

Université Montpellier-II - Master Info-Pro-M2

Parcours GL

Réutilisation et Composants

Partie IV - Composants pour les applications
distribuées - l'Exemple des **Enterprise Java**

Beans - JEE

Notes de Cours - Christophe Dony - 2002-2012

1 Applications Distribuées

1.1 Définitions préalables

- **Applications distribuées** : Applications mettant en jeux différents processus s'exécutant sur des machines distantes.
- **mots clés** : mondialisation, décentralisation des ressources et des connaissances, grid, travail à distance, répartition du calcul, applications multi-niveaux.
- **Network Operating system** système d'exploitation intégrant une couche de communication réseau.
- **Middleware (Intergiciel)** Applications logicielles s'exécutant à la fois côté client et côté serveur et offrant en premier lieu le service de **distribution** et d'hétérogénéité [IBM Mainframe (big-endian, EBCDic) - Unix (Little-endian, 8bit-iso)].

Un intergiciel offre potentiellement en plus les services de :

- **interopérabilité** : communication entre objets écrits avec différents langages. application en C++ faisant appel à un programme distant écrit en JAVA. (Exemple : CORBA).
- **Transactions** : concept à la base de l'échange commercial informatisé, exécution d'une ou plusieurs tâches accédant à une ou plusieurs ressources sur différents serveur (dont des bases de données) offrant un ensemble de garanties :
 - Atomicité : une transaction est soit validée soit entièrement annulée.
 - Cohérence
 - Isolation : Les données manipulées dans une transaction ne sont pas accessibles à d'autres.
 - Durabilité : Le nouvel état des données (après la transaction) est enregistré dans la base.

- **Sécurité**, contrôles d'accès.
- **Nommage** (API JNDI : interface pour se connecter aux services de noms ou d'annuaires).
- gestion transparente de la communication bas-niveau entre divers clients et EJB distants (**J2EE remote connectivity**).

Exemples d'intergiciels.

- Orientés transaction : IBM CICS, Tuxedo
- RPC Middleware,
- Object-Oriented Middleware :
- RMI, OMG/Corba, Microsoft/Com, Java/RMI,
- **ORB** : *Object Request Broker*. Intergiciel, dit bus d'objets ou bus logiciel, assurant l'appel de service à distance.

CORBA (*Common Object Request Broker Architecture*) : Un middleware pour les applications distribuées et l'interopérabilité, incluant un bus logiciel et un ensemble de services.

- **Serveur d'applications** : Autre nom donné au middleware côté serveur. Exemple : JBOSS est un serveur d'applications J2EE libre écrit en Java.
- **Objet distribué** : objet, participant d'une application, accessible à distance via un intergiciel.
- **Application client-serveur** : application dans laquelle des machines clientes utilisent des services délivrées par des serveurs distantes.
- **Applications P2P** "peer-to-peer". Tous les acteurs du système distribué peuvent jouer le rôle de client et de serveur.

– Protocoles réseau

- **Protocole** : langage pour la communication réseau.
- **FTP** : File Transfert Protocol. Protocole de transfert de fichiers basé sur TCP/IP.
- **HTTP** : HyperText Transfert Protocol, le protocole pour les requêtes WEB et l'échange de pages web.
- **SSL** : Secure Socket Layer protocole de communication sécurisé basé sur TCP/IP (encryptage + authentification)
- **HTTPS** : HTTP au dessus de SSL.
- **GIOP** : General inter-ORB protocol.
- **IIOP** : *Internet Inter-ORB Protocol* spécialisation de IIOP pour les réseaux TCP-IP. Tous les ORB s'exécutant sur différentes machines peuvent communiquer via le protocole IIOP.
- **SOAP** : protocole indépendant des langages (basé sur XML) utilisé pour les services WEB.

1.2 Applications multi-tiers (multi-niveaux)

Application multi-tier : Application distribuée architecturée en “niveaux” ou parties.

tier : (anglais) niveau, couche.

Exemple typique : application 3 niveaux (voir figure 1)

- Niveau interfaçage : postes clients accédant à des ressources distantes. Machine légères.
- Niveau serveur de calcul dit *niveau métier* ou encore *niveau milieu* (*middle tier*). Serveur de calcul. Usuellement situé au coeur de l’entreprise.
- Niveau serveur de données. Gestion du système d’information de l’entreprise intégrant les bases de données. Machines dédiées usuellement situées au coeur de l’entreprise et hautement sécurisées.

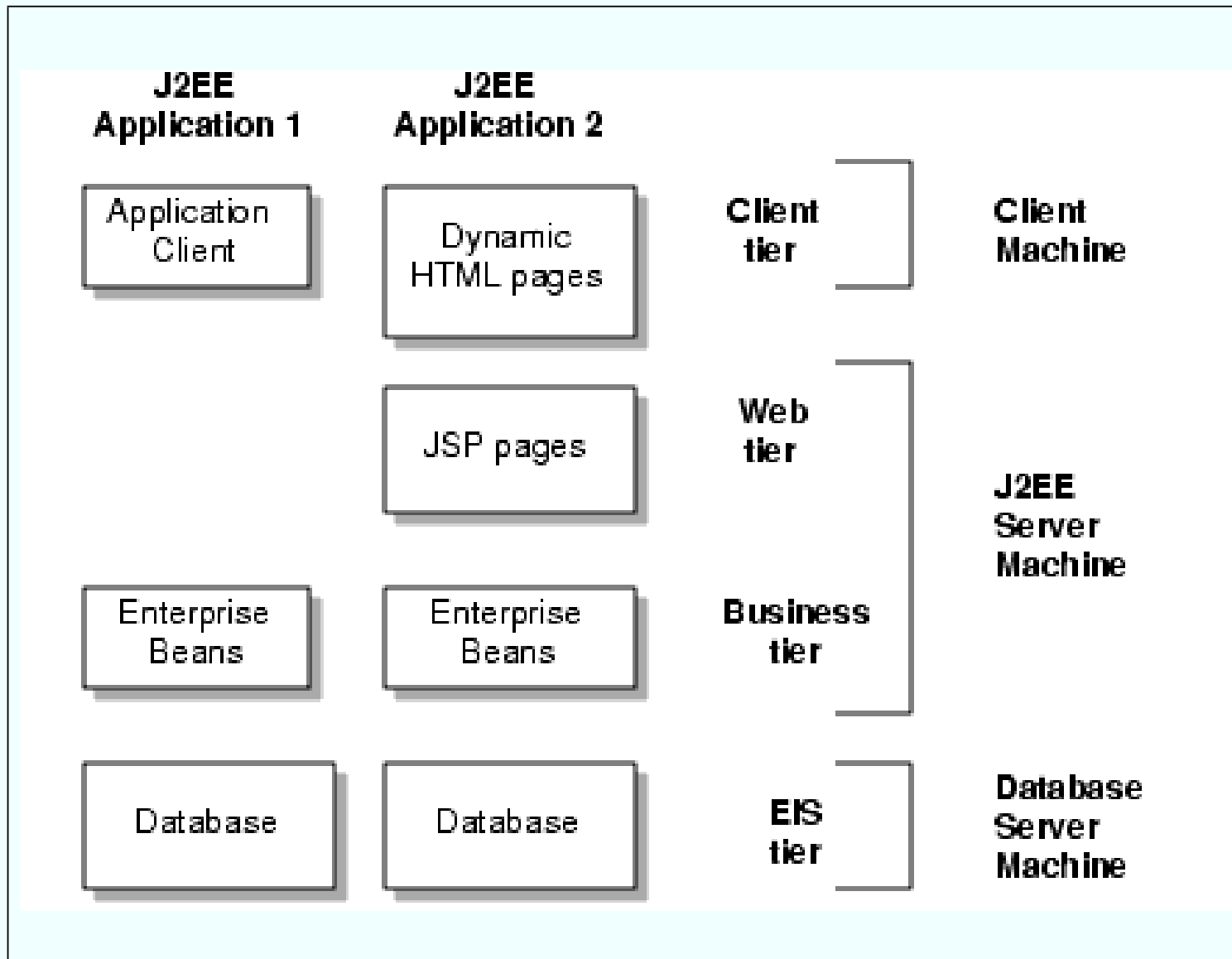


FIG. 1 – Applications 3 niveaux - version j2ee ©j2eeTutorial

2 applications distribués multi-niveaux à base de composants : l'exemple de JEE

JEE (anciennement **J2EE**) : *Java Platform Enterprise Edition*, une solution globale à base de composants pour les applications n-tier en Java.

Application EE (Différence avec application J2SE) : regroupe un ensemble de composants suivant la spécification EE et doit être déployée dans un serveur d'application EE (voir figure 2) pour être exécutée.

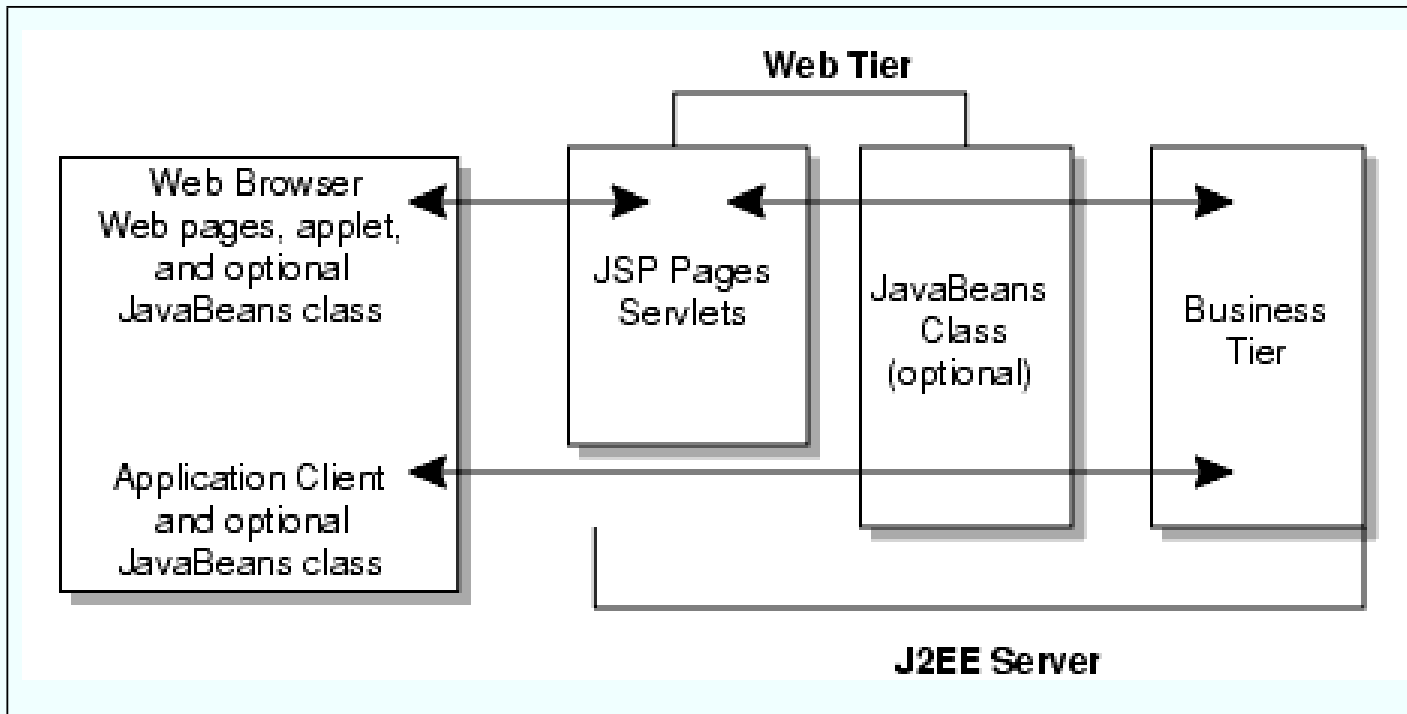


FIG. 2 – Serveur JEE ©j2ee tutorial - Sun

2.1 Composants

Les composants EJB relèvent du concept général de composant (voir cours “introduction aux composants”).

Fourni : Ils proposent différentes fonctionnalités décrites via des interfaces.

Requis : L’expression de ce que requiert un composant EJB n’est pas explicite.

Les EJB sont des composants en ce qu’ils bénéficient d’un ensemble de services fournis par le middleware dans lequel ils sont déployés, représenté par le concept de conteneur.

La norme EJB nomme composants :

- Les **EJB (Enterprise Java Bean)** : objets distribués, serveurs de calcul et/ou de stockage, gérés par des conteneurs
- Les **Composants WEB ou composants d’interface** : programmes (**Servlet, page JSP, Java Beans**), utilisés pour le niveau “présentation”.

– Les **Clients**

Clients **légers** (navigateurs) ou **lourds** (application java) utilisant les composants distants.

2.2 Conteneurs

Conteneur (d'EJB, de Servlet) : entité logicielle constitutive du *middleware*, configurable, gérant et permettant de régler la vie (réation, destruction, mise en réserve (désactivation ou *passivation*, réactivation (sortie de réserve)) ou encore l'exécution des composants et des services.

Un même composant déployé dans des conteneurs différemment configurés aura des comportements différents.

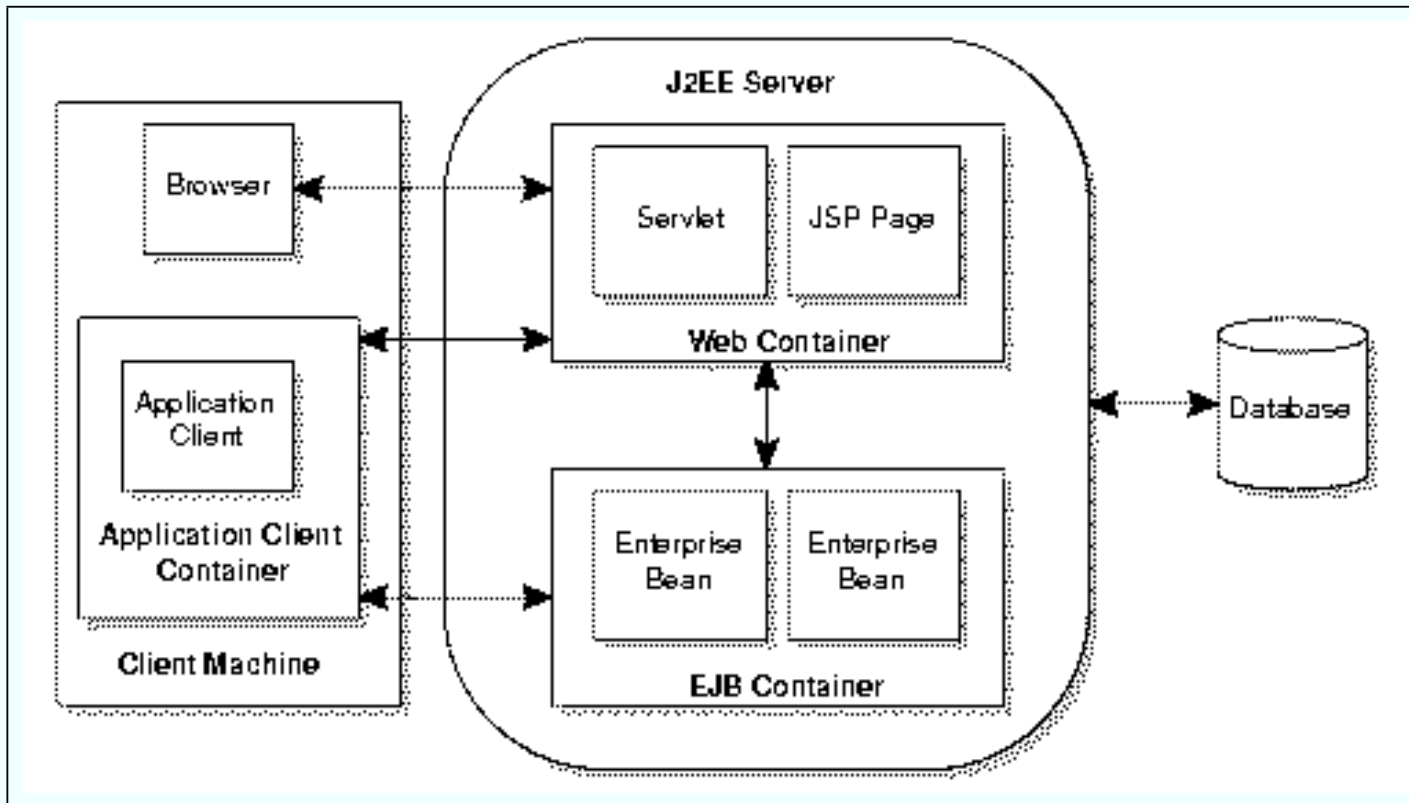


FIG. 3 – Conteneurs et Composants j2ee ©Sun

2.3 Packaging - Déploiement

Application j2ee : ensemble de modules.

Module : entité regroupant un composant (code compilé), ses différents fichiers associés, son conteneur, et un descripteur de déploiement.

Descripteur de déploiement : fichier (XML) qui spécifie les différentes propriétés du module i.e. la configuration des différents services du conteneur.

Descripteur d'application Une application j2ee complète se présente comme un fichier de type **EAR** (*Enterprise ARchive*), sorte de JAR, contenant l'ensemble des modules.

3 Les composants EJB

Un composant EJB représente un composant dans l'application Serveur.

Version 2.0 : Trois sortes d'EJB : *Session*, *Entity*, *Message-Driven*.

Version 3.0 : Les *entity* redeviennent des objets standards pour le programmeur. (POJOS *plain old java objects*).

Les composants orientés message sont récepteurs de messages asynchrones JMS (*Java Messaging Service*) et s'apparentent aux boîtes aux lettres des agents. Ils peuvent distribuer du travail pour différents EJB session associés.

3.1 Objets distribués

Les EJB sont des objets distribués par RMI. Un EJB possède :

- une interface dite “distante” (*remote interface*) définissant le comportement “métier” du composant (ce qu’il sait faire).
- une interface dite “locale” (“home interface”) définissant les méthodes relatives au cycle de vie du composant (*create, remove, ...*).

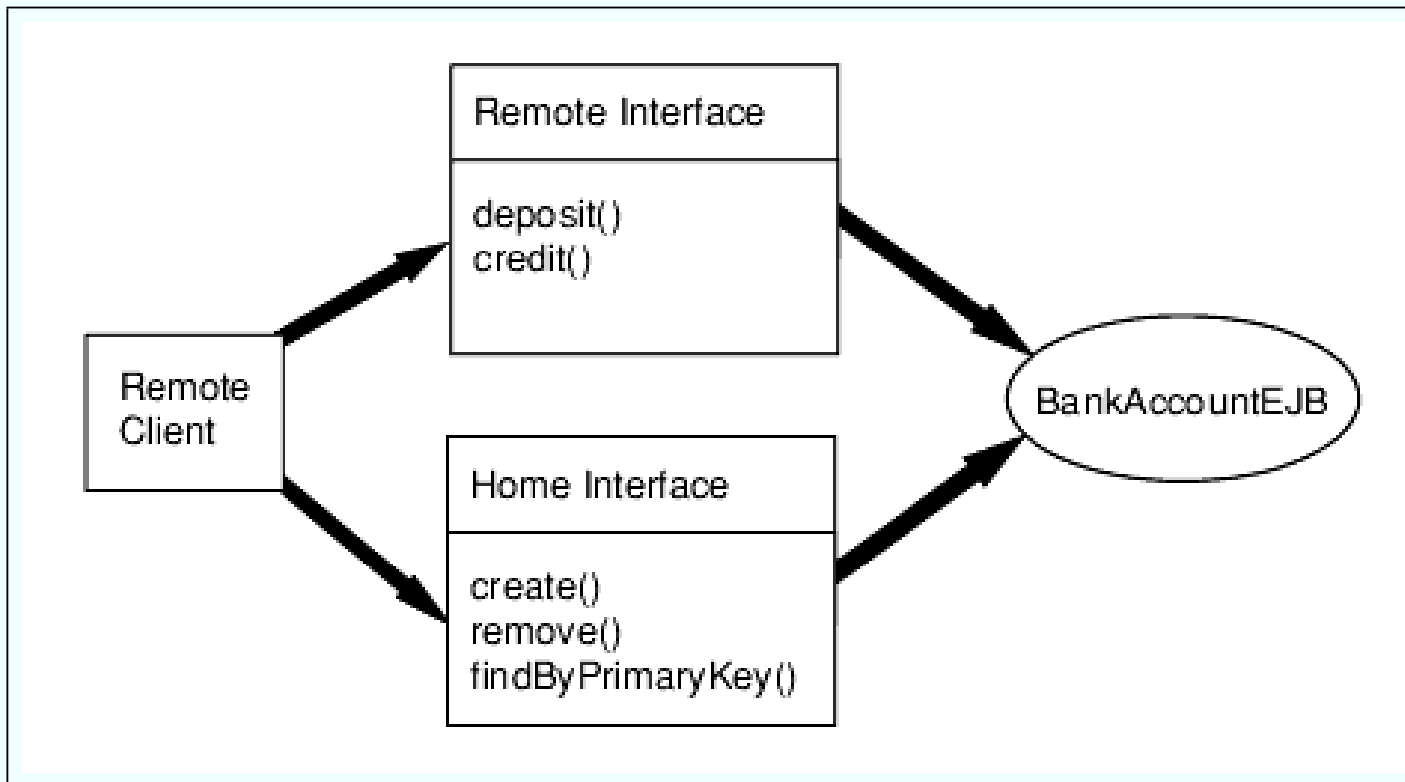


FIG. 4 – Les EJB sont des objets distribués. ©j2ee tutorial - Sun

3.2 Session Beans

Composant dont la durée de vie est une session, capable de fournir un ou des services à des applications clientes.

La création, destruction, optimisation (possibilité de mise en réserve en attendant un autre client (*pooling*) sont gérés par le conteneur.

– **Session Bean sans état (stateless) :**

Collection de services dont chacun est représenté par une méthode.

Chaque service est autonome et ne dépend pas d'un contexte particulier. Aucun état n'est conservé entre deux exécutions de méthodes.

Un session bean servir plusieurs clients simultanément.

Exemple de service fourni : donner l'heure, donner un horaire, appliquer un algorithme sur une donnée, enregistrer une réservation ...

- **Session Bean avec état (statefull)** : composant mémorisant les informations relatives à une conversation avec un client (conversational state).

Exemple : interaction avec un client réalisant des achats sur un site de vente.

Utilisé pour tenir une conversation avec un client. L'état de la "conversation" (conversational state) est stocké dans ses attributs et est retenu pour la durée d'une session. Sert un seul client à la fois.

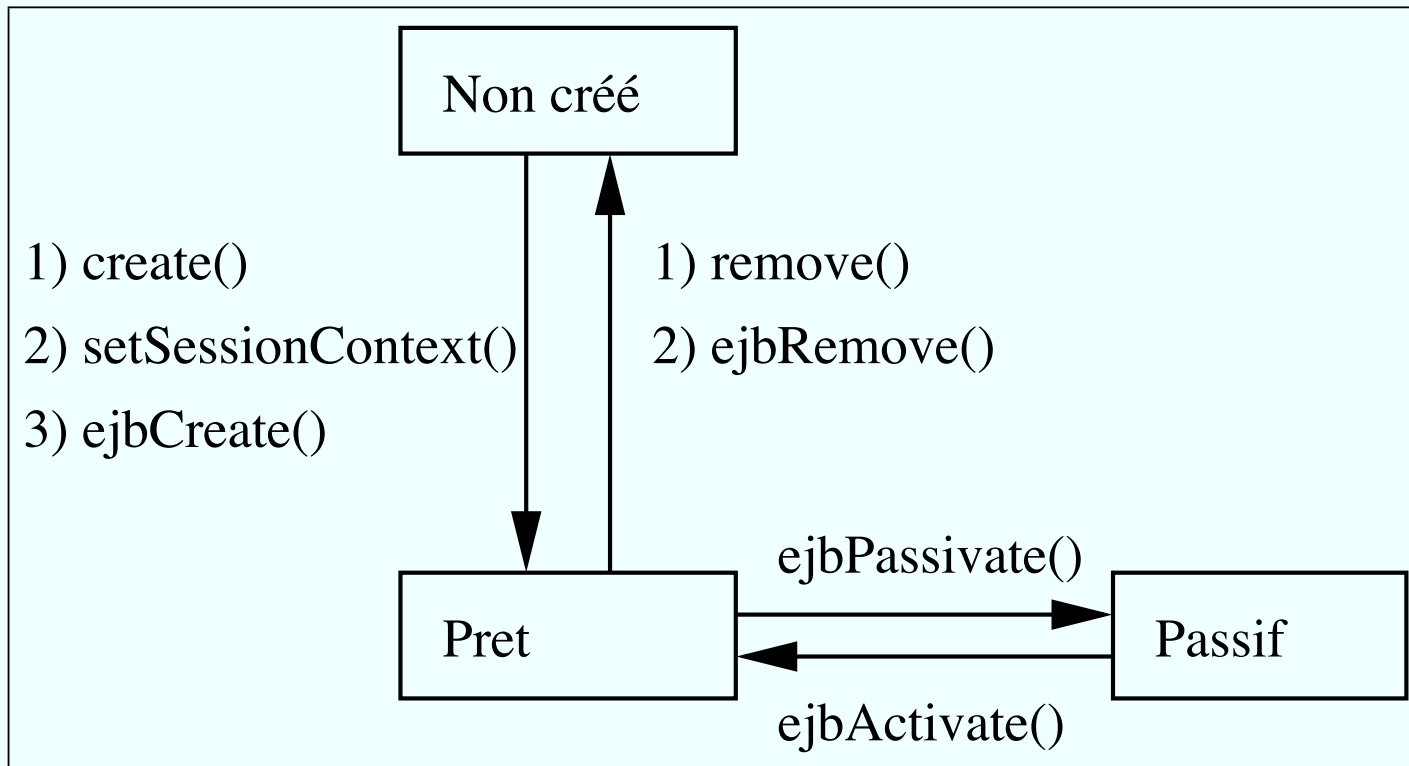


FIG. 5 – Cycle de vie d'un session-bean avec état

Un client crée un ejb en envoyant le message `create` au talon client. `create` est une méthode du framework, paramétrée par `EjbCreate()` (callback). `EjbCreate` permet au programmeur de l'EJB de finaliser l'initialisation si nécessaire.

3.3 Message Driven Bean

Message Driven Bean : Sorte de *stateless session bean* gérant les messages asynchrones.

- Ecouteur de messages *JMS*.
- N'a ni home ni remote interface, ils permet à un conteneur d'affecter un message à n'importe quel composant d'un pool pour des exécution concurrentes de services.
- Quand un message arrive, le conteneur invoque la méthode *onMessage* du MDB, dont le paramètre doit être de l'un des 5 types prédéfinis de messages JMS.
- Avec les EJB3, un schéma d'utilisation typique d'un MDB, via un session bean dit “de façade”, est proposé.

4 Exemple : Création et déploiement d'une application JEE - EJB version 2.0 (Obsolète, pour comprendre les bases)

La norme EJB-3 (voir section 9) a rendu beaucoup plus simple la programmation des EJB et de leurs clients. Les environnements de développement ont également évolué. L'outil *deploytool* utilisé dans cette section préfigure les modes EJB pour *Netbeans* ou *Eclipse*.

L'exemple montre le développement de trois composants :

- un composant métier EJB écrit en Java,
- un composant WEB écrit en JSP,
- un composant client écrit en Java.

L'application peut s'utiliser de deux manières sur un poste client.

- via un navigateur (sur la machine cliente) chargeant et affichant le composant WEB du serveur.
- via une application (sur la machine cliente) qui utilise directement le

composant métier.

La configuration et l'emballage sont réalisés avec l'outil de base de sun : *deploytool*.

4.1 Un EJB convertisseur de devises - EJB version 2.0

Soit à réaliser un EJB de type *stateless session bean* appelé `converterEJB` réalisant une conversion entre monnaies.

4.1.1 Conventions de nommage

Item	Syntax	Example
Enterprise bean name (DD)	"name" EJB	AccountEJB
EJB JAR display name (DD)	"name" JAR	AccountJAR
Enterprise bean class	"name" Bean	AccountBean
Home interface	"name" Home	AccountHome
Remote interface	"name"	Account
Local home interface	Local" name" Home	LocalAccountHome
Local interface	Local" name"	LocalAccount

4.1.2 Interface de l'objet distant (Remote Interface)

L'interface "distante" définit le savoir-faire du composant, utilisable par ses clients.

```
//Fichier Converter.java
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Converter extends EJBObject {
    public double dollarToYen(double dollars) throws RemoteException;
    public double yenToEuro(double yen) throws RemoteException;
}
```

4.1.3 Home Interface

Un composant EJB est un objet distribué qui est manipulé côté client via un talon.

La *home interface* (ou interface de fabrique) définit les méthodes applicables au talon côté client^a.

```
//Fichier ConverterHome.java.  
import java.io.Serializable;  
import java.rmi.RemoteException;  
import javax.ejb.CreateException;  
import javax.ejb.EJBHome;  
  
public interface ConverterHome extends EJBHome {  
    Converter create() throws RemoteException, CreateException;}  
}
```

^aEJB2.0 a introduit le concept de local home interface pour le cas où un objet distribué est local ...

La méthode *create* exécutée côté client provoque, dans le cas d'un *session bean* sans état, l'invocation des méthodes *newInstance*, *setSessionContext* et *ejbCreate* côté serveur. Une fois cette méthode exécutée, le composant, un talon côté client et un autre côté serveur existent, le composant distant est prêt à être utilisé.

4.1.4 Le code métier du composant

Le composant lui-même est défini par une classe qui implémente une des deux interfaces *sessionBean* ou *entityBean* selon ce qu'il est, ici un session bean donc. Le fait qu'il n'ait pas d'état se manifeste par l'absence d'attribut.

```
//fichier ConverterBean.java
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class ConverterBean implements SessionBean {
    public double dollarToYen(double dollars) {
        return dollars * 121.6000; }
    public double yenToEuro(double yen) {
        return yen * 0.0077; }

    public ConverterBean() {}
    public void ejbCreate() {} //requis par l'interface sessionBean
    public void ejbRemove() {} //dites call-back
    public void ejbActivate() {} //permettent de particulariser
    public void ejbPassivate() {} //ces méthodes de base du cycle de vie
    public void setSessionContext(SessionContext sc) {}}
```

4.1.5 Les callbacks : paramétrage par composition du framework JEE

En définissant le code d'un composant, on étend un framework par composition (cf. chapitre 1).

Le framework J2EE gère un composant en laissant au programmeur la possibilité de paramétrer son comportement aux moments importants de son existence.

Si le client souhaite intervenir au moment de son activation, il peut définir la méthode *ejbActivate()*.

Les appels à ces méthodes sont qualifiés des *callback* selon le principe propre aux frameworks (hollywood principle).

4.1.6 Configuration et Emballage

Un ejb est décrit par un fichier xml généralement nommé *ejb-jar*. Il décrit les réglages (options de nommage, options relatives au conteneur, etc) du composant.

Il est par exemple possible d'enregistrer le composant (ici *SimpleConverter*) sous un nom différent de celui de sa classe.

Tous les fichiers représentant un composant ou une application sont emballés dans des fichiers appelés *war* comme *web archive* ou *ear* comme *Enterprise Archive*.

Ces fichiers sont créés automatiquement si on utilise des environnements intégrés de développement.

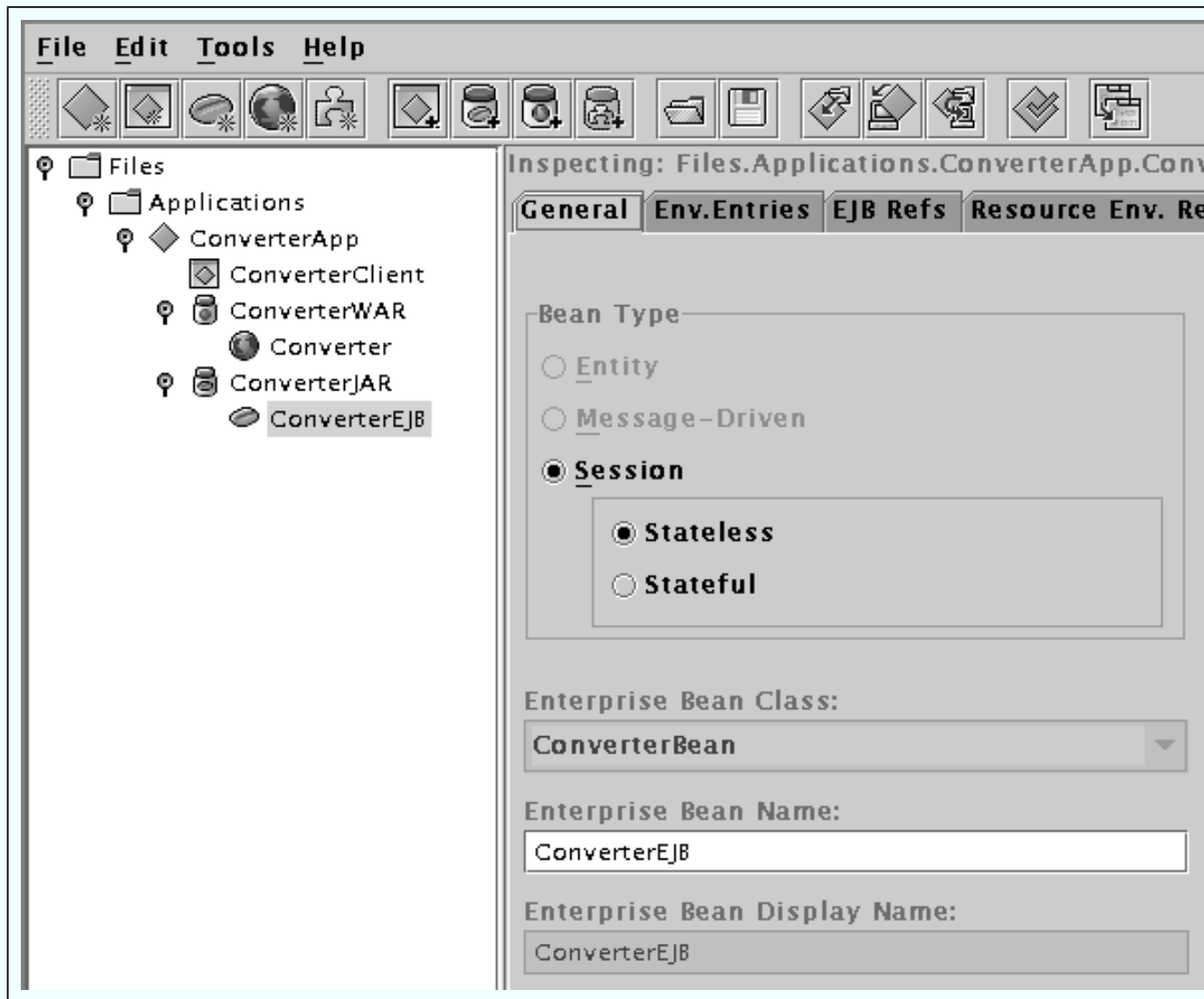


FIG. 6 – Configuration et Emballage d'un EJB avec *sun-deploytool*

4.2 Utilisation d'un EJB par une application cliente

Le code source typique d'un client est constitué de :

- Obtenir une référence (généralement appelée la *fabrique*) qui implémente l'interface home d'un EJB, via JNDI.
- la création d'une instance distante de l'EJB et récupération d'une référence sur cette instance typée par l'interface "distante",
- l'invocation d'une méthode via cette référence.

4.2.1 Recherche du composant

- Création d'une instance d'un objet annuaire (voir l'API *naming* et voir le fichier *jndi.properties*). Les instances de la classe `InitialContext` représentent les annuaires les plus simples.

```
Context initial = new InitialContext();
```

- Recherche, à partir du nom de l'EJB que l'on souhaite utiliser, d'un objet de type "fabrique".

```
Object objref = initial.lookup  
    ("java:comp/env/ejb/SimpleConverter");
```

- La chaîne "java :comp/env" : cuisine interne du serveur j2ee standard non portable).
- mentionner le sous-répertoire ejb indique que l'on recherche un composant ejb (le service d'annuaire permet de rechercher d'autres types de choses, par exemple l'adresse d'une base de donnée),
- on termine par le nom du composant tel qu'il est répertorié dans l'annuaire.

- Obtention d'une référence bien typée sur un objet (*home*) représentant l'objet distant. La mécanique mise en jeu vise à assurer la compatibilité avec CORBA.

```
ConverterHome home =
```

```
    (ConverterHome) PortableRemoteObject.narrow(objref,  
    ConverterHome.class);
```

4.2.2 Création de l'EJB distant

Il est possible d'envoyer via la référence *home* ainsi obtenue tous les messages déclarés dans la *home interface* du composant. La méthode *create* rend une instance d'un objet local au client, de type *Converter* représentant le convertisseur (un *stub*).

```
Converter currencyConverter = home.create();
```

4.2.3 Utilisation de l'EJB distant

L'envoi d'un message au *stub currencyConverter* déclenche le mécanisme d'invocation à distance et de rappatriement du résultat.

```
double amount = currencyConverter.dollarToYen(100.00);
```

4.3 Exemple d'un client lourd

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import Converter;
import ConverterHome;

public class ConverterClient {

    public static void main(String[] args) {
        try {
            Context initial = new InitialContext();
            Object objref = initial.lookup
                ("java:comp/env/ejb/SimpleConverter");
            ConverterHome home =
                (ConverterHome)PortableRemoteObject.narrow(
                    objref, ConverterHome.class);
            Converter currencyConverter = home.create();
```

```
double amount =
    currencyConverter.dollarToYen(100.00);
System.out.println(String.valueOf(amount));
amount = currencyConverter.yenToEuro(100.00);
System.out.println(String.valueOf(amount));
currencyConverter.remove();

} catch (Exception ex) {
    System.err.println("Caught an unexpected exception!");
    ex.printStackTrace();
}
}
```

4.3.1 Emballage du client

L'application cliente est définie par deux JAR.

Le premier contient le code compilé et le descripteur de déploiement.

Le second contient le programme automatiquement généré pour la connexion de

ce client avec les EJB qu'il utilise.

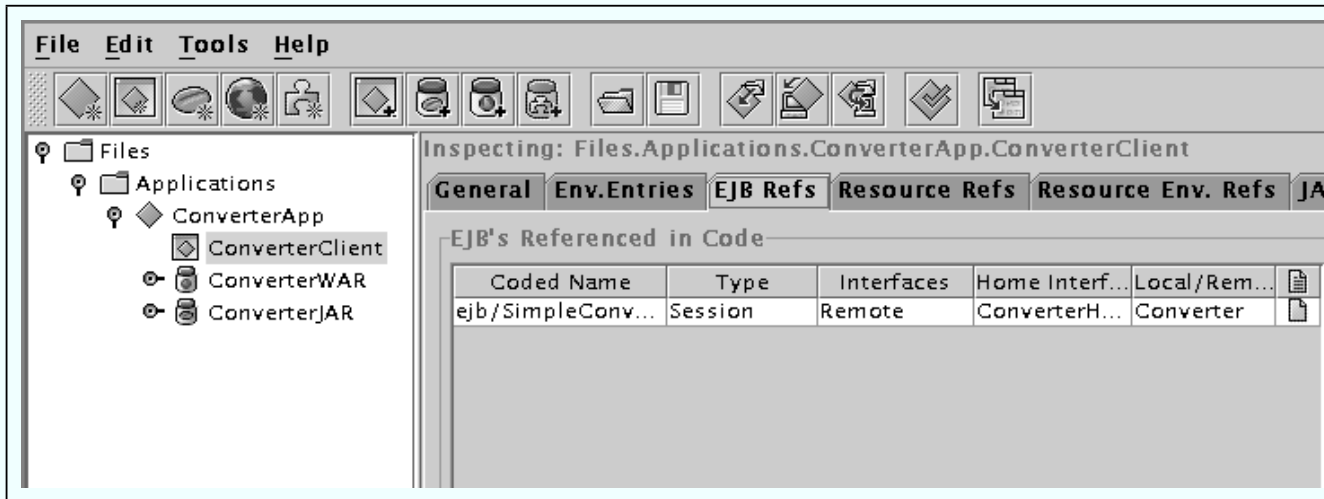


FIG. 7 – Configuration de la référence aux EJB utilisés

5 Composants WEB : les pages JSP

5.1 Généralités

Java Server Page est une technologie Java qui permet de générer dynamiquement, aussi bien côté client que serveur, du code HTML, XML ou autre, donc des pages dont le contenu variera en fonction du contexte.

Page JSP : programme destiné à produire une page HTML.

Les JSP comme les SERVLET sont considérés comme des composants car leur comportement peut être paramétré via un conteneur.

Une page JSP est conceptuellement substituable à une SERVLET. (Une page JSP est potentiellement compilée en une SERVLET).

Une page JSP se présente comme un fichier contenant du source HTML entrecoupé de code Java intégré via différentes sortes de **balises**.

Balises JSP

- **Scriptlet** : balises `<% et %>`.
instructions java (sauf déclarations) exécutés pendant la fabrication de la page finale pour produire différents effets de bord.
- **Expressions** : balises `<%= et %>`
Expression dont la valeur, convertie en chaîne, apparaîtra sur la page en cours de fabrication.
- **Déclaration** : balises `<%! et %>`
Déclaration de classe, de méthode, d'attribut, etc, utilisables dans les scriptlet et expressions précédentes.
- **Directives d'inclusion** : balises `<%@ et %>`
Directive d'inclusion de bibliothèques ou de fichiers.
- **Commentaire** : balises `<%-- et --%>`

Variables pré-définies JSP

- Request (`HttpServletRequest`)
- response (`HttpServletResponse`)
- out (`PrintWriter`) utilisé pour écrire dans la page réponse
- session (`HttpSession`) la session (si elle existe) associée à la requête
- application (`ServletContext`) identique à `getServletConfig().getContext()`.
- config (`ServletConfig`)
- page (`this`)
- pageContext. Accès aux classes JSP spécifiques (`javax.servlet.jsp`)

5.2 Exemple : une page JSP pour l'application "Convertisseur"

```
<%@ page import="javax.ejb.*,
    javax.naming.*,
    javax.rmi.PortableRemoteObject,
    java.rmi.RemoteException" %>

<!-- déclaration d'un variable --%>
<%! private Converter converter = null; %>
<!-- déclaration d'une méthode --%>
<%! public void jspInit() {
    try {
        InitialContext ic = new InitialContext();
        Object objRef = ic.lookup("
            java:comp/env/ejb/TheConverter");
        ConverterHome home = (ConverterHome)PortableRemoteObject.narrow(
                                                    objRef, ConverterHome.class);

        converter = home.create();
    } catch (RemoteException ex) { ... }
} ... %>
```

```
<html> <head> <title>Converter</title> </head>
<body bgcolor="white">
<h1><center>Converter</center></h1> <hr>
<p>Enter an amount to convert:</p>
  <form method="get">
    <input type="text" name="amount" size="25"> <br>
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
  </form>
<% String amount = request.getParameter("amount");
  if ( amount != null && amount.length() > 0 ) {
    Double d = new Double (amount); %> <p>
    <%= amount %> dollars are <%= converter.dollarToYen(d.doubleValue()) %> Yen.<p>
    <%= amount %> Yen are <%= converter.yenToEuro(d.doubleValue()) %> Euro.
  <% } %>
</body>
</html>
```

- Les déclarations contiennent la définition d'une méthode `JspInit` qui intègre la mécanique de connexion vue précédemment avec le client Java.
- Du HTML standard définit le début de page,
- Un formulaire HTML (`<form> ... </form>`) qui contient plusieurs champs de saisie. Un champs de saisie (`<input>`), peut être de différents types prédéfinis (`text`, `password`, `checkbox`, `radio`, `hidden`, `submit`, `button`, `reset`, `file`). Le champ de type `input` lance l'exécution du scriptlet associé.
- Le scriptlet permet récupérer le paramètre et le transformer en `double`.
- Les expressions JSP (entre `<%= %>`) pour intégrer dans la page web calculée, le résultat de l'envoi des messages au convertisseur.

5.3 Configuration et Emballage du composant WEB (figure 8)

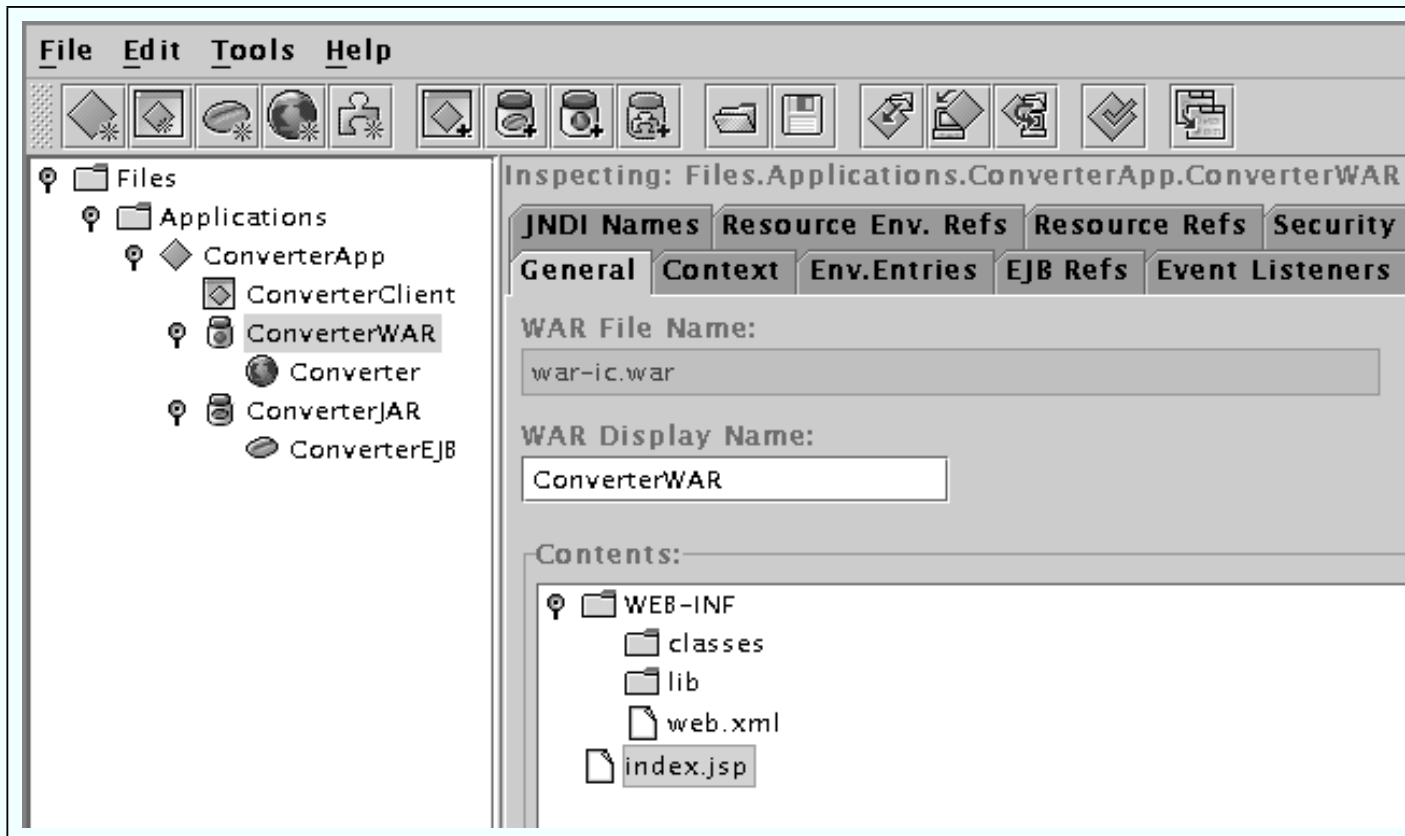


FIG. 8 – Configuration et Emballage d'un client

6 Exécution de l'application (figure 9)

Exécution version client léger via un navigateur utilisant le convertisseur via la page JSP.

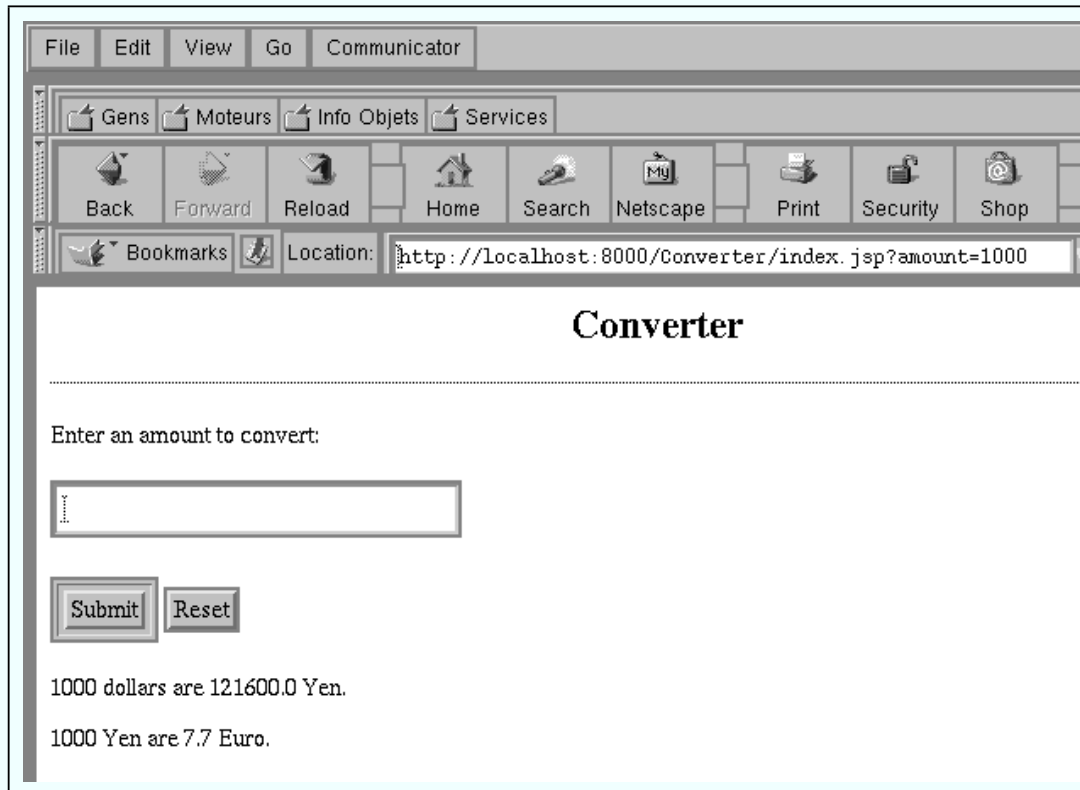


FIG. 9 – Execution de L'application

7 Réalisation et déploiement d'une application avec le serveur JBoss (avec EJB version 2.0)

Cette section permet de se faire une idée l'architecture des fichiers d'une application JEE simple. L'architecture de fichiers est censée rester invisible au développeur utilisant un IDE avancé; mais dans la pratique de la mise au point, il est malheureusement souvent nécessaire de consulter les fichiers de configurations générés pour comprendre où se trouvent les erreurs. Une telle architecture est par ailleurs propre à chaque serveur.

Nous utilisons comme prétexte la réalisation d'une application de type "Hello World".

7.1 L'arbre de déploiement JBOSS (figure 10)

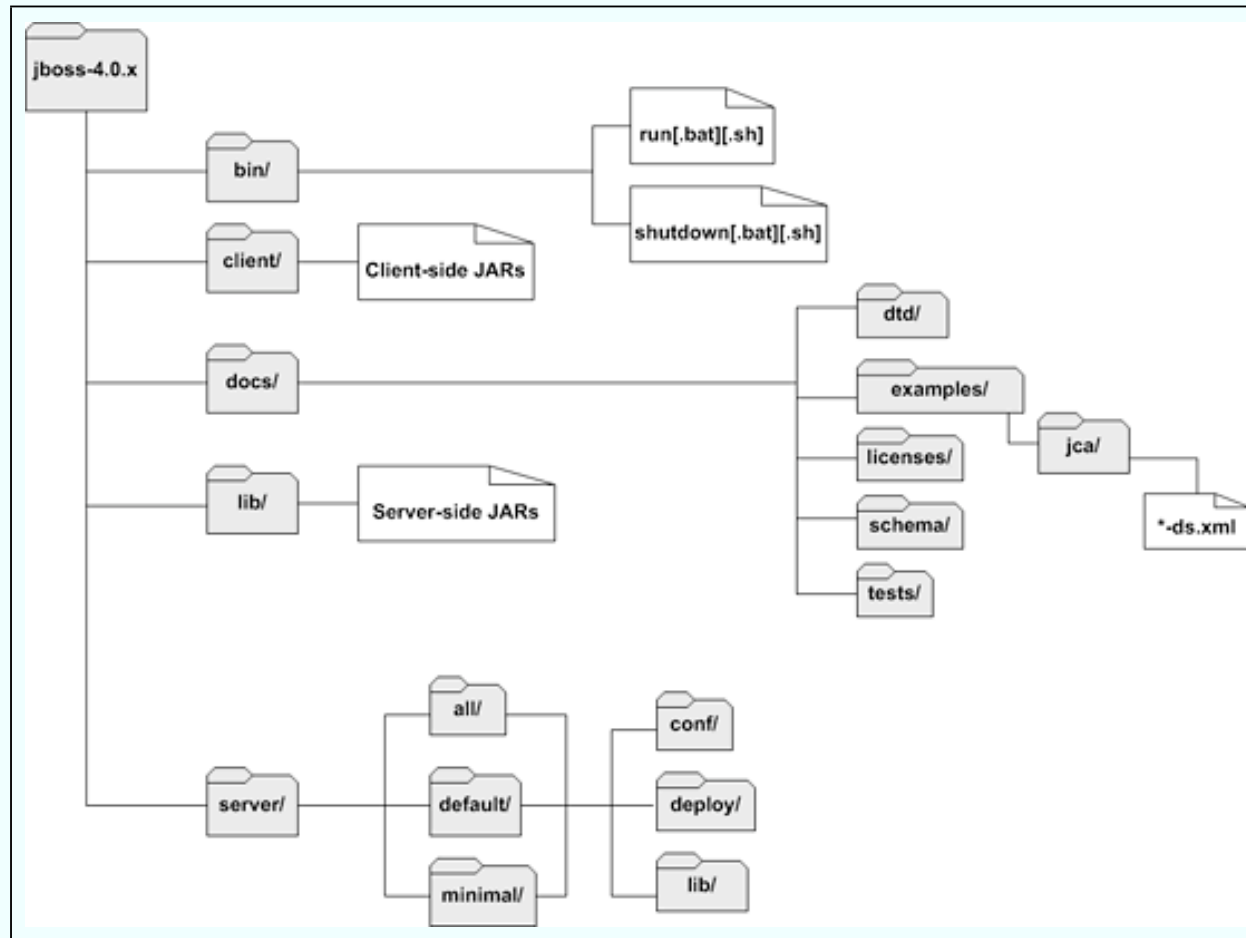


FIG. 10 – Arbre des répertoires JBOSS 4.1 (<http://today.java.net/pub/a/today/2005/03/01/InstallingJBoss.html>)

7.2 Source typique d'une application

```
-rwxr-xr-x  1 dony  dony  4624 13 Dec 20:20 HelloWorld.ear //archive deployable
-rwxr-xr-x  1 dony  dony  2132 13 Dec 20:20 HelloWorld.jar //archive comp. ejb
-rwxr-xr-x  1 dony  dony  2301  6 Dec 15:26 HelloWorld.war //archive comp. web
-rwxr-xr-x  1 dony  dony   490 13 Jun  2002 application.xml //descripteur application
drwxr-xr-x  4 dony  dony   136 18 Jun  2002 build //rep. classes compilées
-rwxr-xr-x  1 dony  dony  2650 13 Dec 20:26 build.xml //makefile ant
-rwxr-xr-x  1 dony  dony   655 13 Jun  2002 ejb-jar.xml //descripteur EJB
-rwxr-xr-x  1 dony  dony   159 13 Jun  2002 jboss.xml //?
drwxr-xr-x  6 dony  dony   204 13 Dec 20:50 src //rep. sources
-rwxr-xr-x  1 dony  dony   789 13 Jun  2002 web.xml //descripteur module
```

7.3 Exemple de fichier descripteur JBOSS de l'unique module de l'application

Fichier web-app.xml

```
<web-app>
  <display-name>Hello World</display-name>
  <description>Hello World</description>
  <servlet>
    <servlet-name>HelloWorldServlet</servlet-name>
    <servlet-class>HelloWorldServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloWorldServlet</servlet-name>
    <url-pattern>/Bonjour</url-pattern>
  </servlet-mapping>
  <ejb-ref>
    <ejb-ref-name>HelloWorldHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>HelloWorldHome</home>
    <remote>HelloWorld</remote>
    <ejb-link>HelloWorld</ejb-link>
  </ejb-ref>
</web-app>
```

7.4 Code de l'EJB

Les Interfaces locales et distantes sont similaires à celle de l'exemple précédent du convertisseur.

```
import javax.ejb.*;

public class HelloWorldEJB implements SessionBean{
    private SessionContext context;
    public String sayHello(){
        return "Aujourd'hui est le meilleur moment de la vie";}
}
```

7.5 Description Jboss de l'EJB

fichier ejb-jar.xml

```
<ejb-jar>
  <display-name>Hello World</display-name>
  <enterprise-beans>
    <session>
      <description>Hello World EJB</description>
      <display-name>HelloWorld</display-name>
      <ejb-name>HelloWorld</ejb-name>
      <home>HelloWorldHome</home>
      <remote>HelloWorld</remote>
      <ejb-class>HelloWorldEJB</ejb-class> \subparagraph

      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

8 Composants WEB : les SERVLETs

8.1 Généralités

Servlet : Application matérialisée par une classe Java (voir `javax.servlet`) destinée à étendre les capacités d'un serveur hébergeant des applications utilisées via un protocole "requête-réponse".

web-servlet : étend un serveur WEB pour traiter des requêtes HTTP et y répondre en fournissant du code HTML.

– est implantée par une sous-classe de **HTTPServlet** (voir package `javax.http.servlet`) dont voici la signature :

```
public abstract class HttpServlet
    extends GenericServlet
    implements Servlet ServletConfig
```

– Toute sous-classe de `HTTPServlet` définissant une *Servlet* concrète peut traiter des requêtes HTTP, dont les deux plus standard, de type "GET" ou "POST", en redéfinissant les méthodes :

- `doGet`, requête HTTP GET :
- `doPost`, requête HTTP POST

- Il est par ailleurs possible de redéfinir `init` et `destroy` liées à la vie de la servlet.

8.2 Cycle de vie d'une Servlet

Il est contrôlé par le conteneur dans lequel la servlet est déployée.

Quand une requête active une servlet, si aucune instance n'existe, le conteneur charge la classe, crée une instance et l'initialise (méthode `init`). Puis le conteneur invoque la méthode appropriée (`doGet(...)` ou `doPost(...)`) et passe en argument les objets représentant la requête et la réponse.

Un client exécute une servlet via son navigateur et une URL (par exemple : `http://localhost:8080/PrincipeServlet`).

8.3 Exemple : Une Servlet pour l'application JBoss "hello world"

```
import java.io.*, java.text.*, java.util.*, javax.servlet.*,
javax.servlet.http.*, javax.naming.*, javax.rmi.*;

public class HelloWorldServlet extends HttpServlet {
    private HelloWorld helloEJB = null;

    public HelloWorldServlet() throws NamingException{
        Context ctx = new InitialContext();
        HelloWorldHome home = (HelloWorldHome)
            PortableRemoteObject.narrow(ctx.lookup("HelloWorld"),
                HelloWorldHome.class);
        try { this.helloEJB = home.create(); }
        catch (Exception e) { }}

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Ceci est un titre</title>");
```

L'exécution (voir figure 11) s'obtient en entrant l'URL déclarée dans la partie configuration.



FIG. 11 – Exécution : un EJB contrôlé par une Servlet

9 JEE-5 et EJB-3

La plate-forme JEE-5 intègre une évolution de la spécification des EJB, matérialisée par l'API EJB-3. Elle apporte un ensemble de nouveautés dont :

- utilisations de nouveaux supports Java 5 : énumérations, annotations, génériques.
- simplification du codage des clients, utilisation du mécanisme d'injection de dépendances.

9.1 Enumérations

```
public enum EtatTransaction {enCours, enEchec, terminée, annulée}  
  
private EtatTransaction et;
```

9.2 Annotations

Java 5 propose le type annotation qui permet d'exprimer des méta-données relatives aux éléments des programmes annotation.

Annotations prédéfinies :

- @deprecated
- @Documented
- @inherited : méta-annotation indiquant qu'une annotation doit être héritée.
Elles ne le sont pas par défaut.
- @overrides

- @retention

Aspect J online documentation : Annotations can have one of three retention policies :

- Source-file retention : Annotations with source-file retention are read by the compiler during the compilation process, but are not rendered in the generated .class files.
- Class-file retention : This is the default retention policy. Annotations with class-file retention are read by the compiler and also retained in the generated .class files.
- Runtime retention : Annotations with runtime retention are read by the compiler, retained in the generated .class files, and also made available at runtime.
- @Target : permet de spécifier la portée de l'annotation, package et déclarations de type (classes, interfaces, enums, annotations), constructeurs, methodes, attributs, paramètres, et variables.

9.3 Injection de dépendances

Mise en correspondance automatisée, au moment du déploiement, entre différents EJB utilisés dans différents modules d'une application.

10 Exemple EJB-3 : session bean avec état et pilotage client via une page JSP

10.1 Un session Bean avec état

Interface distante :

```
import javax.ejb.Remote;

@Remote
public interface CounterService {
    public void incr();
    public void decr();
    public int getValue();
    public void raz();
}
```

Code de l'EJB :

```
import javax.ejb.Stateful;

@Stateful
public class Counter implements CounterService{
    int value = 0;
    public void incr() {value = value + 1;}
    public void decr() {value = value - 1;}
    public void raz() {value = 0;}
    public int getValue() {return value;}
}
```

10.2 Obtenir une référence sur un EJB - JEE5)

10.2.1 Par l'annuaire du serveur d'application

Il n'est plus nécessaire de passer par une fabrique (le "home" de EJB2).

```
try {  
    InitialContext ctx = new InitialContext();  
    CounterRemote cpt = (Counter) ctx.lookup("CounterService/remote");  
}  
catch (NamingException exp) { }
```

10.2.2 Via une injection de dépendance

Référence à un EJB dans le code d'un autre EJB :

```
@EJB  
private CounterService ib;
```

10.3 composant web JSP - JEE5

Une page JSP pour accéder à l'EJB.

Référence à un EJB dans une page JSP :

```
<jsp:useBean id="counter" scope="session" class="ejb.Counter" />
```

Exemple :

```
<html>
  <head>
    <title>Compteur JEE</title>
  </head>
  <body>
    <jsp:useBean id="counter" scope="session" class="ejb.Counter" />
    <%
String s = request.getParameter("operation");
if ("incr".equals(s)) counter.incr();
else if ("decr".equals(s)) counter.decr();
%>
    <h1>Valeur du compteur :
    <jsp:getProperty name="counter" property="value" />
    </h1>
    <form name="Name Input Form" action="index.jsp">
      <input type="submit" value="incr" name="operation"/>
      <input type="submit" value="decr" name="operation"/>
      <input type="submit" value="raz" name="operation"/>
    </form>
  </body>
</html>
```

Pour exécuter (voir figure 12) l'application à partir d'une machine distante :

`http://IPMachineDistante:8080/Increment0-war/index.jsp?operation=incr`

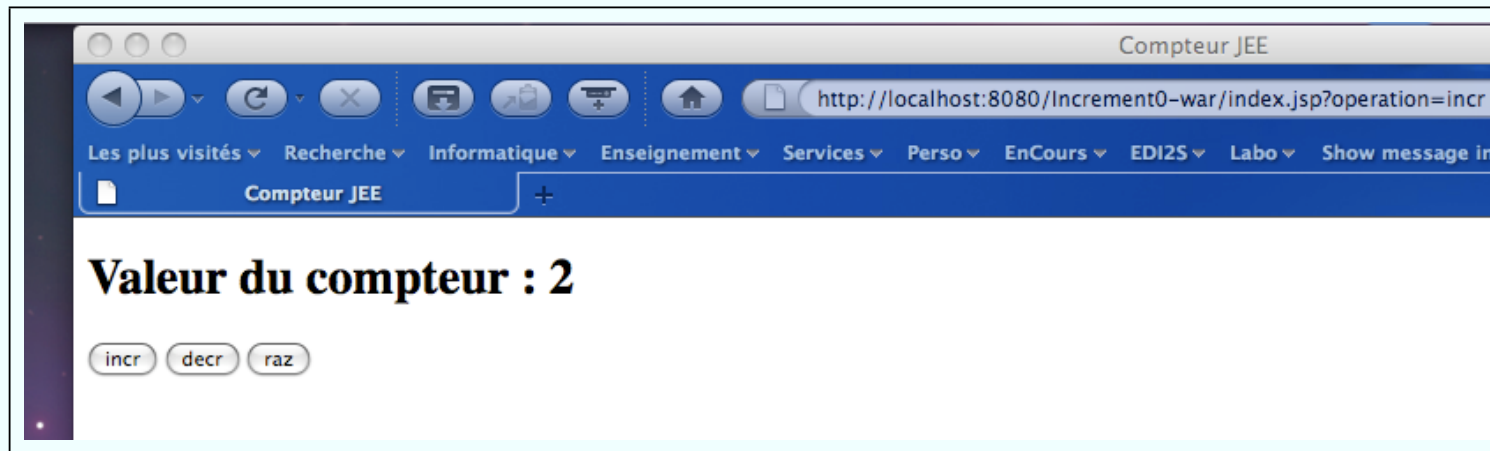


FIG. 12 – Exécution : Counter session bean (EJB3.0) contrôlé via une page JSP

10.4 Composant Servlet - JEE5

Accès au même EJB via une Servlet.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.RequestDispatcher;
import javax.ejb.EJB;
```

```
public class Counter2Servlet extends HttpServlet {
    @EJB
    CounterService c;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        String operation = request.getParameter("operation");
        if (operation.equals("incr")) c.incr();
        if (operation.equals("decr")) c.decr();
        try{
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Accès à un EJB via une Servlet</title>");
            out.println("</head>");
            out.println("<h1> Valeur du compteur: " + c.getValue() + "</h1>");
            out.println("" +
                "<form method=\"GET\" action=\"Counter2\">" +
                "<input type=\"submit\" value=\"incr\" name=\"operation\">" +
                "<input type=\"submit\" value=\"decr\" name=\"operation\">" +
                "</form>");
            out.println("</body>");
            out.println("</html>");}
    }
```

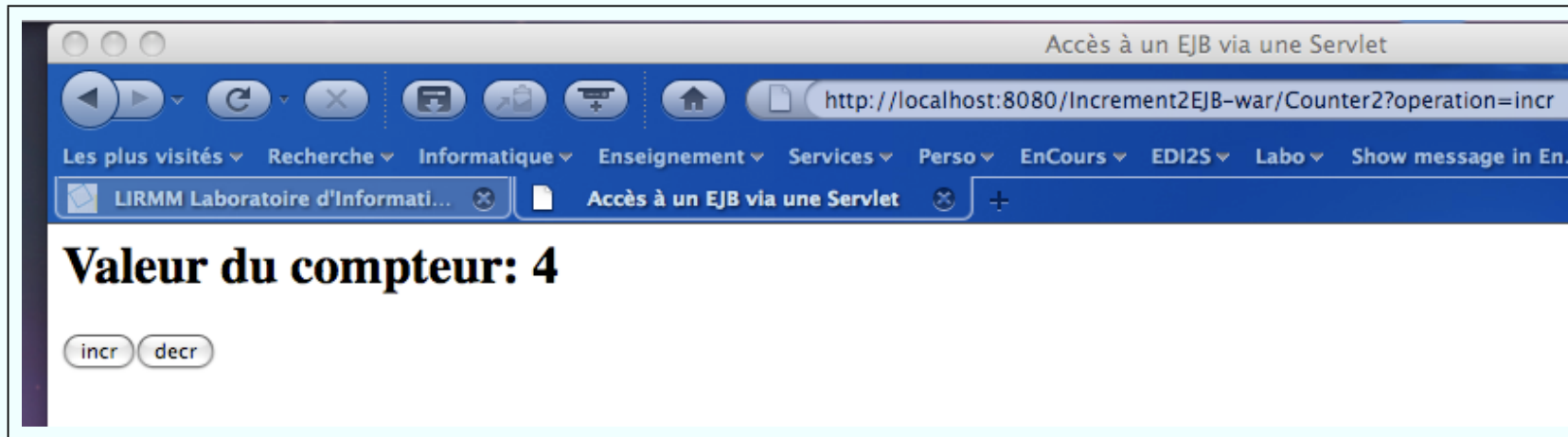


FIG. 13 – Exécution : Counter session bean (v3.0) contrôlé via une Servlet

11 Tiers système d'information

11.1 Entity Beans

Les applications JEE permettent la gestion simplifiée du tier “Système d'information” ou “persistance des données”.

Définition : Un composant de type “entité” représente un objet métier dans un système persistant, ayant une clé primaire, supportant les accès concurrents, compatible avec un système transactionnel.

Le mécanisme standard de persistance est une base de donnée relationnelle.

Un *entity bean* est lié à une table en base de données. La liaison s'appelle un *mapping*.

A chaque classe d'*Entity bean* correspond une table.

Chaque propriété d'un objet correspond à une colonne dans la table.

Chaque instance du bean correspond à un rang dans la table.

11.2 Exemple - Création d'un EntityBean

Soit à définir une application JEE de gestion d'un serveur de *news*. Les news sont entrées via un client léger et sont stockées dans une base de données.

Avec EJB 3, les objets persistants (*entity beans*) d'une application sont des objets standards java (POJO's).

Un mapping est automatiquement réalisé dès qu'une classe d'objet est annotée par `@Entity`.

Cette classe doit posséder un attribut référencé (annotation `@Id`) comme définissant une clé primaire pour le stockage.

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```

```
@Entity
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private String body;
    //accesseurs pour les attributs

    // généré automatiquement
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    @Override
    public int hashCode() { ... }
    @Override
    public boolean equals(Object object) { ... }
    @Override
    public String toString() { return "ejb.NewsEntity[id=" + id + "]; }
}
```

La clé primaire permet de différencier les différentes instances. La stratégie de génération de la clé est paramétrable.

Tout autre attribut (non *static* et non *transcient* d'une classe *entity* sera automatiquement persistant, ses valeurs pour les différentes instances seront stockées dans la base.

Si des instances de la classe doivent être passés en argument, cette classe doit implanter l'interface **Serializable**.

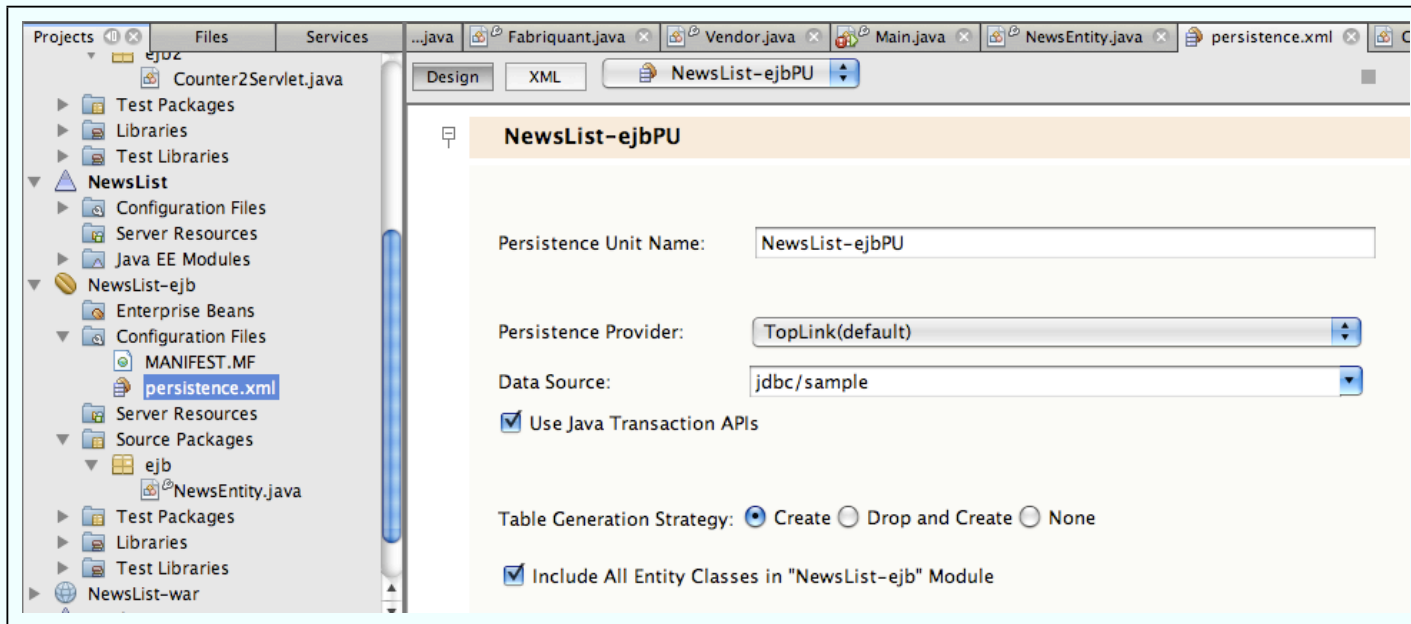


FIG. 14 – Configuration de la persistance.

11.3 Accès aux objets persistants via un session Bean de “façade”- EJB3

Le schéma standard pour accéder aux objets persistants est d'utiliser un *session bean* dit de façade (voir *session bean for entity classes*), qui sera placé dans un conteneur doté d'un *Entity Manager*.

public interface javax.persistence.EntityManager :

An EntityManager instance is associated with a persistence context. A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed. This interface defines the methods that are used to interact with the persistence context. The EntityManager API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

```
@Stateless
public class NewsEntityFacade extends AbstractFacade<NewsEntity> {
    @PersistenceContext(unitName = "NewsApp-ejbPU")
    private EntityManager em;

    protected EntityManager getEntityManager() {
        return em;
    }

    public NewsEntityFacade() {
        super(NewsEntity.class);
    }
}
```

L'annotation `@PersistenceContext` permet de récupérer, par injection de dépendance, une référence sur le gestionnaire d'objets persistants du site de déploiement qui permet l'accès aux `entity beans`.

La classe prédéfinie `AbstractFacade`, permet de générer un session bean façade pour toute classe d'entity bean.

```
public abstract class AbstractFacade<T> {
    private Class<T> entityClass;

    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    protected abstract EntityManager getEntityManager();

    public void create(T entity) {
        getEntityManager().persist(entity);}

    public void remove(T entity) {
        getEntityManager().remove(getEntityManager().merge(entity));
    }

    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }
}
```

Il est ainsi possible d'accéder aux données persistantes :

- Soit directement via `em` dans le code précédent.
- Soit dans tout autre EJB, via une référence vers une instance de `NewsEntityFacade`, comme dans le code suivant.

```
@EJB
private NewsEntityFacadeLocal facade;

NewsEntity e = new NewsEntity("uneNouvelle", "il fait beau et chaud");
facade.create(e);
```

12 Message-Driven Beans

Un Bean de type “Message-driven” est un EJB qui se comporte comme un écouteur de messages *JMS*, c’est-à-dire qui reçoit des messages JMS et les traite de manière asynchrone. Il est associé à une ”destination JMS”, c’est-à-dire une ”Queue” pour les communications ”point à point” ou à un ”Topic” pour les communications de type “publication/souscription”.

Dans l’application `ListNews`, l’utilisation d’un MDB pour réceptionner les demandes de modification (ajout ou destruction de News) à la base de donnée permet

- aux clients émettant ces demandes de ne pas être bloqués en attente d’une réponse,
- de servir de nombreux clients sans blocage.

12.1 Emission de messages JMS

JMS : Java Messaging Service, API fournissant à un middleware (**MOM** : Message Oriented Middleware) la gestion des messages asynchrones.

Concepts JMS

- **Usine de connexion** : entité référencée via JNDI permettant de récupérer une *destination* et d'établir une *connexion* JMS à un service de routage de messages. Une usine encapsule un ensemble de configuration de paramètres. Chaque usine est instance d'une classe implantant `ConnectionFactory`, `QueueConnectionFactory`, ou `TopicConnectionFactory`.
- **Connexion** : une connexion à un serveur JMS est donc obtenue via une usine de connexion.

```
javax.jms.ConnectionFactory connectionFactory;  
javax.jms.Connection connection =  
    connectionFactory.createConnection();
```

- **Session** : permet de grouper des opérations de réception et d'envoi de message, pour une même connexion.

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

- **Destination** : entité utilisée par un client pour spécifier vers qui envoyer les messages qu’il produit ou d’où viennent les messages qu’il consomme.
 - **Topic** : type de *destination* utilisée pour la publication/souscription.
 - **Queue** : type de *destination* utilisée pour la communication de point à point.
Une queue implante une file. Les messages arrivant sont stockées dans la file et en sont extraits lorsqu’ils sont lus.
- **MessageProducer** : objet réalisant l’envoi des messages, auquel il faut associer une “destination”.

```
MessageProducer messageProducer = session.createProducer(queue);
```

- **Message** : entête et corps contenant soit du texte, soit un objet sérialisé (ObjectMessage).

```
ObjectMessage message = session.createObjectMessage();  
NewsEntity e = new NewsEntity();  
e.setTitle("uneNouvelle");  
e.setBody("Il fait beau et chaud");  
message.setObject(e);  
messageProducer.send(message);
```

12.2 Traitement des messages JMS par un MDB

Un **Message-Driven Bean** est un composant EJB spécialisé capable de traiter des messages asynchrones reçus d'un fournisseur JMS sur la **destination** à laquelle il est associé.

Le protocole de réception d'un message est l'activation de la méthode **OnMessage** du MDB.

12.2.1 Exemple d'insertion dans la base de donnée

Le MDB ci-dessous est activé à chaque fois qu'un client poste un message contenant une *news* à insérer dans la base. Il utilise le session bean de façade pour réaliser les accès à la base.

```
@MessageDriven(mappedName = "jms/NewMessage", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
                                propertyValue = "javax.jms.Queue"))})
public class NewMessage implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;
    @EJB
    private NewsEntityFacadeLocal facade;

    public void onMessage(Message message) {
        ObjectMessage msg = null;
        try {
            if (message instanceof ObjectMessage) {
                msg = (ObjectMessage) message;
                NewsEntity e = (NewsEntity) msg.getObject();
                facade.create(e);}
        catch (JMSEException e) {mdc.setRollbackOnly();}
        catch (Throwable te) {te.printStackTrace();}}}
```

12.2.2 Servlet d'affichage de la base de News

```
public class ListNews extends HttpServlet {
    @EJB
    private NewsEntityFacadeLocal newsEntityFacade;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.getSession(true);
        PrintWriter out = response.getWriter();
        try {
            out.println("<html><head><title>Servlet ListNews</title></head>");
            out.println("<body>");
            out.println("<h1>Servlet ListNews at " + request.getContextPath() + "</h1>");
            List news = newsEntityFacade.findAll();
            for (Iterator it = news.iterator(); it.hasNext();) {
                NewsEntity elem = (NewsEntity) it.next();
                out.println(" <b>" + elem.getTitle() + " </b><br />");
                out.println(elem.getBody() + "<br /> ");
            }
        }
    }
}
```

```
out.println("<hr>");
out.println("<a href='PostNews'>Add a news to the database</a>");
out.println("<a href='DeleteNews'>Delete news in the database</a>");
out.println("<hr>");
out.println("</body>");
out.println("</html>");}
finally {out.close();}}
```

12.2.3 Servlet pour envoyer un message JMS d'ajout d'une News à la base

```
public class PostNews extends HttpServlet {
    @Resource(mappedName = "jms/NewMessageFactory")
    private ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/NewMessage")
    private Queue queue;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        // Add the following code to send the JMS message
        String title = request.getParameter("title");
        String body = request.getParameter("body");
        System.out.println("avancee 1");
        if ((title != null) && (body != null)) {
            try {
                Connection connection = connectionFactory.createConnection();
                Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGED
```

```
MessageProducer messageProducer = session.createProducer(queue);
ObjectMessage message = session.createObjectMessage();
// here we create NewsEntity, that will be sent in JMS message
NewsEntity e = new NewsEntity();
e.setTitle(title);
e.setBody(body);
message.setObject(e);
messageProducer.send(message);
messageProducer.close();
connection.close();
response.sendRedirect("ListNews");
```

```
} catch (JMSException ex) { ex.printStackTrace(); }
```

```
PrintWriter out = response.getWriter();
```

```
try {
```

```
    out.println("<html>");
```

```
    out.println("<head>");
```

```
    out.println("<title>Servlet PostMessage</title>");
```

```
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Servlet PostMessage at " + request.getContextPath() + "</h1>");
    out.println("<form>");
    out.println("Title of the message : <input type='text' name='title'><br/>");
    out.println("Message: <textarea name='body'></textarea><br/>");
    out.println("<input type='submit'><br/>");
    out.println("</form>");
    out.println("</body>");
    out.println("</html>");
} finally { out.close(); }
```

13 Références

- tutorial J2EE - ©2001 Sun Microsystems, Inc. All rights reserved.
- tutoriel ECOOP'2001 “Component technologies for the middle tier”, Michael Stal.
- “Client/Server Programming with Java and Corba” - R.Orfali, D. Harkley”.
- “Un petit aperçu de CORBA”, Olivier Gutknecht.
- “Corba Components”, Raphaël Marvie.
- “Corba, des concepts à la pratique” - Jean-Marc Geib, Christophe Gransart, Philippe Merle. Laboratoire d'Informatique Fondamentale de Lille. Dunod.
- “Programmation par composants” - Philippe MERLE, Michel RIVEILL, Techniques de l'ingénieur - Référence H2759 - Dunod 2000.
- “Technologies et architectures internet” - P-Y. Cloux, D. Doussot, A. Géron - Dunod 2000,
- Enterprise JavaBeans, Richard Monson-Haefel, O'Reilly, 2002
- EJB 3 - Des concepts à l'écriture du code- Guide du développeur. Laboratoire SUPINFO des technologies SUN. Dunod 2006.
- Tutorial NetBeans. [http ://www.netbeans.org/kb/docs/javaee/ejb30.html](http://www.netbeans.org/kb/docs/javaee/ejb30.html) - Septembre 2009.