Développement d'applications distribuées avec Java/JEE

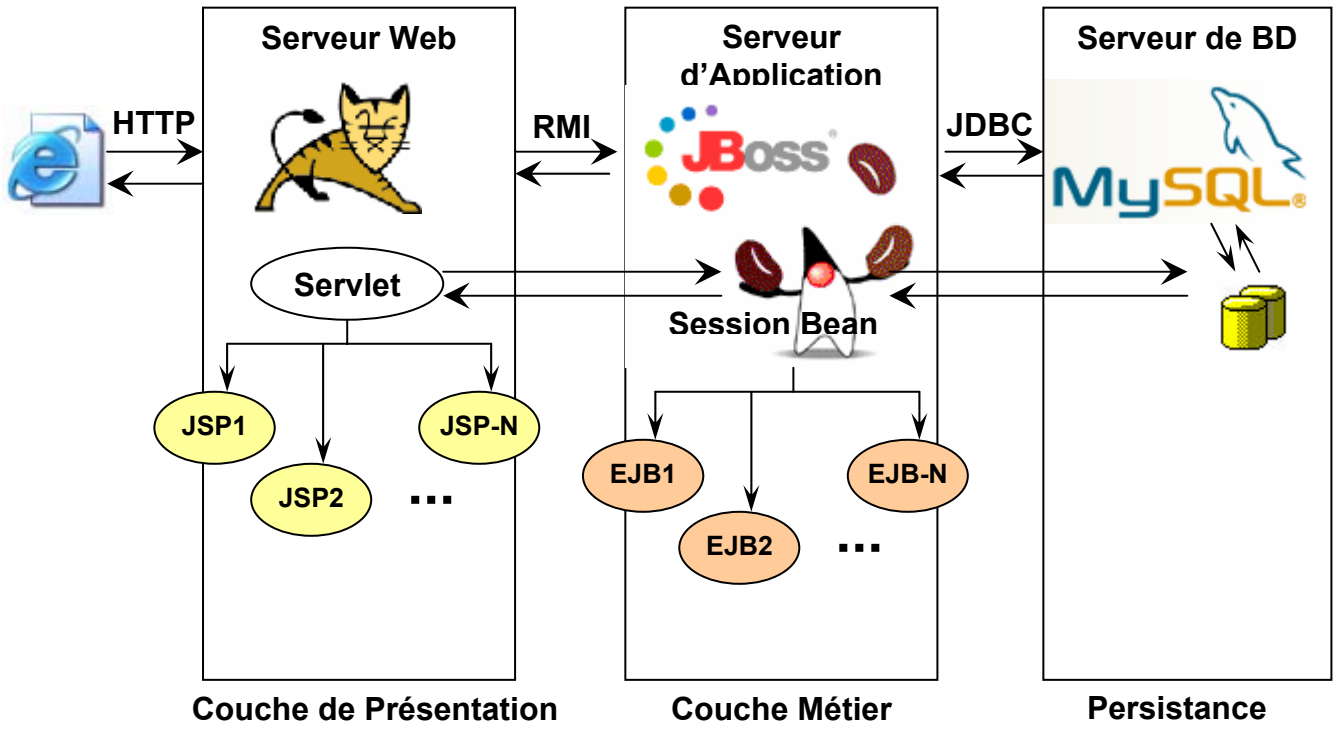
Par N. Chenfour

Site Web : http://www.acs.ma - Email : formation@acs.ma
Tél./Fax : 035 64 18 61 - GSM : 063 05 10 39
Résidence Fès-Carrefour, Bureau N° 6, Lot Walili, Avenue des FAR, Fès - Maroc.

Table des Matières

Chapitre 1. Architecture Générale JEE

1.1 Architecture



1.2 Les Servlets

Les Servlets sont à la base de la programmation Web JEE. Toute la conception d'un site web en Java repose sur ces éléments que sont les Servlets.

Une Servlet est une classe Java qui tourne sur la machine du serveur web, et qui est invoquée lorsqu'un navigateur client appelle l'URL liée à ce serveur.

La Servlet génère alors automatiquement du code HTML qu'elle envoie au client demandeur.

1.3 Les JSP

Une page JSP est une page HTML contenant du code Java. Nous avons déjà dit que la servlet permet de générer du code HTML. Cependant on se trouve limité par le code java et la génération de chaîne de caractères (entre guillemets) contenant du code HTML. Difficile donc à écrire correctement, à corriger et à maintenir (ce qui nécessite une recompilation explicite de la Servlet à chaque modification du code HTML contenu dans celle-ci.

Tous ces problèmes sont résolus à l'aide des JSPs. Ces deux entités peuvent alors collaborer ensemble à coté du modèle de données pour constituer un modèle de développement web robuste, souple et satisfaisant : le modèle MVC2.

1.4 RMI

RMI ou encore, Remote Method Invocation est une technologie développée et fournie par Sun à partir du JDK 1.1 pour permettre des appels distants de méthodes d'objets qui tournent en réseau dans un environnement distribué.

L'appel coté client d'une telle méthode est un peu plus compliqué que l'appel d'une méthode d'un objet local. Il consiste à obtenir une référence sur l'objet distant puis à simplement appeler la méthode à partir de cette référence.

La technologie RMI se charge de rendre transparente la localisation de l'objet distant, son appel et le renvoi du résultat.

En fait, elle utilise deux classes particulières, le stub et le skeleton, générées automatiquement avec le JDK.

Le stub est une classe qui se situe côté client et le skeleton est son homologue coté serveur. Ces deux classes se chargent d'assurer tous les mécanismes d'appel, de communication, d'exécution, de renvoi et de réception du résultat.

1.5 EJB

Les Entreprise Java Bean ou EJB sont des composants serveurs qui tournent sous un serveur d'application permettant de fournir un certain nombre de services métiers par l'intermédiaire d'un premier type d'EJB ; les EJB de session (ou Session Beans). Ils fournissent aussi des services d'accès base de données par l'intermédiaire d'un 2ème type d'EJB : les beans entités « Entity Beans ».

Le but des EJB est de faciliter la création d'applications distribuées pour les entreprises.

Une des principales caractéristiques des EJB est de permettre aux développeurs de se concentrer sur les traitements orientés métiers car les EJB et l'environnement dans lequel ils s'exécutent prennent en charge un certain nombre de traitements tel que la gestion des transactions, la persistance des données, la sécurité, ...

1.6 JDBC

Java DataBase Connectivity est une API Java permettant d'accéder à des bases de données de façons transparente par rapport à la base de données utilisée. Les procédures de connexion, de manipulation et d'administration sont les mêmes indépendamment du SGBD : oracle, MySQL, ODBC ou autre.

1.7 Struts

Struts (http://struts.apache.org/), ou encore « **Apache Struts** », est un framework open source géré par le groupe Jakarta de la communauté Apache, utilisé pour faciliter le développement des applications web J2EE.

Il a été créé par Craig McClanahan qui est passé à la fondation Apache en 2000 en sous-partie du projet **Jakarta**.

L'objectif du framework Struts est de permettre la mise en place d'une **architecture MVC** (Modèle-Vue-Contrôleur) plus aisément. Il se base pour cela sur la technologie des Servlets (d'une manière transparente) et celle des JSP en les étendant et en donnant accès à des objets améliorant l'approche de ces dernières. Cela débouche sur une meilleure structuration du code d'une application web Java ; ce qui ainsi une meilleure maintenabilité du code.

En plus, Struts se démarque aujourd'hui (avec sa version 2 ; résultat d'une fusion entre le projet Struts et le projet WebWork) par le fait qu'il n'impose aucune implémentation d'interface ou extension de classe du framework. Les objets qu'il gère sont des objets Java ordinaires (ou encore des POJO [*Plain Old Java Objects*]). Les classes ainsi réalisées (sauf si le développeur intègre délibérément les API Struts) sont totalement indépendantes du framework.

Par ailleurs, le framework Struts offre une vaste bibliothèque de balise permettant de réaliser toute la logique de présentation d'une manière simple et assistée.

1.8 Hibernate

Les informations ou données, manipulées par une application quelconque sont généralement sauvegardées dans une base de données qui pour la grande majorité des applications est une base de données relationnelle. L'application quant à elle est basée sur les mécanismes de la programmation orientée objets. L'utilisation d'un outil d'ORM (Object Relationnal Mapping) s'impose alors afin de soulager le développeur de l'effort considérable qu'il devrait fournir pour réaliser l'ensemble des classes lui permettant de convertir les objets en des tuples et inversement.

Hibernate est l'un des outils les plus populaires, stables et robuste permettant le mapping objet/relationnel pour les applications Java. Il s'agit dont d'un ORM (Objet Relational Mapper) qui consiste à faire le lien entre la représentation objet des données et sa représentation relationnelle basée sur un schéma SQL.

Hibernate est un framework open source dont le fondateur est Gavin King, qui fait entre autre partie de l'équipe de développement de JBOSS.

Non seulement, Hibernate s'occupe du transfert des classes Java dans les tables de la base de données (et des types de données Java dans les types de données SQL), mais il permet aussi de requêter les données et propose des moyens de les récupérer. Il peut donc réduire de manière considérable le temps de développement réalisé habituellement à base de l'API JDBC avec plusieurs lignes de code permettant de gérer tous les transferts des objets nécessaires. Le framework Hibernate libère ainsi le développeur de plus que 90% des tâches de programmation liées à la persistance des données. Il offre aussi la gestion automatique du mapping dan les différents cas que sont la composition, les associations, l'héritage, le polymorphisme ainsi que la prise en charge des collections Java.

Hibernate rentre ainsi dans le cadre des frameworks basés sur les POJO (**P**lain **O**ld **J**ava **O**bject) offrant à un POJO la possibilité d'être persitant

avec un minimum d'effort; en utilisant un simple moteur de persistance. Le schéma idéal est alors le suivant :

```
PojoClass pojo = new PojoClass() ;

PersistanceEngine engine = PersistanceEngineFactory.createEngine();

engine.save(pojo);
```

1.9 Spring

Spring (http://www.springframework.org) est un framework open source créé par *Rod Johnson* qui le décrit pour la première fois en 2002 dans son livre (sous le nom *Interface21*) *Expert One-on-One : J2EE Design and Development*. L'objectif désigné du framework est d'offrir une alternative viable aux EJB plus simple à utiliser dans le cadre du développement d'applications professionnelles.

Spring est d'abord un conteneur d'objets Java dans le même sens que Tomcat est un conteneur de servlets ou JBoss un conteneur d'EJB.

Néanmoins, Spring se démarque des conteneurs cités du fait que les objets qu'il gère sont des objets Java ordinaires (nommés POJO [*Plain Old Java Objects*] dans la littérature).

Par ailleurs, Spring dispose d'une implémentation AOP qui lui permet d'offrir de nombreux services et ce, sans que les classes Java aient à implémenter une quelconque interface propriétaire. Spring fournit également une riche API qui simplifie la programmation de certaines tâches et l'utilisation d'outils open source répandus.

Chapitre 2.
Les Servlets & Tomcat

2.1 Définition

Une Servlet est une classe Java qui tourne sous un serveur Web (nous utiliserons le serveur Apache Tomcat). La Servlet doit implémenter l'interface « **javax.servlet.Servlet** » directement, ou indirectement en étendant la classe « **javax.servlet.http.HTTPServlet** ».

Lorsqu'une servlet est appelée par un client, la méthode « **service()** » est exécutée. Celle-ci est le principal point d'entrée de toute servlet et accepte deux objets en paramètres :

L'objet **ServletRequest** encapsulant la requête du client, c'est-à-dire qu'il contient l'ensemble des paramètres passés à la servlet.

L'objet **ServletResponse** permettant de renvoyer une réponse au client. Il est ainsi possible de créer des en-têtes HTTP (headers), d'envoyer des cookies au navigateur du client, ...

Afin de développer une servlet fonctionnant avec le protocole HTTP, il suffit de créer une classe étendant « **HttpServlet** ».

La classe HttpServlet redéfinie la méthode service en redirigeant la requête vers l'une des deux méthodes « doGet » ou « doPost » en fonction de la méthodes du protocole HTTP utilisée : GET ou bien POST, il suffira alors de redéfinir la méthode adéquate afin de traiter la requête :

```
public void doGet(HttpServletRequest req, HttpServletResponse res);

public void doPost(HttpServletRequest req, HttpServletResponse res);
```

2.2 Cycle de vie d'une servlet

Le cycle de vie d'une servlet est assuré par le conteneur de servlet (Tomcat) comme suit :

1. le serveur crée un pool de threads auxquels il va pouvoir affecter chaque requête
2. La servlet est chargée au démarrage du serveur ou lors de la première requête
3. La servlet est instanciée par le serveur
4. La méthode *init()* est invoquée par le conteneur
5. Lors de la première requête, le conteneur crée les objets Request et Response spécifiques à la requête
6. La méthode *service()* est appelée à chaque requête. Les objets *Request* et *Response* lui sont passés en paramètre
7. Grâce à l'objet *Request*, la méthode *service()* va pouvoir analyser les données en provenance du client
8. Grâce à l'objet *Response*, la méthode *service()* va fournir une réponse au client
9. La méthode *destroy()* est appelée lors du déchargement de la servlet. La servlet est alors signalée au *garbage collector*.

Pour créer une servlet il est indispensable de mettre en oeuvre l'interface *javax.servlet.Servlet* permettant au conteneur d'assurer le cycle de vie de la servlet.

La mise en place de l'interface (généralement réalisée en étendant *javax.servlet.GenericServlet* ou *javax.servlet.HttpServlet*) fait appel à cinq méthodes :

- la méthode *init()*
- la méthode *service()*
- la méthode *getServletConfig()*
- la méthode getServletInfo()
- la méthode destroy()

2.3 Installation du Serveur Tomcat

1- Installer Tomcat (exemple 6.0.10) :

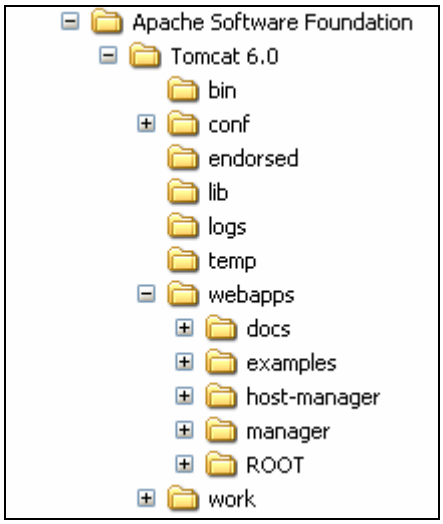
Le serveur Tomcat évolue avec l'évolution des Servlet et des JSP (nous utiliserons la version 6.0)

Version Apache Tomcat	Version Servlet/JSP
6.0.x	2.5/2.1
5.5.x	2.4/2.0
4.1.x	2.3/1.2
3.3.x	2.2/1.1

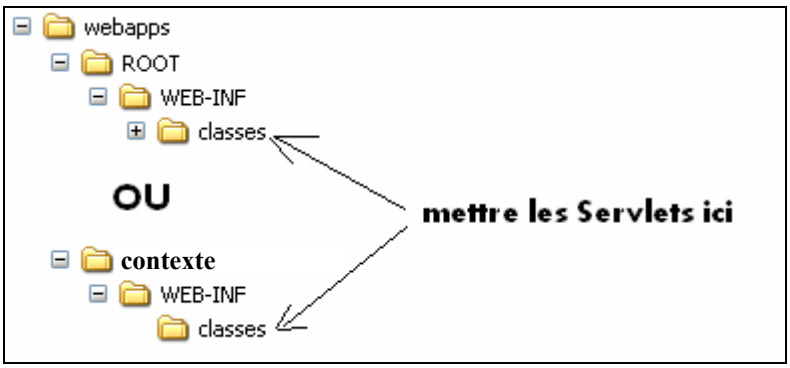
2- Le répertoire d'installation par défaut est :

```
C:\Program Files\Apache Software Foundation\Tomcat 6.0
```

3- On obtient l'arborescence suivante:



4- On développe les Servlets dans le répertoire « **webapps** », soit dans le sous répertoire ROOT soit dans un nouveau sous répertoire de webapps appelé un **context** (appelé par exemple « contexte »). Dans tous les cas, le répertoire de travail doit avoir l'architecture suivante :

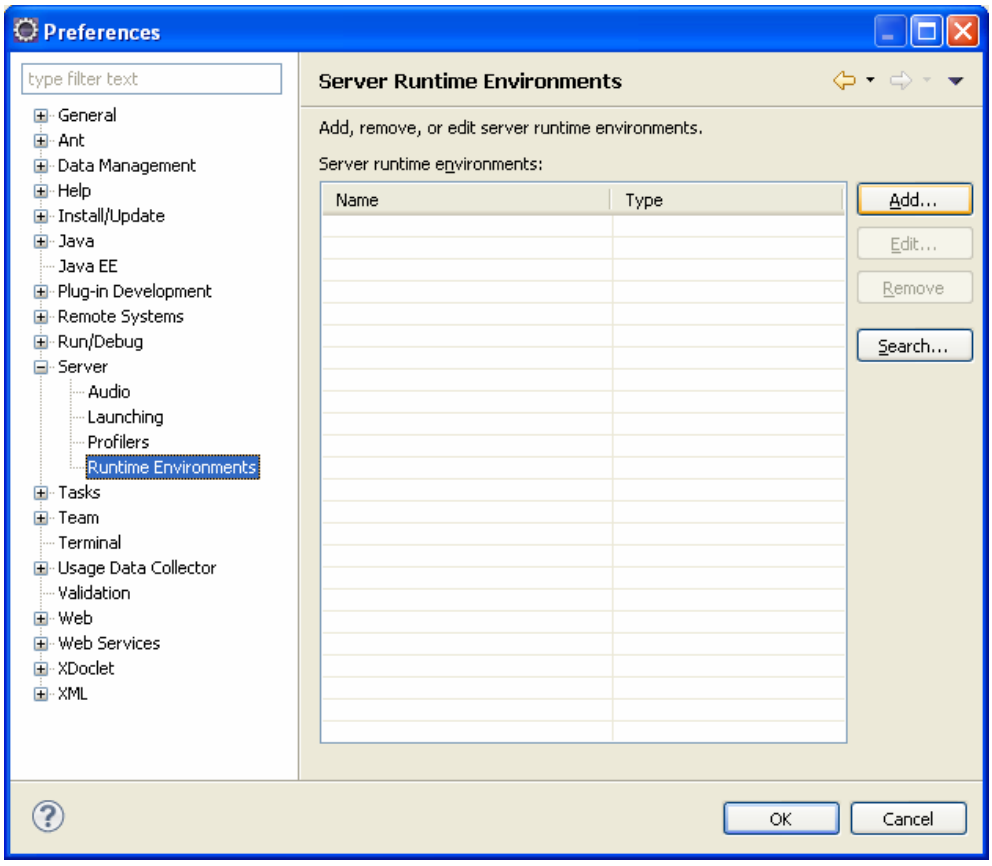


2.4 Configuration de Tomcat sous Eclipse JEE

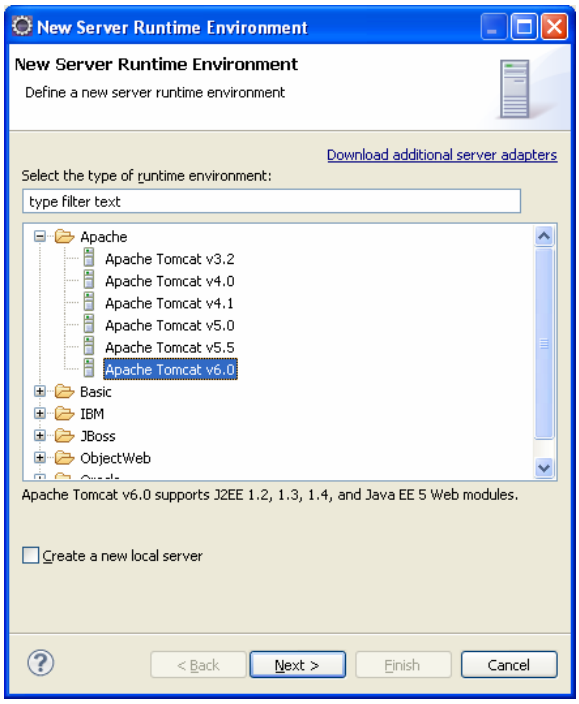
Commencer tout d'abord par la configuration de l'environnement Eclipse.

Pour cela allez au menu « Window/Preferences »

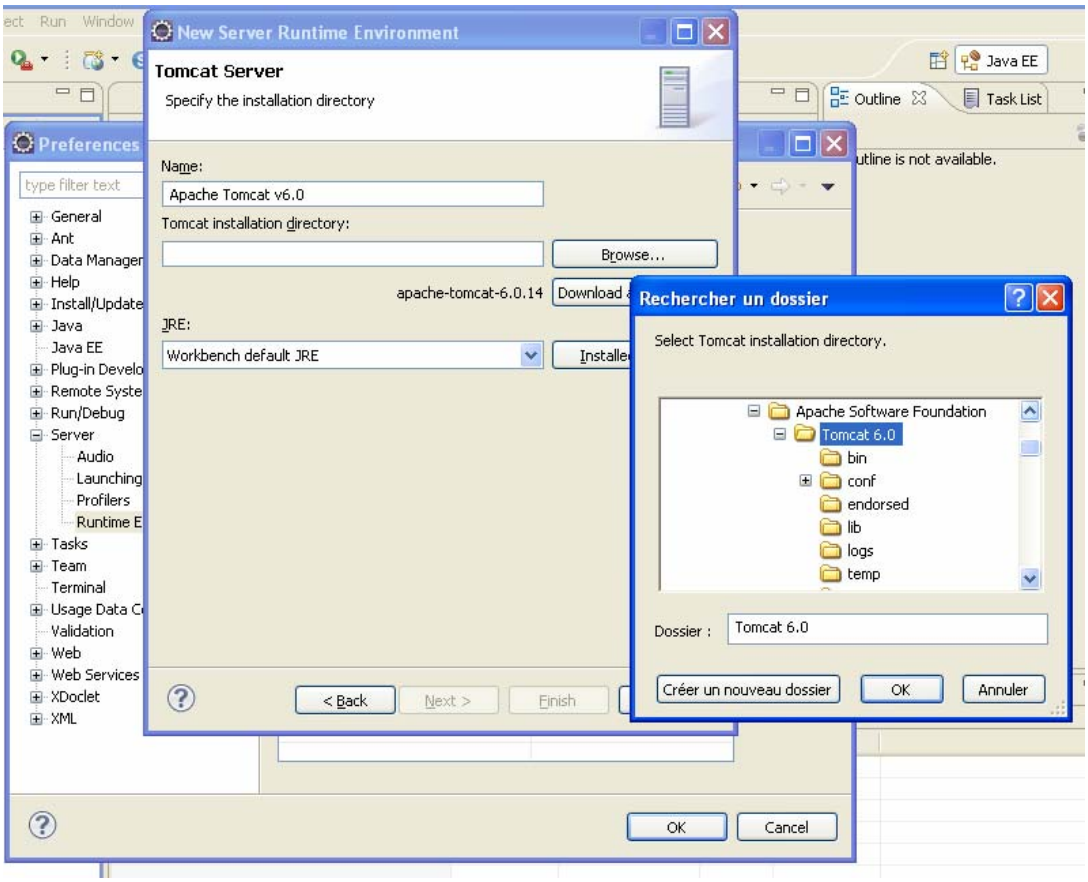
Ensuite « +Server/Runtime Environments », puis cliquer sur le bouton « add… »



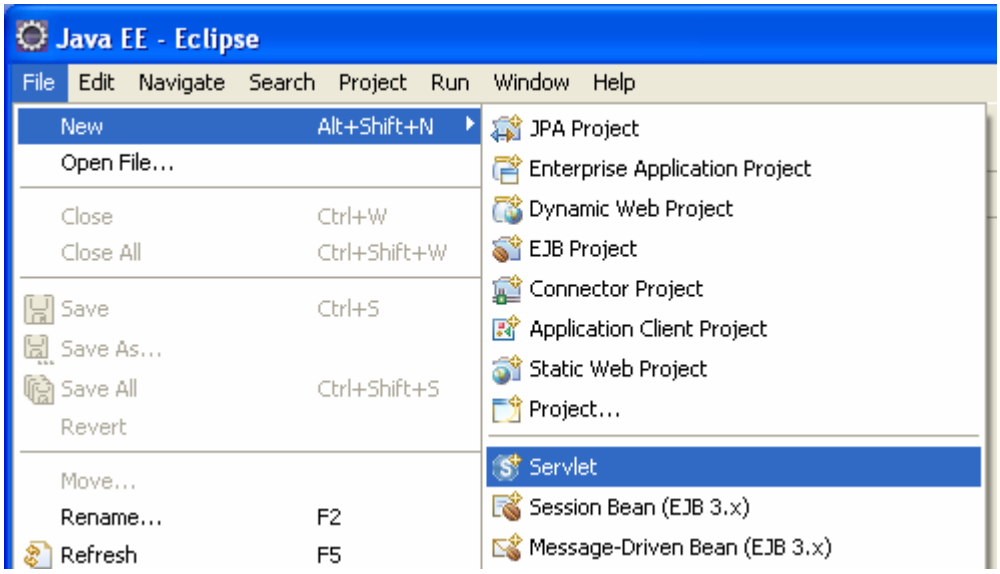
Sélectionner « Apache / Apache Tomcat v6.0 », puis « Next »



On obtient une nouvelle fenêtre dans laquelle il faut cliquer sur le bouton « browse » pour sélectionner le répertoire d'installation de Tomcat :



Créer ensuite votre application Web par le biais d'un projet web dynamique « Dynamic Web Project ». En fin créer la Servlet :



2.5 Développement d'une application Web basée sur les Servlets :

☆ Code de la Servlet :

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

public class FirstServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                throws ServletException {
     try {
             response.setContentType("text/html");
             PrintWriter out = response.getWriter();
             out.println("<html><body>");
             out.println("<h1 align='center'>First Servlet</h1>");
             out.println("</body></html>");
     }
     catch(Exception e) {}
    }
}
```

On peut ajouter à la Servlet une méthode doPost pour prendre en charge aussi bien les requêtes GET que les requêtes POST :

```
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                throws ServletException
    {
           doGet(request, response);
    }
```

☆ Descripteur de déploiement

Il s'agit du fichier « web.xml » se trouvant dans le répertoire WEB-INF du répertoire de travail :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
   version="2.4">


<!-- JSPC servlet mappings start -->
   <servlet>
      <servlet-name>FirstServlet</servlet-name>
      <servlet-class>FirstServlet</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>FirstServlet</servlet-name>
      <url-pattern>/f1.html</url-pattern>
   </servlet-mapping>
</web-app>
```

☆ Invocation de la servlet depuis l'Explorateur Internet :

- Si on a travaillé dans le répertoire ROOT :

  http://localhost:8080/f1.html

- Si on a travaillé dans un autre répertoire : exemple « contexte »

  http://localhost:8080/contexte/f1.html

2.6 L'objet HttpServletRequest

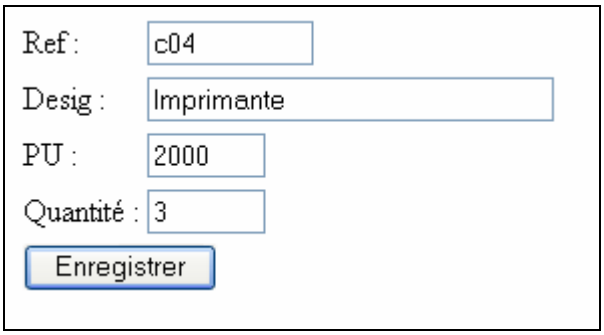
Méthode	Description
String getMethod()	Récupère la méthode HTTP utilisée par le client
String getHeader(String Key)	Récupère la valeur de l'attribut Key de l'en-tête
String getRemoteHost()	Récupère le nom de domaine du client
String getRemoteAddr()	Récupère l'adresse IP du client
String getParameter(String Key)	Récupère la valeur du paramètre Key (clé) d'un formulaire. Lorsque plusieurs valeurs sont présentes, la première est retournée
String[] getParameterValues(String Key)	Récupère les valeurs correspondant au paramètre Key (clé) d'un formulaire, c'est-à-dire dans le cas d'une sélection multiple (cases à cocher, listes à choix multiples) les valeurs de toutes les entités sélectionnées
Enumeration getParameterNames()	Retourne un objet *Enumeration* contenant la liste des noms des paramètres passés à la requête
String getServerName()	Récupère le nom du serveur
String getServerPort()	Récupère le numéro de port du serveur

2.7 L'objet HttpServletResponse

Méthode	Description
String setStatus(int StatusCode)	Définit le code de retour de la réponse
void setHeader(String Nom, String Valeur)	Définit une paire clé/valeur dans les en-têtes
void setContentType(String type)	Définit le type MIME de la réponse HTTP, c'est-à-dire le type de données envoyées au navigateur
void setContentLength(int len)	Définit la taille de la réponse
PrintWriter getWriter()	Retourne un objet « **PrintWriter** » permettant d'envoyer du texte au navigateur client.
ServletOutputStream getOutputStream()	Définit un flot de données à envoyer au client, par l'intermédiaire d'un objet **ServletOutputStream**, dérivé de la classe **java.io.OutputStream**
void sendredirect(String location)	Permet de rediriger le client vers l'URL *location*

2.8 Traitement des formulaires

1- On écrit un fichier HTML pour le formulaire :



Code du Formulaire :

```
<html>
<body>
<form name="FormName" action="RepProduit.html" method="post">
  <table border="0">
    <tr><td>Ref : </td><td><input type="text" name="Ref" size="10"></td>
    <tr><td>Desig : </td><td>
      <input type="text" name="Desig" size="30"></td>
    <tr><td>PU : </td><td><input type="text" name="PU" size="6"></td>
    <tr><td>Quantité : </td><td><input type="text" name="Q" size="6"></td>
    <tr><td colspan=2><input type="submit" value="Enregistrer"></td></tr>
  </table>
 </form>
</body>
</html>
```

Avec "**RepProduit.html**" étant le nom de mapping de la Servlet :

2- Code de La Servlet :

```
import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.*;
public class RepProduit extends HttpServlet {


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

      try {
            String ref = request.getParameter("Ref");
            String desig = request.getParameter("Desig");
            String pu = request.getParameter("PU");
            String q = request.getParameter("Q");

            Produit p = new Produit(ref, desig, pu, q);


            …
      }
      catch(Exception e) {}
   }

   public void doPost(HttpServletRequest request, HttpServletResponse
                     response) throws ServletException {
            doGet(request, response);
   }

}
```

2.9 Le Descripteur de déploiement

Le descripteur de déploiement d'une application web est un fichier nommé web.xml situé dans le répertoire WEB-INF du répertoire racine de l'application web. Il contient les caractéristiques et paramètres de l'application. Cela inclut la description des servlets utilisées, ou les différents paramètres d'initialisation.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
...
</web-app>
```

Dans le fichier web.xml, tous les paramètres de configuration sont définis entre les balises web-app.

2.9.1 Description de l'application

```
<web-app>
    ...
    <display-name>…</display-name>
    <description>… </description>
    ...
</web-app>
```

Deux balises relèvent de la description de l'application. La première, « display-name » permet de donner un nom à l'application. Il s'agit de description, pour s'y retrouver dans les fichiers. Cela n'a pas d'incidence sur le fonctionnement de la webapp. La balise « description » permet de fournir une description plus détaillée. De même, elle est inopérante techniquement parlant.

2.9.2 Déclaration des servlets

Il convient ensuite de déclarer les servlets que l'application utilise. C'est-à-dire les servlets potentiellement accessibles par les clients, au travers d'une URL.

```
<web-app>
    ...
    <servlet>
        <servlet-name>maservlet</servlet-name>
        <servlet-class>com.acs.jee.FirstServlet</servlet-class>
        <description>une servlet simple.</description>
        <init-param>
            <param-name>…</param-name>
            <param-value>…</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    ...
</web-app>
```

- La balise <servlet> … </servlet> doivent encadrer la déclaration d'une servlet.
- On doit affecter un nom à chaque servlet déclarée, dans une balise <servlet-name>. Ce nom n'est pas nécessairement le nom de la classe de la servlet. Il s'agit d'un identifiant interne au fichier web.xml, réutilisé par la suite.
- La balise <description> permet de fournir des informations supplémentaires sur la servlet, si on le souhaite. Cela n'a pas d'utilité technique, cette description n'est visible qu'ici.

- Enfin, il est bien sûr indispensable de définir la classe de la servlet, dans la balise <servlet-class>. Rappelons que cette classe doit se trouver dans le répertoire WEB-INF/classes de l'application, en respectant les packages. Dans notre cas, la classe devra être : WEB-INF/classes/com/acs/jee/FirstServlet.class.
- Il est possible de spécifier des paramètres d'initialisation pour une servlet. Il s'agit de paramètres chargés en même temps que la servlet, et qu'elle peut récupérer. Ce genre de paramètre se déclare au sein de balises <init-param>. La sous-balise <param-name> permet alors de définir le nom du paramètre, sa valeur étant spécifiée dans la sous-balise <param-value>. L'application peut récupérer la valleur de ce paramètre à l'aide de la méthode suivante :

  getServletConfig().getInitParameter("..."),

  le paramètre à fournir à la méthode étant un param-name de ce fichier, pour en récupérer la param-value.
- La balise load-on-startup demande que la servlet soit chargée dès le démarrage du serveur (et non lors de sa première sollicitation). Le nombre entier situé à l'intérieur de ces balises représente l'ordre de chargement. S'il y avait plusieurs servlets dans ce fichier, on pourrait définir par exemple la position 2 de chargement pour une autre...

2.9.3 Mapping des servlets

```
<web-app>
    ...
    <servlet-mapping>
        <servlet-name>maservlet</servlet-name>
        <url-pattern>/uneservlet</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Le mapping d'une servlet sert à indiquer au serveur quelle servlet charger pour une requête donnée du client. Remarquons que les URL des servlets sont relatives à l'URL du contexte (la webapp) auquel elles appartiennent.

Effectuer un mapping est très simple, et se déroule au sein d'une balise servlet-mapping. Il faut spécifier le nom de la servlet à mapper (balise servlet-name). Ce nom doit correspondre à un nom de servlet défini dans la déclaration des servlets. Puis, dans la balise url-pattern, définir l'URL par laquelle cette servlet est accessible. Il s'agit bien d'une pattern, car il est possible d'utiliser des caractères spéciaux. L'instruction suivante par exemple <url-pattern>/test/*</url-pattern> permet de configurer l'accès à la servlet dès que l'URL contiendra la séquence "/test/" (n'importe quelle URL commençant par "/test/").

2.9.4 Paramètres de contexte

Il est également possible de définir des paramètres de contexte pour une application web. A la différence des paramètres d'initialisation de servlets, ceux-ci seront accessibles par toutes les servlets de l'application web dont ce fichier est le descripteur de déploiement.

```
<web-app>
    ...
    <context-param>
        <param-name>…</param-name>
        <param-value>…</param-value>
        <description>…</description>
    </context-param>
    ...
</web-app>
```

Les balises param-name et param-value permettent respectivement de définir le nom du paramètre et sa valeur. Quant à la balise description, elle autorise comme d'habitude l'apposition d'une petite description, non opérante techniquement.

Une servlet peut récupérer un tel paramètre grâce à la méthode **getServletContext().getInitParameter("applicationName")**, le paramètre à fournir à la méthode étant un param-name de ce fichier, pour en récupérer la param-value.

2.9.5 Fichiers index

```
<web-app>
    ...
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        …
    </welcome-file-list>
    ...
</web-app>
```

Un descripteur de déploiement permet également de spécifier les fichiers d'accueil de l'application web. Dans notre exemple, si le path du Context de l'application web est "/appli", l'URL "http://localhost:port/appli" appelle la page d'accueil. Le serveur cherchera donc la page index.html dans le document root de l'application. Si elle n'existe pas (mieux vaut alors ne pas la fournir dans cette liste !), le fichier index.jsp sera recherché. Notons que, dans Tomcat 5 (spécification 2.4 des servlets), l'on peut définir une servlet comme fichier d'accueil, en insérant dans la liste son servlet-name.

2.9.6 Mapping de types MIME

```
<web-app>
    ...
    <mime-mapping>
        <extension>jpg</extension>
        <mime-type>image/jpeg</mime-type>
    </mime-mapping>
    ...
</web-app>
```

Lorsque le serveur recevra une requête pour une ressource statique (fichier HTML, image, fichier multimédia...), il enverra la ressource s'il la trouve. Dans la réponse HTTP, il devra paramétrer l'en-tête Content-Type (type de contenu). Pour ce faire, il se basera sur ces types MIME mappés, en fonction de l'extension du fichier à retourner. Donc, si par exemple un fichier .jpg lui est demandé, il définira le Content-Type comme image/jpeg. Ceci est présenté ici à titre d'information. Notons bien que les mapping de types MIME sont prévus dans le fichier web.xml général de Tomcat (TOMCAT_HOME/conf/web.xml). On peut donc en rajouter directement dans ce fichier. Néanmoins, il est utile de savoir que l'on peut en définir dans tous les descripteurs de déploiement, et limiter ainsi leur portée à l'application concernée.

2.9.7 Sécurisation des resources web

Contraintes de sécurité

Protéger l'accès à une ressource consiste à lui appliquer une contrainte de sécurité. Cette contrainte sera définie dans le descripteur de déploiement **web.xml**. Toute l'application ne sera pas nécessairement sécurisée, nous allons pouvoir définir quelles ressources sont à protéger.

Une contrainte de sécurité se définie entre les balises **<security-constraint>** et **</security-constraint>**. Exemple :

```
<web-app>
    ...
    <security-constraint>
        <display-name>...</display-name>
        <web-resource-collection>
            <web-resource-name>...</web-resource-name>
            <url-pattern>...</url-pattern>
            <url-pattern>...</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>...</role-name>
            <role-name>...</role-name>
        </auth-constraint>
    </security-constraint>
    ...
</web-app>
```

Comme dans les différentes configurations de déploiement, on fournit un nom pour notre contrainte de sécurité <**display-name**>.

L'élément <**web-resource-collection**> décrit les ressources pour lesquelles la contrainte va s'appliquer. La balise <**web-resource-name**> donne un nom à cet ensemble de ressources. Nom qui pourra être réutilisé dans certaines demandes d'authentification.

L'élément <**url-pattern**> permet de définir pour quelles URL la contrainte doit être mise en oeuvre. C'est donc ici que l'on sélectionne les ressources à protéger. Comme d'habitude, il s'agit d'un *pattern*, on peut donc notamment utiliser le caractère '*'. Si on met par exemple :

<url-pattern>/acs/*</url-pattern>

alors toutes les URL commençant par "/acs/" seront concernées par la protection. Comme d'habitude, il s'agit d'une URL relative à celle du contexte de l'application.

Grâce aux balises <**http-method>**, on peut définir les méthodes HTTP concernées par la contrainte. On pourra ainsi indiquer que l'accès à telle ressource par la méthode GET doit être protégée, mais pas l'accès à cette même ressource par la méthode POST. Dans l'exemple, nous avons protégé les deux.

Une fois les ressources à protéger sont définies, il convient de préciser quels utilisateurs, et plus exactement quels rôles peuvent y avoir accès. dans la balise <**auth-constraint**>, on peut indiquer les <**role-name**> ayant le droit d'accéder aux ressources définies plus haut. Ces noms de rôles doivent bien sûr correspondre aux rôles connus par Tomcat.

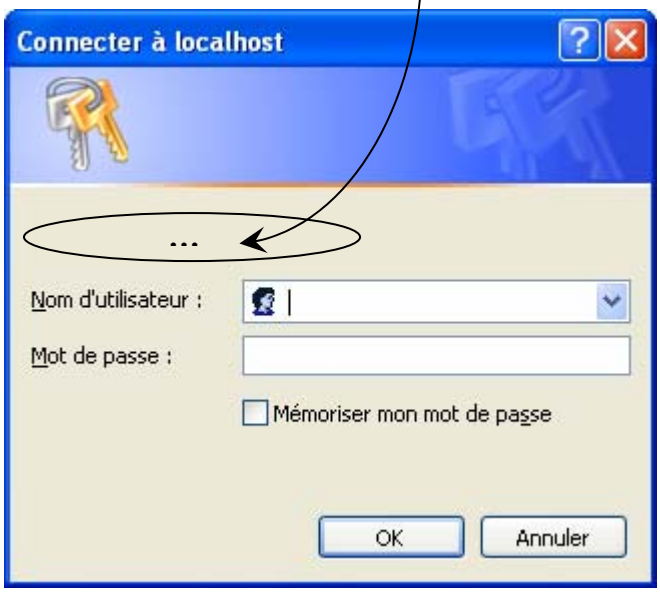
Une fois les contraintes de sécurité définies, nous devons spécifier le mode de *login* (ou authentification) des utilisateurs. Il existe plus qu'une solution :

- authentification basique utilisant une boîte de dialogue standard générée automatiquement.
- Ou par formulaire personnalisé par le développeur.

La configuration du mode de login à l'aide de l'authentification basique est réalisée à l'aide de la balise <**login-config**>. Le contenu de cet élément varie selon la méthode d'authentification. Nous nous intéressons ici à la méthode basique. Elle consiste en une boîte de dialogue, gérée par le navigateur, qui demande le *login* et le mot de passe. Elle envoie ensuite l'information au serveur qui se charge des vérifications nécessaires. Il est à noter que si l'on peut mettre plusieurs contraintes de sécurités dans une webapp, un seul **login-config** est acceptable :

```
<web-app>
    ...
    <security-constraint>
        ...
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>...</realm-name>
    </login-config>
    ...
</web-app>
```

Cela aura pour effet, lorsque l'utilisateur demandera une URL protégée, d'afficher la boîte de dialogue suivante dans le navigateur :



C'est le navigateur qui propose cette boîte de dialogue. Elle diffère donc d'un navigateur à l'autre (ici, Internet Explorer). Le *login* et le mot de passe saisis sont transmis au serveur. De son côté, il se charge de vérifier que le nom d'utilisateur est connu, et le mot de passe est correct. Puis il vérifie que cet utilisateur a bien l'un des rôles nécessaires pour accéder à la ressource, donc qu'il répond à la contrainte de sécurité.

2.10 Les cookies

Les cookies représentent un moyen simple de stocker temporairement des informations chez un client, afin de les récupérer ultérieurement. Concrètement il s'agit de fichiers textes stockés sur le disque dur du client après réception d'une réponse HTTP contenant des champs appropriés dans l'en-tête.

Les cookies font partie des spécifications du protocole HTTP qui permet d'échanger des messages entre le serveur et le client à l'aide de requêtes et de réponses HTTP.

Les requêtes et réponses HTTP contiennent des en-têtes permettant d'envoyer des informations particulières de façon bilatérale. Un de ces en-têtes est réservé à l'écriture de fichiers sur le disque: les cookies.

L'en-tête HTTP réservé à l'utilisation des cookies s'appelle Set-Cookie, il s'agit d'une simple ligne de texte de la forme:

Set-Cookie : NOM=VALEUR; domain=NOM_DE_DOMAINE; expires=DATE

Il s'agit donc d'une chaîne de caractères commençant par Set-Cookie : suivie par des paires clés-valeur sous la forme CLE=VALEUR et séparées par des virgules.

L'API servlet propose un objet permettant de gérer de façon quasi-transparente l'usage des cookies, il s'agit de l'objet Cookie.

L'objet Cookie

La classe **javax.servlet.http.Cookie** permet de créer un objet Cookie encapsulant toutes les opérations nécessaires à la manipulation des cookies.

Ainsi, le constructeur de la classe Cookie crée un cookie avec un nom et une valeur passés en paramètre. Il est toutefois possible de modifier la valeur de ce cookie ultérieurement grâce à sa méthode **setValue()**.

Conformément à la norme HTTP 1.1, le nom du cookie doit être une chaîne de caractères ne contenant aucun caractère spécial défini dans la RFC 2068 (Il vaut mieux donc utiliser des caractères alphanumériques uniquement). Les valeurs par contre peuvent inclure tous les caractères hormis les espaces ou chacun de ces caractères :

[ ] ( ) = , " / ? : ;

Envoi du cookie

L'envoi du cookie vers le navigateur du client se fait grâce à la méthode addCookie() de l'objet HttpServletResponse.

void addCookie(Cookie cookie)

Etant donnée que les cookies sont stockés dans les en-têtes HTTP, et que celles-ci doivent être les premières informations envoyées, la création du cookie doit se faire avant tout envoi de données au navigateur (le cookie doit être créé avant toute écriture sur le flot de sortie de la servlet)

```
Cookie MonCookie = new Cookie("nom", "valeur");


response.addCookie(MonCookie);
```

Récupération des cookies du client

Pour récupérer les cookies provenant de la requête du client, il suffit d'utiliser la méthode getCookies() de l'objet HttpServletRequest

Cookie[] getCookies()

Cette méthode retourne un tableau contenant l'ensemble des cookies présents chez le client. Il est ainsi possible de parcourir le tableau afin de retrouver un cookie spécifique grâce à la méthode getName() de l'objet Cookie().

Récupération de la valeur d'un cookie

La récupération de la valeur d'un cookie se fait grâce à la méthode getValue() de l'objet Cookie :

String Valeur = Cookie.getValue()

2.11 Les variables d'environnement

Les variables d'environnement sont, comme leur nom l'indique, des données stockées dans des variables permettant au programme d'avoir des informations sur son environnement.

Le langage Java fournit une méthode correspondant à chaque variable d'environnement. Voici la liste des méthodes de l'objet **HttpServletRequest** permettant de récupérer des variables d'environnement :

Variable d'environnement	Méthode associée
AUTH_TYPE	getAuthType()
CONTENT_LENGTH	getContentLength()
CONTENT_TYPE	getContentType()
HTTP_ACCEPT	getHeader("Accept")
HTTP_REFERER	getHeader("Referer")
HTTP_USER_AGENT	getHeader("User-Agent")
PATH_INFO	getPathInfo()
PATH_TRANSLATED	getPathTranslated()
QUERY_STRING	getQueryString()
REQUEST_METHOD	getRequestMethod()
REMOTE_ADDR	getRemoteAddr()
REMOTE_HOST	getRemoteHost()
REMOTE_USER	getRemoteUser()
SCRIPT_NAME	getScriptName()
SERVER_NAME	getServerName()
SERVER_PROTOCOL	getServerProtocol()
SERVER_PORT	getServerPort()

La méthode **getHeader()** de l'objet **HttpServletRequest** permet de récupérer la valeur d'une variable d'environnement dont le nom est passé en paramètre :

```
public String getHeader(String nomEnTete)
```

Il est possible de récupérer l'ensemble des noms des en-têtes disponibles dans un objet Enumeration grâce à la méthode **getHeaderNames()** de l'objet **HttpServletRequest** :

```
public Enumeration getHeaderNames()
```

L'objet Enumeration retourné peut alors être parcouru grâce à ses méthodes hasMoreElements() et nextElement() :

2.12 Suivi de Session

L'objet **HttpSession** permet de mémoriser les données de l'utilisateur, grâce à une structure similaire à une table de hachage, permettant de relier chaque id de session à l'ensemble des informations relatives à l'utilisateur.

Ainsi en utilisant un mécanisme tel que les cookies (permettant d'associer une requête à un id) et l'objet HttpSession (permettant de relier des informations relatives à l'utilisateur à un id), il est possible d'associer facilement une requête aux informations de session.

L'objet **HttpSession** s'obtient grâce à la méthode **getSession()** de l'objet **HttpServletRequest**.

Gérer les sessions

La gestion des sessions se fait de la manière suivante :

- Obtenir l'ID de session
  - Si GET: en regardant dans la requête
  - Si POST: en regardant dans les en-têtes HTTP
  - Sinon dans les cookies
- Vérifier si une session est associée à l'ID
  - Si la session existe, obtenir les informations
  - Sinon
    - Générer un *ID de Session*
    - Si le navigateur du client accepte les cookies, ajouter un cookie contenant l'ID de session
    - Sinon ajouter l'ID de session dans l'URL
    - Enregistrer la session avec l'ID nouvellement créé

Obtenir une session

La méthode getSession() de l'objet HttpServletRequest permet de retourner la session relative à l'utilisateur (l'identification est faite de façon transparente par cookies ou réécriture d'URL) :

HttpSession getSession(boolean create)

L'argument « **create** » permet de créer une session relative à la requête lorsqu'il prend la valeur **true**.

Etant donnée que les cookies sont stockés dans les en-têtes HTTP, et que celles-ci doivent être les premières informations envoyées, la méthode **getSession()** doit être appelée avant tout envoi de données au navigateur.

Obtenir des informations d'une session

Pour obtenir une valeur précédemment stockée dans l'objet HttpSession, il suffit d'utiliser la méthode **getAttribute()** de l'objet HttpSession (celle-ci remplace dans la version 2.2 de l'API servlet la méthode getValue() qui était utilisée dans les versions 2.1 et inférieures).

Object getAttribute("cle")

La méthode getAttribute() retourne une valeur de type Object. Si l'attribut passé en paramètre n'existe pas, la méthode getAttribute() retourne la valeur null.

Stocker des informations dans une session

Le stockage d'informations dans la session est similaire à la lecture. Il suffit d'utiliser la méthode **setAttribute()** (putvalue() pour les versions antérieures à la 2.2) en lui fournissant comme attributs la clé et la valeur associée.

Object setAttribute("cle","valeur")

L'exemple suivant stocke dans la session la page d'appel de la servlet (une page ayant fait un lien) :

```
HttpSession session = request.getSession(true);

session.setAttribute("referer",request.getHeader("Referer"));
```

Invalider une session

Pour supprimer une session, il suffit de faire appel à la méthode invalidate() de l'objet HttpSession

```
HttpSession session = request.getSession(true);

session.invalidate();
```

Autres méthodes de HttpSession

Méthode	Description
long getCreationTime()	Retourne l'heure de la création de la session
Object getAttributes(String Name)	Retourne l'objet stocké dans la session sous le nom *Name*, *null* s'il n'existe pas
String getId()	Génère un identifiant de session
long getCreationTime()	Retourne la date de la requête précédente pour cette session
String[] getValueNames()	Retourne un tableau contenant le nom de toutes les clés de la session
Object getValue(String Name)	Retourne l'objet stocké dans la session sous le nom *Name*, *null* s'il n'existe pas
void invalidate()	Supprime la session
boolean isnew()	Retourne *true* si la session vient d'être créée, sinon *false*
void putValue(String Name, Object Value)	Stocke l'objet *Value* dans la session sous le nom *Name*
void removeValue(String Name)	Supprime l'élément *Name* de la session
void setAttribute(String Name, Object Value)	Stocke l'objet *Value* dans la session sous le nom *Name*
int setMaxInactiveInterval(int interval)	Définit l'intervalle de temps maximum entre deux requête avant que la session n'expire
int getMaxInactiveInterval(int interval)	Retourne l'intervalle de temps maximum entre deux requête avant que la session n'expire

Chapitre 3.
Les JSP

3.1 Introduction

Les JSP (Java Server Pages) constituent une technologie Java permettant la génération de pages web dynamiques. Une JSP est habituellement constituée :

- de données et de tags HTML
- de tags JSP
- de scriptlets (code Java intégré à la JSP)

Les fichiers JSP possèdent par convention l'extension .jsp.

3.2 Les Tags JSP

Il existe trois types de tags :

- **tags de directives** : <%@ ... %>
- **tags de scripting** : ils permettent d'insérer du code Java dans la servlet générée.
- **tags d'actions** : <jsp:…>

Remarque :

Les noms des tags sont sensibles à la casse.

3.3 Les tags de directives <%@ ... %>

Syntaxe :

<%@ directive attribut="valeur" ... **%>**

Il existe 3 directives :

Directive	Description
page	permet de définir des options de configuration
include	permet d'inclure des fichiers dans la JSP
taglib	permet de définir des tags personnalisés

3.3.1 La directive page

Cette directive doit être utilisée dans toutes les pages JSP : elle permet de définir des options qui s'appliquent à toute la JSP.

Quelques options importantes de la directive Page :

Option	Valeur
contentType	Une chaîne contenant le type mime
errorPage	URL à ouvrir en cas d'erreur
extends	Une classe
import	Une classe ou un package.*, peuvent être séparés par ,
langage	Par défaut "java"

Exemples :

```
<%@ page language="java"
        contentType="text/html; charset=ISO-8859-1"
         pageEncoding="ISO-8859-1"%>

<%@page import="java.util.Vector"%>
```

3.3.2 La directive include

Cette directive permet d'inclure un fichier dans une JSP. Le fichier inclus peut être un fragment de code JSP, HTML ou Java.

Si le fichier inclus est un fichier HTML, celui ci ne doit pas contenir de tag <HTML>, </HTML>, <BODY> ou </BODY>

Syntaxe :

```
<%@ include file="chemin du fichier" %>
```

3.3.3 La directive taglib

Cette directive permet de déclarer l'utilisation d'une bibliothèque de tags personnalisés.

Cette directive possède deux attributs :

- **prefix** : un préfix qui servira d'espace de noms pour les tags de la bibliothèque dans la JSP
- **uri** : l'URI de la bibliothèque telle que définie dans le fichier de description

Exemple :

```
<%@ taglib prefix="s" uri="/struts-tags" %>
```

3.4 Les tags de scripting

Il existe trois tags pour insérer du code Java :

- **Tag de déclaration :**

  le code Java est inclus dans le corps de la servlet générée. Ce code peut être la déclaration de variables d'instances ou de classes ou la déclaration de méthodes.

  Syntaxe :

```
<%!
    ...
%>
```

- **Tag d'expression :**

  Evalue une expression et insère le résultat sous forme de chaîne de caractères dans la page web générée.

  Syntaxe :

```
<%= ... %>
```

- **Tag de scriptlets :**

par défaut, le code Java est inclus dans la méthode service() de la servlet.

```
<%
        ...
%>
```

Exemple :

```
<body>
   …
   <%
         for(int i=0;i < 20; i++)
               response.getWriter().println(i + "<br>");

   %>
   …
</body>
```

3.5 Les variables implicites

Les spécifications des JSP définissent plusieurs objets utilisables dans le code dont les plus utiles sont :

Object	Classe	Rôle
out	javax.servlet.jsp.JspWriter	Flux en sortie de la page HTML générée
request	javax.servlet.http.HttpServletRequest	Contient les informations de la requête
response	javax.servlet.http.HttpServletRequest	Contient les informations de la réponse
session	javax.servlet.http.HttpSession	Gère la session

3.6 Les tags de commentaires

Il existe deux types de commentaires avec les JSP :

- **les commentaires visibles dans le code HTML**

syntaxe :

```
<!-- ... <%= expression %> ... -->
```

- **les commentaires invisibles dans le code HTML**

syntaxe :

```
<%-- ... --%>
```

3.7 Tags d'actions

3.7.1 Le tag <jsp:useBean>

Le tag <jsp:useBean> permet de localiser une instance ou d'instancier un bean pour l'utiliser dans la JSP.

Syntaxe :

```
<jsp:useBean attribut="valeur" … >
        ...
</jsp :useBean>
```

Liste de quelques attributs :

Attribut	description
id	Nom de la variable qui va contenir la référence sur le bean
scope	Portée durant laquelle le bean est défini et utilisable. Les valeurs possibles sont : **page**|request|session|application
class	Classe du bean : pleinement qualifiée : nom.du.package.Classe
type	Généralement l'interface de base : mère de la classe du bean

Exemple :

```
<jsp:useBean
      id="produit01" scope="session"
      class="com.acs.lab.jsp.Produit"
 />
```

3.7.2 Le tag <jsp:setProperty >

Le tag <jsp:setProperty> permet d'affecter des valeurs aux propriétés du Bean (en utilisant implicitement leur setters).

Syntaxe :

Il existe trois façons de mettre à jour les propriétés soit à partir des paramètres de la requête soit avec une valeur :

- Affecter automatiquement toutes les propriétés avec les valeurs des paramètres correspondants de la requête. Dans ce cas, les noms des propriétés et ceux des paramètres doivent être identiques.

```
<jsp:setProperty
      name="idDuBean"
      property="*"
/>
```

- alimenter automatiquement une propriété avec le paramètre de la requête correspondant. Si l'attribut « param » est omis, alors le nom de la propriété et celui du paramètre doivent être identiques.

```
<jsp:setProperty
      name="idDuBean"
      property=" nomDeLaPropriété "
      [param=" nomDuparamètre "]
/>
```

- ☆ alimenter une propriété avec la valeur précisée

```
<jsp:setProperty
        name="idDuBean"
        property="nomDeLaPropriété"
        value="{string | <%= expression%>}"
/>
```

3.7.3 Le tag <jsp:getProperty>

Permet de récupérer et d'afficher la valeur d'une propriété du bean.

Syntaxe :

<jsp:getProperty name="*idDuBean*" **property**=" *nomPropriété*" **/>**

- L'attribut « **name** » indique le nom du bean tel qu'il a été déclaré dans le tag <jsp:useBean>.
- L'attribut « **property** » indique le nom de la propriété en question.

3.7.4 Le tag de redirection <jsp:forward>

Cette balise permet de rediriger la requête vers une autre URL pointant vers un fichier HTML, JSP ou un servlet.

Syntaxe :

<jsp:forward page="{*relativeURL* | <%= *expression* %>}" **/>**

3.7.5 Le tag <jsp:include>

Ce tag permet d'inclure le contenu généré par une JSP ou une servlet dynamiquement au moment ou la JSP est executée. C'est la différence avec la directive include avec laquelle le fichier est inséré dans la JSP avant la génération de la servlet.

Syntaxe :

<jsp:include page="relativeURL" **/>**

Chapitre 4. Accès aux Bases de données avec JDBC

4.2 Chargement d'un pilote JDBC

La classe DriverManager gère la liste des drivers chargés par l'intermédiaire de la classe « Class », et offre donc la possibilité d'ouverture de connexions à travers les pilotes chargés. Le chargement d'un driver est effectué comme suit :

```
Class.forName( "classe du driver" );
```

Une fois cette étape effectuée JDBC est pret à se connecter à la base de donnée.

4.3 Connexion à la base de donnée

Pour se connecter à une base de données il vous faut un identifiant, un mot de passe et l'adresse de connexion au serveur.
La classe gérant les connexions est Java.SQL.Connection et la méthode permettant de créer une connexion est getConnection. Une connexion s'effectue de la manière suivante :

```
Connection db = DriverManager.getConnection("URL", "user" , "pwd" );
```

La classe Connection renvoie une exception SQLException si une erreur d'accès à la base de données se produit.

Une fois la connexion effectuée, il est désormais possible d'exécuter des requêtes sur la base de données.

Cas d'une base de données MySQL

Pour MySQL il faut utiliser un driver spécifique disponible gratuitement sur http://dev.mysql.com/downloads/

```
private String driver = "com.mysql.jdbc.Driver";
private String bridge = "jdbc:mysql:";

private Connection db;
private DatabaseMetaData dbm;


//Etape N° 1 : Chargement du Pilote d'acces base de données
try {
      Class.forName(driver);
} catch (Exception e) {
      System.out.println("Erreur de chargement du Driver");
}

//Etape N° 2 : ouverture d'une connexion base de données
url = bridge + "/" + host + "/" + dbName;
try {
      db = DriverManager.getConnection(url, user,
                                                pwd);
      dbm = db.getMetaData();
} catch (Exception e) {
      System.out.println("Erreur de Connexion");
}
```

Cas d'une base de données Oracle

Un driver spécifique est aussi disponible pour Oracle, celui-ci se trouve dans les répertoires d'installation d'oracle.

L'ouverture de la base de données est effectuée en respectant le même scénario, mais avec les paramètres suivants :

```
private String driver = " oracle.jdbc.driver.OracleDriver";
private String bridge = " jdbc:oracle:thin:";

url = bridge + "@" + host + ":1521:" + dbName;
```

Cas d'une base de données Access via ODBC

Un driver natif est fourni avec le JDK permettant l'accès aux bases de données ACCESS publiées avec les pilotes ODBC.

L'ouverture de la base de données est aussi effectuée en respectant le même principe. Les paramètres sont les suivants :

```
private String driver = " sun.jdbc.odbc.JdbcOdbcDriver";
private String bridge = " jdbc:odbc:";

url = bridge + dbName;
```

4.4 Exécution des requêtes SQL

Une requête est un ordre SQL envoyé à la base et exécuté par celle-ci. Une requête peut renvoyer des données s'il s'agit d'un SELECT par exemple, ou modifier le contenu de la base s'il s'agit d'un INSERT par exemple.

- L'exécution des requêtes se fait à travers un objet de type « **java.sql.Statement** ». ce dernier s'initialise avec la méthode **createStatement** de la classe Connection
- Dans le cas d'une requête « SELECT », les données retournées seront stockées dans un objet de type « **java.sql.ResultSet** ».
- La classe gérant les requêtes est et celle gérant les resultset est **java.sql.ResultSet**.
- Les requêtes s'exécutent ensuite avec la méthode **executeQuery()** de l'objet « **Statement** » pour une requête SELECT, « **executeUpdate()** » pour les autres requêtes DML (INSERT, UPDATE et DELETE), et finalement « **execute()** » pour les requêtes DDL.
- Un **resultset** contenant plusieurs lignes se parcourt à l'aide de la méthode « next » qui renvois false lorsqu'il n'y a plus de lignes

retournées. L'accès aux colonnes de chaque ligne se faisant via les méthodes **get** suivis du nom du type de la colonne telles que getInt(), getString(), …. Ces méthodes get accèdent aux colonnes via leur nom ou leur index (commençant à partir de 1).

Voici un exemple d'une requête Select :

```
Statement sql = db.createStatement();
ResultSet rs = st.executeQuery( "SELECT * FROM produit" );
while( rs.next() ) {
        System.out.println( rs.getString( "desig" ) );
        System.out.println( rs.getDouble(3) );
}
```

Remarques

La classe **ResultSet** possède de nombreuses méthodes permettant de travailler sur les requêtes. Ainsi il est possible de changer la ligne courante avec absolute, ou encore de se déplacer directement à la première ou dernière ligne avec **first** et **last**. Il est possible de supprimer une ligne du résultat et de la base de données avec **deleteRow**, d'insérer des lignes avec **insertRow** ou encore de modifier des valeurs avec **update** suivit du type de donné a modifier (**updateInt** par exemple).

Les classes Statement et ResultSet renvoient une exception SQLException si une erreur d'accès à la base de données se produit. Il est possible de récupérer les erreurs renvoyées par le SGBD avec la méthode getWarnings qui renvois un objet de type SQLWarning
Une fois vos requêtes effectuées il est nécessaire de fermer la connexion.

Les connexions, les Statement et les ResultSet peuvent être terminés par le développeur ou automatiquement par le « Garbage Collector ». Il est fortement recommandé de fermer toutes les ressources utilisées lorsqu'elles ne sont plus nécessaires afin d'éviter tous problèmes.

La fermeture d'un objet Statement, Connection ou ResultSet se fait via leurs méthodes « **close** ».

```
rs.close();
sql.close();
db.close();
```

4.5 Les requêtes préparées ou pré compilées : « Prepared Statement »

La classe PreparedStatement est directement dérivée de la classe Statement. Elle reprend donc les mêmes méthodes et fonctionne d'une manière similaire. Mais contrairement au Statement classique, la requête est fournie des la création de l'objet. Elle est alors immédiatement envoyée au SGBD qui la compile. Ainsi un PreparedStatement n'exécute pas une requête SQL mais une requête SQL pré compilée. Cela à pour effet une réduction très forte du temps d'exécution lorsque vous exécutez plusieurs fois la même requête. De plus il y a possibilité de fournir des paramètres aux requêtes permettant ainsi d'utiliser les PreparedStatement avec des requêtes similaires et non forcément identiques.

La création d'un PreparedStatement se fait via l'appel de la méthode createPreparedStatement, de la classe Connection, à laquelle on passe la requête. Les paramètres étant désigné par des "?".

```
PreparedStatement pst = conn.preparedStatement( "INSERT INTO ma_table VALUES( ? )" );
```

Une fois la requête prête il va falloir lui donné les valeurs des paramètres avant de l'exécuter. Il faut pour cela utiliser les méthodes set suivit du type du paramètre (setInt par exemple). Il faut spécifié l'index du paramètre (indexé a 1) puis la valeur du paramètre.
Une fois tous les paramètres définis on appel la méthode d'exécution voulue (execute, executeQuery ou executeUpdate).

```
PreparedStatement sql = conn.preparedStatement( "INSERT INTO
ma_table VALUES( ? )" );
for( int cmp = 0 ; cmp < 10 ; cmp++ ) {
  sql.setInt( 1 , cmp );
  sql.executeUpdate();
}
```

Une fois son utilisation terminée, il faut cette fois-ci encore fermer le PreparedStatement à l'aide de la méthode close.

```
sql.close();
```

Les transactions :

Il est possible d'annuler tous ou certains des changements effectués avec la méthode rollback couplée a l'utilisation de setSavePoint et de les valider avec la méthode commit, il est également possible de confirmer automatiquement tous les ordres passés avec autoCommit.

4.6 Appel de procédures stockées

1. Préparer l'appel :

CallableStatement st ;

3 cas se présentent :

1. procedure sans paramètres :

st = db.**prepareCall(**"{call nomProcedure}" **)**;

2. procedure avec paramètres :

st = db.**prepareCall(**"{call nomProcedure(?, …, ?)}" **)**;

3. fonction qui retourne un résultat :

st = db.**prepareCall(**"{ ? = call nomProcedure (?, …, ?)}" **)**;

2. Définir les valeur des paramètres d'entrée :

Utiliser les méthodes **setXXX(,)** du **Statement**. Exemple :

```
st.setInt(1, valeur);
```

3. Fixer le type des paramètres de sortie :

Utiliser la fonction **registerOutParameter** du statement. Exemple :

```
st.registerOutParameter(2, java.sql.Types.FLOAT);
```

4. Exécution de la procédure :

```
st.execute();
```

5. Récupération du résultat d'appel :

Utiliser les méthodes **getResultSet()** et **getMoreResults()** du **Statement**.

6. Récupération des valeurs des paramètres de sortie :

Utiliser les méthodes **getXXX(,)** du **Statement**. Exemple :

```
System.out.println("Résultat = " + st.getFloat(2));
```

4.7 Gestion des transactions

Le mode par défaut est « Auto Commit »:

- connexion.setAutoCommit(false);
- connexion.commit();
- connexion.rollback();

4.8 Accès à la structure de la base de données

Méta données sur les ResultSet

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

Informations disponibles :

- nombre de colonnes,
- Libellé d'une colonne,
- table d'origine,
- type associé à une colonne,
- la colonne est-elle nullable?
- etc.

Méta données sur la base

```
DataBaseMetaData dbmd = connexion.getMetaData();
```

Informations disponibles :

- tables existantes dans la base,
- nom d'utilisateur,
- version du pilote,
- etc.

4.9 Extensions JDBC

Nouvelle version des « ResultSet »

Il existe quatre types de Result Set:

- **scroll-insensitive**: Vision figée du résultat de la requête au moment de son évaluation.
- **scroll-sensitive**: Le Result Set montre l'état courant des données (modifiées/détruites).
- **read-only**: Pas de modification possible donc un haut niveau de concurrence.
- **updatable**: Possibilité de modification donc pose de verrou et faible niveau de concurrence.

Exemple :

```
Statement sql = con.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM Produit");
```

Déplacement dans un « ResultSet »

```
rs.first();
rs.beforeFirst();
rs.next();
rs.previous();
rs.afterLast();
rs.absolute( n );
rs.relative( n );
```

Modification d'un « ResultSet »

Modification:

```
rs.absolute(100);
rs.updateString("Desig", "Clavier");
rs.updateInt("Prix", 150);
rs.updateRow();
```

Destruction:

```
rs.deleteRow();
```

Insertion de lignes:

```
rs.moveToInsertRow();
rs.updateString("Desig", "Clavier");
rs.updateInt("Prix", 150);
rs.insertRow();
rs.first();
```

Exécution des requêtes en batch

```
connexion.setAutoCommit(false);
Statement st = connexion.createStatement();
st.addBatch("INSERT ...");
st.addBatch("DELETE ...");
st.addBatch("UPDATE ...");
…
int[] nb = st.executeBatch();
```

4.10 Gestion des DataSources sous Tomcat

L'interface **DataSource** a été ajoutée à l'API JDBC (**javax.sql.DataSource**) pour définir les paramètres de connexion à une base de donnée.

La configuration des datasources sous tomcat peut être réalisée de différentes manières. Nous présentons 2 solutions :

1ère Solution :

Fournir les paramètres de connexion sous forme d'une ressource dans le fichier de configuration « context.xml » du répertoire « conf » se trouvant dans le répertoire d'installation de Tomcat.

Voici un exemple :

```
<Context>
  ...
  <Resource
       name="jdbc/stock"
       type="javax.sql.DataSource"
       username="root"
       password=""
       driverClassName="com.mysql.jdbc.Driver"
       url="jdbc:mysql://localhost/stock"
  />
</Context>
```

2ème Solution :

Faire la même configuration, mais à travers un fichier « context.xml » du sous répertoire « MATA-INF » de l'application Web.

Dans tous les cas l'ouverture de connexion pourra être réalisée ensuite à l'aide des instructions suivantes :

```
try {
 DataSource ds = (DataSource) new
InitialContext().lookup("java:comp/env/NomDeLaRessource");

 Connection db = ds.getConnection();
}
```

Exemple :

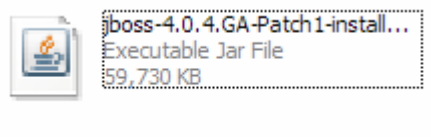
```
try {
    DataSource ds = (DataSource) new
InitialContext().lookup("java:comp/env/jdbc/stock ");

    Connection db = ds.getConnection();
}
```

Chapitre 5. Les EJB3 avec JBoss et Eclipse

4.1 Installation de JBoss avec le support des EJB 3

JBoss est un serveur d'application J2EE développé à partir de 1999 par un français Marc FLEURY. Ce serveur est écrit en Java et il est distribué sous licence LGPL.

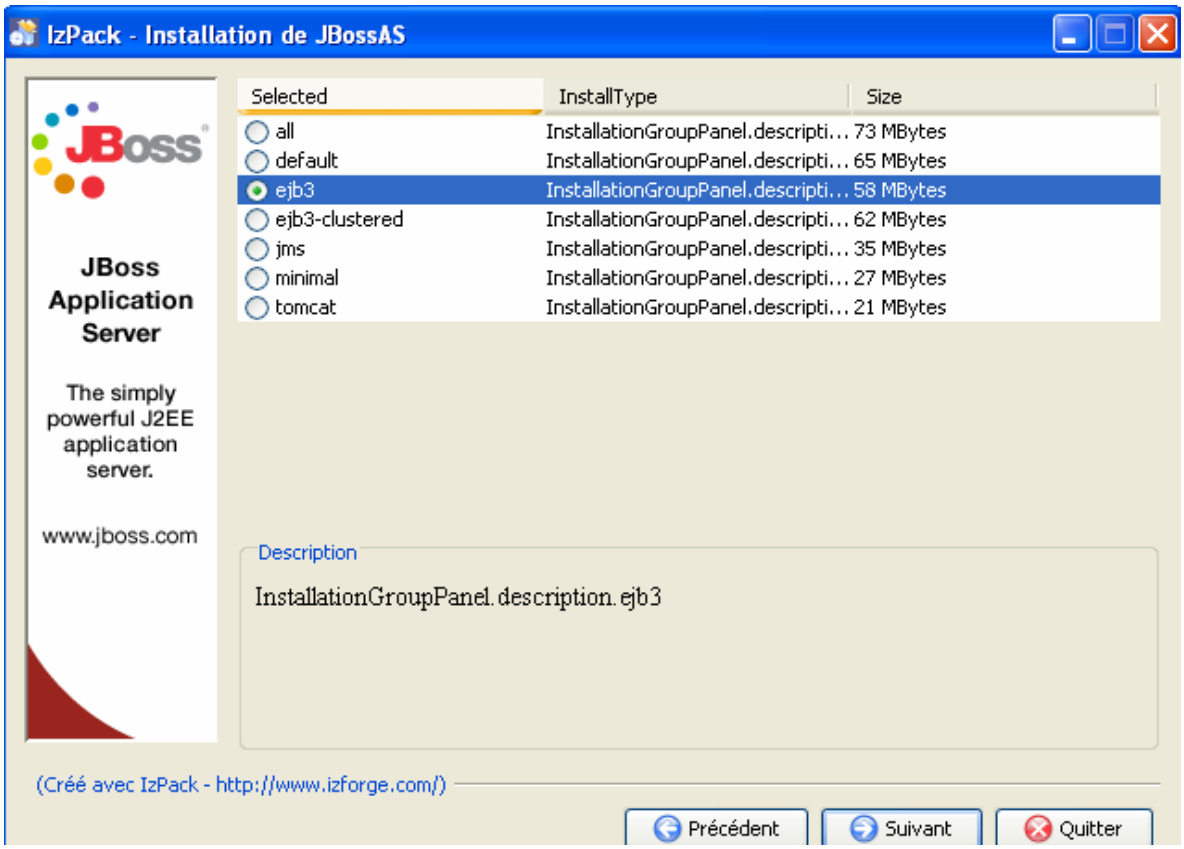
JBOSS est disponible en téléchargement gratuit sur le lien suivant : http://www.jboss.org/download/



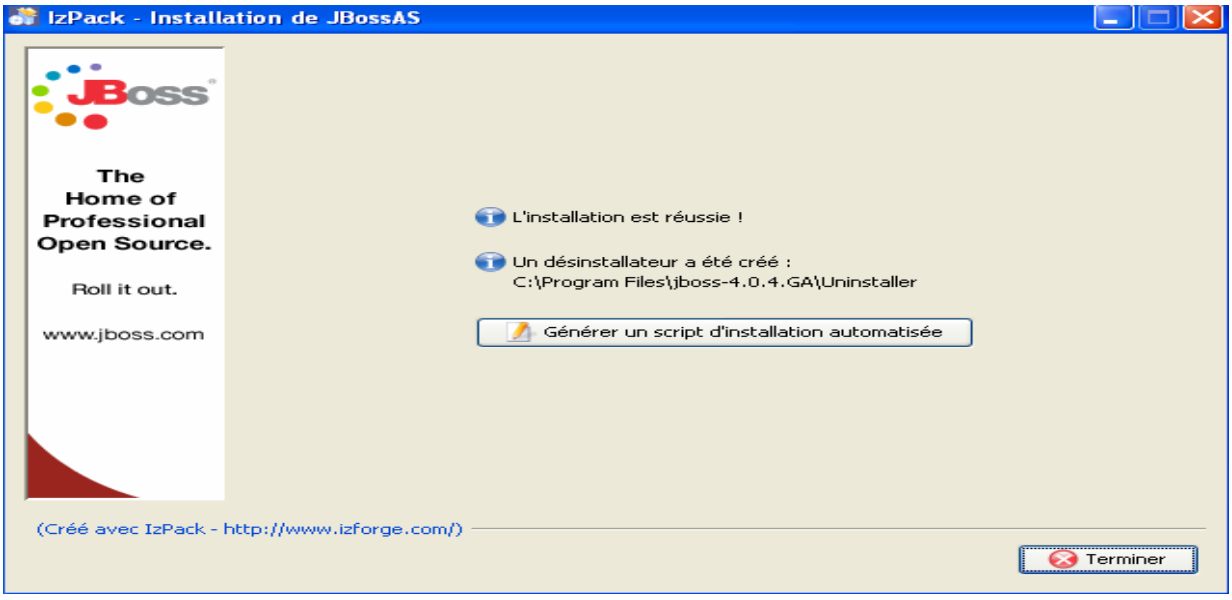
Exécuter l'installeur en double-cliquant sur le fichier suivant :

jboss-4.0.4.GA-Patch1-installer.jar

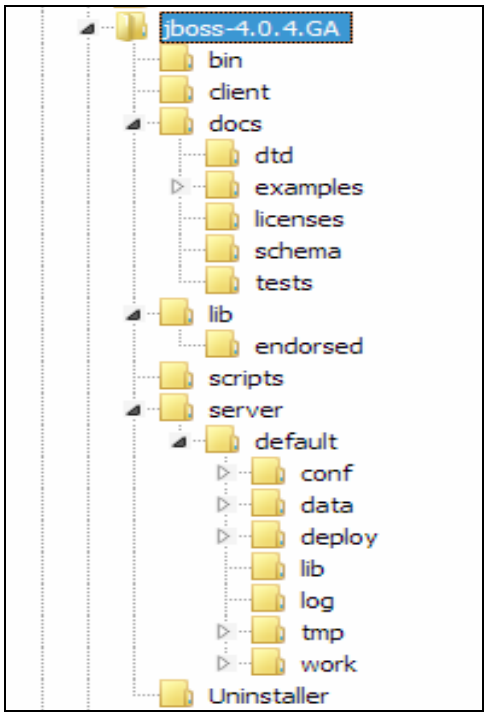
On clique sur OK puis suivant, puis choisir le support « ejb3 »:



On clique sur suivant puis suivant pour terminer l'installation :



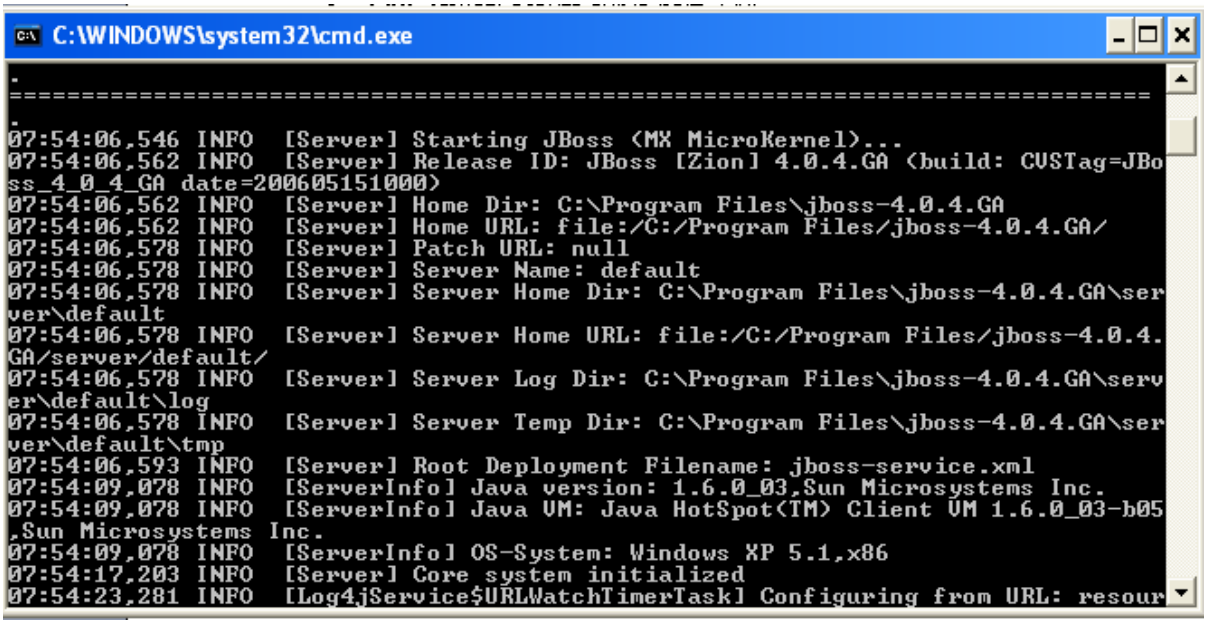
L'arborescence du répertoire d'installation de JBOSS:



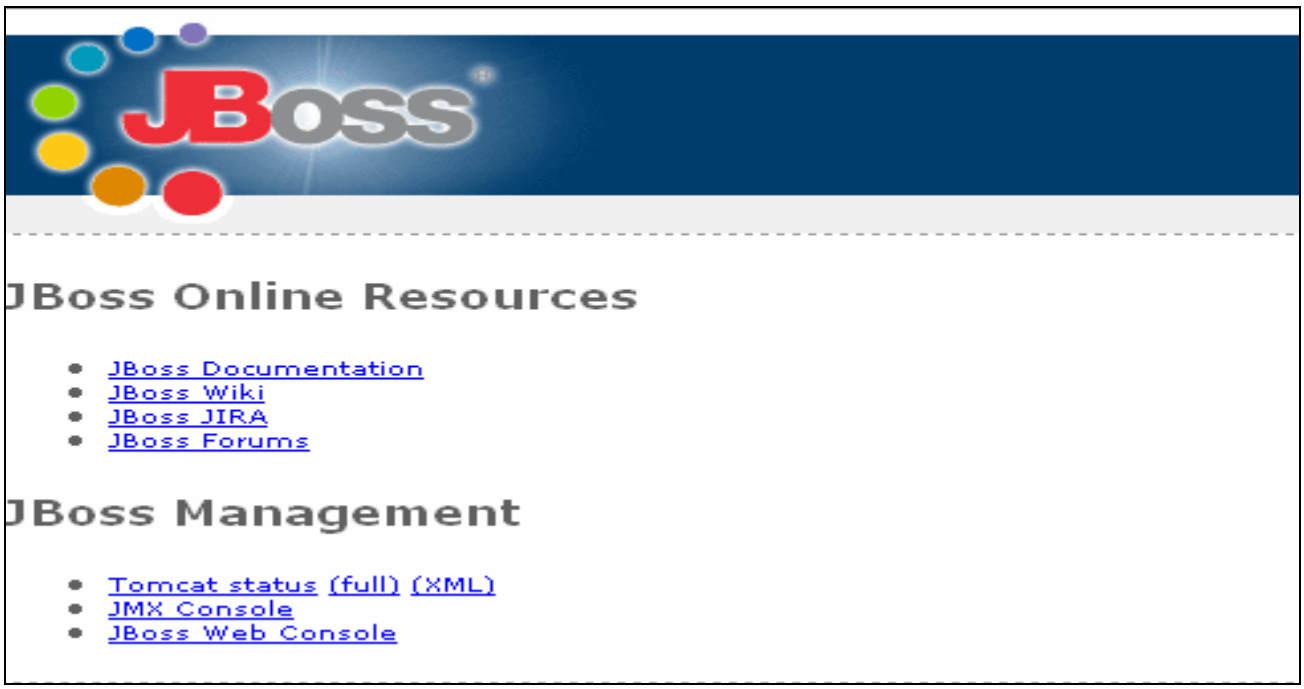
- **bin** contient un fichier run.bat qui permettra de démarrer le serveur JBOSS.
- **lib contient les librairies utilisées par JBOSS**
- **log contient les logs de fonctionnement de JBOSS**
- **server/default contient l'environnement qui sera utilisé par** défaut au lancement de JBoss.
    - **Conf** contient certains éléments de configuration. On y trouve par exemple la configuration log4j utilisé par JBOSS pour les applications qu'il héberge.
    - **Deploy** contient les applications hébergées par JBoss. On y trouve notamment :
        - jbossweb-tomcat55.sar qui est le Tomcat embarqué par JBoss
        - C'est dans ce répertoire que seront déployé les EJB

Afin de vérifier que l'installation et la configuration Jboss est correcte, on démarre JBoss en exécutant le « run.bat » qui se trouve dans le chemin suivant :

Jboss-4.0.4.GA\bin\run.bat

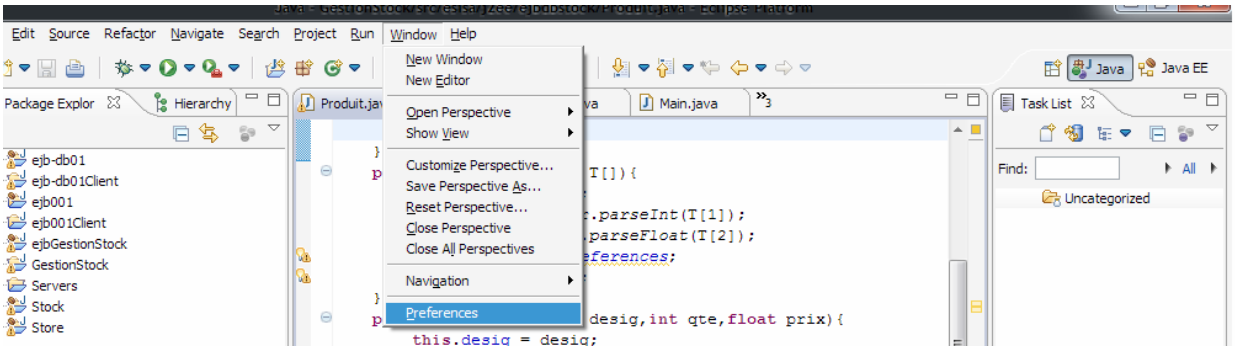


Démarrer la page d'accueil : http://localhost:8080/



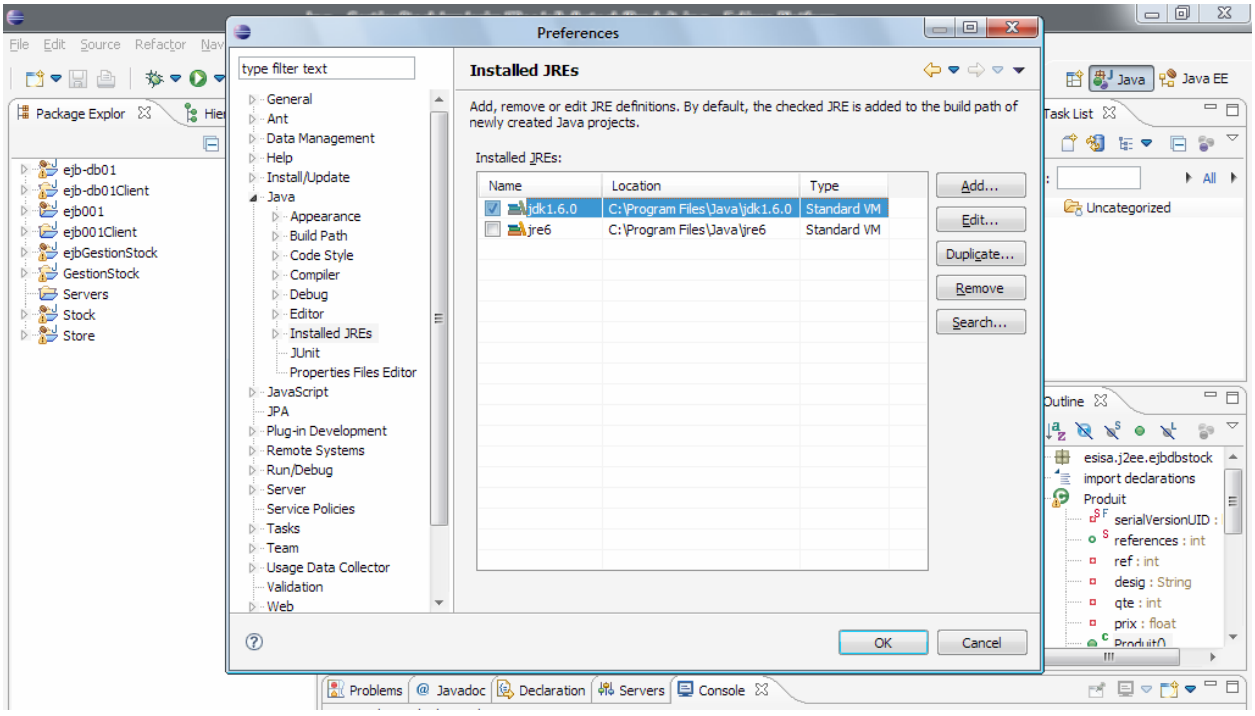
4.2 Configuration Eclipse

Nous utilisons Eclipse JEE (Ganymede ou Galileo) disponible sur le site : http://www.eclipse.org sous la rubrique DOWNLODS

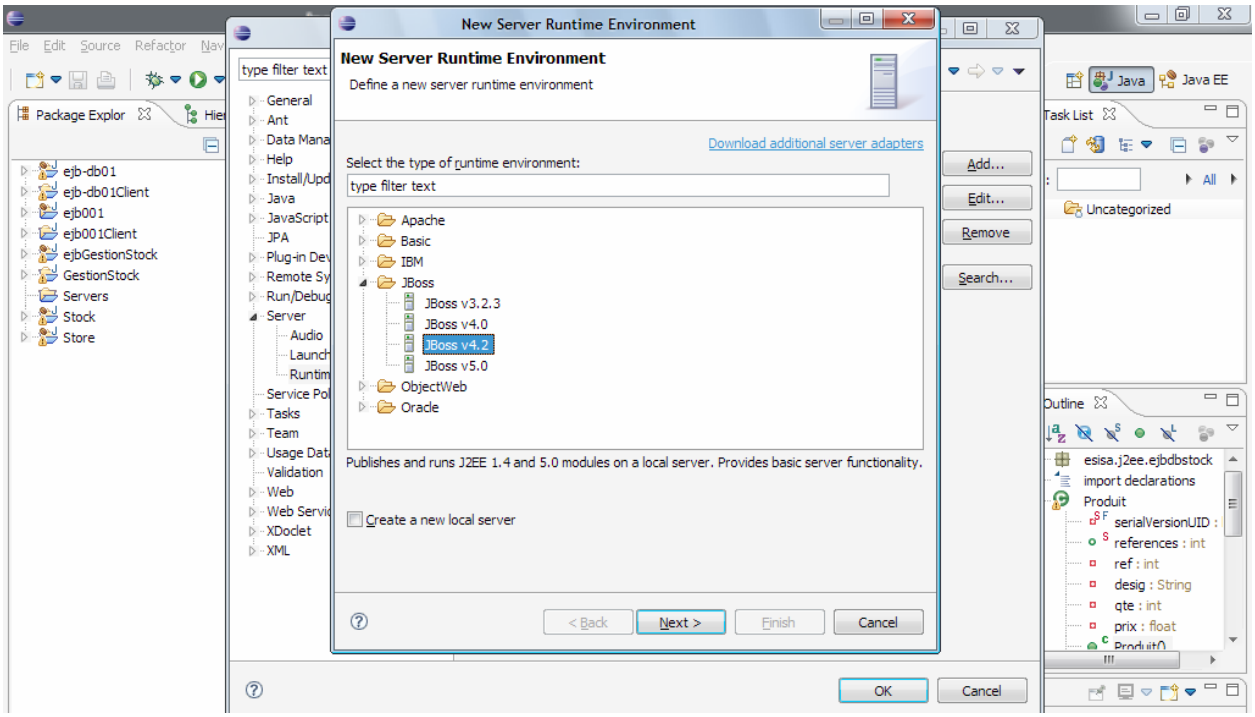


Aller par la suite sur le sous-item Installed JREs sous l'item JAVA, pour configuer le JDK. On ajoute et on sélectionne alors le chemin vers le JDK.

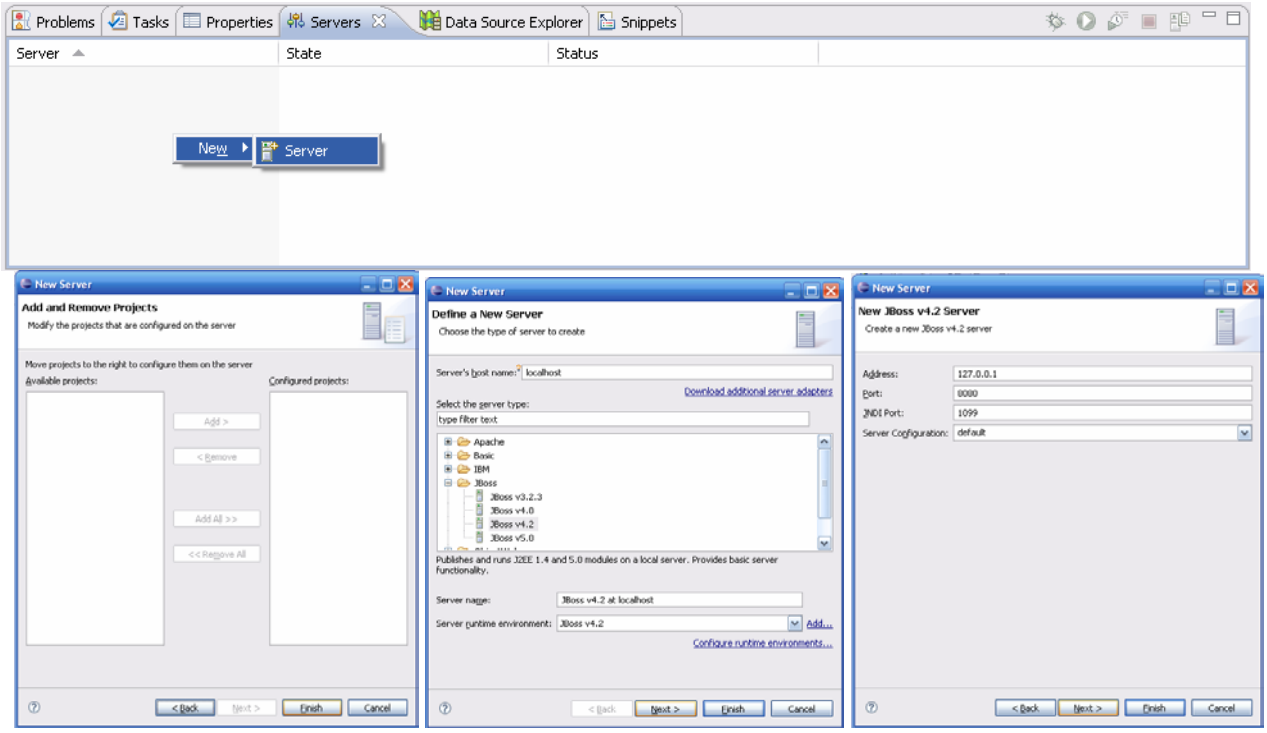
On obtient la fenêtre suivante :



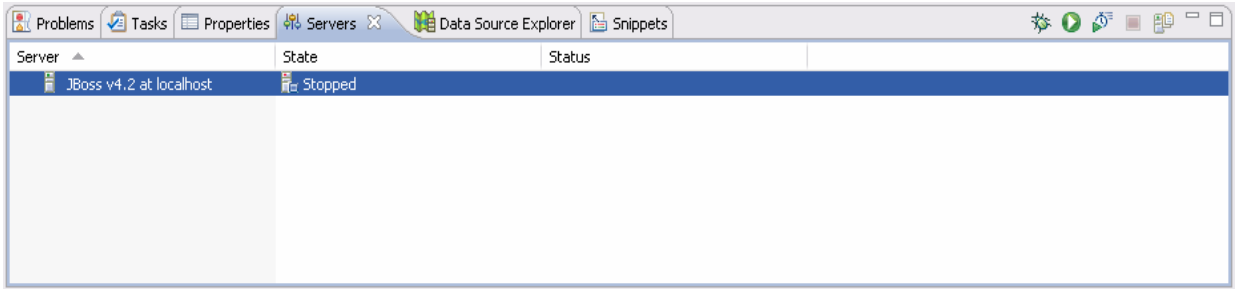
On ajoute ensuite sous la rubrique « Server » / « Runtime Environments » le chemin vers le serveur JBoss :



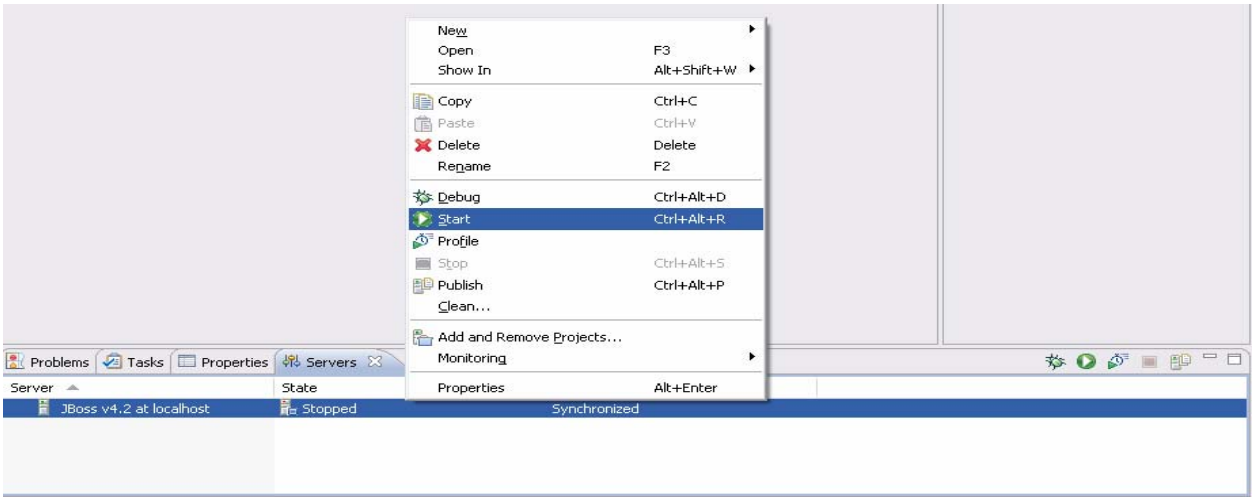
On ajoute ensuite le serveur dans la vue « Servers » en bas de la fenêtre Eclipse : Clique bouton droit, en suite « New » puis « Server »…

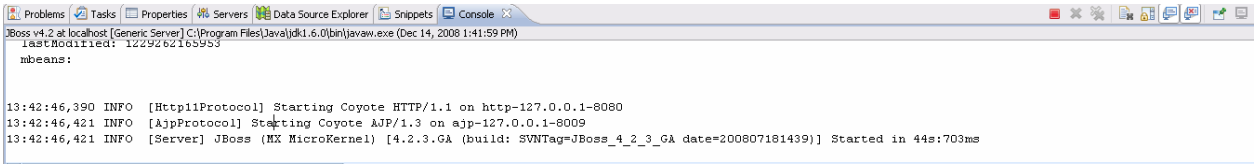


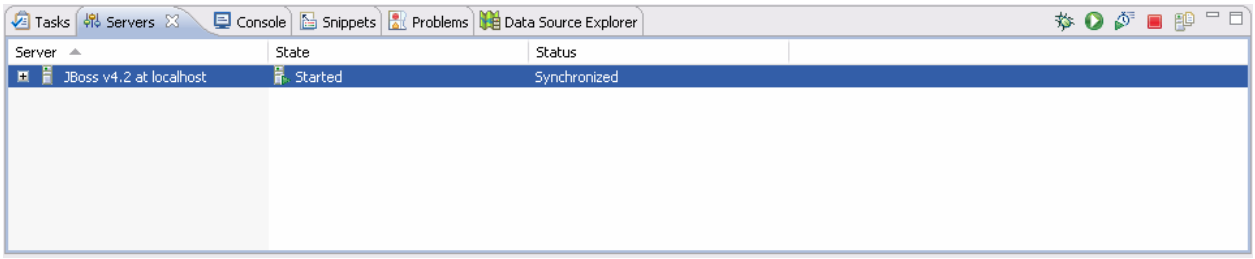
L'icône du serveur JBOSS devrait alors apparaître dabs la vue « Servers »



Pour le faire démarrer, cliquez dessus :



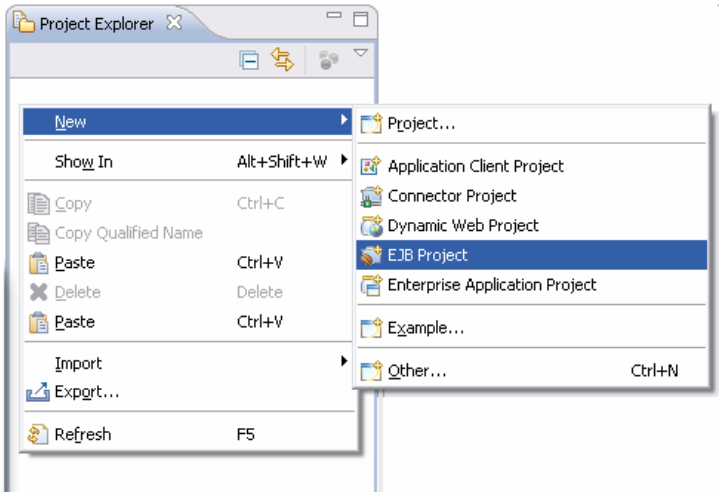


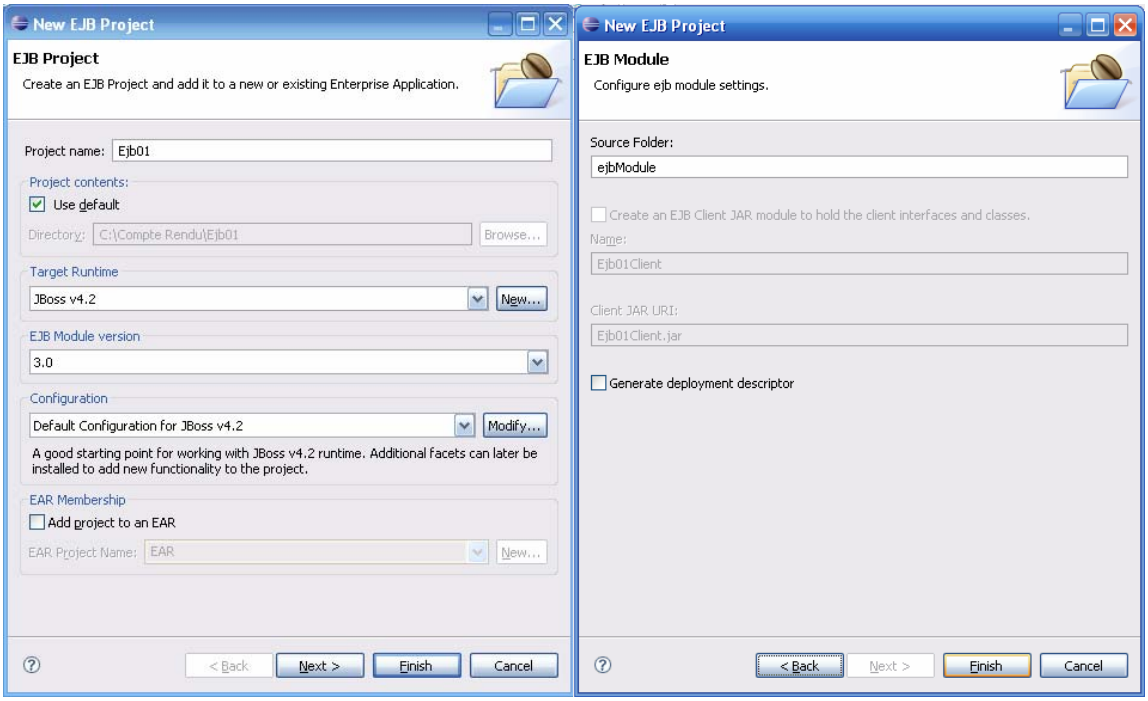


4.3 Création d'un Bean session avec Eclipse et JBoss

Etape N° 1 : Création du Projet :

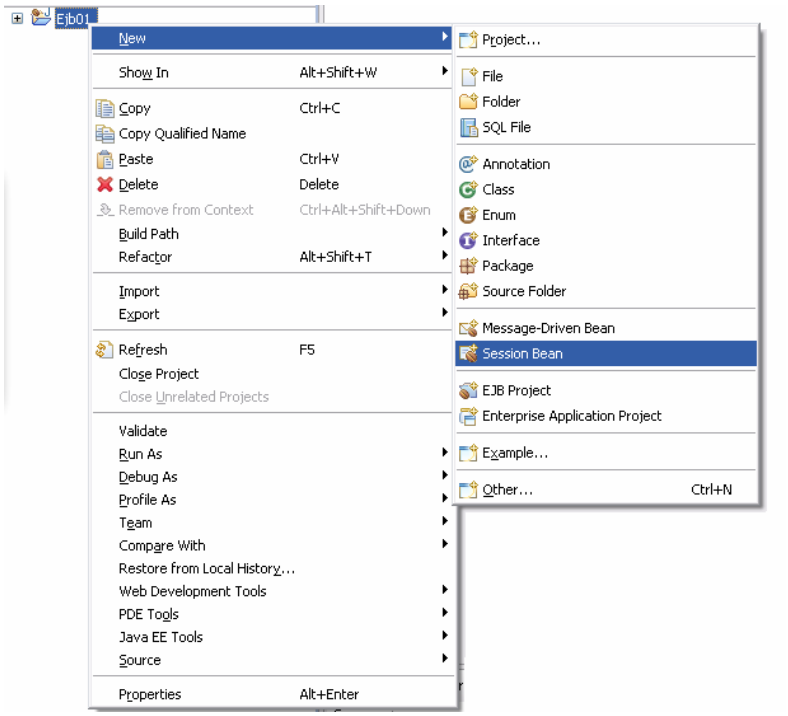
➔ Créer un EJB Project.

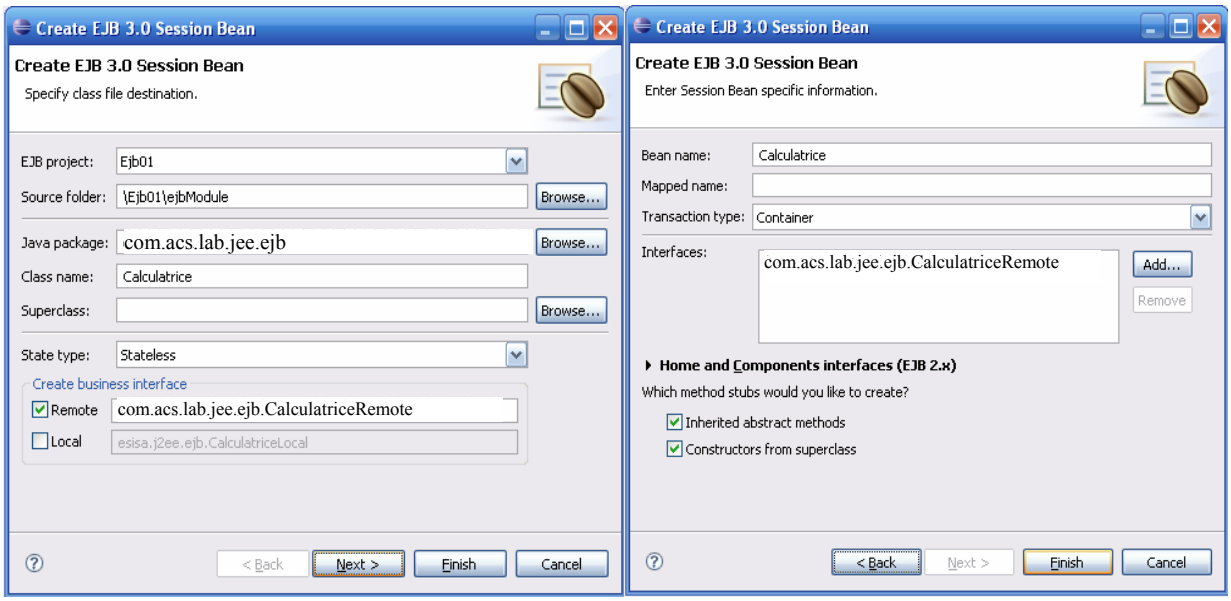




Il faut s'assurer que l'environnement d'exécution cible « Target Runtime » est bien le serveur « JBoss v4.2 », et que la version des EJB est la version 3.0.

Etape N° 2 : Création du Bean session :





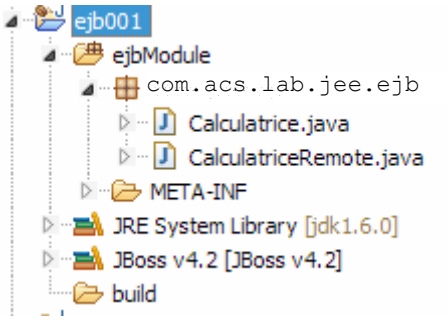
Il faut s'assurer de créer un package dans lequel sera inséré l'ejb session.

Dans le champ « State type » choisir : statless ou statfull,

Dans cet exemple on utilise « stateless », en sachant que l'implémentation de l'autre type est tout à fait similaire.

Préciser le type de « business interface »: Local ou Remote. Dans notre exemple, on utilise une interface distante (Remote).

On obtient :



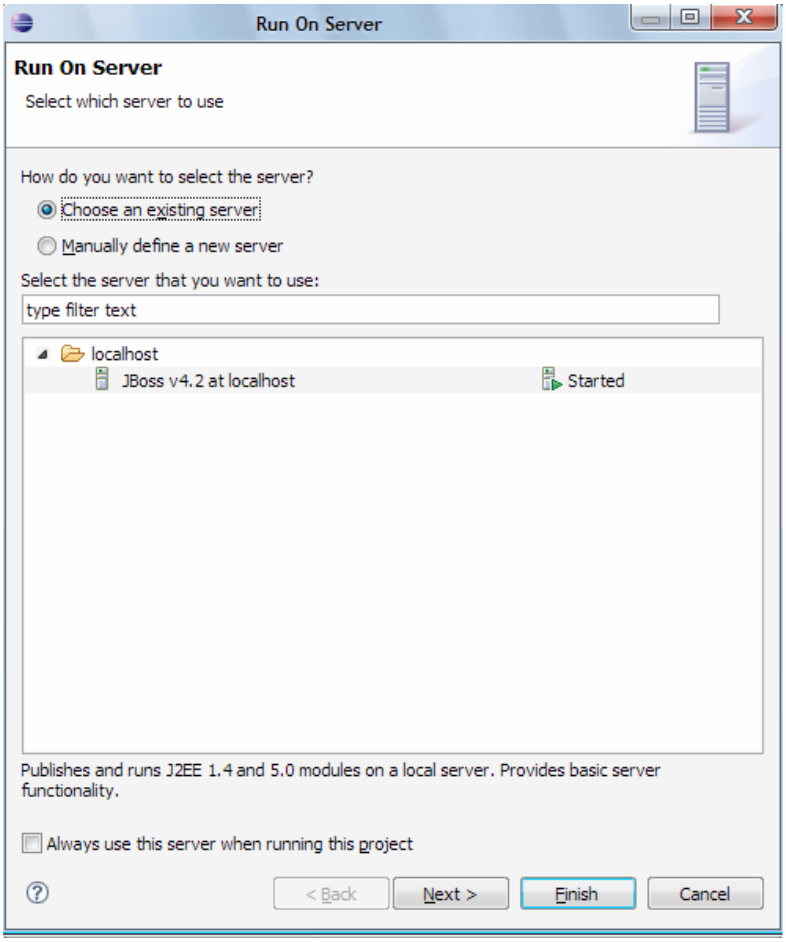
L'exemple est une simple calculatrice capable de faire la somme de réels.

```
package com.acs.lab.jee.ejb;
import javax.ejb.Remote;

@Remote
public interface CalculatriceRemote {
        public double sum (double x, double y);
}
```

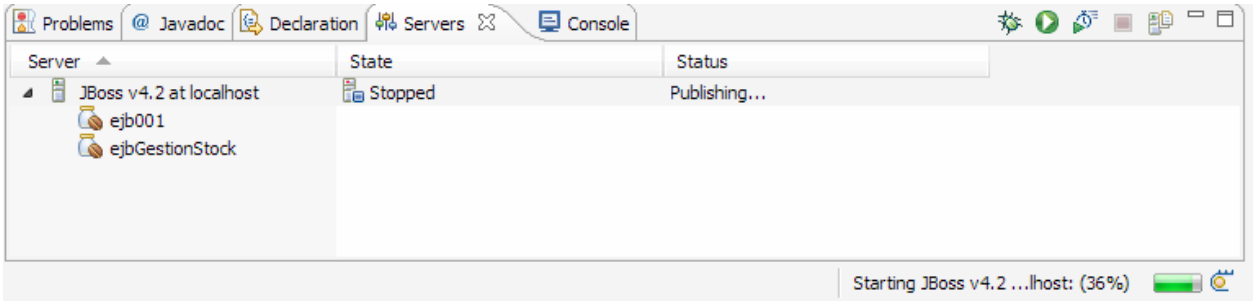
```
package com.acs.lab.jee.ejb;

import javax.ejb.Stateless;

@Stateless
public class Calculatrice implements CalculatriceRemote {

        public Calculatrice() {

        }

        public double sum(double x, double y) {
                return x+y;
        }
}
```

Etape N° 3 : Déploiement du Bean :

Pour déployer le Bean : click droit sur le projet, puis « Run As », et ensuite « Run on Server ».
On obtient la fenêtre suivante, on clique alors « Next » ensuite « Finish »

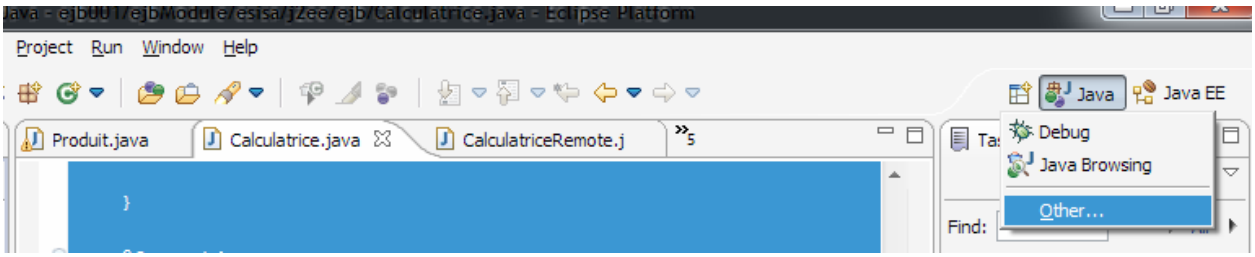


Nous pouvons remarquer que l' EJB s'ajoute comme suit:

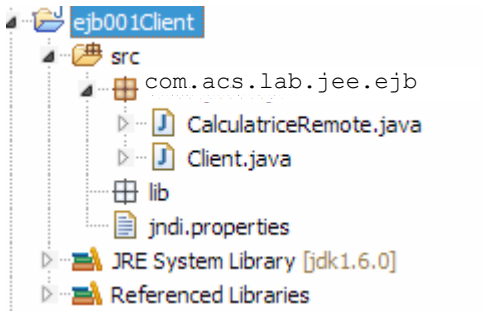


Etape N° 4 : Création du client :

Il s'agit d'une application Java ordinaire. Faut-il alors sélectionner la perspective « Java » :



Créer ensuite un projet Java « Le client ». Celui-ci doit contenir l'interface « Remote » (sans les annotations) :



A remarquer que le client peut être une entité de toute forme : application avec ou sans interface graphique, un bean, une Servet ou une JSP ou un autre EJB.

Le stub est une représentation locale de l'objet distant. Il implémente l'interface remote mais contient une connexion réseau pour accéder au Skelton de l'objet distant.

Le mode d'appel d'un EJB suit toujours la même logique :

- Obtenir une référence qui implémente l'interface home de l'EJB grâce à JNDI.
- Créer une instance qui implémente l'interface remote en utilisant la référence précédemment acquise.
- Appel des méthodes de l'EJB.

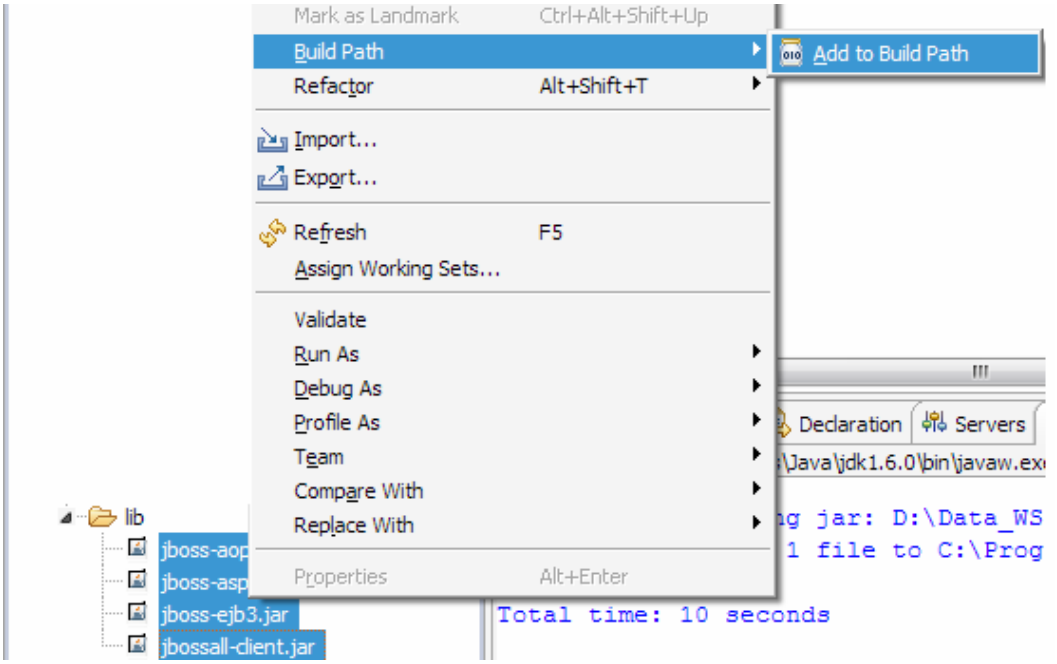
La classe du client :

```
package com.acs.lab.jee.ejb;

import javax.naming.Context;
import javax.naming.InitialContext;

public class Client {
  public static void main(String[] args) {
        try{
            Context  context = new InitialContext();
                CalculatriceRemote cal =
                  (CalculatriceRemote)context.lookup("Calculatrice/remote");
                System.out.println("somme = " + cal.sum(10, 20));
        } catch(Exception e){   e.printStackTrace();     }
  }
}
```

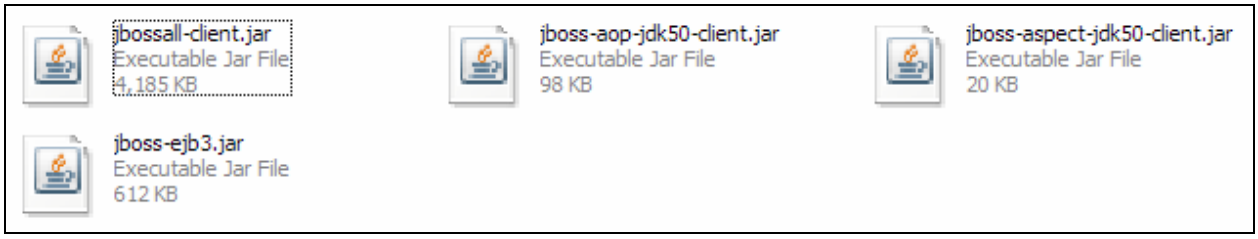
L'appel d'un EJB par un client repose sur les APIs JNDI. Pour que cela fonctionne, il faut définir un fichier « **jndi.properties** » qui indique au client les paramètres nécessaires pour localiser l'EJB :

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
```

Le client a aussi besoin d'un certain nombre de bibliothèques qu'il faut ajouter au « Buid Path » de l'application :



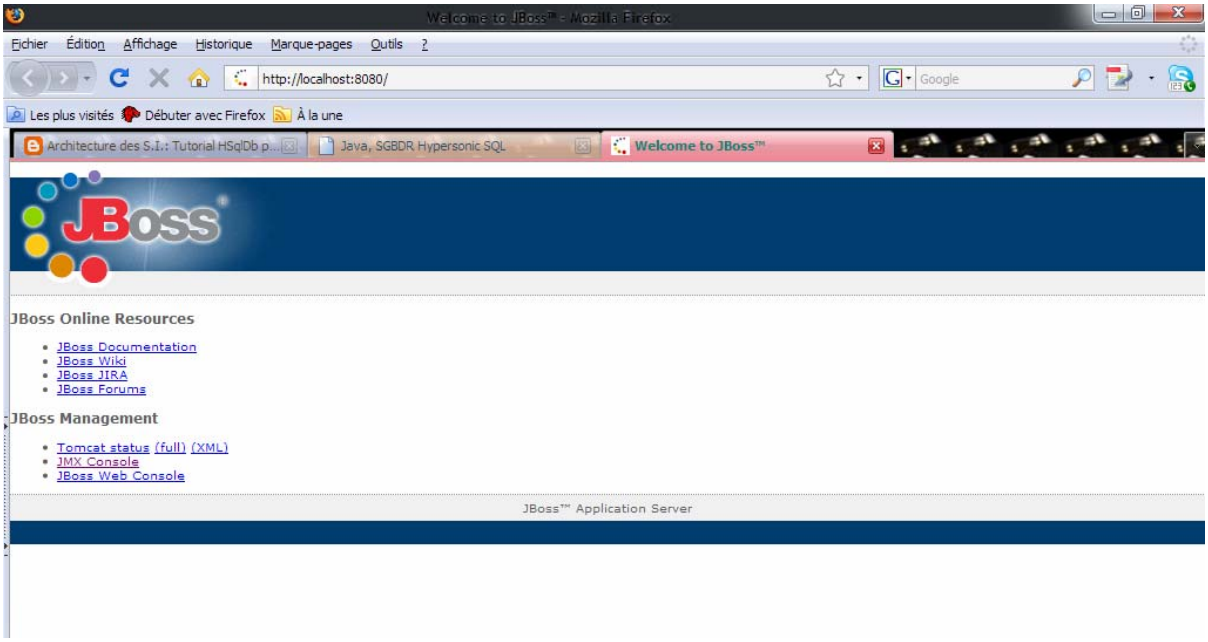
Les biliotèques sont les suivantes :



4.4 Création d'un Bean Entity :

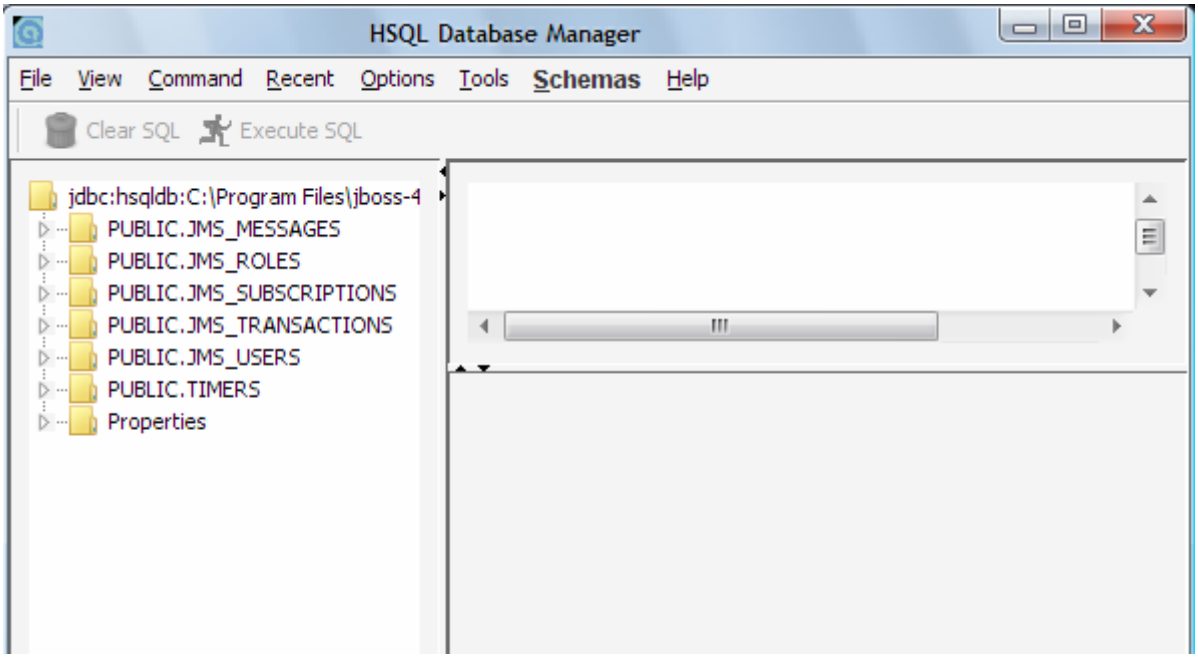
4.4.1 HSQLDB et JBOSS :

La livraison de JBOSS inclus la base de données « Hypersonic ». Le fonctionnement de cette base repose sur un jar : hsqldb.jar. Cette archive contient à la fois le code du serveur et le code d'un client d'administration. Hypersonic SQL est un petit système de gestion de bases de données écrit en Java, il est livré avec un administrateur de base se nommant : l'outil Database Manager, qu'on pourrait retrouver à l'aide de la JMX Console depuis la page d'accueil de JBoss :



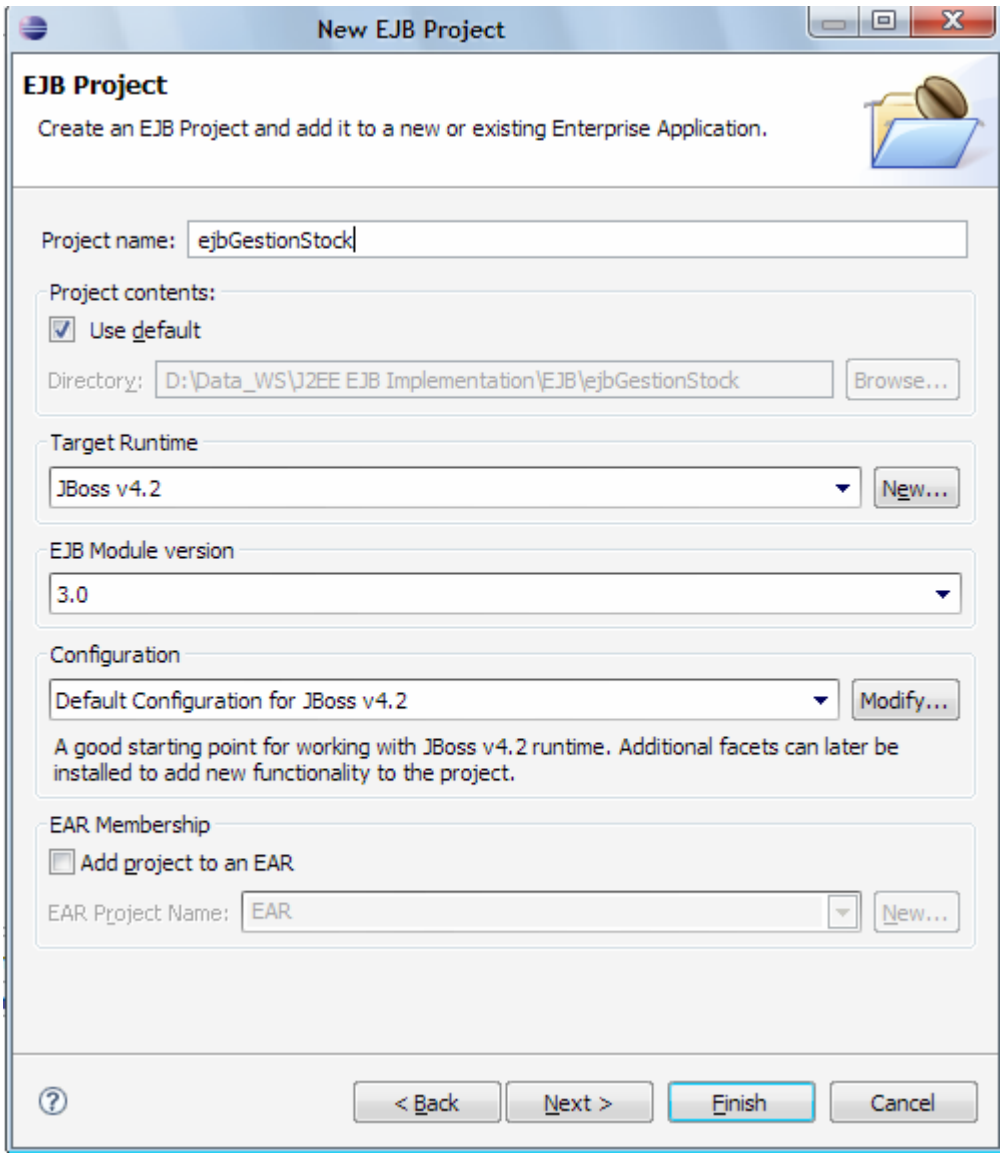
➔

Appuyez sur Invoke pour obtenir la fenêtre du manager :



4.4.2 Créer le bean « Entity »:

Tout d'abord nous allons créer un EJB Project comme avant, y ajouter une classe « Produit » par exemple qui va représenter le « Entity bean ».



```
package com.acs.lab.jee.ejb;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Produit implements Serializable {
	@Id
	private String ref;
	private String desig;
	private double PU;
	private int QS;

	public Produit() {
		super();
	}
	public Produit(String ref, String desig, double pu, int qs) {
		super();
		this.ref = ref;
		this.desig = desig;
		PU = pu;
		QS = qs;
	}
	…..GETTER ET SETTER
	public String toString(){
		return "("+ref+")" +desig+ ":"+PU+"==>"+QS;
	}
}
```

Remarquons :

- l'ajout de deux nouvelles annotations : @Entity,@Id.
- La première est pour indiquer que la classe est l'entité à persister, tandis que la seconde précise la clé primaire à prendre en compte lors de la création de la table.
- La classe doit être Serializable.

4.4.3 Réaliser un « Session Bean » :

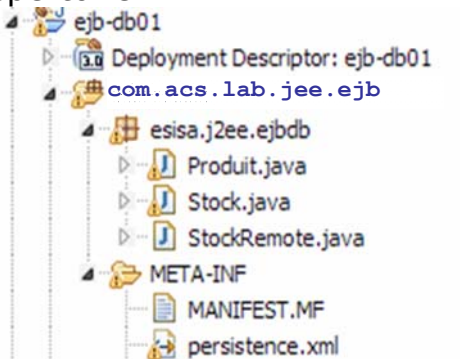
```
package com.acs.lab.jee.ejb;
import java.util.List;
import javax.ejb.Remote;

@Remote
public interface StockRemote {
	public void insert(Produit p);
	public Produit select(String ref);
	public List<Produit>selectAll();
}
```

```
package com.acs.lab.jee.ejb;

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

@Stateless
public class Stock implements StockRemote {
	@PersistenceContext
	private EntityManager manager;

	public Stock() {

	}

	public void insert(Produit p) {
		this.manager.persist(p);
	}

	public Produit select(String ref) {
		return this.manager.find(Produit.class,ref);
	}

	public List<Produit> selectAll() {
		String q = "SELECT * FROM Produit";
		Query query=manager.createQuery(q);
		List<Produit> list=(List<Produit>)query.getResultList();
		return  list;
	}
}
```

Il y'a eu l'ajout de l'annotation **@PersistenceContext**, en plus de l'objet **EntityManager** qui aura le rôle de gérer et faciliter la communication.

4.4.4 Fichier « persistence.xml » :

Fichier ajouté sous le répertoire META-INF.



Il s'agit d'un fichier de configuration permettant d'établir la connexion avec l'unité de persistance adéquate.

- Cas de la base de données Hypersonic :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>

	<persistence-unit name="EJB01">
		<jta-data-source>java:/DefaultDS</jta-data-source>
		<properties>
			<property name="hibernate.hbm2ddl.auto" value="update" />
		</properties>
	</persistence-unit>
</persistence>
```

Selon la balise <jta-data-source>, nous remarquons que la BD utilisée est celle par défaut dont le fichier de configuration se trouve dans le chemin :

C:\ProgramFiles\jboss4.0.4.GA\server\default\deploy**hsqldb-ds.xml**

Dont voici un extrait :

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- The Hypersonic embedded database JCA connection factory config -->

<!-- $Id: hsqldb-ds.xml,v 1.15.2.1 2006/01/10 18:11:03 dimitris Exp $ -->

<datasources>
  <local-tx-datasource>

    <!-- The jndi name of the Data Source, it is prefixed with java:/ -->
    <!-- Datasources are not available outside the virtual machine -->

    <jndi-name>DefaultDS</jndi-name>

</datasources>
```

- Cas de la base de données MySQL :

Dans ce cas, il faut ajouter le fichier MySQL Connector « mysql.jar » dans le répertoire « lib » suivant :

C:\Program Files\jboss-4.0.4.GA\server\default\lib

Le fichier persisatnce.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
      <persistence-unit name="MySQLbean">
            <jta-data-source>java:/MySqlDS</jta-data-source>
            <provider>org.hibernate.ejb.HibernatePersistence</provider>
            <class>com.acs.lab.jee.ejb.Produit</class>
            <properties>
              <property name="hibernate.dialect"
                        value="org.hibernate.dialect.MySQLDialect" />
              <property name="hibernate.hbm2ddl.auto" value="update" />
            </properties>
      </persistence-unit>
</persistence>
```

Remarquons l'utilisation du provider Hibernate qui est un Framework open source gérant la persistance des objets en base de données relationnelle.

La propriété hibernate.dialect va servir à configurer le dialecte SQL de la base de données.

Ce dialecte va servir à Hibernate pour optimiser certaines parties de l'exécution en utilisant les propriétés spécifiques à la base.

Cela se révèle très utile pour la génération de clé primaire par exemple.

Aussi nous voyons la property hibernate.hbm3dll.auto ayant pour value update et qui signifie une synchronisation de la classe avec la base de donnée.

Le fichier de configuration associé devrait être définie dans le répertoire suivant :

C:\ProgramFiles\jboss4.0.4.GA\server\default\deploy**mysql-ds.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
    <local-tx-datasource>
        <jndi-name>MySqlDS</jndi-name>
        <connection-url>jdbc:mysql://localhost:3306/stock
        </connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>root</user-name>
        <password></password>
    </local-tx-datasource>
</datasources>
```