1 (*queue*)
- n vers m (*topic*)



2.1 Message-driven Bean

Caractéristiques

Message-Driven Bean : *listener processing messages asynchronously*

- consomme des messages asynchrones
- pas d'état (= *stateless session bean*)
- toutes les instances d'une même classe de MDB sont équivalentes
- peut traiter les messages de clients ≠

Quand utiliser un MDB

- éviter appels bloquants
- découpler clients et serveurs
- besoin de fiabilité : protection *crash* serveurs

Vocabulaire : producteur/consommateur

2.1 Message-driven Bean

Concepts

MDB basé sur Java Messaging Service (JMS) java.sun.com/jms

ConnectionFactory	fabrique pour créer des connexions vers une <i>queue/topic</i>
Connection	une connexion vers une <i>queue/topic</i>
Session	période de temps pour l'envoi de messages dans 1 <i>queue/topic</i>
	peut être rendue transactionnelle
	similitude avec les notions de sessions JDBC, Hibernate, ...

Processus

1. Création d'une connexion
2. Création d'une session (* : éventuellement plusieurs sessions par connexion)
3. Création d'un message
4. Envoi du message
5. Fermeture session
6. Fermeture connexion

2.1 Message-driven Bean

Producteur

```
public class MyProducerBean {  
  
    @Resource(name="jms/QueueConnectionFactory") // l'id de la factory  
    private ConnectionFactory connectionFactory;  
  
    @Resource(name="jms/ShippingRequestQueue") // l'id de la queue  
    private Destination destination;  
  
    public void produce() {  
        Connection connection = connectionFactory.createConnection();  
        Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);  
        MessageProducer producer = session.createProducer(destination);  
  
        TextMessage message = session.createTextMessage();  
        message.setText("Hello World!");  
        producer.send(message);  
  
        session.close();  
        connection.close();  
    }  
}
```

2.1 Message-driven Bean

Consommateur

MDB = classe

- annotée @MessageDriven

- implémentant interface MessageListener

- méthode void onMessage(Message)

```
@MessageDriven(name="jms/ShippingRequestProcessor")  
public class MyConsumerBean implements MessageListener {  
  
    public void onMessage( Message m ) {  
        TextMessage message = (TextMessage) m;  
        ...  
    }  
}
```

2.2 Timer Bean

Timer bean

Déclenchement d'actions périodiques

- automatiquement
- programmiquement

Définition automatique de Timer beans

- annotation @Schedule pour définir la périodicité

```
@Stateless  
public class NewsLetterGeneratorBean implements NewsLetterGenerator {  
    @Schedule(second="0", minute="0", hour="0",  
              dayOfMonth="1", month="*", year="*")  
    public void generateMonthlyNewsLetter() { ... }  
}
```

➤ déclenchement tous les 1ers du mois à 00h00

2.2 Timer Bean

Timer bean

Définition programmatique de Timer beans

- @Timeout : méthode exécutée à échéance du *timer*
: void <methodname>(javax.ejb.Timer timer)
- @Resource : attribut de type javax.ejb.TimerService
- utilisation des méthodes de TimerService pour créer des *timers*
 - createTimer(long initialDuration, long period, Serializable info)

```
public class EnchereBean {  
    @Resource TimerService ts;  
  
    public void ajouterEnchere( EnchereInfo e ) {  
        ts.createTimer(1000,25000,e); }  
  
    @Timeout  
    public void monitorerEnchere( Timer timer ) {  
        EnchereInfo e = (EnchereInfo) timer.getInfo(); ... }  
}
```


2.3 Intercepteur

Intercepteur

Permettent d'implanter des traitements avant/après les méthodes d'un *bean*

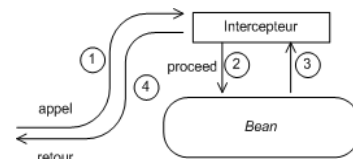
- `@Interceptors` : les méthodes devant être interceptées
- `@AroundInvoke` : les méthodes d'interception

profil de méthode

`Object <methodname>(InvocationContext ctx) throws Exception`

`javax.interceptor.InvocationContext`

- permet d'obtenir des informations (introspecter) sur les méthodes interceptées
- fournit une méthode `proceed()` pour exécuter la méthode interceptée



2.3 Intercepteur

Intercepteur

Plusieurs méthodes dans des classes ≠ peuvent être associées à `MyInterceptor`

```
public class EnchereBean {
    @Interceptors(MyInterceptor.class)
    public void ajouterEnchere( Bid bid ) { ... } }

public class MyInterceptor {
    @AroundInvoke
    public Object trace( InvocationContext ic ) throws Exception {
        // ... code avant ...

        java.lang.reflect.Method m = ic.getMethod();
        Object bean = ic.getTarget();
        Object[] params = ic.getParameters();
        // éventuellement modification paramètres avec ic.setParameters(...)

        Object ret = ic.proceed(); // Appel du bean (facultatif)

        // ... code après ...

        return ret; } }
```

2.4 Méthode asynchrone

Méthode asynchrone

- invocation asynchrone des méthodes d'un *bean*

```
@Stateless
public class OrderBillingServiceBean implements OrderBillingService {
    @Asynchronous
    public void billOrder(Order order) { ... }
}
```

- ne remplace pas *message-driven bean*
- s'implifie le développement pour beaucoup de cas asynchrones simples
- définition d'un objet dit futur en cas de résultat

```
@Asynchronous
public Future<OrderStatus> billOrder(Order order) { ... }
```

- interface `java.util.concurrent.Future` (depuis JDK 6)
- méthodes `isDone()`, `get()`

Plan

1. Composant EJB
 - 1.1 Session Bean
 - 1.2 Entity Bean
2. Fonctionnalités avancées
3. Services
 - 3.1 Transaction
 - 3.2 Sécurité
4. Design patterns EJB

3.1 Transactions

Service de transactions

Assure des propriétés **ACID** pour des transactions plates

Exemple classique : un transfert bancaire (débit, crédit)

- atomicité : soit les 2 opérations s'effectuent complètement, soit aucune
- cohérence : le solde d'un compte ne doit jamais être négatif
- isolation : des transferts // doivent fournir le même résultat qu'en séq.
- durabilité : les soldes doivent être sauvegardés sur support stable

3.1 Transactions

Définition des transactions

2 modes

- BMT (*Bean Managed Transaction*) : API JTA
- CMT (*Container Managed Transaction*) : annotations

3.1 Transactions

Définition des transactions

BMT

- démarcation explicite avec begin/commit/rollback
- possibilité granularité plus fine qu'une méthode

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class MyBean implements MyBeanItf {
    public void transfert() {
        try {
            ut.begin();
            Account a1 = em.find(Account.class, "Bob");
            Account a2 = em.find(Account.class, "Anne");
            a1.credit(10.5);
            a2.withdraw(10.5);
            ut.commit();
        } catch (Exception e) { ut.rollback(); } }
}
```

```
@PersistenceContext
private EntityManager em;

@Resource
private UserTransaction ut; }
```

3.1 Transactions

Définition des transactions

CMT

- toute la méthode est considérée comme un bloc transactionnel
- *commit* par défaut en fin de méthode

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class MyBean implements MyBeanItf {
    @TransactionAttribute(TransactionAttributeType.xxxxx)
    public void transfert() {
        try {
            Account a1 = em.find(Account.class, "Bob");
            Account a2 = em.find(Account.class, "Anne");
            a1.credit(10.5);
            a2.withdraw(10.5);
        } catch (Exception e) { ut.setRollbackOnly(); } }
}
```

3.1 Transactions

Granularité des transactions

Attribut transactionnel avec 6 valeurs

- REQUIRED
- REQUIRES_NEW
- SUPPORTS
- NOT_SUPPORTED
- MANDATORY
- NEVER

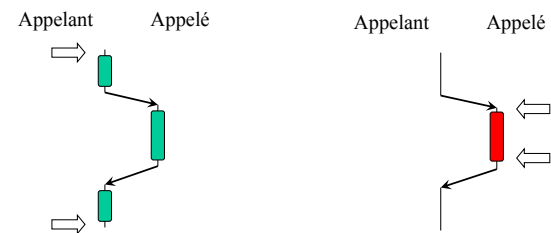
2 cas pour le *bean* appelant

- soit il s'exécute dans une transaction
- soit il s'exécute en dehors de tout contexte transactionnel

3.1 Transactions

Granularité des transactions

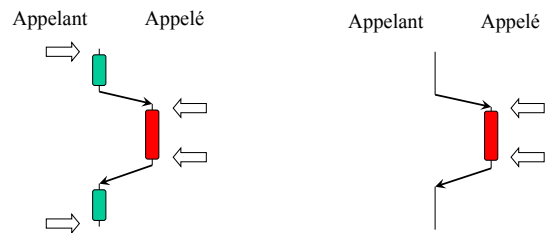
REQUIRED



3.1 Transactions

Granularité des transactions

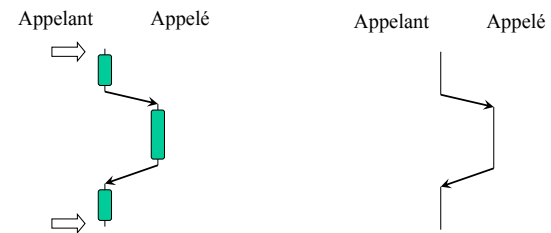
REQUIRES_NEW



3.1 Transactions

Granularité des transactions

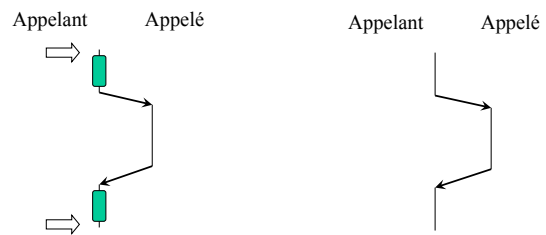
SUPPORTS



3.1 Transactions

Granularité des transactions

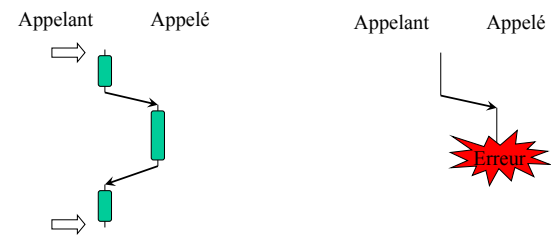
NOT_SUPPORTED



3.1 Transactions

Granularité des transactions

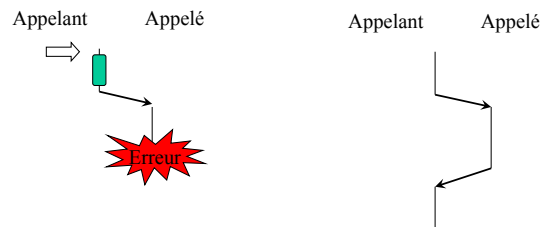
MANDATORY



3.1 Transactions

Granularité des transactions

NEVER



Plan

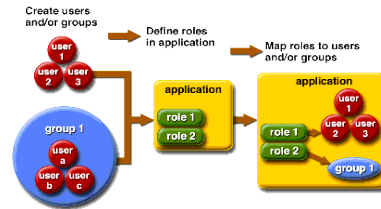
1. Composant EJB
 - 1.1 Session Bean
 - 1.2 Entity Bean
2. Fonctionnalités avancées
3. Services
 - 3.1 Transaction
 - 3.2 Sécurité
4. Design patterns EJB

3.2 Sécurité

Service de contrôle d'accès

- contrôler l'accès aux méthodes d'un bean
- authentifier l'utilisateur

À base de rôles



5 annotations

```
javax.annotation.security.PermitAll
javax.annotation.security.DenyAll
javax.annotation.security.RolesAllowed
javax.annotation.security.DeclareRoles
javax.annotation.security.RunAs
```

3.2 Sécurité

Annotations	Target		EJB or its super classes	Servlet or web libraries	Descriptions
	TYPE	METHOD			
@PermitAll	X	X	X		Indicates that the given method or all business methods of the given EJB are accessible by everyone.
@DenyAll		X	X		Indicates that the given method in the EJB cannot be accessed by anyone.
@RolesAllowed	X	X	X		Indicates that the given method or all business methods in the EJB can be accessed by users associated with the list of roles.
@DeclareRoles	X		X	X	Defines roles for security checking. To be used by <code>EJBContext.isCallerInRole</code> , <code>HttpServletRequest.isUserInRole</code> , and <code>WebServiceContext.isUserInRole</code> .
@RunAs	X		X (not for non-EJB super classes)	X (for servlet only)	Specifies the run-as role for the given components.

3.2 Sécurité

Service de contrôle d'accès

Exemple

```
@Stateless
@RolesAllowed("user")
public class Client {

    public String getClientNameById(long id) { return ... }

    @RolesAllowed("admin")
    public void addClient( ... ) { ... }
}
```

3.2 Sécurité

Service de contrôle d'accès

Pour les cas plus complexes : @DeclaresRoles

Exemple 2

```
@Stateless
@DeclaresRoles({"user", "admin"})
public class Client {

    @Resource private SessionContext sc;

    public void forUsersThatAreNotAdmins() {
        if (sc.isCallerInRole("user") && !sc.isCallerInRole("admin")) {
            ...
        } else {
            ...
        }
    }
}
```

Plan

1. Composant EJB
 - 1.1 Session Bean
 - 1.2 Entity Bean
2. Fonctionnalités avancées
3. Services
 - 3.1 Transaction
 - 3.2 Sécurité
4. Design patterns EJB

4. Design Patterns EJB

Gabarit de conception (*design pattern*)

Problèmes de codage récurrents

- parcourir un arbre de données dont les noeuds sont typés
 - maj une fenêtre en fonction de modifications sur des données (et vice-versa)
 - ...
- ⇒ design pattern (DP) : solutions reconnues d'organisation du code

But

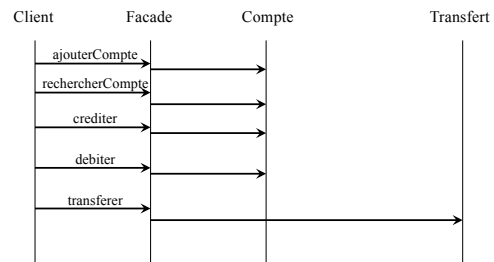
- améliorer la clarté, la compréhension du code
- mettre en avant des éléments d'architecture logicielle

4. Design Patterns EJB

DP Session facade

Pb : nombreuses dépendances entre les clients et les beans

Solution : présenter aux clients **une seule interface façade** (*stateless session bean*)

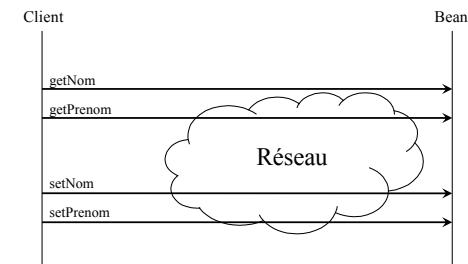


4. Design Patterns EJB

DP Data Transfert Object

Aussi connu sous le terme : Value Object

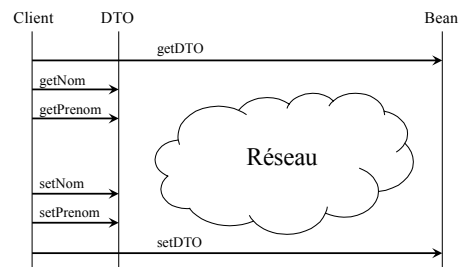
Pb : nombreux échanges réseaux pour de simples get/set



4. Design Patterns EJB

DP Data Transfert Object

Solution : transmettre une instance par valeur



4. Design Patterns EJB

Autres DP

- application service
centraliser un processus métier s'étendant sur plusieurs *beans*
- composite entity
rassembler dans 1 seul EB les données persistantes de plusieurs EB
- transfert object assembler
aggregation de plusieurs DTO
- service activator
invoquer des services de façon asynchrone