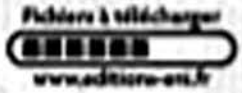
Java EE

Guide de développement d'applications web en Java



Jérôme LAFOSSE

Collection epsilon

Java EE

Guide de développement d'applications web en Java

Jérôme LAFOSSE

Résumé

Ce livre sur le développement d'applications web en Java s'adresse à tout **développeur** qui souhaite disposer de tous les détails des différentes étapes de réalisation d'une application web : **l'analyse**, la **modélisation**, le **codage**, la mise en **production**, les **tests** et la **maintenance**.
Le livre suit une **démarche progressive** et s'appuie sur une étude de cas d'un développement d'une **boutique de vente en ligne**. Il est découpé en sept chapitres progressifs qui peuvent également être étudiés de manière autonome.
Le premier chapitre présente le **langage Java**, explique les règles de nommage et les bonnes pratiques à adopter lors des développements de projets Java EE. Le chapitre 2 est consacré à la mise en place du serveur Java EE de référence, **Tomcat, sous Windows et Linux**. Les chapitres 3 et 4 explorent en détail les **servlets** et les **JavaServer Page** (JSP), en application avec l'étude de cas et le modèle **MVC**. Le chapitre 5 présente les **bases de données** en Java EE et détaille la mise en place de **JDBC** et des technologies associées. Le chapitre 6 concerne le développement Java EE à l'aide d'un framework. En accord avec les standards actuels en entreprise, **Struts** a été utilisé pour ce livre. Cependant les explications sont valables pour d'autres **frameworks Java** (description des outils proposés par un framework Java tant en terme de validation de données que d'approche **MVC II**).
Enfin, le dernier chapitre est consacré aux **techniques avancées** Java EE et permet de **déployer** un véritable projet sur un serveur en production à partir d'un nom de domaine.
Le code lié à l'étude de cas traitée dans le livre est en téléchargement sur cette page.
L'auteur propose à ses lecteurs un lieu d'échanges via le site www.gdawj.com qui apporte également un certain nombre d'éléments complémentaires (FAQ, outils, application déployée...).

L'auteur

Ingénieur en informatique et diplômé du CNAM, **Jérôme Lafosse** intervient comme consultant, concepteur et formateur sur les technologies Java. Spécialiste des technologies web, il travaille à promouvoir les outils et solutions Open Source pour le développement de projets Internet. Il enseigne également la plate-forme Java Entreprise Edition et la conception de projets Web en Licence et Master. Son expérience pédagogique s'allie à ses compétences techniques et offrent au lecteur un guide réellement opérationnel sur le développement d'applications web en Java.

Avant-propos

La réalisation de sites Web passe par différentes étapes : l'analyse, la modélisation, le codage, la mise en production, les tests et la maintenance. Toutes ces phases de conception sont longues, complexes et doivent être maîtrisées en détail pour mener à bien les projets Internet. Pour ce type de projet, différents langages de programmation sont utilisés comme PHP, Ruby, Perl, .NET ou Java. Java est reconnu actuellement comme l'un des meilleurs langages de programmation objet pour la réalisation de projets Internet complexes avec son API spécifique, Java EE.

Ce guide détaillé suit une démarche progressive et vous aidera à créer des applications Web complexes et fonctionnelles. Tous les concepts de la création d'un projet professionnel sont abordés dans ce livre, de la prise en main du langage, à l'installation de l'environnement, à la configuration d'un serveur Web Java jusqu'à la création et la mise en production d'un projet. Cet ouvrage permet de percevoir le développement d'applications Web en Java dans sa globalité.

Mon objectif est de fournir un guide complet de développement d'applications Web en Java sur les deux principaux environnements de développement que sont Windows et Linux, sans faire l'impasse sur une partie du cycle de développement. Il s'agit d'explications et de conseils concrets, illustrés par une étude de cas réaliste de boutique de vente en ligne.

Les lecteurs sont d'ailleurs invités à échanger via le site www.gdawj.com, véritable complément du livre puisqu'il propose un exemple de l'application déployée, un forum de questions, des outils complémentaires pour le développement d'applications web,etc.

Organisation du guide

Le guide est divisé en sept chapitres spécifiques et autonomes :

- Le chapitre 1 (Objectifs et spécifications de Java EE) présente le langage Java, les règles de nommage ainsi que l'installation de l'environnement de développement.
- Le chapitre 2 (Le serveur d'applications Apache-Tomcat) est consacré à la mise en place du serveur Java de référence, Tomcat.
- Le chapitre 3 (Les JavaServer Page) aborde la programmation de Servlets avec les classes, objets et méthodes.
- Le chapitre 4 (Les Servlets) explore en détail le développement de pages Web au travers des pages JSP.
- Le chapitre 5 (Java et les bases de données) présente les bases de données en Java ainsi que les solutions techniques et les outils adaptés à la persistance des données.
- Le chapitre 6 (Framework Java EE) est consacré à l'étude d'un framework de développement Java nommé Struts.
- Le chapitre 7 (Techniques avancées) est dédié aux techniques avancées en développement Java EE.

À qui s’adresse ce guide ?

Que vous ayez peu de connaissances en développement Web ou que vous soyez un expert Java EE, ce guide a pour objectif de vous présenter en détail et de façon exhaustive, toutes les étapes de réalisation d’applications Internet à partir d’un projet concret mais facilement portable. Au cours de mes études et de mon parcours professionnel, j’ai étudié les langages du Web, l’installation d’un serveur Java, la mise en œuvre d’un framework, les bases de données pour Internet, les Servlets/JSP et le développement de projets mais aucun ouvrage ne regroupait tous ces aspects de conception et certaines parties importantes n’étaient pas détaillées, comme s’il manquait quelques clés essentielles qui ne s’acquièrent qu’avec l’expérience. La démarche pédagogique de ce guide est de ne pas faire l’impasse sur des points essentiels d’un projet Web comme le déploiement sur un serveur en production avec un nom de domaine, la mise en place d’un pool de connexion sur une base de données quelconque ou encore la configuration complète d’un serveur Java. L’ouvrage utilise pour cela une mise en page adaptée afin de mettre en valeur les concepts essentiels à partir de schémas, graphiques, etc.

Les conventions

1. Les conventions du guide, de Java, de codage et les règles de nommage

Dans le cycle de vie d'un produit logiciel, la phase de maintenance représente la majeure partie du temps (environ 80 %). De même, un logiciel est rarement développé par une seule et même personne. C'est une équipe entière qui réalise le projet de développement avec tous les avantages mais aussi toutes les contraintes que cela implique.

La réussite d'un projet dépend beaucoup de l'homogénéité dans le codage. Cette étape essentielle passe par la mise en œuvre de conventions strictes respectées par toute l'équipe impliquée dans le projet. La plupart des outils de développement (IDE) proposent des fonctionnalités pour permettre cette homogénéité mais il existe des règles que seuls les développeurs doivent appliquer. Le fait de vouloir à tout prix avoir un code esthétique et de faible complexité peut être un frein aux performances. À l'inverse, la course à la performance peut découler sur un code peu lisible. Il revient donc aux développeurs de trouver le bon équilibre entre les conventions et les contraintes du projet (projet axé sur la performance, projet OpenSource...).

2. Les conventions du guide

Tout au long de ce guide seront utilisées des conventions pour la rédaction des explications, les parties de code, les rappels et les résumés. Les explications seront rédigées sous format textuel et graphique avec un maximum de clarté. Les parties de code seront bien différenciées par la police de caractères et seront encadrées. Les schémas viendront appuyer une explication et permettront d'expliquer sous forme graphique une convention, un choix, une spécification ou un exemple.

3. Les conventions de codage

Des conventions de codage sont utilisées tout au long de ce document. Par défaut, les conventions de codage pour la plupart des projets OpenSource suivent ces instructions. Par exemple, si la parenthèse *{* est après la condition *if*, le code n'est pas correct.

Tous les blocs de code doivent commencer par une nouvelle ligne.

```
public class MaClasse
{
  public void maMethode()
  {
      if (xxx)
      {
      }
  }
}
```

Chaque conditionnelle contient les parenthèses ouvrantes et fermantes.

```
//Correct
if (expression)
{
    //le code
}
//Incorrect
if (expression)
//le code
```

Une indentation se trouve après chaque instruction. Les noms des fonctions et des paramètres, ne doivent pas avoir de préfixe, commencent par une minuscule et chaque partie de mot est en majuscule.

```
public class MaClasse
{
    private String maChaine;
    public void maMethode(String monParametre)
    {

    }
```

```
}
```

La version de l'application (du code) est précisée dans chaque fichier d'extension *.java*.

```
@version 2.8
```

Le nom de l'auteur est précisé.

```
@author jeromelafosse
```

Chaque importation de paquetage (paquet) Java est pleinement qualifiée.

```
//Correct
import java.util.Date;
import java.net.HttpURLConnection;

//Incorrect
import java.util.*;
import java.net.*;
```

4. Les conventions Java

Pour utiliser efficacement le langage de programmation Java, il existe plusieurs conventions à connaître et à appliquer. Les instructions Java se terminent par un point-virgule. Les instructions Java utilisent des accolades { } pour indiquer le début et la fin du corps. Un corps peut contenir plusieurs instructions.

La présentation avec les accolades alignées sur la même colonne que le premier caractère de l'instruction Java sera utilisée. Cette présentation ajoute des lignes de code, mais elle est plus facile à lire. Il est important de mettre en retrait (espace ou tabulation) les instructions Java qui comportent un corps. Il est également nécessaire de toujours utiliser la même mise en retrait du code.

Java recourt comme tout langage à plusieurs mots-clés, c'est-à-dire des mots réservés exclusivement au langage. Il ne sera donc pas possible d'utiliser ces mots-clés comme noms ou valeurs de variables.

Voici une liste non exhaustive de ces mots-clés : *abstract*, *boolean*, *break*, *case*, *catch*, *char*, *class*, *continue*, *do*, *double*, *else*, *extends*, *false*, *final*, *finally*, *float*, *for*, *if*, *import*, *instanceof*, *int*, *interface*, *new*, *null*, *package*, *private*, *protected*, *public*, *return*, *static*, *super*, *switch*, *this*, *true*, *try*, *void*, *while*...

Le code doit être succinct, cela facilite la maintenance et la lisibilité de l'ensemble. Il vaut mieux découper parfois des méthodes et ajouter des commentaires. Les recommandations doivent être appliquées sur l'intégralité du projet et non avec parcimonie. Si les règles sont plus ou moins appliquées, si le code change d'un fichier à l'autre, la compréhension sera difficile. L'application uniforme des règles est un gage de maintenabilité.

Fichiers

Les noms des fichiers sources portent l'extension *.java* et le code (*bytecode*) généré porte l'extension *.class*. Les fichiers de propriétés portent l'extension *.properties* (langues, traductions...) et les fichiers de configuration l'extension *.xml*. Le fichier de construction du projet porte le nom *build.xml* (Ant) et le fichier de description du projet est appelé *README*.

Un projet de développement utilise plusieurs répertoires. Il est généralement composé du répertoire */src* qui contient les sources, du répertoire */build* qui contient les classes compilées, */docs* qui contient la documentation du projet et */log* qui contient les traces d'exécutions et d'erreurs du projet.

Souvent, pour un projet minime, seuls les répertoires */src*, */build* et */docs* sont utilisés.

Les classes doivent être regroupées en packages (paquetages ou paquets).

Sources

Il est recommandé de ne pas dépasser 2000 lignes de code par fichier. Si tel est le cas, il est important d'optimiser le code, de vérifier s'il n'existe pas de redondance et de découper les fonctionnalités en plusieurs classes (boîte à outils par exemple).

Formatage

Il faut configurer l'éditeur pour que la tabulation écrive huit caractères espace (configuration par défaut). L'entrée dans un bloc impose l'ajout d'une indentation. Des blocs de même niveau doivent débuter sur la même colonne, c'est-à-dire avoir la même indentation. Les lignes blanches doivent être utilisées pour séparer des portions de code et les méthodes. Les espaces peuvent être utilisés en quantité mais sous certaines conditions. Le caractère espace est

proscrit avant les points-virgules, avant les crochets des tableaux et entre une variable et les opérateurs de pré/post incrément. Par contre, l'espace est autorisé après les virgules, avant et après les accolades, avant et après chaque opérateur, après les mots réservés du langage et entre le nom d'une méthode et la parenthèse de ses paramètres.

```
//Correct
maMethode (a, b, c, d);
for (i = 0; i < 100; i++) {
++count;
(MaClasse)maVariable.get(i);

//Incorrect
MaMethode (a,b,c,d);
++ count;
(MaClasse) maVariable.get(i);
```

Nommage

Les noms utilisés doivent être explicites, c'est-à-dire que le nom doit expliquer le contenu de l'objet, le rôle de la méthode... Les acronymes qui apparaissent dans les noms doivent être passés en minuscules (sauf la première lettre du premier mot). Il est conseillé de mettre les identifiants en langue anglaise pour des projets internationaux.

Package/Paquetage

Les noms des paquetages doivent être en minuscules. Ces noms doivent être pleinement qualifiés (comme une URL) et reprennent le nom du projet, l'URL du site... En général, la technique consiste à retourner l'URL du projet. Exemple : monprojet.com devient : com.monprojet.monpaquet

```
//Correct
package com.monprojet.monpaquet;
//Incorrect
package Com.MonProjet.MonPaquet;
```

Classes et interfaces

Les noms des classes doivent être en minuscules, hormis les initiales des mots qui les composent.

```
//Correct
class MaClasseFavorite;

//Incorrect
class maclassefavorite;
class maClassefavorite;
```

Méthodes

Les noms des méthodes doivent être en minuscules hormis les initiales des mots qui composent les mots (sauf la première lettre).

```
//Correct
public void maMethodeFavorite() {
//Incorrect
public void mamethodefavorite() {
```

Les accesseurs directs (*getters* et *setters*) des attributs d'une classe doivent être préfixés d'un *get* pour la lecture de l'attribut et d'un *set* pour l'écriture. Le préfixe *is* doit être utilisé pour les méthodes qui retournent un booléen.

```
//Correct
public int getNiveau() {
public void setNiveau(int niveau) {
public boolean isVisible() {
//Incorrect
public int recupererLeNiveau() {
public void ecrireLeNiveau(int niveau) {
public boolean estIlVisible() {
```

Nous pouvons également utiliser d'autres mots pour les recherches, les suppressions, les ajouts, la fermeture de connexions... (*find*, *delete*, *add*, *close*...).

Attributs, variables et paramètres

Les attributs des classes , les variables ainsi que les paramètres des méthodes doivent être en minuscules hormis les initiales des mots qui les composent (sauf le premier). Les variables de boucle doivent porter une seule lettre : *i*, *j*, *k*... Les signes dollars (*$*) et soulignement (_) sont proscrits.

```
//Correct
Voiture prochaineVoiture = voitures.get(this.id + 1);
float laTaille = 145.5;
//Incorrect
Voiture a = voitures.get(this.id + 1);
float la_Taille = 145.5;
```

Les collections doivent être nommées au pluriel.

```
//Correct
Vector comptes;
Collection banques;
Object[] mesObjets;
//Incorrect
Vector compte;
Collection banque;
Object[] monObjet;
```

Constantes

Les noms des constantes doivent être écrits entièrement en majuscules. Le séparateur de mot est le caractère de soulignement (underscore).

```
//Correct
static final int LOG_CONSOLE = 1;
//Incorrect
static final int LOGCONSOLE = 1;
static final int console_Log = 1;
static final int Console_LOG = 1;
```

Commentaires

Les commentaires sont essentiels dans un code source. Ils permettent de documenter le projet à l'intérieur même du code source en vue de la génération de la documentation via l'outil JavaDoc. Il existe en Java deux types de commentaires :

- les commentaires mono-ligne qui permettent de désactiver tout ce qui apparaît sur la même ligne // ;
- les commentaires multilignes qui permettent de désactiver tout le code qui se trouve entre les deux délimiteurs /* */.

Il est important de réserver les commentaires multilignes aux blocs utiles à la JavaDoc et à l'inactivation de portions de code. Les commentaires mono-ligne permettent de commenter le reste, à savoir, toute information de documentation interne aux lignes de code.

```
/*
* La classe MaClasse permet telles fonctionnalités…
*/
public class MaClasse() {
// Recuperer un objet de la collection
monFichier = (Fichier)fichiers.get((int)item.getIdFichier());
```

Déclarations

Les variables doivent être déclarées ligne par ligne. L'initialisation doit se faire lors de la déclaration lorsque cela est possible. Les variables doivent être déclarées au plus tôt dans un bloc de code. Les noms des méthodes sont accolés à la parenthèse ouvrante listant leurs paramètres. Aucun espace ne doit y être inséré.

```
//Correct
int niveau = 10;
void maMethode() {
```

```
//Incorrect
int niveau;
niveau = 10;
void maMethode () {
```

Ordre

L'ordre de déclaration des entités du code source doit être le suivant (qui est plus ou moins naturel) :

- Les attributs de la classe (1-> statiques, 2->publiques, 3->protégés, 4-> privés).
- Les méthodes de la classe (1->statiques, 2->publiques, 3->protégées, 4->privées).

Instructions

Une ligne de code ne peut contenir qu'une seule instruction.

```
//Correct
count++;
i--;
println("Bonjour");
//Incorrect
count++; i--; println("Bonjour");
```

Définitions de J2EE/Java EE

De nombreuses possibilités existent pour réaliser des applications Internet depuis plusieurs années. Des langages ont été créés, des architectures et des environnements de travail ont été conçus pour répondre aux besoins et faciliter la tâche des développeurs. Sun (le concepteur de Java) a donc mis en place un ensemble de technologies pour réaliser des applications Web. Ces technologies sont regroupées sous le nom J2EE (*Java 2 Entreprise Edition*), désormais Java EE.

Depuis la version 5, le chiffre 2 a disparu pour faciliter la compréhension de la version et ne pas mélanger le chiffre 2 avec le numéro de version.

La plate-forme Java EE s'appuie entièrement sur le langage Java. Java EE est donc une norme, qui permet à des développeurs, entreprises et SSII de développer leur propre application qui implémente en totalité ou partiellement les spécifications de SUN. En simplifiant, il est possible de représenter Java EE comme un ensemble de spécifications d'API, une architecture, une méthode de packaging et de déploiement d'applications et la gestion d'applications déployées sur un serveur compatible Java.

Une API est un ensemble de librairies ou bibliothèques de fonctions destinées à être utilisées par les programmeurs dans leurs applications.

Java Entreprise Edition est destiné aux gros (très gros) systèmes d'entreprises. Les librairies utilisées fonctionnent difficilement sur un simple PC et requièrent une puissance beaucoup plus importante (notamment au niveau de la mémoire).

Java Entreprise Edition est apparue à la fin des années 90. Cette évolution apporte au langage Java une plate-forme logicielle robuste et complète pour le développement. La plate-forme Java EE a souvent été remise en cause, mal utilisée et mal comprise. Des outils OpenSource sont venus la concurrencer. Ces remarques et la concurrence ont permis à Sun d'améliorer son produit et d'éditer des versions de plus en plus abouties. Java EE ne remplace en aucun cas J2SE. Au contraire, J2SE est la base de Java EE qui est plus complet et qui est axé sur le Web. La plate-forme J2SE offre des outils de développement d'applications client/serveur, applications graphiques fenêtrées et Applets.

La plate-forme J2SE est composée des éléments suivants :

- **La machine virtuelle Java (JVM)** : permet d'exécuter des applications Java. Elle constitue une passerelle et permet une portabilité entre les architectures (Windows, Linux, Mac...).
- **La bibliothèque de classes Java** : un ensemble de composants logiciels prêt à l'emploi.
- **Les outils de développement** : le compilateur javac , un interpréteur Java nommé java, le générateur de documentation javadoc, la console de supervisation Jconsole... La plate-forme Java EE est une extension de la plate-forme J2SE. Elle permet un développement d'applications qui vont s'exécuter sur un serveur d'applications. Les applications seront utilisées par des clients légers (comme des navigateurs Web) ou bien des applications lourdes (IHM). La dernière version stable de Java EE est la version Java EE 5.0 et fonctionne avec le JDK 5.0 et 6.0.

1. Pourquoi choisir Java EE

Il existe actuellement beaucoup d'autres plates-formes de développement qui sont basées sur d'autres langages (C#, PHP5, .NET...). Les principaux avantages d'utiliser Java EE (et donc Java) sont la portabilité, l'indépendance, la sécurité et la multitude de librairies proposées.

Le développement d'applications d'entreprise nécessite la mise en œuvre d'une infrastructure importante. Beaucoup de fonctionnalités sont utilisées et développées, le but étant de produire des applications sûres, robustes et faciles à maintenir. Certains services sont d'ailleurs récurrents comme : l'accès aux bases de données, l'envoi de mails, les transactions, la gestion de fichiers, la gestion d'images, le téléchargement, le chargement ou upload, la supervision du système...

C'est pour cela que l'architecture Java EE est intéressante car tous les éléments fondamentaux sont déjà en place. Pas besoin de concevoir une architecture , des librairies et des outils spécialement adaptés. Cela nécessiterait un temps et un investissement considérables.

Enfin, la plate-forme Java EE est basée sur des spécifications, ce qui signifie que les projets sont portables sur n'importe quel serveur d'applications conforme (Tomcat, JBoss, WebSphere...) à ces spécifications. Cette implémentation est gratuite et permet de bénéficier de la totalité de l'API sans investissement. La plate-forme Java EE est la plus riche des plates-formes Java et offre un environnement standard de développement et d'exécution d'applications d'entreprise multi-tiers.

Le fait que Java EE soit standardisé a contribué à son adoption par de très nombreux éditeurs de logiciels/outils informatique. Ces éditeurs associés à Sun Microsystems font partie du **JCP** (*Java Community Process*). Le Java Community Process regroupe les entreprises suivantes : Sun, IBM, Oracle, Borland, Nokia, Sony, la fondation Apache, ObjectWeb... L'objectif du JCP est de définir les spécifications des technologies basées sur Java.

Chaque demande de modification est appelée une **JSR** (*Java Specification Request*).

2. L'API Java EE (JDBC, Servlets, JSP)

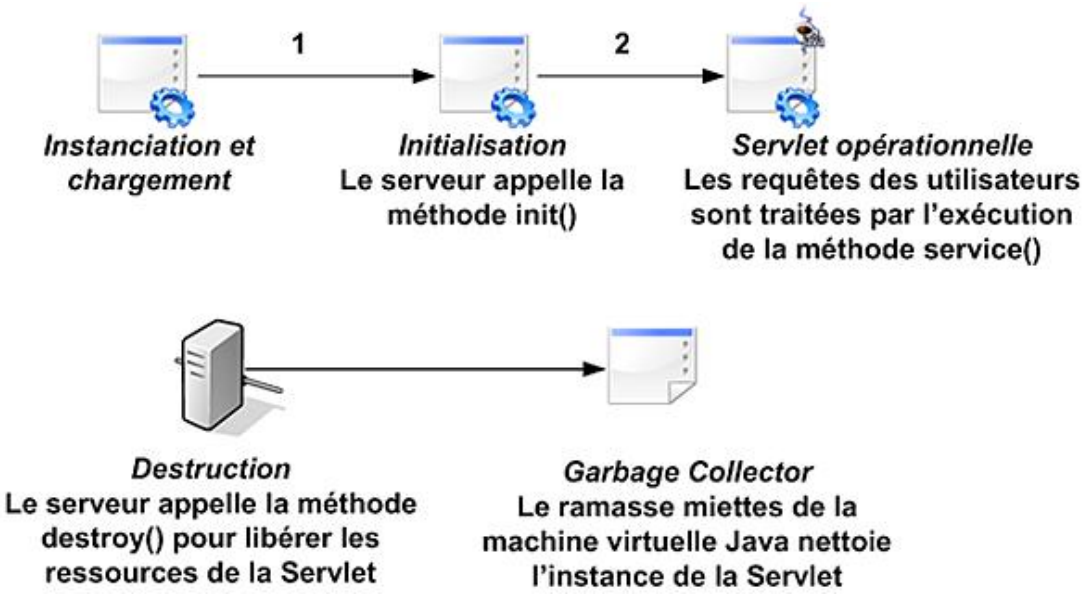
La plate-forme Java EE est composée de plusieurs API (ensemble de libraires et services). Java EE fait intervenir trois types de composants logiciels (Servlets, JSP, EJB).

a. Les Servlets

L'API Servlet fournit les éléments nécessaires à la conception de composants Web dynamiques avec le langage Java. Les Servlets sont des composants logiciels entièrement écrits en Java. Les Servlets effectuent des traitements côté serveur en réponse aux requêtes des clients distants. Une Servlet est chargée en mémoire lors de son premier appel. De même, il n'existe qu'une seule instance d'une Servlet en mémoire, le serveur utilise alors un thread global pour traiter les demandes émises par les clients.

Cycle de vie d'une Servlet

Une Servlet est une classe Java. Cette classe doit être chargée puis interprétée par une machine virtuelle Java (celle du serveur d'applications). La Servlet est alors prête à recevoir des requêtes et à renvoyer des réponses. Lorsque l'application ou le serveur s'arrête, la Servlet est détruite, puis son instance est nettoyée par le ramasse-miettes de la machine virtuelle.



Les Servlets permettent de développer des pages dynamiques, dont le contenu est créé à la volée sur demande. C'est le cas par exemple, lorsqu'un client souhaite obtenir la liste des articles d'une boutique pour une plage de prix. Les pages HTML sont alors générées dynamiquement en fonction de critères spécifiques (prix, dates, recherches...).

Une Servlet est un composant Java qui implémente l'interface *javax.servlet.Servlet*. Cette interface permet de gérer les requêtes du client, dirigées vers la Servlet en question. Le serveur reçoit une demande adressée à une Servlet sous la forme d'une requête HTTP. Il transmet alors la requête à la Servlet concernée par le traitement puis renvoie la réponse fournie par celle-ci au client. La Servlet est gérée par le conteneur de Servlets Java EE. Lorsque le serveur Web reçoit la requête du client, il charge la Servlet (si elle n'est pas encore chargée) et invoque l'interface *javax.servlet.Servlet* afin de satisfaire la requête.

Ce type de programmation est très proche du développement CGI mais offre les outils pour gérer les cookies, sessions, accès aux bases de données et autres avec une excellente portabilité.

Exemple de Servlet simple :

```
public class PremiereServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, java.io.IOException
    {
    }

    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, java.io.IOException
    {
        doGet(request, response);
    }
}
```

b. Les JSP (Java Server Page)

L'API JSP permet de développer des pages Web dynamiques rapidement à partir d'un squelette HTML et de données incluses directement dans chaque page. Les JSP permettent d'insérer des bouts de code Java (scriptlets) directement dans le code HTML.

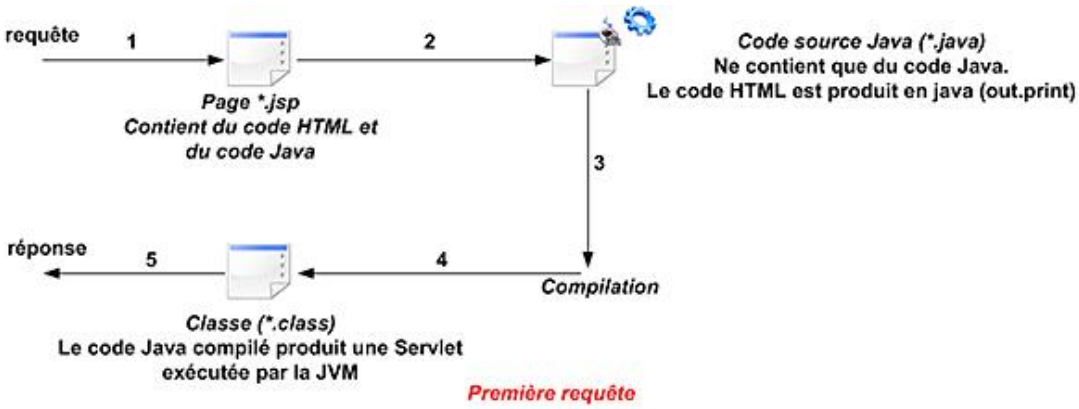
Du point de vue de la structure, une JSP est très proche d'une page PHP ou bien ASP. Une page JSP est un fichier qui porte l'extension *.jsp* ou *.jspf* (pour les fragments de code).

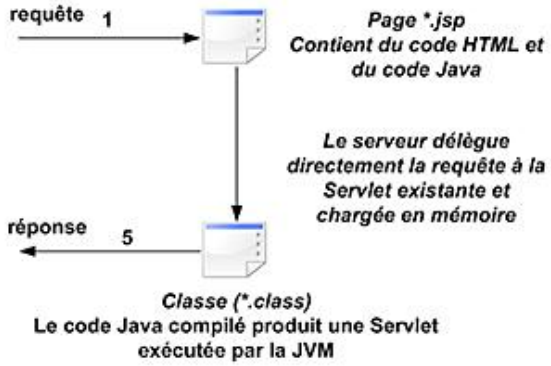
Lors de la création de Servlets, le but est de construire des composants capables de produire un service (essentiellement du code HTML). Toutefois, ce principe est parfois complexe pour des personnes qui ne sont pas habituées à la programmation objet et au code 100% Java. C'est pour ces raisons que les développeurs de chez SUN ont inventé JSP.

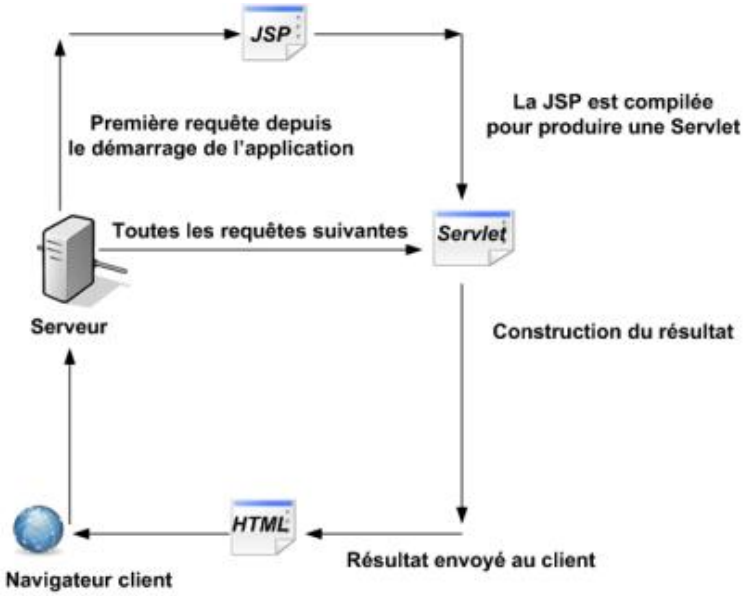
La page JSP est transformée en classe Java puis compilée en Servlet par le serveur d'applications. Ce traitement est réalisé par le serveur d'applications au premier appel de la page et à chaque fois que cette page est modifiée par un programmeur.

C'est cette étape qui nécessite un serveur d'applications Java EE, un compilateur Java et qui par conséquent nécessite pour la majorité l'installation de Java avec un JDK plutôt qu'un JRE.

Cycle de vie d'une JSP







Le résultat de la compilation (essentiellement du code HTML) est renvoyé au client. Cette technologie est simple, légère et rapide. Les développeurs de pages JSP peuvent ainsi mélanger du contenu statique et du contenu dynamique.

```
<%
System.out.println("Ma première Servlet");
out.println("<h1>Ma première Servlet<h1>");
%>
```

Il est donc possible d'avoir des équipes de développement séparées avec une personne spécialiste de HTML/XHTML et du design et un programmeur Java qui réalise les scriptlets. Les JSP sont exécutées sous la forme de Servlets, elles disposent donc des mêmes fonctionnalités que celles-ci et peuvent ainsi manipuler les sessions, les bases de données, les mails...

c. LES EJB (Entreprise Java Bean)

Les EJB sont des composants métier distribués, c'est-à-dire qu'ils sont invocables par le réseau. Un composant EJB est une classe qui possède des attributs et méthodes pour mettre en application la logique métier. L'API EJB fournit un ensemble de services (persistance, transaction...) de gestion de composants. Il existe plusieurs (trois) types d'EJB : les beans sessions, les beans entités et les beans contrôlés par message.

- EJB session : il permet de maintenir des informations sur les clients et les traitements qu'ils réalisent.
- EJB entité : c'est un composant persistant, son état est sauvegardé dans une base de données.
- EJB piloté par message : les EJB message sont semblables aux EJB session mais sont invoqués différemment (par le biais de Java Message Service).

La mise en place d'EJB nécessite l'utilisation d'un serveur d'applications capable de gérer ces EJB. Actuellement, les serveurs GlassFish, JBoss et Jonas existent dans le domaine du libre. Le serveur Tomcat ne permet pas d'utiliser les EJB.

Il existe ensuite au sein de Java EE, la plate-forme de Services avec JDBC, JNI, JavaMail, JTA, RMI, JAAS et XML.

JDBC (Java DataBase Connectivity)

L'API JDBC permet de faciliter l'obtention de connexions JDBC vers des sources de données (essentiellement des bases de données, mais également annuaire...). L'API fournit les librairies pour se connecter aux bases de données et pour la gestion des transactions.

JNDI (Java Naming and Directory Interface)

L'API JNDI permet d'accéder à des services de nommage ou d'annuaire (LDAP par exemple). Cette API est par exemple utilisée pour se connecter à une source de données pour des accès à la base de données ou la gestion des accès (associée aux Realms). JNDI permet d'implémenter un service de nommage. L'ensemble des ressources que le serveur d'applications met à disposition via ces API de services, doit être enregistré avec un nom logique unique, permettant aux applications de rechercher cette ressource dans le serveur.

JavaMail

L'API JavaMail fournit des fonctionnalités de gestion de courrier électronique (transfert, type de contenu, pièces jointes...). JavaMail permet la création et l'envoi de messages électroniques via Java. Cette API permet de manipuler les protocoles de messagerie Internet comme POP, IMAP, SMTP. JavaMail n'est pas un serveur de courrier mais plutôt un outil qui permet d'interagir avec ce type de serveur.

JPA (Java Persistance API)

Les entités Beans ont été développées pour le modèle de persistance en Java EE. Ce modèle de composants avait de nombreux détracteurs. Pour apporter des solutions à ce problème, de nouvelles spécifications, des outils de mapping objet/relationnel comme TopLink et Hibernate ont été développés. Java EE apporte donc un nouveau modèle de persistance nommé JPA. JPA s'appuie sur JDBC pour communiquer avec la base de données mais permet d'éviter de manipuler directement les fonctionnalités de JDBC et le langage SQL.

3. Les autres API

Parmi les autres API Java EE, il faut citer :

JMS (*Java Message Service*) permet d'accéder à un service de messages pour une gestion asynchrone des composants. Le composant appelant poste un message (en arrière-plan) à destination d'une file d'attente de messages hébergés par le serveur d'applications puis continue son traitement sans attendre.

RMI (*Remote Method Invocation*) permet de concevoir des applications distribuées en Java. RMI permet l'appel de fonctionnalités à distance par le biais de la communication réseau.

JTA (*Java Transaction API*) permet de mettre en place une gestion des transactions dans des applications distribuées (commit, rollback...). Le principe des transactions est de considérer un ensemble d'opérations comme une seule. Ce type de service est obligatoire pour des traitements bancaire. Par exemple, une application bancaire qui permet de réaliser des virements entre deux comptes va d'abord débiter le premier compte et ensuite créditer le second compte. Si le débit puis le crédit aboutissent sans problème, alors la transaction est validée. JDBC permet de gérer les transactions sur une base de données locale mais si les données sont réparties, il faudra alors utiliser les transactions JTA. JTA permet en effet de gérer les transactions distribuées qui font intervenir différentes bases de données.

JCA (*J2EE Connector Architecture*) : ce connecteur permet à Java EE d'utiliser des gros systèmes tels que les mainframes.

JAAS (*Java Authentication and Autorisation Service*) est un mécanisme de sécurité géré par le serveur d'applications.

XML n'est pas véritablement une API Java mais il est très utilisé pour la mise en place et la configuration des applications. De même, XML est la base d'un nouveau mode de communication entre les applications qui est appelé Web Service. Par exemple, JAXP (*Java API for XML Parsing*) analyse des fichiers ou données XML, JAX-RPC (*Java API for XML based RPC*) utilise des Web Services et JAXB (*Java API for XML Binding*) permet de générer des classes Java à partir de schémas XML ou inversement.

Actuellement, la version de Java EE est 5.0 associée à l'API Servlet 2.5, et à l'API JSP 2.1 (et à Apache-Tomcat 5.X-6.X).

Encodage des applications Java

1. Présentation

Les ordinateurs travaillent avec des bits ou suites d'octets (un octet=8 bits). Les chaînes de caractères, entiers, réels sont donc codées sous forme d'octets. Les ordinateurs utilisent donc un procédé qui consiste à transformer les chaînes de caractères en octets, associé à une technique afin de relire les chaînes d'origine. C'est ce procédé qui est appelé encodage.

Il existe plusieurs encodages qui utilisent plus ou moins le même nombre d'octets, donc de caractères disponibles comme ISO-8859-1, ASCII, UTF-8... L'encodage UTF-8 est le plus pratique pour échanger des textes constitués de caractères UNICODE (standard du consortium Unicode). Ce consortium a pour but de répertorier tous les caractères utilisés dans les différentes langues et d'associer à chacun un code noté sous forme hexadécimal. L'encodage UTF-8 est compatible avec l'encodage ASCII ce qui est très pratique lors des développements informatiques.

Lors des développements d'applications Java et/ou Java EE, il n'est pas rare de constater de nombreux problèmes d'encodage des applications tant au niveau des messages présents dans un navigateur, que des fichiers de propriétés d'une application ou encore de l'encodage des caractères saisis au clavier. La mise en œuvre d'une application Java nécessite donc la gestion de plusieurs paramètres. La difficulté est qu'il ne faut pas en oublier un seul sous peine de constater l'affichage de ''hiéroglyphes'' à la place du caractère souhaité.

2. Les fichiers

La première contrainte à vérifier est que tous les fichiers (HTML, JSP, JSPF, XML, XSLT...) de l'application développée soient dans l'encodage souhaité. Pour cela, la plupart des IDE peuvent se paramétrer afin de sélectionner l'encodage. Avec Eclipse, nous pouvons sélectionner l'encodage dans le menu **Fenêtre - Préférences - Editeurs - Codage du fichier texte** et **Types de contenu**.

3. Les pages JSP et JSPF

Il est nécessaire de déclarer dans chaque page JSP ou JSPF d'en-tête l'encodage utilisé. Pour cela, il faut utiliser la directive JSP adaptée.

```
<jsp:directive.page contentType="text/html;charset=UTF-8" />
ou
```

```
<%@ page contentType="text/html;charset=UTF-8" %>
```

Il est possible également de centraliser l'encodage dans le fichier de configuration et de déploiement de l'application *web.xml* du serveur Tomcat.

```
<jsp-config>
 <jsp-property-group>
  <description>Config. de l'encodage des pages JSP</description>
  <url-pattern>*.jsp</url-pattern>
  <page-encoding>UTF-8</page-encoding>
 </jsp-property-group>
</jsp-config>
```

4. Les pages HTML/XHTML

Il est également important de prévenir le navigateur client de l'encodage qu'il doit utiliser pour afficher la page HTML/XHTML. Cette directive est précisée avec la balise *meta* et le paramètre *content*.

```
<head>
<meta equiv="Content-Type" content="text/html;charset=UTF-8">
...
</head>
```

Il est également possible de préciser l'encodage d'une feuille de style externe à l'aide de la directive placée en tout début de fichier.

```
@charset "UTF-8";
```

5. Les feuilles de style XSL

Si des transformations XSLT sont utilisées dans notre application, il est nécessaire de déclarer également explicitement l'encodage dans ces pages.

```
<xsl:output method="xml" omit-xml-declaration="yes" encoding="UTF-8"
indent="yes" />
```

6. Code Java

Du point de vue du code Java, il est possible d'utiliser un filtre qui va forcer le serveur d'applications Java à lire les paramètres de la requête dans l'encodage souhaité et qui va renvoyer les réponses avec le même encodage.

```
package application.filters;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class EncodingFilter implements Filter {
public static final String ENCODING = "encoding";
private String encoding;

public void init(FilterConfig filterConfig) throws ServletException
{
    this.encoding = filterConfig.getInitParameter(ENCODING);
}
public void doFilter(ServletRequest req, ServletResponse resp,
FilterChain filterChain)throws IOException, ServletException
{
    req.setCharacterEncoding(encoding);
    resp.setContentType("text/html;charset="+encoding);
```

```
    filterChain.doFilter(req, resp);
}
public void destroy() {}

}
```

Il est alors possible de déclarer ce filtre au sein du fichier de configuration de l'application *web.xml*. Les filtres étant exécutés dans l'ordre de déclaration, ce mapping doit être le premier déclaré dans le fichier de configuration.

```
<filter>
 <filter-name>encodingfilter</filter-name>
 <filter-class>application.filters.EncodingFilter</filter-class>
 <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
 </init-param>
</filter>
<filter-mapping>
 <filter-name>encodingfilter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

L'encodage peut aussi être géré au sein d'une Servlet générique. Chaque Servlet du projet devra alors ensuite hériter de cette Servlet.

```
package application.servlets;
import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public abstract class EncodingServlet extends HttpServlet {
public static final String ENCODING = "encoding";
private String encoding;
public void init(ServletConfig servletConfig)throws ServletException
{
    this.encoding = servletConfig.getInitParameter(ENCODING);
}

public void doGet(HttpServletRequest req, HttpServletResponse resp)
throws IOException, ServletException
{
    req.setCharacterEncoding(encoding);
    resp.setContentType("text/html;charset="+encoding);
}
public void doPost(HttpServletRequest req, HttpServletResponse resp)
throws IOException, ServletException
{
    request.setCharacterEncoding(encoding);
    response.setContentType("text/html;charset="+encoding);
}
}
```

7. Encodage de la JVM

Il est important d'exécuter la JVM dans l'encodage voulu. Le traitement des chaînes de caractères doit être le même que le reste de l'application. C'est au lancement de la JVM, donc au lancement du serveur Java EE, que l'encodage est spécifié à l'aide de l'argument : `-Dfile.encoding=UTF-8`

Avec Tomcat, cet argument est spécifié dans le fichier de lancement du serveur, *catalina.sh*.

```
JAVA_OPTS="$JAVA_OPTS "-Dfile.encoding=utf-8"
```

Il est parfois également nécessaire de vérifier l'encodage des URL de l'application. Avec Tomcat, cet encodage est déclaré explicitement via l'attribut *URIEncoding* sur le connecteur Coyote. Voici la ligne du fichier *server.xml* concerné :

```
<Connector port="8080" maxHttpHeaderSize="8192"... URIEncoding="UTF-8" />
```

Le code suivant est très utile car il permet la transformation d'une chaîne de caractères dans un encodage précis.

```
public static String transformStringEncoding(String
init,String encodingBefore, String encodingAfter)
{
    try
    {
        return new String(init.getBytes(encodingBefore),
encodingAfter);
    }
    catch (UnsupportedEncodingException uee)
    {
        return null;
    }
}
```

8. Gestion de l'encodage

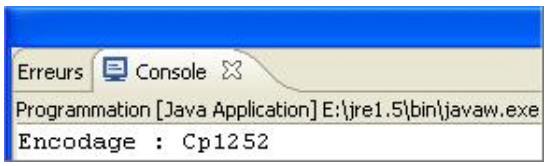
Il est tout à fait possible en Java de gérer l'encodage à utiliser. Pour les flots d'entrée par exemple, la connexion est réalisée à l'aide de la classe *InputStreamReader*. Le jeu de caractères à utiliser peut être alors précisé. Par défaut, le jeu de caractères est fonction du système d'exploitation utilisé et de la localisation (ex : fr_FR UTF-8).

Avec un flot, il est possible de préciser l'encodage utilisé :

```
InputStreamReader i=new InputStreamReader(is,"UTF-8");
```

Le nom du jeu de caractères utilisé par défaut est obtenu en programmation avec la méthode *System.getProperty()* et le paramètre *file.encoding*.

```
package com.betaboutique.classes;

public class Programmation {
   public static void main(String[] args) {
       System.out.println("Encodage : "+System.getProperty("file.encoding"));
 }
}
```



Lors des développements Web, il est assez courant que les paramètres reçus par les méthodes HTTP GET et POST ne soient pas dans un format correct. Les problèmes portent alors sur les accents, les caractères spéciaux... Pour cela, la transformation d'un encodage peut être forcé en utilisant les octets. Le code ci-dessous permet de transformer un paramètre reçu en caractères UTF-8.

```
String parametre=(String)request.getParameter("parametre");
String parametreUTF8=new String(parametre.getBytes(),"UTF-8");
```

De même pour les envois d'informations en programmation Java à travers des flux, l'encodage et les transformations de jeux de caractères sont utilisés.

```
package com.betaboutique.classes;
import java.io.InputStream;
import java.net.URL;
import java.net.URLEncoder;

public class Programmation {

  public static void main(String[] args) {
```

```
    try
    {
    //paramètre à envoyer en direction du flux
    String parametreaenvoyer="mon paramètre";
    String unautreparametreaenvoyer="un autre paramètre";
    String unautreparametreeniso="un paramètre en iso";
    URL url=new URL("http://www.essai.com"+"?parametre1="
+URLEncoder.encode (parametreaenvoyer,"UTF-8")
    +"&parametre2="+URLEncoder.encode(new String
(unautreparametreaenvoyer.getBytes(), "UTF-8"),"UTF-8")
    +"&parametre3="+URLEncoder.encode(unautreparametreeniso,"ISO-8859-1"));

    //poster les informations dans le flux
    InputStream fluxLecture=url.openStream();
    //fermer le flux
    fluxLecture.close();
    }
    catch(Exception e)
    {
    System.out.println("Erreur de connexion");
    }

  }
}
```

Les architectures Web

1. Les types d'architectures

Dans les applications Web, la communication entre le client et le serveur est réalisée selon le protocole TCP/IP qui est chargé du routage des données. Le transit des informations s'effectue selon le protocole HTTP pour le Web, les données sont alors transmises entre le client et le serveur via TCP/IP. On distingue alors deux types de clients :

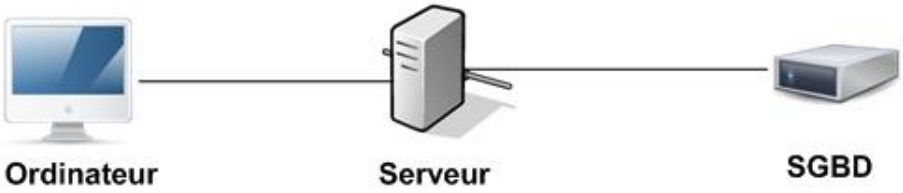
- **Le client léger** : il est aussi appelé client Web car le module d'exécution est alors un navigateur. Les applications clientes sont composées de pages HTML/XHTML voire DHTML avec l'utilisation du langage client JavaScript .

- **Le client lourd** : il s'agit d'une application composée d'une interface graphique évoluée ou en mode console. Dans l'idéal, les clients lourds communiquants ne contiennent que la logique présentation (affichage des données). Tous les traitements sont délégués à des composants métier distants.

Il existe actuellement un grand nombre d'architectures utilisées pour le Web.

L'architecture 2-tiers est composée de deux éléments, un client et un serveur. Cette architecture physique simple peut être représentée de cette façon :

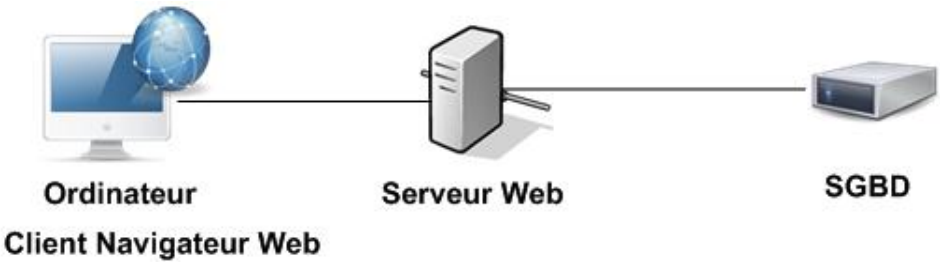
Cette architecture peut aussi être représentée avec un serveur de base de données (SGBD), le schéma est alors le suivant :



Dans ce type d'architecture, le client assume les tâches de présentation et communique uniquement avec le serveur d'applications. Le client est dit ''lourd''. Ce type d'architecture peut être développé très rapidement en fonction de la complexité du projet. Il existe un très grand nombre d'outils de développement et de langages pour les architectures 2-tiers.

Du point de vue des inconvénients, le problème d'évolutivité, de maintenance et la mise en place lors de projets complexes peuvent être cités.

Dans l'architecture 3-tiers, le client est constitué d'un simple navigateur Internet et communique avec le serveur. Cette architecture est composée de trois éléments ou trois couches. La couche présentation ou affichage est le client ''léger'' dans la mesure où il ne fait aucun traitement. La couche fonctionnelle ou métier est en général un serveur Web. Et enfin, la couche de données est liée au serveur de bases de données (SGBD).



- **La couche présentation** (de premier niveau) souvent appelée IHM (*Interface Homme Machine*) correspond à la partie visible et interactive. Cette partie est réalisée pour le Web en HTML en général avec JavaScript, Flash...

- **La couche métier** (de second niveau) correspond à la partie fonctionnelle de l'application. Les opérations à réaliser, les fonctions d'accès aux données et les traitements sont mis à la disposition des utilisateurs et invoqués par leurs requêtes. Pour fournir ces services, elle s'appuie parfois sur la couche accès aux données et en retour renvoie à la couche présentation les résultats qu'elle a calculés.

- **La dernière couche** (de troisième niveau) gère l'accès aux données du système. Ces données peuvent être stockées sur le même système (fichiers, fichiers XML, base de données, images...) ou sur d'autres systèmes. L'accès aux données est transparent pour la couche métier et correspond uniquement à la préoccupation de la couche accès aux données.

D'une manière générale cette abstraction améliore la maintenance du système. Parmi les avantages de cette architecture, la flexibilité de l'ensemble peut être citée. La partie client est composée uniquement d'affichage (pas de programmation, de requêtes SQL...). De fait, des modifications peuvent être réalisées au niveau du SGBD sans que cela apporte un impact sur la couche client. De même, par la suite toute nouvelle technologie peut être introduite sans tout remettre en question. Du point de vue développement, la séparation entre le client, le serveur et le SGBD permet une spécialisation des développeurs et une meilleure répartition des tâches et fonctions (développeur de modèle/designer, programmeur, administrateur de bases de données...).

Le gros inconvénient de ce modèle (et le principal), est l'expertise qu'il est nécessaire d'avoir et qui est assez longue à obtenir pour bien maîtriser chaque tiers et interconnexions. Les coûts de développement d'une architecture 3-tiers sont plus élevés que pour du 2-tiers.

L'architecture n-tiers a été pensée pour pallier les limitations des architectures 3-tiers et concevoir des applications puissantes et simples à maintenir. D'un point de vue théorique, cette architecture permet de solutionner les problèmes suivants :

- Elle permet l'utilisation de clients riches.

- Elle sépare nettement tous les niveaux de l'application.

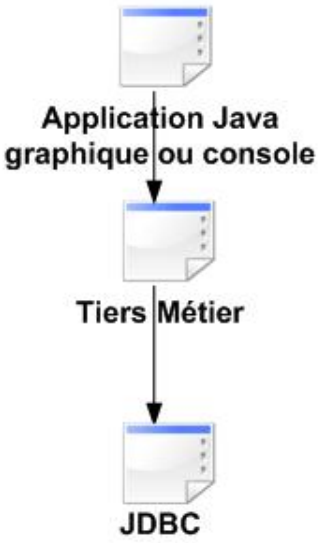
- Elle facilite la gestion des sessions.

- Elle offre de grandes capacités d'extension.

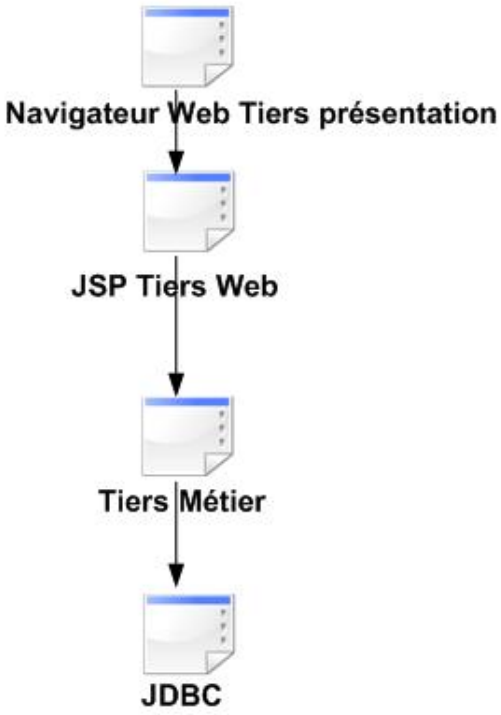
Une définition possible de ce type d'architecture est : une architecture 3-tiers dans laquelle le traitement des données (couche accès aux données ou middleware) contient lui-même plusieurs couches multipliant ainsi les tiers.

Les types d'architectures en Java EE

Dans ce cas, l'application cliente peut être développée avec des composants graphiques (Swing par exemple) et faire appel à des règles métier EJB qui accèdent à une base de données.

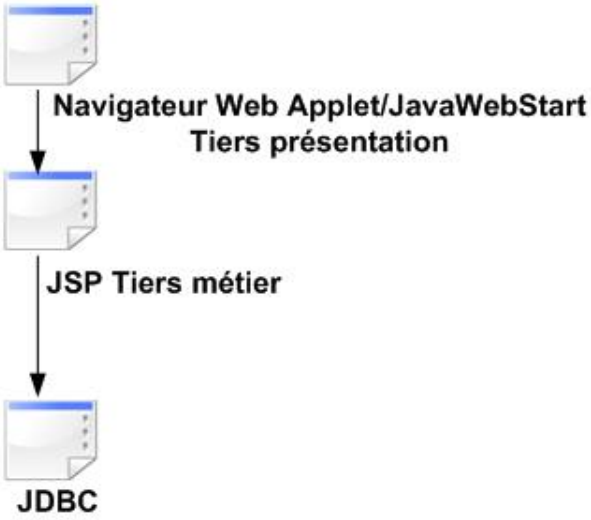


Il est aussi possible de trouver des applications clientes avec JSP, EJB et base de données. Le client est un navigateur Web. Les pages JSP accèdent aux règles métier et construisent le contenu HTML fourni au navigateur.

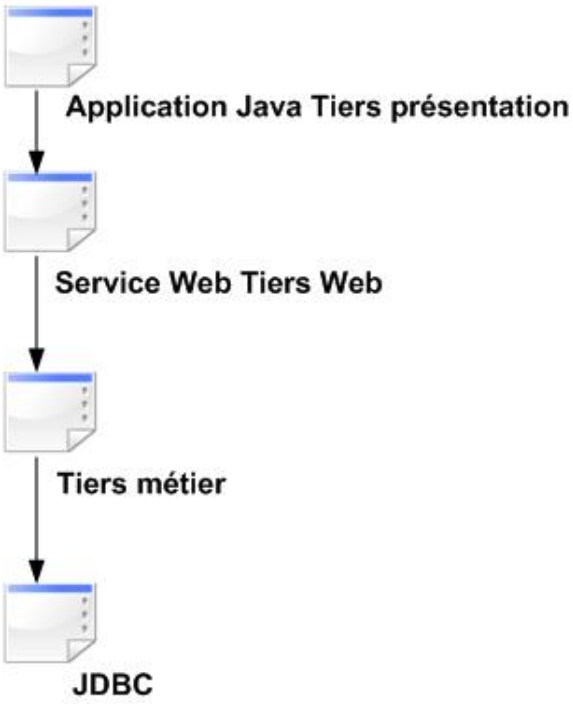


Un dérivé de l'architecture précédente est une Applet cliente avec JSP et base de données. Dans ce cas, le client est encore un navigateur mais une Applet permet d'obtenir une interface utilisateur plus complexe (mais aussi plus lourde) et plus interactive. Cette Applet peut accéder au contenu produit par des JSP. Celles-ci obtiennent des données nécessaires grâce à JDBC.

Une autre version de l'architecture précédente est **JWS** (*Java Web Start*) avec JSP et base de données. C'est pratiquement le même cas que l'architecture avec Applet si ce n'est que le client est une application téléchargeable et utilisable de manière autonome (sans navigateur) mais uniquement avec une connexion réseau.



Le dernier type d'architecture repose sur l'intégration de services Web. Une application cliente accède aux données grâce à un service Web programmé en Java.



Les architectures proposées par Java EE reposent sur le découpage des applications en plusieurs tiers aux responsabilités clairement séparées. Les programmeurs vont développer des composants qui seront hébergés par un serveur d'application Java EE (Tomcat, JBoss...). Les applications distribuées (réalisées sous forme de composants distincts) permettent de diviser le logiciel en plusieurs couches appelées tiers (chaque couche représente un tiers). Le modèle le plus courant étant l'architecture 3-tiers/n-tiers. Cette division facilite la maintenance et l'adaptabilité du produit.

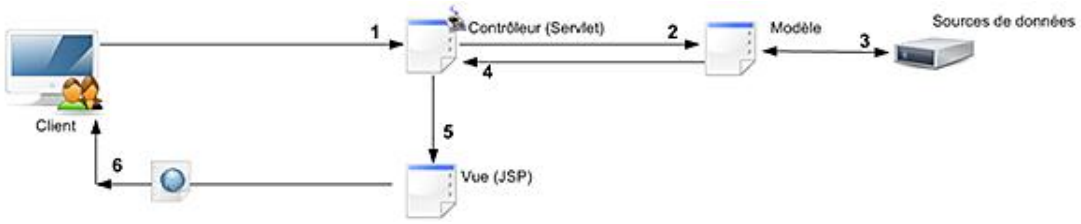
2. L'architecture MVC (Model View Controller)

L'architecture MVC proposée par Sun est la solution de développement Web côté serveur qui permet de séparer la partie logique/métier de la partie présentation dans une application Web. C'est un point essentiel du développement de projets car cela permet à toute l'équipe de travailler séparément (chacun possède ses fichiers, ses logiciels de développement et ses composants).

Cette architecture trouve son origine dans le langage SmallTalk au début des années 1980, ce n'est donc pas un modèle (design pattern) nouveau uniquement lié à Java EE. L'objectif principal est de diviser l'application en trois parties distinctes : le **modèle**, la **vue** et le **contrôleur**.

Dans l'architecture MVC, nous retrouvons :

- Le **modèle** qui est représenté par les EJB et/ou JavaBeans et/ou systèmes de persistances (Hibernate, objets sérialisés en XML, stockage de données par le biais de JDBC...).
- La **vue** qui est représentée par les JSP.
- Le **contrôleur** qui est représenté par les Servlets.

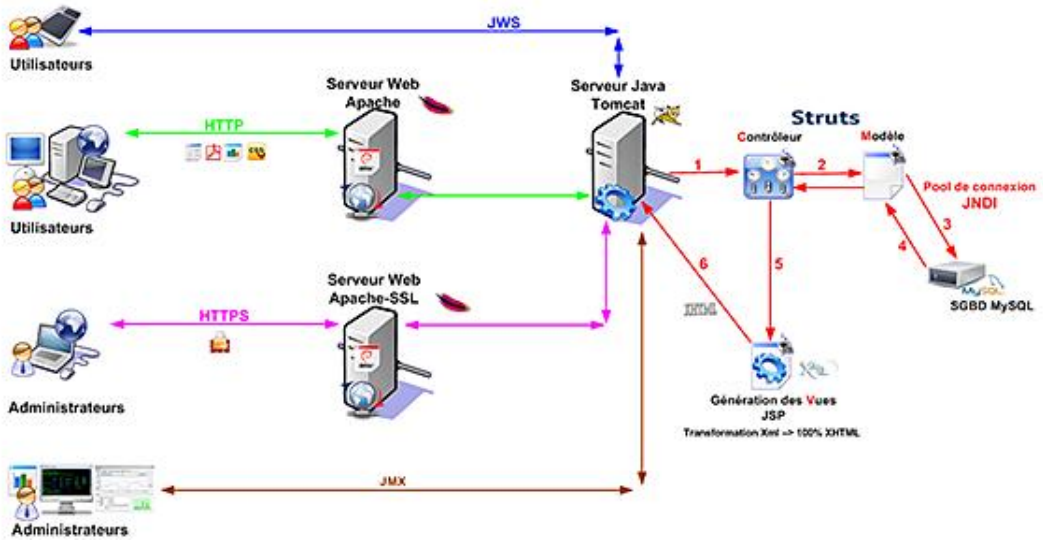


Principe de fonctionnement de l'architecture MVC

- 1. Le client envoie une requête HTTP au serveur. C'est en général une Servlet (ou un programme exécutable côté serveur) qui traite la demande.
- 2. La Servlet récupère les informations transmises par le client et délègue le traitement à un composant métier adapté.
- 3. Les composants du modèle manipulent ou non des données du système d'information (lecture, écriture, mise à jour, suppression).
- 4. Une fois les traitements terminés, les composants rendent la main à la Servlet en lui retournant un résultat. La Servlet stocke alors le résultat dans un contexte adapté (session, requête, réponse...).
- 5. La Servlet appelle la JSP adéquate qui peut accéder au résultat.
- 6. La JSP s'exécute, utilise les données transmises par la Servlet et génère la réponse au client.

Les composants sont bien sûr plus nombreux mais également plus simples. Leurs spécificités font qu'ils pourront être développés par des spécialistes : les Servlets et EJB par des développeurs Java, les JSP par des développeurs et Webdesigner, les accès aux données par des spécialistes SQL... Ce découpage permet également une maintenance plus aisée du système. Ainsi, le changement de la charte graphique sera opéré facilement en utilisant les vues sans toucher au modèle et au contrôleur.

Afin de faciliter l'utilisation du modèle MVC dans les architectures Java EE, des frameworks (outils composés de spécifications, librairies, outils...) de développement entièrement basés sur ce modèle ont été développés (Apache Struts/Spring). Le schéma complexe ci-dessous reprend un exemple de mise en place d'une telle architecture dans un projet d'entreprise.



3. Les différents modules Java EE

Comme indiqué précédemment, les architectures Java EE offrent de multiples possibilités. Il est donc nécessaire d'organiser les différents éléments d'une application en fonction de leur rôle.

Module Web

Le module Web contient les éléments d'une application Java EE qui vont permettre l'utilisation de cette application au travers d'un navigateur Web et de toute application utilisant le protocole HTTP. Les éléments regroupés dans ce module Web sont les Servlets, les JSP et les ressources statiques de l'application Internet (images, JavaScripts, fichiers statiques...). Il y a également des bibliothèques dynamiques développées en Java fournies sous forme de fichiers *.jar* et qui sont utilisées par les Servlets et/ou JSP (manipulation d'images, de fichiers...). Les modules Web possèdent un descripteur de déploiement, le fichier *web.xml*. L'ensemble des fichiers est regroupé dans un fichier d'extension *.war* signifiant WebARchive.

Module EJB/composants métier

Les composants EJB sont constitués de fichiers de code Java. Il y a aussi dans ce module des bibliothèques au

format *.jar*. Les modules sont ensuite assemblés en archive d'extension *.jar*.

Module Client

Il est possible d'utiliser un client riche avec une interface graphique développée en utilisant les API de programmation Java comme Swing et/ou AWT. Un module client permet un assemblage en classes et fournit un descripteur de déploiement. Le module client est un fichier d'archive portant l'extension *.jar*.

Mise en place de l'environnement

L'interface Java EE permet de créer des sites Web dynamiques avec une technologie Java. La mise en place d'un environnement Java EE nécessite l'utilisation d'un serveur d'applications capable d'exécuter le code et de répondre aux requêtes des clients. GlassFish, Jonas, JBoss, WebSphere et Apache-Tomcat font partie de ces serveurs d'applications Java.

Il est également nécessaire d'utiliser un environnement de développement évolué. Il n'est pas possible de développer de manière confortable des centaines de fichiers sources, la documentation, les fichiers de configuration avec un simple éditeur de texte et le compilateur en ligne de commandes.

Il existe plusieurs grands IDE Java : Eclipse et ses différentes versions, JBuilder, NetBean...

Eclipse est très puissant, il dispose d'une grande panoplie de plug-ins pour l'interfacer avec Tomcat, pour manipuler Mysql, pour gérer les fichiers XML, les JSP, le code JavaScript... Enfin, point important, Eclipse est un projet OpenSource gratuit.

Pour la réalisation des pages et des exemples, la version Lomboz d'Eclipse sera utilisée (Eclipse + ensemble de plug-ins pour Java EE). Cette version complète d'Eclipse a été développée par le consortium ObjectWeb.

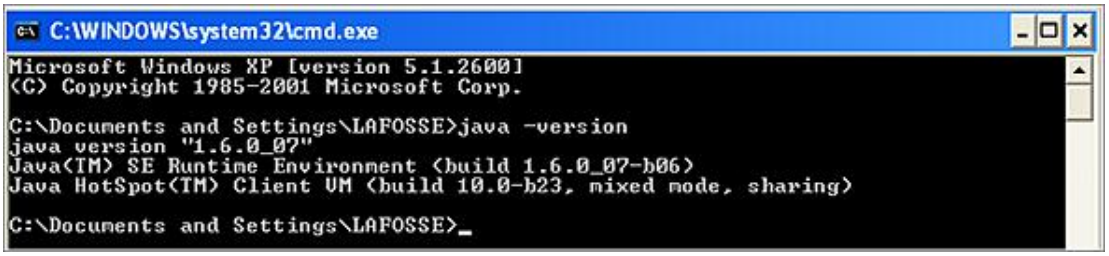
L'installation sera expliquée pour un système Windows et pour un système Linux. Vous remarquez alors l'intérêt de développer en Java, tout est portable et indépendant de la plate-forme de développement. Nous vérifierons l'installation complète de l'environnement en déployant une application très simple. Il est essentiel de tester l'installation avec un petit projet plutôt que de rencontrer des problèmes ensuite lors de la réalisation d'exemples complexes.

1. Installation du JDK (Java Development Kit)

Le Java Development Kit (couramment abrégé en JDK) est l'environnement dans lequel le code Java est compilé pour être transformé en bytecode afin que la JVM (machine virtuelle Java) puisse l'interpréter/l'exécuter.

La première étape nécessaire à l'installation de l'environnement est la mise en place des bibliothèques de développement Java. En effet, avant l'installation du serveur Java EE, il est impératif que le JDK soit installé sur la machine où celui-ci sera installé. Le serveur d'applications fonctionne en Java et a donc besoin lui aussi du JDK pour travailler. Le JDK fournira un compilateur Java nécessaire pour le traitement des JSP. Avant d'installer une version de développement Java, il est nécessaire de vérifier si le système actuel ne possède pas déjà une version de Java.

Pour vérifier si le système actuel possède une version de Java, il faut ouvrir une invite de commandes Ms-DOS et lancer la commande : `java -version`



Nous pouvons voir que la version indiquée dans cet exemple est Java 1.6 (ou également appelée 6.0). Le compilateur utilisé est J2SE (il aurait été aussi possible d'utiliser Java EE). Si cette commande n'indique aucun résultat, c'est que le JDK n'est pas installé ou que la variable d'environnement PATH n'est pas correctement configurée sur le système.

Les variables d'environnement sont des raccourcis utilisés par Windows et Linux pour désigner certains dossiers (exécutables) du disque dur (ex : pas besoin de saisir /usr/bin/java/javac, le raccourci javac suffit). Une fois les variables d'environnement correctement configurées, l'exécution de vos applications sera beaucoup plus simple.

Nous pouvons également obtenir des informations sur l'API Java en utilisant sous Windows le panneau de configuration. L'icône Java ouvre une fenêtre avec les onglets suivants : **Général** - **Mise à jour** - **Java** - **Sécurité** et **Avancé**.

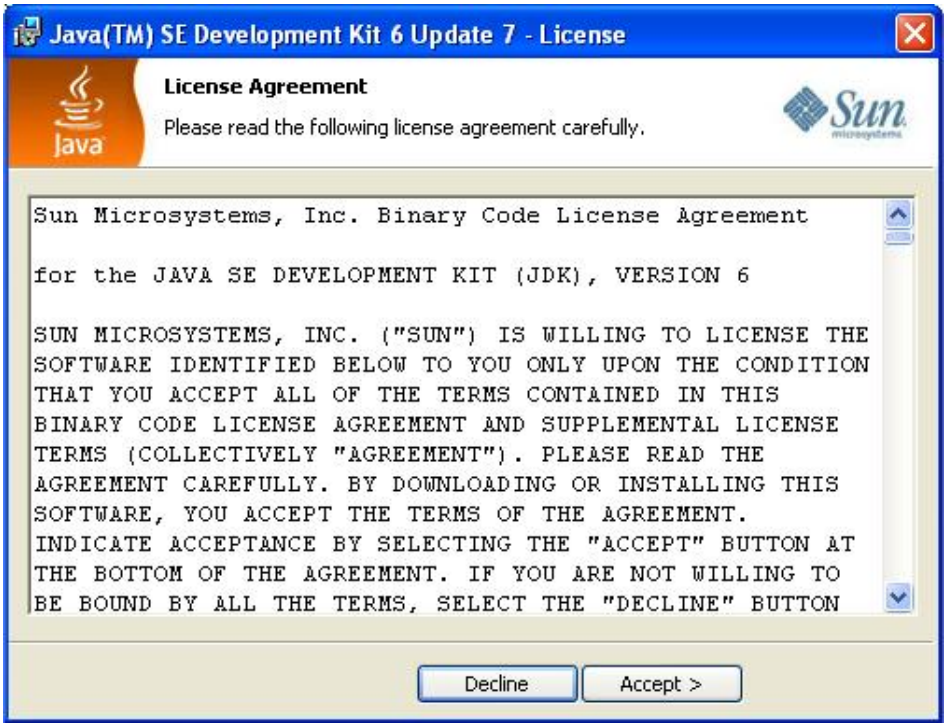
a. Installation sous Windows

Une fois ces vérifications effectuées, l'installation de Java EE s'effectue simplement en exécutant le fichier téléchargé sur le site de Sun Microsystems (http://java.sun.com/). Le programme d'installation démarre puis indique les étapes à suivre.

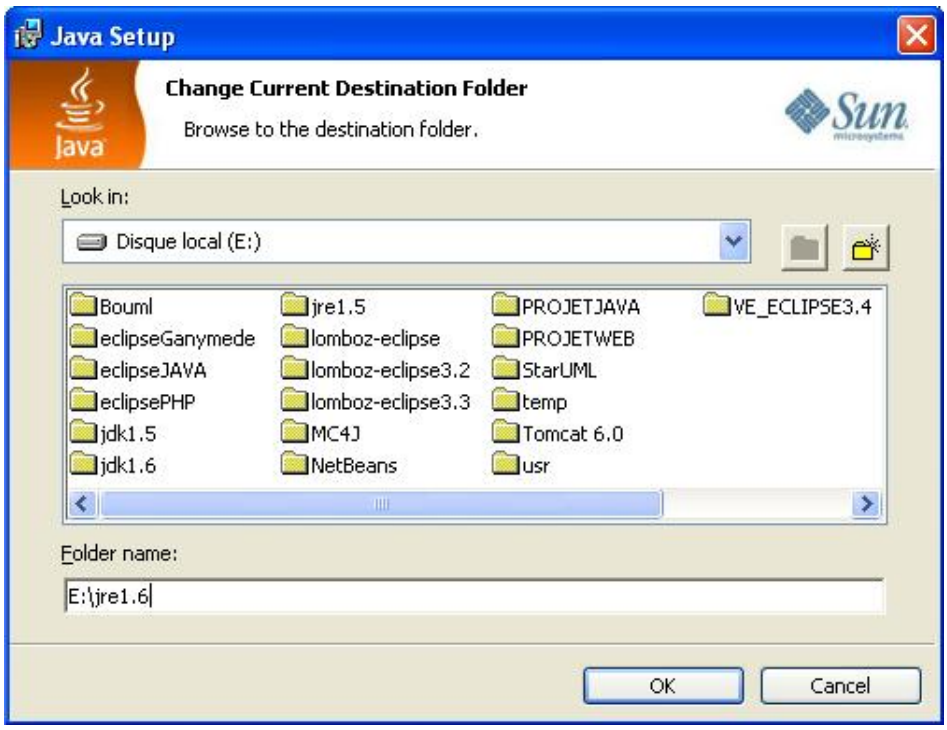
La page suivante permet de télécharger le JDK/Java EE/ Java SE : http://java.sun.com/javase/downloads/index.jsp

La version Windows Platform d'environ 160 Mo permet d'installer l'environnement de développement complet. L'installation ne pose pas de problème particulier, il suffit de double cliquer sur l'exécutable téléchargé et l'installation s'effectue toute seule après avoir indiqué le répertoire d'installation des bibliothèques.

1 - Lancement de l'installation



2 - Acceptation des conditions



3 - Choix du répertoire d'installation

4 - Fin de l'installation

b. Installation sous Linux

L'installation sous Linux nécessite le téléchargement de la librairie au format *.bin*.

Toujours sur la même page, il est possible de télécharger la version du JDK pour Linux.

La page suivante permet de télécharger le JDK/J2EE/JSE :

https://sdlc5b.sun.com/ECom/EComActionServlet;jsessionid=BB61072B1EEB17E1232E99D84013FACB

La version Linux Platform - Java EE SDK d'environ 150 Mo permet une installation sous Linux. L'installation sous

Linux est un peu différente de celle sous Windows. Après le téléchargement de la bibliothèque, il est nécessaire de copier celle-ci dans un répertoire de librairies Linux (en général */usr/local/src*).

Exemple avec le jdk-1.6.0 :

```
#mv /home/jdk-1_6_0-linux-i586.bin /usr/local/src
```

Puis, il faut rendre ce paquet exécutable.

```
#chmod +x jdk-1_6_0-linux-i586.bin
```

L'installation peut être ensuite lancée.

```
#./jdk-1_6_0-linux-i586.bin
```

Tout le répertoire est alors décompacté dans */usr/local/src*.

Il faut maintenant déplacer le répertoire du JDK dans un répertoire final d'installation (en général */usr/local*).

```
#mv /jdk.6.0_00 /usr/local
```

Il faut ensuite se déplacer dans le répertoire */usr/local*.

```
#cd /usr/local
```

Un lien (raccourci) appelé *jdk* peut être créé, il sera plus simple d'accès que le nom complet jdk.6.0_00.

```
#ln -s jdk.6.0_00 jdk
```

Il reste à placer les droits sur les fichiers de la librairie Java.

```
#chmod 755 jdk.6.0_00
```

Java est désormais installé mais il faut encore paramétrer les variables d'environnement pour pouvoir lancer les commandes directement (sans indiquer le chemin complet, ex : */usr/local/jdk/java*). Dans l'état actuel nous ne pouvons pas lancer Java directement :

```
#java -version (not found...)
```

Il existe plusieurs solutions pour cela :

1. Éditer le fichier */etc/profile* et ajouter la ligne suivante dans le fichier.

```
PATH=$PATH:/usr/local/jdk/bin
```

2. Éditer le fichier */root/.bashrc* et ajouter la ligne suivante dans le fichier.

```
export PATH=$PATH:/usr/local/jdk/bin
```

3. Éditer le fichier de l'utilisateur connecté et ajouter la ligne suivante dans le fichier.

```
export PATH=$PATH:/usr/local/jdk/bin
```

4. Exporter la variable d'environnement PATH directement en ligne de commande dans un shell.

```
export PATH=''$PATH:/usr/local/jdk/bin''
echo $PATH (pour vérifier)
```

Pour que les modifications soient effectives, il faut fermer toutes les fenêtres et ouvrir un nouveau terminal.

Attention, par la suite nous réaliserons un script de démarrage de Tomcat (serveur d'applications) qui exportera lui-même les variables d'environnement.

Vous pouvez désormais vérifier que l'installation du JDK sous Linux est opérationnelle.

```
#java -version
```

Installation du serveur d'applications Java EE (Tomcat)

Un serveur Java EE est aussi appelé serveur d'applications (applications signifiant applications Web). L'utilisation d'un serveur Java EE est obligatoire pour le développement de pages Web dynamiques en Java EE.

Un serveur HTTP classique reçoit des requêtes HTTP et renvoie des réponses mais il ne connaît pas les Servlets, les JSP... Il est donc essentiel d'utiliser un programme appelé moteur de Servlets qui est contenu dans le serveur Java EE et qui permet de pallier ce manque. Dans la plupart des cas, le serveur Java EE contient également un serveur HTTP mais il n'est pas aussi puissant que les serveurs spécialisés du monde informatique pour les contenus statiques (Apache).

Il existe un grand nombre de serveurs qui répondent à cette norme (Tomcat, WebSphere, JRun, JBoss, GlassFish...). Nous utiliserons le serveur Apache-Tomcat de la fondation Apache. Il est très important de comprendre que ces serveurs servent uniquement à fournir une plate-forme d'exploitation de l'API Java EE fournie par SUN. Donc, si le serveur répond à la norme et au standard Java EE, le choix n'a alors que peu d'importance. La différence sera observée d'un point de vue rapidité, sécurité, facilité d'utilisation, montée en charge, gestion ou pas des EJB...

Les versions majeures de Tomcat correspondent toutes à une implémentation de référence des technologies Servlet et JSP. Voici un bref rappel des relations entre les versions des technologies Java EE et les versions de Tomcat.

Spécifications J2EE	API Servlet	API JSP	Apache Tomcat
J2EE 1.2	2.2	1.1	3.X
J2EE 1.3	2.3	1.2	4.X
J2EE 1.4	2.4	2.0	5.X
Java EE 5.0	2.5	2.1	5.X - 6.X

Comme indiqué précédemment, il est impératif que le JDK soit déjà installé avant l'installation du serveur d'applications.

Le serveur Tomcat 6 est disponible en libre téléchargement. Les versions binaires de Tomcat sont en fait constituées de classes Java et sont donc portables entre les systèmes d'exploitation et les plates-formes matérielles.

Il existe trois formats d'archives binaires :

- Les archives au **format ZIP** : une fois le répertoire décompressé, le serveur est directement opérationnel après configuration. Ce format est intéressant pour les administrateurs car il permet une mise à jour rapide en cas de changement de version du serveur. De plus, la configuration du système n'est pas modifiée, l'installation est transparente.

- Les archives au **format TAR.GZ** : c'est le format le plus commun sous les systèmes Linux.

- Les **installeurs Windows** : au **format EXE** permettent une installation à partir d'un assistant qui réalise également la configuration. C'est la méthode la plus simple pour installer Tomcat sur le système de Microsoft.

1. Quelle version choisir ?

Tomcat 6.X est une adaptation de Tomcat 5.5 pour Java 5.0 (JDK 1.5) avec des améliorations concernant les performances. La version 1.5 de Java a vu apparaître de nombreuses modifications dans le langage. Tomcat 6.X requiert donc au minimum une machine virtuelle Java 1.5 pour fonctionner correctement.

Pour le guide et le projet, la version de Tomcat utilisée est la version 6.0.16.

Installation sous Windows

L'installation de Tomcat version 6 suppose le téléchargement de la bibliothèque depuis le site de la fondation Apache à l'adresse suivante : http://tomcat.apache.org/

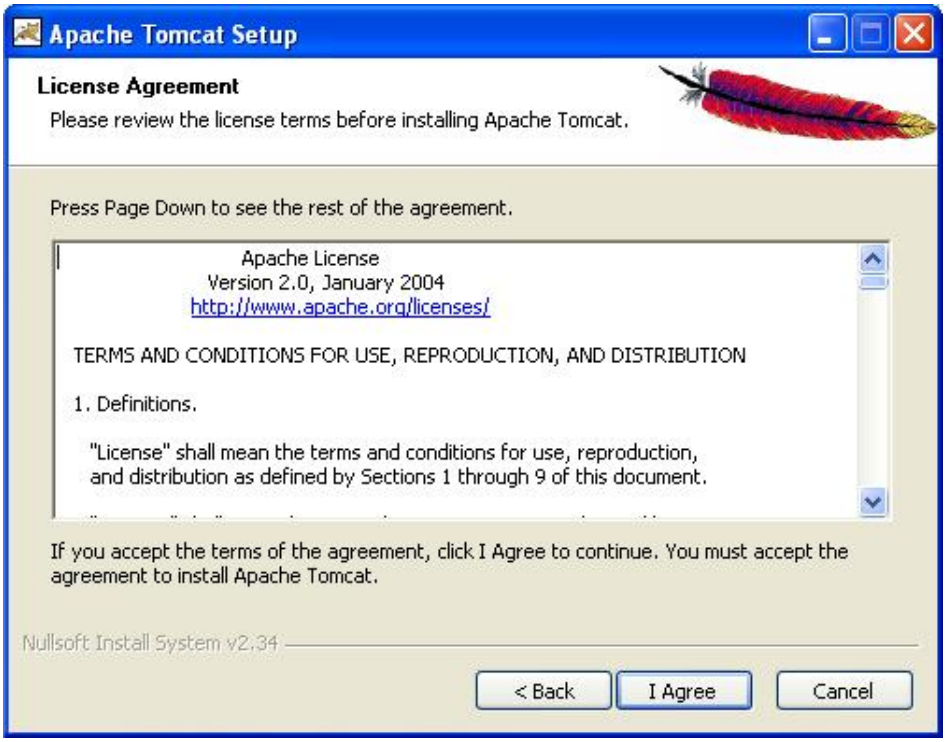
La section download permet de choisir la version adaptée à nos besoins. Les liens Core sont alors utilisés au format souhaité (Windows Service Installer par exemple) afin de télécharger Tomcat pour Windows (http://apache.miroir-francais.fr/tomcat/tomcat-6/v6.0.16/bin/apache-tomcat-6.0.16.exe).

Une fois la bibliothèque téléchargée, il est demandé d'indiquer le chemin de la JVM (*Java Virtual Machine*) sur le système (d'où l'importance d'installer le JDK avant le serveur).

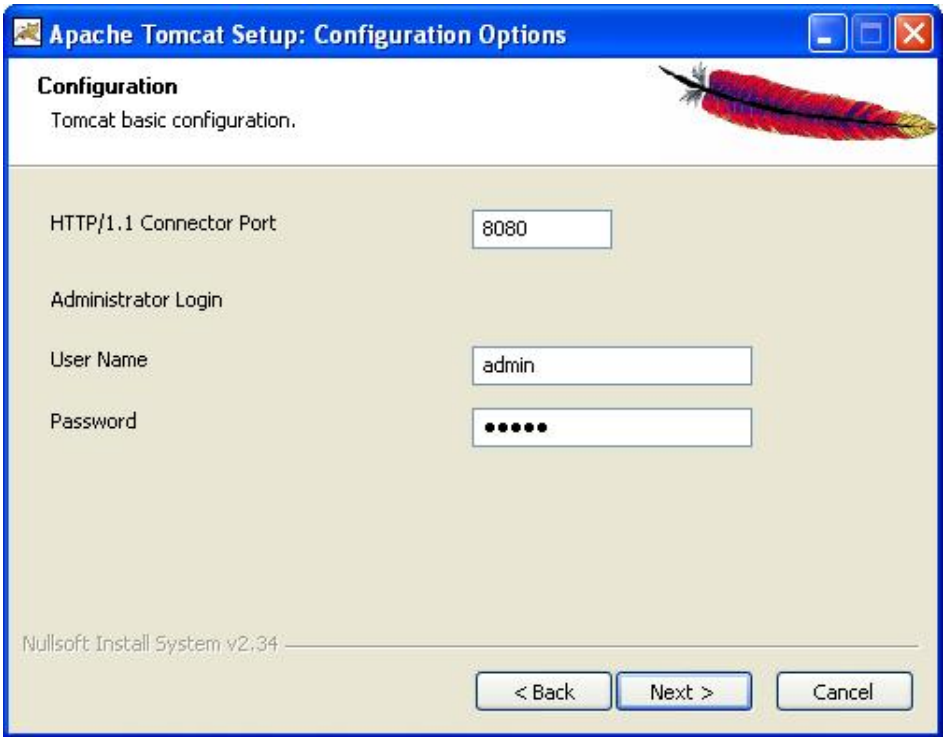
Tomcat 6 utilise un certain nombre de ports TCP/IP pour fonctionner. Il faut donc s'assurer que ces ports ne sont pas déjà utilisés : **Port 8080** : port du connecteur HTTP Tomcat, **Port 8005** : port d'arrêt du serveur, **Port 8809** : port du connecteur JK.

L'installation avec l'installeur Windows permet de créer les entrées dans le menu Démarrer de Windows ainsi qu'un service pour Tomcat permettant, si nécessaire, le démarrage de celui-ci au lancement du système. L'installeur propose également de créer un utilisateur Tomcat pour administrer et gérer les applications (un super administrateur).

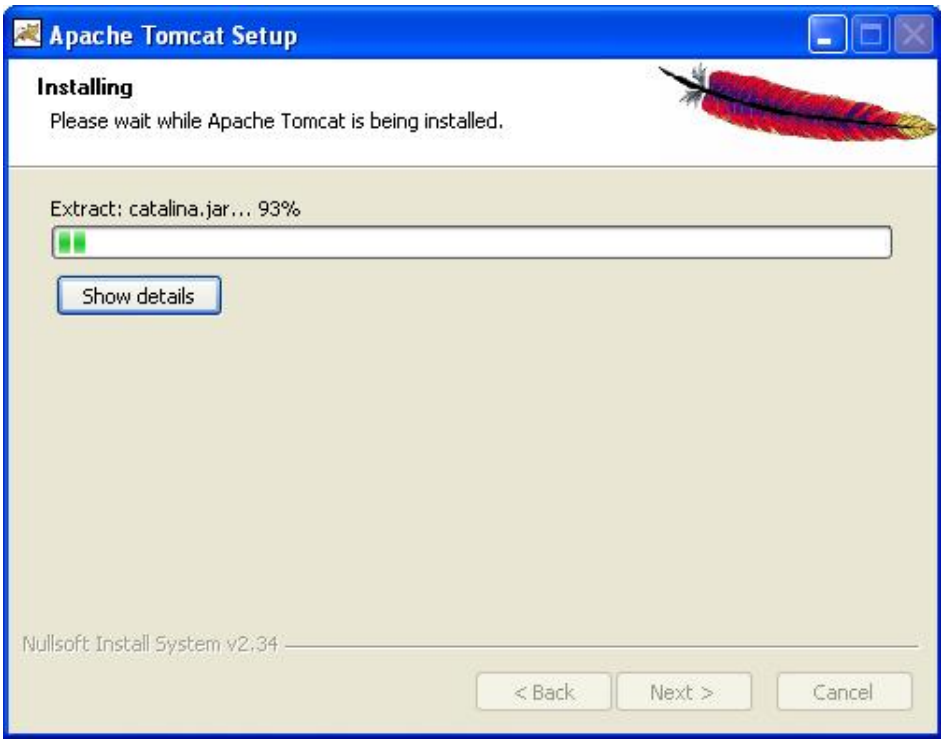
1 - Lancement de l'installation



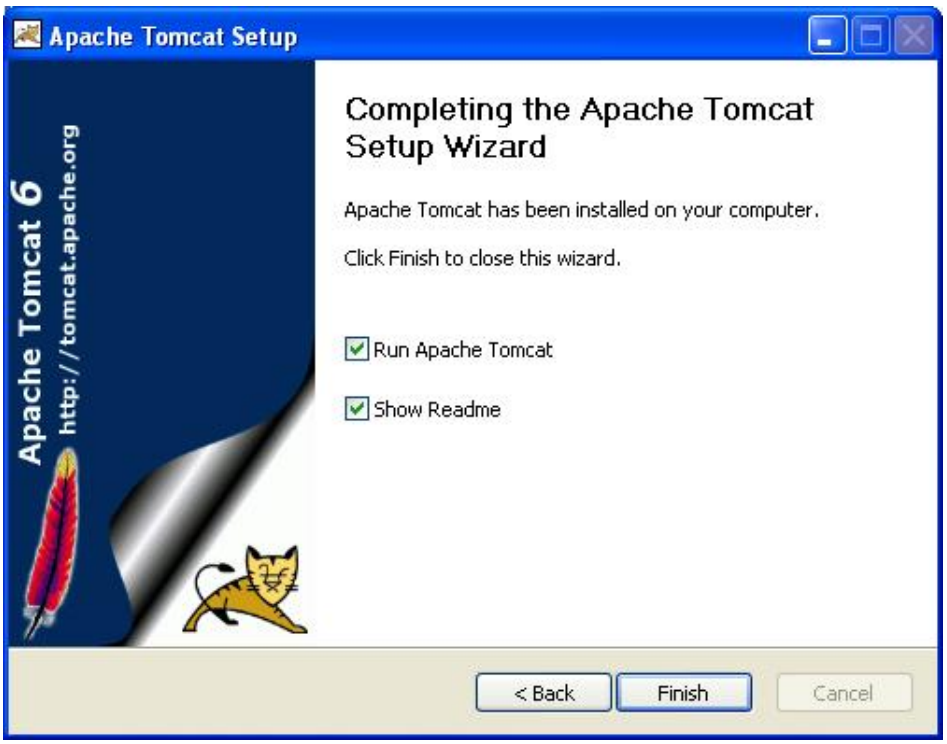
2 - Acceptation des conditions



3 - Création de l'utilisateur (User Name : admin, Password : admin)

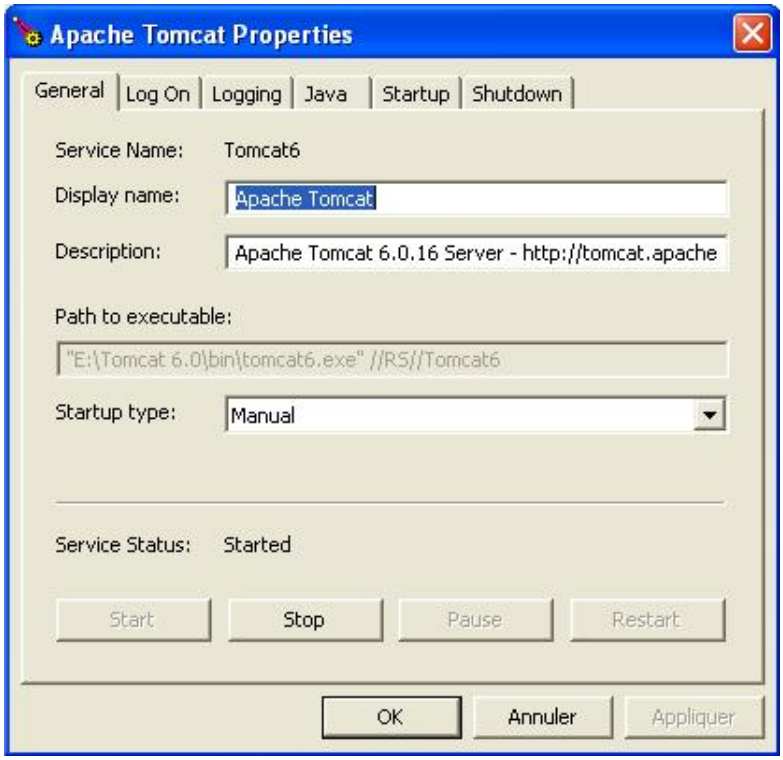


4 - Suite de l'installation



5 - Fin de l'installation

Après l'installation, le fonctionnement du serveur peut être testé en lançant le moniteur Tomcat situé dans le menu démarrer. Le moniteur se lance dans le systray (barre des tâches à côté de l'horloge Windows) avec une icône rouge. Le service de monitoring est lancé mais le serveur pas encore. Pour réellement lancer le serveur Tomcat, il faut soit double cliquer sur l'icône soit faire un clic droit et start service.



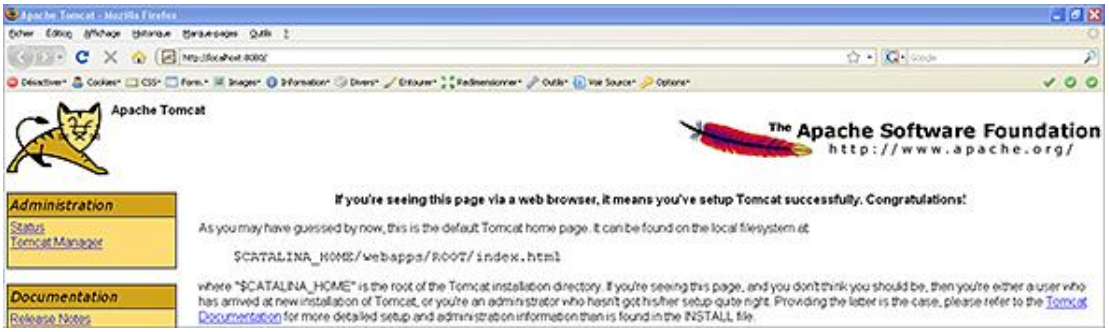
Dans cette fenêtre de monitoring, Tomcat peut être démarré en cliquant sur **Start**, stoppé avec le bouton Stop... Les autres onglets de la fenêtre permettent de gérer les logs, le JDK associé au serveur, le lancement au démarrage de la machine, les opérations d'arrêt du serveur...

Vérifier que le serveur Java EE est actif

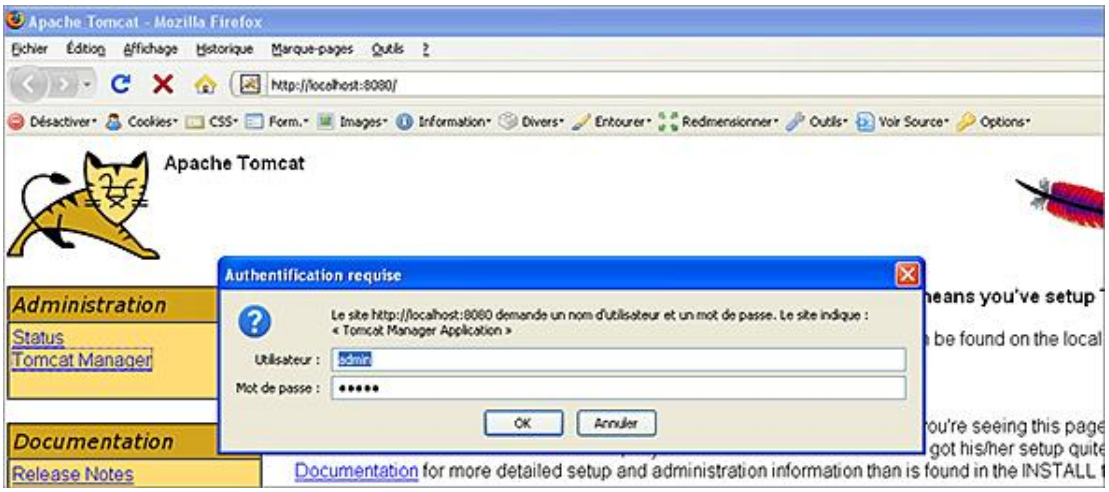
Afin de vérifier que le serveur est correctement installé et opérationnel (capable d'exécuter des Servlets et pages

JSP), nous pouvons lancer un navigateur et saisir l'adresse suivante : http://localhost:8080/

Le nom localhost correspond à l'adresse locale de la machine (le serveur de la machine locale) et le port 8080 est le port HTTP pour Tomcat.



Suivant le serveur utilisé (Tomcat, JRun...) le fonctionnement est identique, seul le port d'accès HTTP change (8080, 8200...). Le lien Tomcat Manager du menu permet de gérer les différentes applications déployées sur le serveur. Une authentification est nécessaire. Les coordonnées (identifiant et mot de passe) correspondent à celles saisies lors de l'installation du serveur (User Name : admin, Password : admin).



Il peut être intéressant d'installer la partie administration de Tomcat. Pour cela, il faut télécharger le ZIP Administration Web Application (http://tomcat.apache.org) et décompresser ce fichier dans le répertoire d'installation de Tomcat. Il sera alors possible d'accéder à la partie administration du serveur à l'adresse suivante : http://localhost:8080/admin.

Pour vérifier le fonctionnement du serveur, nous pouvons lancer les exemples présents, par défaut, sur le serveur avec des Servlets et des JSP.

Pour l'installation sous Linux, voir le chapitre Le serveur d'applications Apache-Tomcat consacré à Tomcat.

Installation de l'environnement de développement (IDE) Eclipse

1. Présentation

Eclipse est l'environnement de développement (spécialisé pour le langage Java) qui sera utilisé dans cet ouvrage. Le choix d'Eclipse repose essentiellement sur sa gratuité, sa facilité d'utilisation, sa puissance de développement et surtout ses nombreux plug-ins (bibliothèques additives). Il existe actuellement beaucoup de versions d'Eclipse de base (versions 2.1, 3.X). La version utilisée dans cet ouvrage correspond à Eclipse 3.3 Europa.

2. Installation

Eclipse a été décliné en plusieurs versions spécifiques pour des développements orientés. Par exemple, pour le développement Java EE, la version Lomboz d'Eclipse développée par le consortium ObjectWeb (http://www.objectweb.org/) est actuellement l'une des plus poussées et stables (parseur XML, syntaxe JSP, CSS, HTML, XHTML...).

La version utilisée de Lomboz regroupe Eclipse et les plug-ins pour le développement Java EE.

La version téléchargée est la suivante : http://lomboz.objectweb.org/downloads/drops/R-3.3-200710290621/

Il existe une version pour Windows et une autre pour Linux. Il est important de télécharger le projet complet (Lomboz Complete Installation) afin de disposer d'un système robuste et stable.

Installation sous Windows

L'installation ne pose pas de problème particulier. Le fichier téléchargé au format *.zip* doit être décompressé dans le répertoire où nous souhaitons installer Eclipse/Lomboz. Comme pour le serveur d'applications, il est important de bien installer avant l'IDE un JDK Java. Lors de l'installation d'Eclipse, il sera alors demandé de préciser le répertoire d'installation du JDK.

Si au lancement de Windows nous obtenons une fenêtre d'erreur de ce type : *JVM Terminated. Exit code=1...*, il faut alors supprimer le répertoire *.metadata* qui se trouve dans le répertoire des projets d'Eclipse (*Workspace*) et relancer l'IDE.

Installation sous Linux

Le fichier téléchargé au format *.tar.gz* doit être copié dans le répertoire des sources.

```
#cp lomboz.tar.gz /usr/local/src
```

Il faut ensuite dézipper et détarrer cette archive.

```
#gunzip lomboz.tar.gz
```

Un répertoire nommé lomboz est alors créé. Il faut déplacer ce répertoire dans */usr/local*.

```
#mv -r lomboz /usr/local/lomboz
```

Il faut enfin positionner des droits corrects sur les fichiers.

```
#chown -R tomcat:tomcat /usr/local/lomboz
```

Nous pouvons ensuite lancer l'IDE Eclipse.

```
#/usr/local/lomboz/eclipse&
```

Le lancement d'Eclipse est donc effectué en tapant la commande shell adaptée (depuis le répertoire d'installation d'Eclipse ou avec le chemin complet).

```
#eclipse -vm /usr/local/jdk
```

Cette commande permet de lancer Eclipse en précisant le répertoire d'installation du JDK. Par défaut, les projets sont stockés dans le répertoire *workspace* situé dans le répertoire d'Eclipse. Il est également possible de modifier ce paramètre avec l'option *-data*.

```
#eclipse -data '/home/lafosse/mesprojets'
```

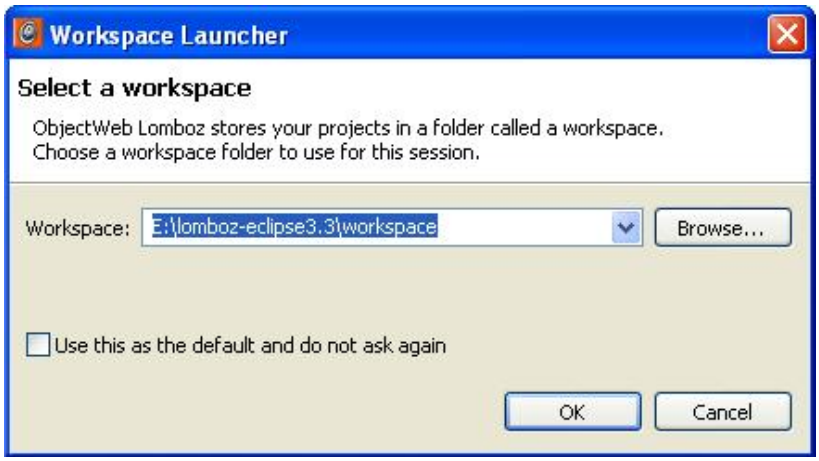
3. Les plug-ins Eclipse

Un plug-in est un composant logiciel additionnel qui permet à un logiciel principal d'apporter de nouvelles fonctionnalités. Il existe une multitude de plug-ins plus ou moins intéressants et fiables pour Eclipse qui sont classés par catégorie (J2EE, graphiques...) à cette adresse URL : http://www.eclipseplugincentral.com

4. Lancement d'Eclipse et paramétrage du serveur Java EE (Tomcat)

Au lancement d'Eclipse, un splashscreen (écran d'attente) est affiché et permet de charger en arrière-plan les très nombreuses librairies de l'environnement.

Il faut ensuite valider le *workspace* (répertoire de travail) et l'environnement s'ouvre.



Installation du plug-in de gestion du serveur Tomcat

Eclipse possède un plug-in développé par la société SYSDEO qui permet de gérer le serveur d'applications Tomcat. Ce plug-in permet de disposer d'une barre d'outils et de démarrer, arrêter et redémarrer Tomcat directement depuis Eclipse. L'installation est très simple, il suffit de télécharger le plug-in Eclipse-Tomcat à cette adresse : http://www.eclipsetotale.com/tomcatPlugin.html

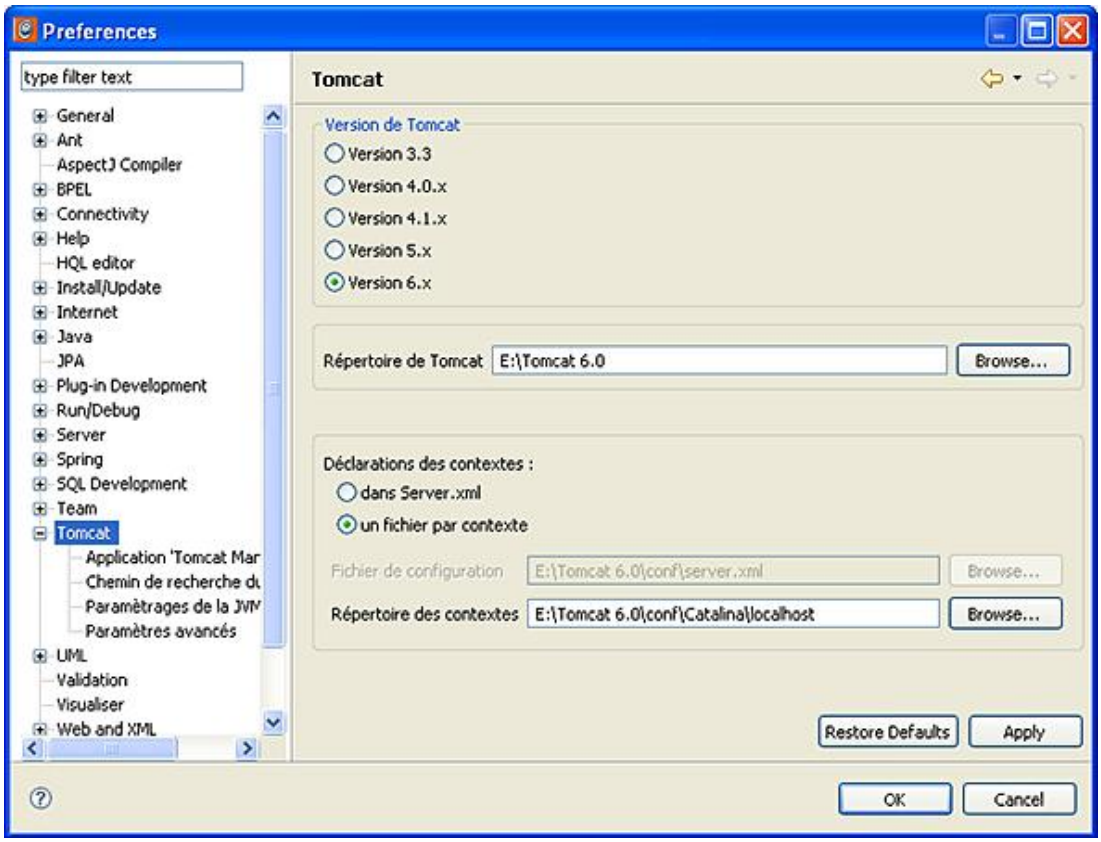
Il est important de choisir le plug-in qui correspond à la version d'Eclipse installée. Pour l'utilisation de ce guide et le développement du projet, le plug-in utilisé est *tomcatPluginV321.zip*.

Une fois le téléchargement terminé, il faut décompresser ce fichier dans le répertoire des plug-ins d'Eclipse (*/lomboz-eclipe3.3/plugins/*).

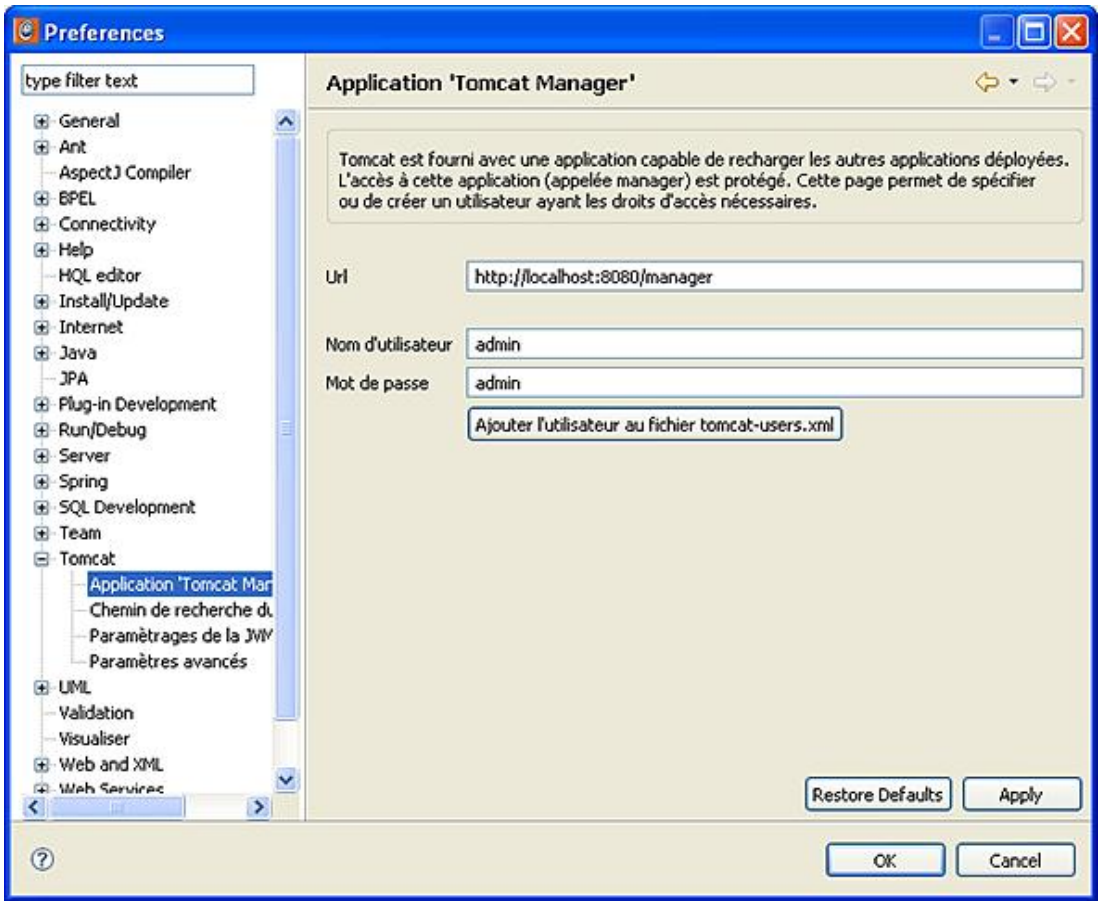
Avant l'installation d'un nouveau plug-in Eclipse, il est important de fermer l'environnement de développement, d'installer le plug-in et de relancer l'environnement.

Lors du premier lancement, il est important de paramétrer correctement l'IDE avec le serveur Java Tomcat et de vérifier le JDK utilisé. Pour contrôler Tomcat depuis Lomboz et éviter de toujours recharger les pages, démarrer/arrêter Tomcat avec le menu démarrer, il faut réaliser les opérations suivantes : **Menu Windows - Preferences - Tomcat**.

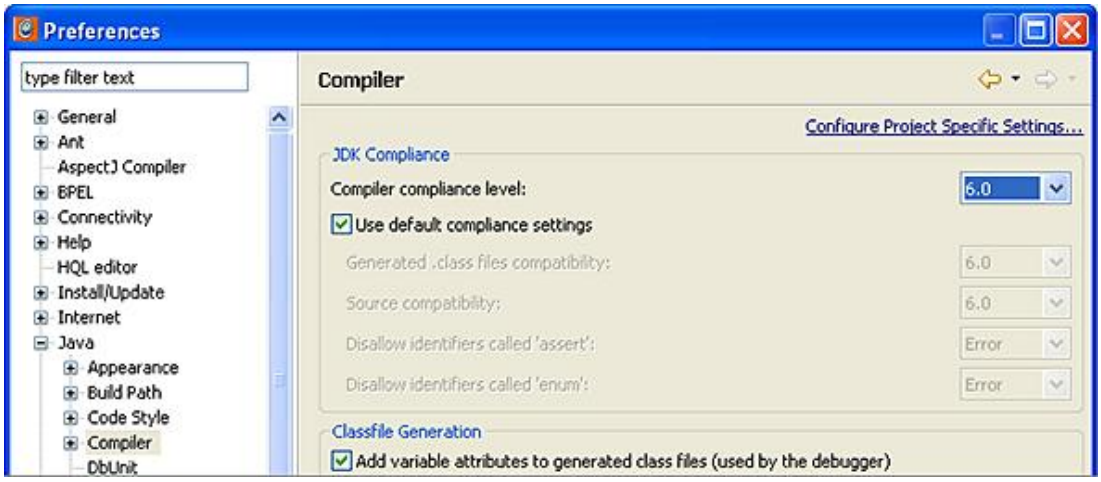
Dans la fenêtre principale, il faut indiquer la version de Tomcat utilisée (Version 6.x), le répertoire d'installation de Tomcat et les déclarations de contextes. Les déclarations de contextes permettent de préciser si un seul fichier est utilisé pour déclarer les applications ou si au contraire, il y a un fichier par application. Il est préférable d'utiliser un fichier par application ou contexte pour des raisons pratiques de maintenabilité.



Dans la fenêtre **Application Tomcat Manager**, il faut vérifier que l'URL est bien http://localhost:8080/manager et indiquer l'identifiant et le mot de passe Tomcat (ceux utilisés lors de l'installation du serveur, par défaut : User Name : admin - Password : admin).



Enfin, l'installation de l'environnement est terminée en vérifiant la compatibilité et la version du JDK utilisé par Eclipse : **Windows - Preferences - Java - Compiler**.



Dans la fenêtre, il est visible que le compilateur Java JDK utilisé est la version 6.0 (JDK 1.6). Nous pouvons désormais relancer Eclipse et observer la nouvelle barre d'outils proposée.

5. En résumé

Ce chapitre nous a présenté les conventions utilisées dans ce guide et au sein de Java ainsi que la gestion de l'encodage en Java.

La plate-forme Java EE a été expliquée en détail ainsi que ses services associés tels que les Servlets, JSP, EJB, JDBC... Dans un troisième temps, les architectures Web ont été présentées afin de montrer la pertinence d'un choix physique avant le développement d'un projet.

Enfin, l'environnement Java EE a été mis en place avec l'installation du JDK Java, du serveur d'applications, de l'environnement de développement et du paramétrage de l'ensemble.

Qu'est-ce que Tomcat ?

1. Présentation et définition

Apache-Tomcat est le serveur d'applications Java du projet Jakarta de la fondation Apache. Ce serveur libre, sous licence Apache permet d'exécuter des applications Web développées avec les technologies Java (Servlets, JSP...).

Apache-Tomcat trouve ses origines au tout début de l'apparition des technologies Servlets et JSP Java lorsque Sun Microsystems décide de donner le code de son serveur Java Web Server à la fondation Apache (1999). Aujourd'hui, Tomcat est pour Sun Microsystems, le serveur de référence pour les technologies Java EE Servlet et JSP. Tomcat est un moteur de Servlets fiable, évolutif et adapté à l'utilisation professionnelle. Il est actuellement utilisé dans le monde entier et mis en application au sein de domaines très variés.

2. La fondation Apache

Le serveur Web Apache a été développé par Rob McCool en 1994. La première version de ce serveur Web est rendue disponible en Avril 1995 sous le nom d'Apache (A Patchy Server). Aujourd'hui, le serveur Web Apache est le serveur le plus utilisé de la planète. En 1999, les développeurs à l'origine d'Apache fondent l'Apache Software Foundation. Cette organisation à but non lucratif développe de nombreux projets et logiciels libres (le serveur Tomcat, des librairies pour le développement Internet, le serveur Web Apache, des bibliothèques de balises...).

3. Le projet Jakarta

Jakarta est un des très nombreux projets de la fondation Apache. Jakarta divise ses projets en trois grandes catégories :

- les serveurs d'applications ;
- les bibliothèques, outils et API ;
- les frameworks.

Le serveur d'applications Tomcat appartient à la première catégorie des projets Apache. Parmi les autres projets, il y a :

- *JMeter* : outil de mesure de performances des applications Web ;
- *Log4J* : bibliothèque de gestion des fichiers journaux (logs) et traces de programmation ;
- *Struts* : le framework de développement Web en Java le plus célèbre ;
- *ANT* : l'outil d'automatisation des applications Web ;
- *Commons* : un ensemble de bibliothèques de programmation Java.

Actuellement, le projet Tomcat a pris une telle ampleur qu'il n'est plus considéré comme un sous-projet Jakarta (de la catégorie serveurs d'applications) mais comme un projet complet dénommé Apache-Tomcat.

4. Évolutions de Tomcat

La première version de Tomcat est la version 3.X qui est l'implémentation des technologies Servlets 2.2 et JSP 1.1. Cette version a été conçue à partir du code source donné par Sun Microsystems à la fondation Apache. À partir de 2000, le serveur a été complètement modifié et donne alors naissance à la version 4.X. Le serveur possède alors un nouveau moteur de Servlets baptisé Catalina (Servlets 2.3 et JSP 1.2).

Tomcat 5.X est apparu récemment et implémente les Servlets 2.4 et JSP 2.0. Cette version apporte des nouveautés au niveau du monitoring (intégration de JMX - *Java Management Extension*) ainsi que plusieurs optimisations (mémoire,

configuration du serveur...). Tomcat 5.X intègre le support de la version Java 5.0. La dernière version de Tomcat 6.X permet l'utilisation de Java 6.0. Cette version repose sur les Servlets 2.5 et JSP 2.1.

Le serveur Jakarta Tomcat est développé depuis ses premières versions en Java. Les applications hébergées par Tomcat sont elles-mêmes écrites en Java, l'intégration est alors totale et robuste. Aujourd'hui, la version 6.X de Tomcat sait tirer profit des améliorations apportées à la plate-forme Java SE, notamment en terme de performance.

Installation de Tomcat

1. Quelle version choisir ?

Actuellement, Tomcat propose une version 6.X stable qui est supportée par la majorité des environnements de développement. La version 6.X de Tomcat utilise la spécification Java EE 5 ainsi que l'API Servlet 2.5 et l'API JSP 2.1. C'est cette version de Tomcat qui sera utilisée tout au long de ce guide et qui est parfaitement gérée par Eclipse pour les opérations de démarrage, d'arrêt et de redémarrage du serveur. Le serveur Tomcat 6.X est disponible en libre téléchargement sur le site Internet d'Apache à cette adresse : http://tomcat.apache.org

2. Installation sous Windows

Pour l'installation sous Windows, vous pouvez vous référer au premier chapitre de ce guide de développement d'applications Web en Java. La démarche à suivre est expliquée en détail avec une progression étape par étape.

3. Installation sous Linux

Comme indiqué dans le premier chapitre de ce guide, le serveur d'applications a besoin d'un JDK Java pour fonctionner correctement. Il est donc nécessaire d'avoir installé un JDK 1.5 ou JDK 1.6 fonctionnel avant de procéder à l'installation du serveur Tomcat. La version de Tomcat utilisée dans ce guide est *apache-tomcat-6.0.16* et peut être téléchargée à cette adresse : http://tomcat.apache.org/download-60.cgi.

Lors de l'installation, nous allons préparer le serveur de façon à ce qu'il puisse dialoguer par la suite avec le serveur Web. Le système Linux utilisé tout au long de ce guide est une version Debian (Etch) stable.

Vérifier les paquets installés

Avant toute chose, il est nécessaire de mettre à jour les paquets installés.

```
#apt-get update
#apt-get upgrade
```

Mise en place du serveur Web Apache

Apache sera utilisé par la suite lors du déploiement d'un exemple complet sur un serveur en production.

Les serveurs déjà installés sur le système sont désinstallés afin de paramétrer l'ensemble de manière stable et uniforme.

```
#apt-get remove --purge apache
#apt-get remove --purge apache2
#apt-get remove --purge apache-perl
#apt-get remove --purge apache-ssl
```

Nous allons ensuite, détruire les répertoires des précédentes installations.

```
#rm -rf /etc/apache
#rm -rf /etc/apache2
#rm -rf /etc/apache-ssl
#rm -rf /etc/apache-perl
```

Nous procédons à l'installation du serveur Web Apache (1.3 ou 2.0 au choix).

```
#apt-get install apache
```

L'étape suivante consiste à installer le JDK 1.5 ou JDK 1.6 Java. Il existe de très fortes dépendances de la plate-forme Java EE 5 et Tomcat 6.X avec Java 5. Il est donc impératif d'installer un serveur Tomcat 6 sur un JDK 5.0 au minimum.

Installation du JDK 1.5/1.6

Pour l'installation du jdk sous Linux, vous pouvez vous référer au chapitre Objectifs et spécifications de Java EE de ce guide de développement d'applications Web en Java.

Installation de Tomcat 6.0.X

L'installation de Tomcat à partir d'une archive est assez simple. Il est nécessaire de télécharger sur le site d'Apache-Tomcat l'archive du serveur au format *.bin*. La version utilisée pour ce guide est la suivante : *apache-tomcat-6.0.16.tar.gz*.

Avec la version 6.0.X de Tomcat, la partie administration n'est pas incluse mais fait partie d'une option. Il est donc important de télécharger également la partie administration du serveur : *apache-tomcat-6.0.16-admin.tar.gz*.

Après le téléchargement, il faut copier l'archive dans le répertoire des sources Linux (*/usr/local/src*).

```
#cp apache-tomcat-6.0.16.tar.gz /usr/local/src
```

Il faut ensuite détarrer l'archive dans le répertoire */usr/local*.

```
#tar -xzf apache-tomcat-6.0.16.tar.gz -C /usr/local
```

Il faut maintenant créer un lien pour référencer Tomcat directement.

```
#cd /usr/local
#ln -s apache-tomcat-6.0.16 ./tomcat
```

Il sera donc possible de référencer Tomcat de cette manière sans se soucier de la version utilisée.

```
#cd /usr/local/tomcat
```

Désormais, il est nécessaire de créer un utilisateur système, dédié à Tomcat et d'indiquer que cet utilisateur est le propriétaire de Tomcat.

```
#groupadd tomcat
#useradd -g tomcat -d /usr/local/tomcat tomcat
#chown -R tomcat:tomcat apache-tomcat-6.0.16
#chmod 770 apache-tomcat-6.0.16
```

Le serveur de base est désormais opérationnel. Nous pouvons alors vérifier son fonctionnement en lançant le serveur et en ouvrant un navigateur à l'adresse suivante : http://localhost:8080/.

```
#/usr/local/tomcat/bin/startup.sh
```

Pour un fonctionnement correct et afin d'éviter de positionner les variables d'environnement à chaque démarrage ou dans les fichiers spécifiques, il est nécessaire de créer un script de démarrage et d'arrêt de Tomcat.

```
#! /bin/bash
# LAFOSSE JEROME
NAME="apache-tomcat6.0.16"

#variables d'environnement
TOMCAT_HOME=/usr/local/tomcat
CATALINA_HOME=/usr/local/tomcat
JAVA_HOME=/usr/local/jdk
CATALINA_OPTS="-Dfile.encoding=iso8859-1"
TOMCAT_USER=tomcat
LC_ALL=fr_FR
#exporter les variables
export TOMCAT_HOME CATALINA_HOME JAVA_HOME CATALINA_OPTS TOMCAT_USER LC_ALL
cd $TOMCAT_HOME/logs

case "$1" in
 start)
 echo -ne "Demarrer $NAME.\n"
 /bin/su $TOMCAT_USER $TOMCAT_HOME/bin/startup.sh
 ;;

 stop)
 echo -ne "Arreter $NAME.\n"
 /bin/su $TOMCAT_USER $TOMCAT_HOME/bin/shutdown.sh
```

```
;;

 *)
 echo "Usage : /etc/init.d/tomcat {start|stop}"
 exit 1
 ;;
 esac
 exit 0
```

Il est aussi possible de modifier légèrement le script afin de gérer la partie redémarrage du serveur et ainsi d'éviter de réaliser des arrêts/démarrages pour un simple redémarrage.

```
...
restart)
echo -ne "Redemarrer $NAME.\n"
/bin/su $TOMCAT_USER $TOMCAT_HOME/bin/shutdown.sh
sleep 7
/bin/su $TOMCAT_USER $TOMCAT_HOME/bin/startup.sh
;;
...
```

Il est nécessaire de copier ce script dans le répertoire dédié aux services de la machine Linux et de le rendre exécutable.

```
#cp tomcat /etc/init.d/
#chmod 755 /etc/init.d/tomcat
```

Le propriétaire de ce script doit être le super utilisateur root car c'est le seul utilisateur qui a le droit de démarrer des services.

```
#chown root:root /etc/init.d/tomcat
```

Nous pouvons désormais tester le fonctionnement du serveur en utilisant les commandes suivantes :

```
#/etc/init.d/tomcat start
#/etc/init.d/tomcat stop
```

Pour activer le script au démarrage du système afin de lancer Tomcat dès l'amorçage du système, il est possible d'utiliser la commande : `#chkconfig - -add /etc/init.d/tomcat`

À ce stade de l'installation le serveur Tomcat est opérationnel. Il est désormais possible de lancer un navigateur et de se connecter sur la page d'accueil du serveur à l'adresse suivante : http://localhost:8080/

Suivre les fichiers journaux

Il est souvent très utile de suivre les traces lors d'un démarrage ou arrêt du serveur. De même, les fichiers journaux permettent de renseigner les administrateurs sur les problèmes de connexion au SGBD, les exceptions, les traces de programmation... Les fichiers journaux de Tomcat sont stockés dans le répertoire : */usr/local/tomcat/logs*.

Une bonne habitude est d'ouvrir en permanence une console avec la commande suivante qui trace en temps réel le fichier de logs du serveur.

```
#tail -f /usr/local/tomcat/logs/catalina.out&
```

Le nom donné au service de Tomcat 6.0.X (moteur de Servlets) est Catalina.

4. Mise en place de la partie administration de Tomcat

Par défaut, la partie administration du serveur n'est pas installée. Si nous cliquons sur le lien **Tomcat Administration** ou que nous accédons à l'adresse suivante : http://localhost:8080/admin/ le message suivant est affiché : *Tomcat's administration web application is no longer installed by default. Download and install the "admin" package to use it*. Nous allons donc procéder à l'installation de la partie administration en utilisant la librairie *apache-tomcat-6.0.16-admin.tar.gz*.

Il est nécessaire de commencer l'installation en détarrant l'archive dans le répertoire des sources Linux.

```
#tar -xzf apache-tomcat-6.0.16-admin.tar.gz /usr/local/src
```

Le répertoire *apache-tomcat-6.0.16* va être créé. Il faut alors copier tout le contenu de ce répertoire (qui possède plus de fichiers que la version d'origine) dans le répertoire d'installation de Tomcat.

```
#cd /usr/local/src/apache-tomcat-6.0.16
#cp -r * /usr/local/tomcat/
```

Le serveur est presque opérationnel, il faut juste remettre les bons droits sur les fichiers et répertoires et redémarrer le serveur (arrêt puis démarrage).

```
#chown -R tomcat:tomcat /usr/local/apache-tomcat-6.0.16
#chmod 770 apache-tomcat-6.0.16
#/etc/init.d/tomcat stop
#/etc/inid.t/tomcat start
```

Nous pouvons désormais nous connecter à l'adresse suivante http://localhost:8080/admin/ mais il est nécessaire de créer un utilisateur manager Tomcat.

Pour cela, il est nécessaire d'éditer le fichier : */usr/local/tomcat/conf/tomcat-users.xml*.

```
#vi /usr/local/tomcat/conf/tomcat-users.xml
```

La ligne suivante est ajoutée, elle permet de créer un utilisateur avec l'identifiant *admin* et le mot de passe *admin* puis on redémarre le serveur.

```
<user username='admin' password='admin' roles='admin,manager'/>
```

```
#/etc/init.d/tomcat stop
#/etc/init.d/tomcat start
```

En fait, la partie administration de Tomcat est une application/webapp fournie sous forme de fichiers *.tar* ou *.gz*. La seule différence avec les autres webapps est que le projet n'est pas livré sous la forme d'une archive *.war* directement déployable. Il faut donc copier les fichiers de configuration au bon endroit.

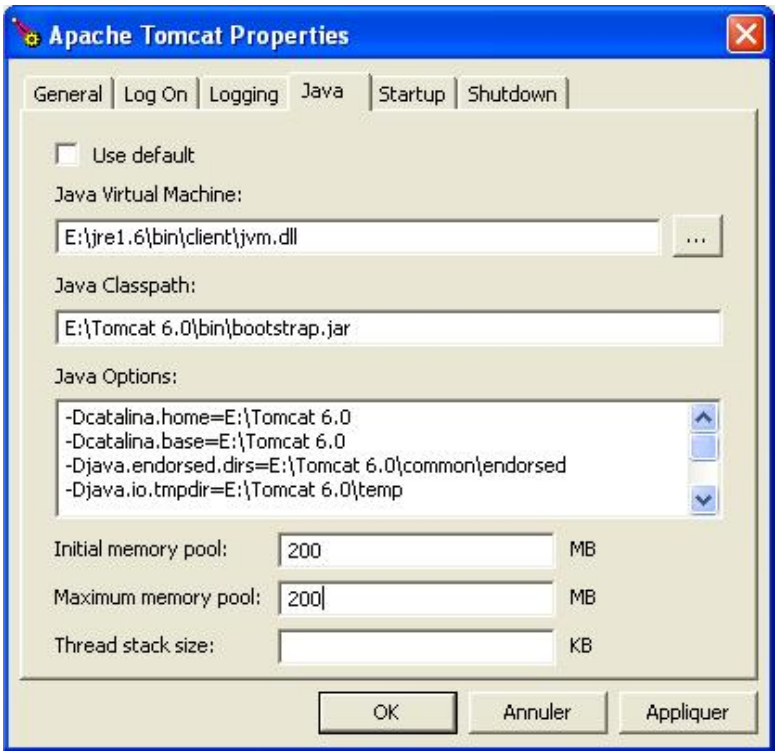
Pour éviter certains problèmes au démarrage du serveur, il est nécessaire d'éditer le fichier */usr/local/tomcat/conf/server.xml* et de mettre en commentaire la ligne suivante qui est appelée APR pour *Apache Portable Runtime* et qui empêche parfois Tomcat de fonctionner correctement : `<Listener className='org.apache.catalina.core.AprLigecycleListener'/>`

5. Augmenter la mémoire allouée à Tomcat

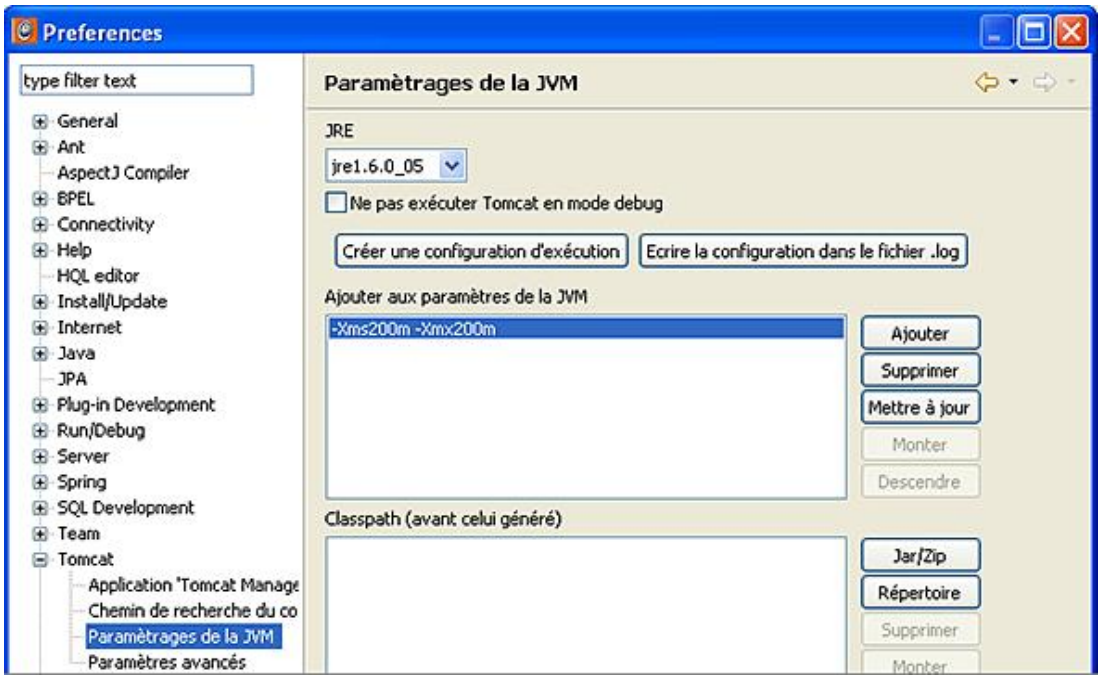
Tomcat est développé en langage Java et comme tout programme qui utilise ce langage, il est assez gourmand en terme de mémoire. Il est donc parfois nécessaire d'augmenter la mémoire allouée à Tomcat afin qu'il puisse compiler plus rapidement les pages et répondre au plus vite aux requêtes clients (même en phase de développement un serveur rapide est plus souple).

Sous Windows

Sous Windows, la mémoire allouée à Tomcat peut être paramétrée avec la console de gestion située dans le systray. Il faut ouvrir l'onglet **Java** et paramétrer les champs **Initial memory pool** et **Maximum memory pool**. Pour l'utilisation de ce guide et de ses exemples, 200 Mo seront alloués à Tomcat.



Une autre solution pour la gestion de la mémoire allouée à Tomcat est d'utiliser Eclipse et l'onglet **Windows - Preferences**. Dans la partie réservée à Tomcat, il est possible de préciser la mémoire initiale (*-Xms*) et la mémoire maxi (*-Xmx*). Dans cet exemple, 200 Mo sont alloués à Tomcat. Il est important de ne pas allouer trop de mémoire par rapport aux possibilités de la machine afin d'éviter des blocages lors de déploiements/codages.



Sous Linux

Pour augmenter la mémoire allouée à Tomcat sous Linux, il est nécessaire d'éditer le fichier de configuration */usr/local/tomcat/bin/catalina.sh* et d'ajouter la ligne suivante en début de fichier. Dans cet exemple, 2 Go de RAM sont alloués à Tomcat pour un serveur en production.

```
JAVA_OPTS="-Xms2048m -Xmx2048m"
```

Coupler Tomcat et le serveur Web Apache

1. Présentation

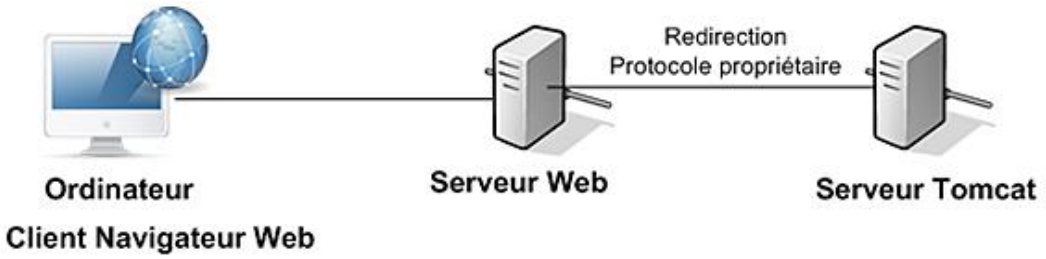
Si nous ne souhaitons pas installer un serveur en production, cette partie est alors facultative. En effet, elle ne permet pas d'améliorer l'environnement de développement mais elle est recommandée pour un déploiement sur un serveur en production.

Dans une architecture en production, il est recommandé d'utiliser un serveur Web en frontal d'un serveur d'applications. Ces recommandations sont également appliquées dans le cas de l'utilisation d'un conteneur Web comme Tomcat. L'utilisation d'un serveur Web en frontal est nécessaire pour des raisons de performance, de sécurité et de configurabilité.

- **Performance** : le moteur HTTP de Tomcat est beaucoup plus lent que le moteur HTTP d'un serveur Web dédié à cette tâche. Le serveur Web permet de délivrer les contenus statiques comme les pages HTML, le code JavaScript, les images du site, les feuilles de style... Tomcat sera utilisé alors pour servir uniquement les contenus dynamiques en Java.

- **Sécurité** : le serveur Web est utilisé en frontal et isole le conteneur Web d'Internet. Le conteneur est alors au plus près des données et il est moins sollicité pour des services simples.

- **Configurabilité** : un serveur Web comme Apache dispose d'une plus grande palette de services (point de vue HTP) que Tomcat (*.htaccess*, gestion des droits, urlrewriting, alias, annuaire...).



2. Un connecteur pour l'intégration du serveur Web

L'intégration d'un serveur Tomcat avec un serveur Web se fait au travers d'un connecteur configuré au sein de Tomcat et d'une extension ajoutée au serveur Web. Un connecteur Tomcat est une classe Java qui supporte un protocole réseau spécifique et propriétaire. La librairie d'extension du serveur Web est chargée dynamiquement par le serveur Web lors de son démarrage et permet un dialogue entre les deux serveurs. Plusieurs connecteurs existent comme le module *mod_jserv* pour le serveur JServ et le module *mod_webapp* pour Tomcat 4.X. Ces modules sont désormais abandonnés au profit du connecteur JK.

Le **connecteur JK** utilise le protocole AJP (*Apache JServ Protocol*) dans sa version 1.3 (AJP13). Ce connecteur est plus performant mais il offre également le support de plusieurs systèmes d'exploitation, de plusieurs serveurs Web (Apache, IIS, Lotus Domini) et du protocole HTTPS. Ce connecteur est aujourd'hui la référence pour le couplage d'un serveur d'applications avec un serveur Web.

a. Fonctionnement

Le connecteur JK utilise donc le protocole AJP13 et nécessite l'installation du module *mod_jk* pour fonctionner avec Apache. Le connecteur JK permet ainsi d'utiliser le serveur Web en frontal et de déléguer certaines tâches au serveur Tomcat.

Les requêtes des clients sont envoyées au serveur Web Apache qui retourne alors directement les contenus statiques comme les images, JavaScript, pages HTML, ... Pour les requêtes avec du contenu dynamique, le module *mod_jk* du serveur Web est alors sollicité et délègue certaines tâches au serveur d'applications Tomcat.

La configuration de Tomcat avec un serveur Web utilise la notion de *worker* ou travailleur. Un travailleur est lié à une instance de serveur Tomcat. Un travailleur est caractérisé par un nom d'hôte ou une adresse IP et un numéro de port (comme une socket/prise). Le travailleur AJP13 représente une instance de Tomcat en fonctionnement et il est utilisé comme plug-in pour le serveur Apache. Le module *mod_jk* agit alors comme un routeur de requêtes vers le serveur Tomcat.

b. Installation du module mod_jk

L'extension d'Apache qui supporte le connecteur JK est le module *mod_jk*. Ce module est livré sous forme de binaires (ou code source).

Le module *mod_jk* fonctionne sur le port 8009 de Tomcat. Par défaut, le fichier de configuration de Tomcat *server.xml* propose un connecteur AJP13 qui fonctionne sur le port 8009 : `<Connector port='8009' enableLookups='false' redirectPort='8443' protocol='AJP/1.3' />`

Avant de commencer l'installation, il faut mettre en place sous Linux Debian, le paquet *apache-dev* qui permet de compiler les modules d'Apache.

```
#apt-get install apache-dev
```

L'installation de Tomcat 6.X avec une intégration pour un serveur Web ne requiert aucune configuration particulière. Pour utiliser *mod_jk*, il faut un connecteur compatible configuré dans le fichier *server.xml* de l'instance de serveur Tomcat 6.X. Cette configuration existe désormais par défaut avec Tomcat 6.X et propose un connecteur qui fonctionne sur le port 8009. Ensuite, il est nécessaire de télécharger le connecteur (la dernière version) sur le site d'Apache-Tomcat (http://tomcat.apache.org). Le lien suivant est utilisé : *The Apache Jakarta Tomcat Connector*. Pour ce guide, c'est la version *jakarta-tomcat-connectors-1.2.15-src.tar.gz* du connecteur qui a été utilisée. L'installation commence en copiant la bibliothèque source dans le répertoire des sources Linux.

```
#cp jakarta-tomcat-connectors-1.2.15-src.tar.gz /usr/local/src
```

Ensuite, l'archive est détarré(e) :

```
#tar -xzf jakarta-tomcat-connectors-1.2.15-src.tar.gz
```

Il faut ensuite compiler le fichier source présent dans le répertoire */usr/local/src/jakarta-tomcat-connectors-1.2.15-src/jk/native*.

```
#./configure -with-apxs=/usr/bin/apxs
#make
```

L'option *-with-apxs* permet de préciser l'emplacement de la commande apxs qui est utilisée pour construire les modules d'Apache. Un fichier source compilé va alors être généré. Ce fichier source est alors copié dans le répertoire des librairies Apache.

```
#cp apache-1.3/mod_jk.so.0.0.0 /usr/lib/apache/1.3/mod_jk.so
```

Il faut alors positionner les droits corrects sur le module.

```
#chown root:root /usr/lib/apache/1.3/mod_jk.so
#chmod 644 /usr/lib/apache/1.3/mod_jk.so
```

c. Configurer le module mod_jk

Il nous reste à configurer le fichier de gestion d'Apache pour qu'il charge dynamiquement lors de son démarrage le module *mod_jk*. Pour cela, il faut éditer selon la version d'Apache le fichier *httpd.conf* ou *modules.conf* et ajouter les lignes suivantes :

```
LoadModule jk_module /usr/lib/apache/1.3/mod_jk.so
JkWorkersFile /etc/apache/workers.properties
JkLogFile /usr/local/tomcat/logs/mod_jk.log
JkLogLevel warn
```

La première opération permet de réaliser le chargement du module *mod_jk* dans le serveur Apache (*LoadModule*) . La directive *JkWorkersFile* permet de spécifier l'emplacement du fichier de configuration du module. La directive *JkLogFile* précise l'emplacement d'un fichier journal réservé au module et la directive *JkLogLevel* indique le type de messages enregistrés dans ce fichier.

Il ne reste maintenant plus qu'à configurer le serveur Web Apache et à créer le fichier *workers.properties*. Il faut commencer par éditer le fichier de configuration d'Apache */etc/apache/httpd.conf* puis ajouter les lignes suivantes pour notre hôte virtuel.

```
<VirtualHost *>
ServerName monserveur.com
#les parties statiques de mon application sont gérées par Apache
Alias /images /usr/local/tomcat/webapps/monapplication/images
Alias /css /usr/local/tomcat/webapps/monapplication/css
DocumentRoot /usr/local/tomcat/webapps/monapplication
#les requêtes ne sont transmises à Tomcat que pour les servlets et JSP
<Location "/*.jsp">
JkMount worker1
</Location>
<Location "/*.do">
JkMount worker1
</Location>
#pages d'accueil autorisées
DirectoryIndex index.html index.htm index.jsp
</VirtualHost>
```

Vous remarquez que le serveur sera capable de définir la page *index.jsp* comme page d'accueil de l'application. De même, tous les contenus autres que les Servlets (extension *.do*) et les JSP (extension *.jsp*) seront traités par le serveur Web qui est plus approprié.

La directive *JkMount* est très importante car c'est elle qui permet au serveur Apache d'accéder aux applications Tomcat. Elle permet en effet, de spécifier un travailleur pour l'accès au contexte. Il y aura donc une redirection de requêtes utilisateur à destination du travailleur. La directive *JkUnMount* permet de réaliser l'inverse et donc de ne pas rediriger les requêtes utilisateurs à destination de ressources particulières.

Exemple :

```
JkUnMount /usr/local/tomcat/webapps/monapplication/mesimages/*.gif
```

D'autres directives d'Apache sont utilisables avec le *mod_jk*. *JkAutoAlias* permet de réaliser un alias du répertoire de Tomcat sur le répertoire de données Apache, *JkLogStampFormat* permet de gérer le format de la date dans le fichier journal du module, *JkExtractSSL* permet de transmettre les informations SSL vers le serveur Tomcat (état *on* par défaut)...

d. Créer le fichier de configuration du travailleur

Une fois le fichier de configuration d'Apache modifié, il faut ensuite créer le fichier de configuration du travailleur indiqué par la directive *JkWorkersFile* dans le fichier d'Apache. Ce fichier permet de gérer la communication entre le serveur Web et le serveur Tomcat. Le fichier utilisé nommé *workers.properties* porte l'extension *.properties* qui est un format très utilisé avec les technologies Java. Ces fichiers sont composés d'un ensemble de paires clé/valeur.

La syntaxe pour le fichier *workers.properties* est la suivante :

```
worker.<nom_du_travailleur>.<directive>=<valeur>
```

La syntaxe de notre fichier est la suivante :

```
workers.tomcat_home=/usr/local/tomcat
workers.java_home=$(JAVA_HOME)
ps=/
#liste des travailleurs (il serait possible de placer plusieurs travailleurs
séparés par des virgules)
workers.list=worker1
#protocole de worker1
workers.worker1.type=ajp13
#nom d'hote pour worker1
workers.worker1.host=localhost
#port du connecteur JK pour worker1
workers.worker1.port=8009
#le facteur de charge
workers.worker1.lbfactor=50
#nombre de connexions AJP maintenues dans le cache
workers.worker1.cachesize=10
#temps pendant lequel la connexion est maintenue dans le cache
workers.worker1.cache_timeout=600
#ne pas couper les connexions inactives
workers.worker1.socket_keepalive=1
```

```
#temps d'expiration lors de la communication entre Apache et Tomcat
workers.worker1.socket_timeout=300
```

Ce fichier est créé dans le répertoire local d'Apache à savoir : */etc/apache/workers.properties* conformément à la directive *JkWorkersFile*. Les paramètres de configuration du fichier *workers.properties* sont, entre autres :

- *worker.list* : permet de spécifier une liste de travailleurs séparés par des virgules.
- *type* : permet de spécifier le type de travailleur.
- *host* : permet de spécifier le nom d'hôte ou adresse IP du serveur Tomcat à contacter pour le travail.
- *port* : indique le numéro de port du connecteur JK.
- *socket_timeout* : indique le temps d'expiration de la communication entre Apache et Tomcat. Si le serveur Tomcat ne répond pas dans le délai indiqué, *mod_jk* génère une erreur et recommence.
- *retries* : positionne le nombre de tentatives de connexions vers Tomcat en cas de non réponse de ce dernier (3 par défaut).
- *socket_keepalive* : permet d'éviter que le firewall coupe les connexions inactives (défaut 0).
- *recycle_timeout* : indique le nombre de secondes au-delà duquel le serveur coupe une connexion AJP en cas d'inactivité (défaut 0, bonne moyenne 300).
- *cachesize* : précise le nombre de connexions AJP maintenues en cache.
- *cache_timeout* : permet de spécifier combien de temps une connexion doit être maintenue dans le cache avant d'être fermée par *mod_jk* afin de réduire le nombre de processeurs de requêtes actifs sur le serveur Tomcat (défaut 0).
- *lbfactor* : permet de gérer le facteur de charge d'un travailleur dans le cas où la répartition de la charge est mise en œuvre par *mod_jk*. Cette valeur permet de préciser quel pourcentage de requêtes l'instance Tomcat sera amenée à traiter (défaut 1).

Une fois la configuration du connecteur JK terminée, il faut redémarrer les deux serveurs et tester l'accès à une Servlet ou une JSP pour notre hôte précédemment défini (http://monserveur.com/index.jsp).

Pour cela, sous Windows, il est possible d'éditer le fichier *C:\WINDOWS\system32\drivers\etc\hosts* et d'ajouter le nom de l'hôte virtuel.

```
127.0.0.1       localhost
127.0.0.1       monserveur.com
```

Sous Linux, nous pouvons réaliser la même opération en éditant le fichier */etc/hosts*.

```
127.0.0.1       localhost
127.0.0.1       monserveur.com
```

Désormais, lorsque l'url suivante : http://monserveur.com sera appelée dans un navigateur, c'est la machine locale (127.0.0.1) qui répondra aux requêtes du client.

```
#/etc/init.d/apache restart
#/etc/init.d/tomcat stop
#/etc/init.d/tomcat start
```

Si la page *.jsp* est affichée, l'installation est correcte. En effet, c'est l'hôte virtuel du serveur Web Apache qui est précisé par le nom de domaine (http://monserveur.com) et le fichier *.jsp* est exécuté par Tomcat par le biais du connecteur.

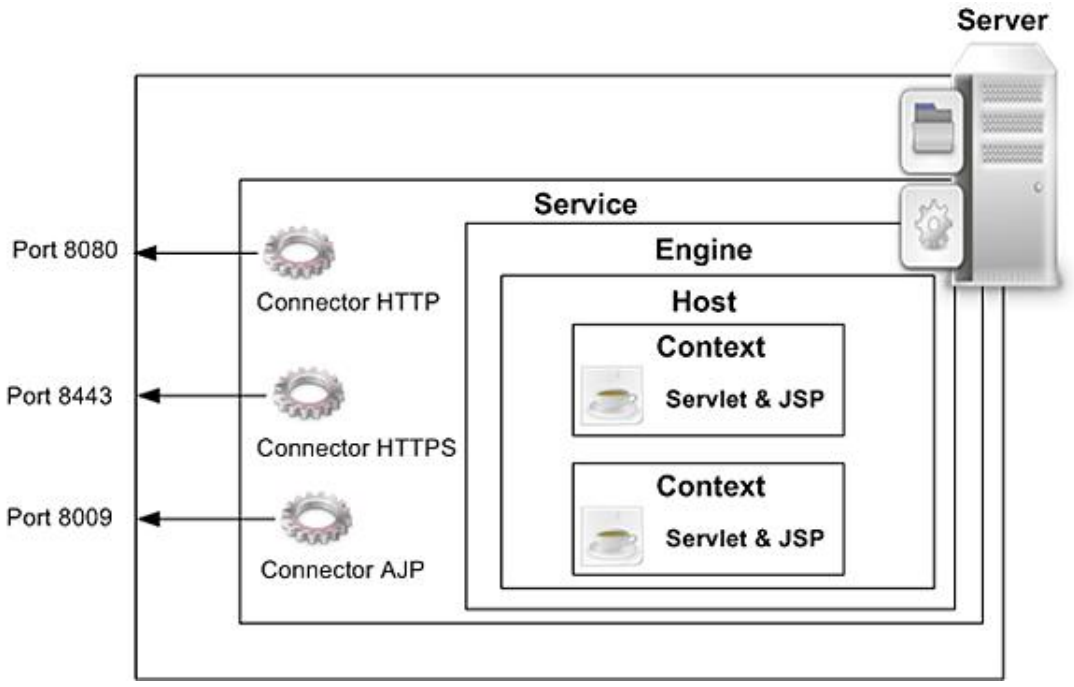
En cas de problème lors du test du connecteur, les fichiers journaux d'Apache et de *mod_jk* peuvent nous aider en indiquant les raisons des dysfonctionnements.

Architecture et configuration de Tomcat

Le serveur d'applications Tomcat utilise une architecture spécifique (spécifique aux projets Java EE) qu'il est nécessaire de bien maîtriser. Tomcat est livré pré-configuré, il est possible de l'utiliser comme cela sans avoir à modifier les fichiers de configuration, mais lors des développements et de la mise en production des applications, il sera nécessaire de bien contrôler l'administration du serveur.

1. Les composants de Tomcat

Tomcat est constitué d'un ensemble de composants dédiés à l'exécution d'un service particulier et précis. Les composants de Tomcat sont appelés des conteneurs (parce qu'ils contiennent eux aussi des composants). Il existe actuellement cinq types de conteneurs : *Server*, *Service*, *Engine*, *Host* et *Context*.



Tous ces conteneurs sont représentés dans le fichier de configuration du serveur *server.xml* qui est le principal fichier de configuration de Tomcat. Chaque conteneur est représenté par une balise XML en suivant une structure arborescente adaptée en conséquence.

2. Arborescence du serveur

La structure des fichiers au sein du répertoire d'installation est présentée dans cette image.

Parmi les répertoires présents, certains sont dédiés à la configuration du serveur et sont donc difficilement modifiables, d'autres peuvent être modifiés suivant les développements.

Le répertoire */bin* contient tous les scripts et fichiers indispensables au bon fonctionnement de Tomcat. Ces exécutables contiennent des scripts shell et des fichiers batch qui permettent de démarrer et d'arrêter le serveur sur les différentes plates-formes prises en charge. Les fichiers d'extension *.bat* sont utilisés sous Windows et les fichiers *.sh* sont utilisés sous Linux.

Le répertoire */lib* est le répertoire partagé de Tomcat. La totalité de son contenu est accessible à toutes les applications Web déployées sur le serveur. Les ressources placées dans ce répertoire peuvent être livrées au format *.class* et donc copiées dans le répertoire */lib/classes* ou sous forme de fichiers *.jar*. Ce répertoire */lib* sera très souvent utilisé pour placer le pilote JDBC de la base de données utilisée. Par exemple, lors du déploiement d'un projet, la librairie *mysql-connector-java-3.1.11-bin.jar* sera copiée dans le répertoire */lib* et sera donc utilisable par toutes nos classes du projet (ce qui est valable pour les librairies *mail.jar*, *xalan.jar*...). Dans le cas où plusieurs applications Web ont toutes besoin d'une même bibliothèque, il peut être plus judicieux de copier cette bibliothèque dans ce répertoire plutôt que dans chacune des applications.

Le répertoire */conf* contient tous les fichiers de configuration de Tomcat avec les quatre fichiers importants que sont *server.xml*, *tomcat-users.xml*, *web.xml* et *catalina.policy*.

Le répertoire */logs* contient tous les fichiers journaux du serveur Tomcat. Il est important de préciser que les fichiers journaux sont automatiquement remplacés tous les jours à minuit, et contiennent dans leur intitulé la date au format anglais. Il existe trois principaux types de fichiers journaux : les fichiers relatifs au serveur, les fichiers relatifs aux applications et aux noms d'hôtes.

Le répertoire */temp* est un répertoire temporaire pour les applications non déployées.

Le répertoire */webapps* est le répertoire par défaut d'installation des applications Java EE. Il contient par défaut des applications d'exemples ainsi que l'application *tomcat-docs* qui fournit la documentation du serveur. Il est tout à fait possible de référencer des applications Web qui ne se trouvent pas dans le répertoire */webapps*. Dans ce cas, il faut préciser le répertoire de l'application de façon précise dans le fichier de configuration *server.xml*.

Le répertoire */work* est utilisé pour le traitement des pages JSP et leur transformation en classes Java. Toutes les Servlets générées à partir de pages JSP seront stockées dans ce répertoire. Chaque application possède alors son propre sous-répertoire (ex : *work/Catalina/localhost/monapplication*) en suivant l'arborescence suivante : *work/<engine>/<host>/<context>* où *<engine>*, *<host>* et *<context>* représentent le nom des conteneurs *Engine*, *Host* et *Context* dans lesquels cette application est installée. Ce répertoire sera d'une grande utilité lors du débuggage de pages JSP. Le code transformé sera en effet accessible. De même, il sera parfois nécessaire de supprimer le contenu de ce répertoire pour notre hôte lorsque l'on voudra regénérer toutes les pages JSP.

Rappels XML

Les fichiers de configuration de Tomcat sont écrits avec le langage XML. Il est donc important de présenter la syntaxe et les balises de ce langage afin de configurer correctement le serveur d'applications. XML (*eXtended Markup Language*) dérive du langage SGML (*Standard Generalized Markup Language*) développé dans les années 80. SGML est un langage très complexe à apprendre et à utiliser. Une version plus simple de ce langage a été proposée pour la présentation de document Web : le HTML (*HyperText Markup Language*).

HTML

HTML est aujourd'hui le standard pour le développement Web. Il commence à être remplacé progressivement par le XHTML (*eXtend Hypertext Markup Language*) qui est assez similaire mais qui respecte les normes XML. HTML est ''un langage'' de description, il est lié à une DTD (*Document Type Definition*) qui permet de vérifier la syntaxe du langage.

XML

XML utilise la simplicité du HTML avec la souplesse de SGML. Le point commun le plus important entre SGML et XML est l'utilisation d'une DTD ou d'un schéma. Cette association n'est pas obligatoire et un fichier XML peut très bien se suffire à lui-même.

Dans un document XML, la mise en forme des données est complètement séparée des données. Les données (le contenu) sont séparées de l'apparence (le contenant). Il sera donc possible de fournir plusieurs types de sorties pour un même fichier de données (image, fichier HTML, fichier XML, fichier PDF...).

Les langages SGML, HTML/XHTML et XML sont en fait composés de balises qui peuvent être comparées à des mots du langage français. Par contre, il y a des règles à respecter pour l'utilisation de ces mots dans un document, ces règles sont appelées : grammaires.

XML permet de séparer le fond de la forme. Cela signifie qu'un document XML ne comporte que des données. Ainsi, pour produire un document HTML à partir d'un fichier XML, il est nécessaire de créer au moins deux fichiers, le premier pour les données et le second pour la mise en forme de ces données. Un troisième fichier peut parfois être utilisé, c'est une DTD ou un schéma permettant de définir les balises et la grammaire utilisées.

Bien formé

Un document XML bien formé est un document XML qui respecte certaines règles.

- Le document doit commencer par une déclaration XML (prologue) .

```
<?version="1.0" standalone="yes" encoding="iso-8859-1"?>
```

- Il ne doit exister qu'une seule balise racine.

```
<maracine>
<balise>donnée1</balise>
<balise>donnée2</balise>
...
</maracine>
```

- Les valeurs des attributs doivent être impérativement encadrées par des guillemets simples ou doubles.

```
<balise attribut1="valeur" attribut2='valeur'>
```

- Toute balise ouverte doit être fermée.

```
<balise>donnée</balise>
```

- Une balise vide doit être obligatoirement fermée.

```
<balise/>
<balise></balise>
```

- Les balises doivent être correctement imbriquées.

```
<baliseparent>
<baliseenfant>donnée</baliseenfant>
```

```
</baliseparent>
```

- Les noms des balises doivent commencer par une lettre ou ''_'', les autres caractères peuvent être des chiffres, des lettres, ''_'', ''.'' ou ''-''.

```
<_balise attribut28='valeur'/>
```

- Les noms des balises et des attributs doivent conserver une casse identique.

```
<mABaLise attriBuT='12'></mABaLise>
<mABaLise attriBuT='14'></mABaLise>
```

- Les noms des balises ne doivent pas commencer par *xml*.

- Le document doit contenir un ou plusieurs éléments. Si le document contient un seul élément, alors ce document sera composé du seul élément racine.

- Le caractère inférieur < est réservé à l'ouverture des balises.

```
<mabalise/>
```

- Les caractères inférieur <, esperluette &, supérieur >, quote ', et double quotes '' doivent être remplacés par leurs entités HTML ou numériques (< & >...).

Valide

Un document XML est dit valide lorsque celui-ci est bien formé et qu'il est conforme à une grammaire.

Structure d'un document XML

Un fichier XML est composé d'un prologue, d'un élément racine et d'un arbre. L'arbre est composé d'éléments imbriqués les uns dans les autres. Le prologue est constitué de la première ligne du document permettant d'indiquer que c'est un document XML et éventuellement de l'association à une DTD.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Élément racine -->
<dvds>
  <!-- Premier enfant -->
  <dvd>
    <!-- Élément enfant titre -->
    <titre>Les infiltrés</titre>
    <prix>17.99</prix>
    <duree>125</duree>
  </dvd>
  <dvd>
    <titre>Le cercle rouge</titre>
    <prix>14</prix>
    <duree>120</duree>
  </dvd>
  <dvd langue="en">
    <titre>Psychose</titre>
    <prix>22</prix>
    <duree>155</duree>
  </dvd>
</dvds>
```

Dans cet exemple, le prologue est indiqué par la première ligne du document *<?xml version="1.0" encoding="ISO-8859-1"?>*, il précise que le document est un fichier XML utilisant l'encodage *ISO-8859-1* (caractères standards du clavier). Dans le prologue, l'attribut *version* précise la version XML utilisée dans le document. L'attribut *encoding* permet de définir le jeu de caractères utilisé. Pour la France, le jeu de caractères standard est *ISO-8859-1*. Pour faire des pages en international (Japonais, Anglais, Français...), il faut utiliser l'encodage *Unicode UTF-8*. Un troisième attribut optionnel nommé *standalone* permet de préciser si oui ou non le fichier XML est lié à une DTD externe pour vérifier sa syntaxe.

- sans utilisation d'une DTD : `<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>`

- avec utilisation d'une DTD : `<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?> <!DOCTYPE dvds SYSTEM "dvds.dtd">`

L'élément *<dvds>* est l'élément racine du document. Il est lui-même constitué de trois éléments *<dvd>*. Dans chacun des éléments de cet exemple vous retrouvez un élément *<titre>*, *<prix>* et *<duree>*. L'élément *<dvd>* possède un attribut nommé langue.

Cette vue représente le code XML de la page précédente ouvert avec un navigateur

Une DTD peut être associée à ce document et permet alors de définir la structure du document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dvds SYSTEM "dvd.dtd"/>
<!-- Élément racine -->
<dvds>
  <!-- Premier enfant -->
  <dvd>
    <!-- Élément enfant titre -->
    <titre>Les infiltrés</titre>
    <prix>17.99</prix>
    <duree>125</duree>
  </dvd>
  <dvd>
    <titre>Le cercle rouge</titre>
    <prix>14</prix>
    <duree>120</duree>
  </dvd>
  <dvd langue="en">
    <titre>Psychose</titre>
    <prix>22</prix>
    <duree>155</duree>
  </dvd>
</dvds>
```

Lors de la déclaration d'une DTD, il faut préciser l'élément racine (*<dvds>* dans l'exemple) et le nom du fichier qui contient la grammaire du document (*dvd.dtd* dans l'exemple). Bien que facultative, une DTD permet de simplement vérifier la validité d'un document XML. Dans un document XML, les commentaires sont utilisés comme dans un document HTML avec la syntaxe suivante *<!-- mon commentaire -->*.

Explications

L'élément racine *<dvds>* est la base du document XML. Il doit être unique et englobe tous les autres éléments. Il s'ouvre juste après le prologue et se ferme à la fin du document. Les autres éléments forment la structure du document. Ce sont donc les branches et les feuilles de l'arborescence. Les éléments contenants sont appelés *élément parent* et les autres éléments imbriqués *élément enfant*. Les éléments peuvent contenir un ou plusieurs attributs. Chaque élément ne peut contenir qu'une fois le même attribut. Un attribut est composé d'un nom et d'une valeur. Un attribut ne peut être présent que dans une balise ouvrante d'un élément (et pas dans la fermante).

Certains caractères ont un sens particulier en XML, il est nécessaire de trouver un remplaçant quand il faut insérer ces caractères dans le document. Il faut alors avoir recours aux entités. Les caractères réservés en XML sont les suivants :

Caractère	Entité
&	&amp ;
<	&lt ;
>	&gt ;
”	&quot ;
’	&aquot ;

Pour les lettres accentuées, il faudra parfois utiliser les entités numériques du type *&#numero;* (où *numero* est une valeur décimale). Par exemple, le caractère codé *é* peut être remplacé par *é*.

Syntaxe d'une DTD

Dans l'exemple précédent, le fichier est correctement affiché dans un navigateur. Le document est valide et bien formé, il peut désormais être utilisé.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dvds SYSTEM "dvd.dtd"/>
<!-- Élément racine -->
<dvds>
  <!-- Premier enfant -->
  <dvd>
    <!-- Élément enfant titre -->
    <titre>Les infiltrés</titre>
    <prix>17.99</prix>
    <duree>125</duree>
  </dvd>
  <dvd>
    <titre>Le cercle rouge</titre>
    <prix>14</prix>
    <duree>120</duree>
  </dvd>
  <dvd langue="en">
    <titre>Psychose</titre>
    <prix>22</prix>
    <duree>155</duree>
  </dvd>
</dvds>
```

Rien ne nous empêche de ne plus avoir de prix et de durée pour chaque dvd. La structure du document est correcte du point de vue des balises mais la grammaire ne l'est pas. C'est là qu'intervient la DTD (fichier dvd.dtd ci-dessous) qui permet de définir la syntaxe que le document XML devra respecter.

Exemple 1 :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT dvds (dvd*)>
<!ELEMENT dvd (titre, prix, duree)>
```

```
  <!ATTLIST dvd   type (thriller | policier | horreur | théâtre) #IMPLIED
   langue CDATA "fr"
   >
<!ELEMENT titre (#PCDATA)>
<!ELEMENT prix (#PCDATA)>
<!ELEMENT duree (#PCDATA)>
```

Exemple 2 :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dvds SYSTEM "dvd.dtd"/>
<!-- Élément racine -->
<dvds>
  <!-- Premier enfant -->
  <dvd>
    <!-- Élément enfant titre -->
    <titre>Les infiltrés</titre>
  </dvd>
  <dvd>
    <titre>Le cercle rouge</titre>
  </dvd>
  <dvd langue="en">
    <titre>Psychose</titre>
  </dvd>
</dvds>
```

Le fichier XML est associé à cette DTD. Il devra donc respecter la grammaire suivante : la balise racine doit être <*dvds*>, la balise racine est composée de balises <*dvd*> et chaque balise <*dvd*> contient une balise <*titre*>, <*prix*> et <*duree*>. La balise <*dvd*> contient deux attributs optionnels type et langue. L'attribut *type* ne peut contenir que les valeurs 'thriller, policier, horreur et théâtre'. La balise langue contient par défaut la valeur *'fr'*.

Le mot clé *SYSTEM* dans le fichier XML indique que le fichier DTD se trouve sur l'ordinateur local et qu'il est disponible en accès privé uniquement. Le mot clé *PUBLIC* indique qu'une ressource est disponible pour tous (accès public) sur un serveur Web distant.

```
<!DOCTYPE dvds SYSTEM "dvd.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Les éléments

Une déclaration d'élément est de la forme :

```
<!ELEMENT nom type_element>
```

nom est le nom de l'élément et *type_element* est le type auquel l'élément est associé. Un élément peut être de type texte, vide (*EMPTY*), séquence ou choix d'éléments. Dans ces deux derniers cas, la liste des éléments enfants est indiquée.

Un élément texte est précisé par *#PCDATA*.

```
<!ELEMENT titre (#PCDATA)>
```

Un élément vide utilise le mot clé *EMPTY*.

```
<!ELEMENT titre EMPTY>
```

Dans le document XML, l'élément vide sera représenté par : <*monelement/*>. Un élément vide peut par contre posséder des attributs.

Lors de la déclaration de séquence ou de choix d'éléments, une indication d'occurence (?, + ou *) peut être attribuée à chaque élément enfant.

```
<!ELEMENT elt0 (elt1, elt2?, elt3+, elt*)>
```

- *elt1* ne contient aucune indication d'occurrence, il doit donc apparaître une seule et unique fois dans l'élément *elt0* (1 et 1 seul).
- *elt2* a pour indication d'occurrence ?, l'élément doit apparaître au maximum une fois et il peut ne pas apparaître

du tout (0 ou 1).

- *elt3* a pour indication d'occurrence +, l'élément doit apparaître au moins une fois et autant de fois que l'auteur le désire (1 ou plusieurs).

- *elt4* a pour indication d'occurrence *, l'élément doit apparaître autant de fois que l'auteur le désire (il peut ne pas apparaître du tout) (0 ou plusieurs).

Les séquences d'éléments

Une séquence d'éléments est une liste ordonnée d'éléments devant apparaître comme éléments enfants de l'élément qui est en train d'être défini. Ce dernier ne pourra contenir aucun autre élément que ceux figurant dans la séquence. Cette liste est composée d'éléments séparés par des virgules et est placée entre parenthèses. Chaque élément doit être déclaré par ailleurs dans la DTD. Dans le fichier XML, les éléments doivent apparaître dans l'ordre de la séquence.

```
<!ELEMENT elt0 (elt1, elt2, elt3)>
```

Il est possible bien sûr d'utiliser les indicateurs d'occurrence.

```
<!ELEMENT elt0 (elt1, elt2?, elt3+, elt*)>
```

La déclaration d'attributs

Des attributs peuvent être présents dans un document XML, la DTD permet donc de définir des contraintes sur ces attributs. Le mot clé de déclaration d'un attribut est *ATTLIST*. Chaque attribut peut être requis, optionnel ou fixe et avoir une valeur par défaut.

Un attribut peut avoir une valeur par défaut.

```
<!ELEMENT elt(...)>
 <!ATTLIST elt attr CDATA "valeur">
```

Un attribut peut être obligatoire.

```
<!ELEMENT elt (...)>
 <!ATTLIST elt attr CDATA #REQUIRED>
```

Un tel attribut est obligatoire. Son absence déclenche une erreur du vérificateur syntaxique sur le fichier XML.

Un attribut peut être optionnel.

```
<!ELEMENT elt (...)>
 <!ATTLIST elt attr CDATA #IMPLIED>
```

Un attribut peut être fixe.

```
<!ELEMENT elt (...)>
 <!ATTLIST elt attr CDATA #FIXED "valeur">
```

L'attribut ne peut donc prendre qu'une seule valeur fixée.

Exemples concrets

La syntaxe XML a été présentée mais il est important de montrer son utilisation avec un simple fichier XHTML et avec un fichier de configuration d'une application Tomcat.

Ci-dessous, un exemple d'un fichier simple en XHTML. Nous remarquons l'utilisation d'une DTD (grammaire) *PUBLIC* proposée par le serveur Web du W3C. Nous pouvons également noter que chaque balise est fermée, correctement imbriquée et que certains éléments possèdent des attributs (*<img/>*).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head><title>Mon Titre</title></head>
<body>
<h1>Ma page</h1>
<img src="monimage.png" width="300"/>
```

```
</body>
</html>
```

Une application déployée avec Tomcat est configurée par l'intermédiaire d'un fichier nommé *web.xml*. Une syntaxe possible de ce fichier est la suivante :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd"
>
<web-app>
    <description>Une application déployée avec Tomcat</description>

    <servlet>
        <servlet-name>DisplaySource</servlet-name>
        <display-name>DisplaySource</display-name>
        <description>display source of sample jsp pages
</description>
        <servlet-class>org.displaytag.sample.DisplaySourceServlet
</servlet-class>

    <servlet-mapping>
        <servlet-name>DisplaySource</servlet-name>
<url-pattern>*.source</url-pattern>
    </servlet-mapping>

    <mime-mapping>
        <extension>css</extension>
        <mime-type>text/css</mime-type>
    </mime-mapping>

<welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

    <error-page>
        <error-code>404</error-code>
        <location>/404.jsp</location>
    </error-page>
</web-app>
```

Nous remarquons que ce fichier de configuration d'une application Tomcat doit être conforme à la DTD *PUBLIC* de Sun Microsystems (<*!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd*">).

Dans un éditeur tel que Eclipse, la syntaxe du fichier de configuration sera donc vérifiée en temps réel grâce à une connexion Internet. Par exemple, il n'est pas possible de placer les balises <*welcome-file-list*> après les balises <*error-page*> ce qui est très utile et permet d'augmenter la fiabilité du projet.

Les fichiers de configuration Tomcat

Le fichier server.xml

Le principal fichier de configuration du serveur Tomcat s'appelle *server.xml* et se trouve dans le répertoire */conf* du serveur. Il fournit à Tomcat tous les paramètres nécessaires pour le fonctionnement, les ports à écouter, les hôtes virtuels, les instances de processeur HTTP et les classes à utiliser pour gérer les connexions entrantes. Ce fichier est au format XML et possède une balise de déclaration XML (prologue) mais il n'est lié à aucun fichier de validation DTD ou schéma. Ce fichier est donc bien formé mais non valide. Comme tout fichier XML, le langage est sensible à la casse et le fichier *server.xml* doit utiliser des balises commençant par une majuscule suivie de lettres en minuscules. Lors du démarrage, Tomcat vérifie la syntaxe des éléments déclarés dans ce fichier. Tomcat comprend un processus principal, le serveur qui contient lui-même plusieurs sous-composants qui sont utilisés pour traiter les requêtes.

```
<?xml version="1.0" encoding="UTF-8"?>
<Server>
  <Listener className="org.apache.catalina.mbeans.Global
ResourcesLifecycleListener"/>
  <Listener className="org.apache.catalina.storeconfig.Store
ConfigLifecycleListener"/>
  <Listener className="org.apache.catalina.mbeans.ServerLife
cycleListener"/>
   <GlobalNamingResources>
    <Environment
      name="simpleValue"
      type="java.lang.Integer"
      value="30"/>
    <Resource
      auth="Container"
      description="User database that can be updated and saved"
      name="UserDatabase"
      type="org.apache.catalina.UserDatabase"
      pathname="conf/tomcat-users.xml"
      factory="org.apache.catalina.users.MemoryUserDatabaseFactory"/>
  </GlobalNamingResources>
  <Service
      name="Catalina">
    <Connector
        port="8080"
        redirectPort="8443"
        minSpareThreads="25"
        connectionTimeout="20000"
        maxSpareThreads="75"
        maxThreads="150"
        maxHttpHeaderSize="8192">
    </Connector>
    <Connector
        port="8009"
        redirectPort="8443"
        protocol="AJP/1.3">
    </Connector>
    <Engine
       defaultHost="localhost"
       name="Catalina">
     <Realm className="org.apache.catalina.realm.UserDatabaseRealm"/>
      <Host
          appBase="webapps"
          name="localhost">
      </Host>
    </Engine>
  </Service>
</Server>
```

L'élément <*Server*> est la racine du fichier *server.xml*. Il représente l'instance de serveur Tomcat. Cet élément utilise trois attributs et contient tous les autres éléments qui contrôlent le serveur. Le premier attribut facultatif est *className*. Cet attribut correspond à la classe à utiliser en tant que serveur Tomcat. Tomcat utilise généralement l'implémentation par défaut. Les deux autres attributs sont *port* et *shutdown*. Ils contrôlent le service pour écouter les commandes d'arrêt du serveur.

L'élément<*Service*> regroupe les éléments qui permettent la connectivité au serveur ainsi que le moteur d'exécution de Tomcat. Le seul attribut obligatoire de cette balise est *name*. Cet attribut permet d'affecter un nom au service qui s'affiche dans les fichiers journaux. Le nom par défaut de cet attribut est *Catalina*.

L'élément<*Executor*> permet de gérer les threads (sous-processus) à l'intérieur de la machine virtuelle Java. Chaque thread est dédié au traitement d'une requête client et à l'envoi de sa réponse. Pour éviter les créations et suppressions inutiles de threads, Tomcat 6.X utilise un mécanisme nommé pool de threads. Plutôt que de détruire un thread après traitement d'une requête client, celui-ci est recyclé. Le pool de threads est dimensionné avec une valeur initiale qui permet de définir combien de threads doivent être créés dedans et le nombre maximum de threads. Ces deux paramètres sont respectivement configurés à partir des attributs *minSpareThreads* et *maxThreads* de l'élément <*Executor*>.

L'élément<*Connector*> : cet élément fils de la balise <*Service*> permet d'implémenter la couche transport des requêtes clients vers le moteur de Servlets Tomcat (Catalina). Depuis la version 5.0 de Tomcat, il existe deux types de connecteurs selon le protocole, à savoir HTTP et AJP. Le choix de transiter par l'un ou l'autre des protocoles se fait avec l'attribut *protocol*. L'attribut *port* permet d'indiquer le port à l'écoute du connecteur. Par défaut, Tomcat utilise le port 8080 pour le HTTP et le port 8009 pour le connecteur AJP. L'attribut *address* permet de spécifier une adresse IP particulière pour écouter les connexions entrantes. Enfin, l'attribut *secure* permet de définir un connecteur HTTPS.

L'élément<*Engine*> est chargé de répartir toutes les requêtes. C'est le moteur de Servlets Catalina de Tomcat. Une application est obligatoirement associée à un élément <*Engine*>. Les deux attributs obligatoires de cet élément sont *name* et *defaultHost*. L'attribut *name* permet d'identifier le moteur de Servlets et l'attribut *defaultHost* permet de définir lequel des éléments <*Host*> de la configuration va recevoir les requêtes en cas de non correspondance de nom d'hôte (hôte par défaut).

L'élément<*Host*> : le conteneur <*Host*> permet de configurer les attributs à associer à un hôte unique. La possibilité de configurer une seule machine pour servir plusieurs hôtes offre une souplesse considérable. L'hébergement virtuel est une technique utilisée par les serveurs HTTP, permettant d'héberger plusieurs sites distincts à une même adresse IP. Par exemple, lorsqu'un client saisit l'adresse : http://www.monsite.com/accueil.html, la requête suivante est envoyée

```
GET /accueil.html HTTP/1.1
Host : www.monsite.com
```

Le système DNS va permettre au navigateur de trouver l'adresse IP du serveur Web à contacter et la requête sera envoyée au serveur indiqué. Un autre site http://www.monentreprise.com peut être hébergé sur la même machine et donc posséder la même adresse IP. C'est dans ce cas précis que l'élément <*Host*> est utilisé pour identifier de façon unique et précise le site. Donc autant d'hôtes que nécessaire peuvent être configurés dans un serveur Tomcat et c'est l'attribut *name* qui permet de préciser le nom d'hôte.

Si un client accède au serveur directement par l'adresse IP, le serveur sera alors dans l'incapacité de résoudre le nom d'hôte. L'attribut de configuration *defaultHost* de l'élément <*Engine*> permet donc de définir l'hôte qui sera contacté dans ce cas précis.

Il peut être intéressant de mapper plusieurs noms de domaine sur un contenu unique. Par exemple, les sites www.monentreprise.com et monentreprise.com peuvent être utilisés avec le même hôte. Dans ce cas, les noms d'hôtes doivent retourner un contenu identique, il faut alors utiliser l'élément <*Alias*> à l'intérieur du conteneur <*Host*>.

```
<Engine defaultHost="www.monsite.com">
    <Host name="www.monsite.com">
    ...
    </Host>
    <Host name="www.monentreprise.com">
    <Alias>monentreprise.com</Alias>
    ...
    </Host>
</Engine>
```

L'autre attribut obligatoire est *appBase*. Il permet de spécifier le répertoire racine dans lequel sont stockées les applications accessibles via cet hôte. La valeur par défaut est *webapps* et correspond au répertoire */webapps* de Tomcat.

Les autres attributs de configuration permettent de gérer le déploiement des applications. Entre autres, l'attribut *autoDeploy* permet d'indiquer à Tomcat si les applications déposées dans le répertoire des webapps (indiqué donc par *appBase*) doivent être automatiquement déployées sans redémarrage du serveur pour un hôte. Par défaut la valeur est *autoDeploy="true"* ce qui est très intéressant pour un serveur en développement, mais il oblige Tomcat à surveiller en permanence le contenu de ce répertoire. Ce procédé est très coûteux en terme de ressources sur un serveur en production.

L'attribut *liveDeploy* indique qu'un nouveau projet déposé dans le répertoire /webapps doit être rechargé automatiquement au niveau du projet et pas d'un hôte.

L'attribut *deployOnStartup* rend disponible toutes les applications au démarrage de Tomcat, ce qui est évidemment la valeur par défaut.

L'attribut *unpackWARs* permet de décompresser les fichiers d'archives WAR ce qui est le cas par défaut.

L'attribut *deployXML* permet d'autoriser le déploiement des applications via les fichiers de contexte XML.

Enfin, le répertoire *workDir* permet de spécifier un répertoire de travail pour les applications. Les classes des Servlets et des JSP seront générées dans ce répertoire. Chaque application possède son propre sous-répertoire. Par défaut, ce répertoire est placé de cette façon : */work/Catalina/nom_hôte*.

L'élément<*Context*> qui est une balise fille de l'élément <*Host*> permet de déployer une application Web dans Tomcat. Un contexte permet de relier une URL à une application Web.

Cet élément est utilisé pour déclarer explicitement une application, il peut être utilisé dans le fichier *server.xml* ou bien dans les fichiers de contexte XML, c'est cette dernière méthode qui est préconisée avec Tomcat 6.X. La balise <*Context*> possède deux attributs obligatoires afin de préciser le répertoire qui contient l'application et l'URL pour accéder à cette application.

L'attribut *docBase* permet de faire référence au répertoire des données de l'application ou bien directement au fichier WAR de l'application.

L'attribut *path* permet d'indiquer le chemin de contexte de cette application Web. Ce chemin commence toujours par le caractère /, chaque application doit posséder une valeur unique de cet attribut. L'attribut facultatif *reloadable* permet d'activer une surveillance des répertoires */WEB-INF/lib* et */WEB-INF/classes* de l'application.

Chaque modification apportée au contenu de ces répertoires sera automatiquement prise en compte par Tomcat qui rechargera alors automatiquement l'application. Il est préférable d'utiliser l'outil *manager* du serveur d'applications pour recharger les classes et ainsi éviter un gaspillage inutile des ressources.

L'attribut facultatif *workDir* permet de spécifier le répertoire de travail pour l'application. Si cet attribut est spécifié dans l'hôte, il surcharge alors celui défini dans l'élément <*Host*>.

L'attribut facultatif *cookie* permet d'indiquer si le serveur utilise les cookies pour gérer les sessions utilisateurs. La valeur par défaut est true.

```
<Host name="monsite.com" appBase="/usr/local/tomcat/webapps/monsite"
debug="0" unpackWARs="true" autoDeploy="false" liveDeploy="false">
    <Context path="" docBase="." debug="1" reloadable="false">
    </Context>
</Host>
```

Tomcat utilise un contexte par défaut qui est mis en œuvre dans le fichier */conf/context.xml*.

L'élément<*Realm*> : la plate-forme Java EE définit un mécanisme standard basé sur la notion de rôles pour la gestion des authentifications dans les applications Web. Un <*Realm*> peut être défini en tant qu'élément enfant des balises <*Engine*>, <*Host*> ou <*Context*>. Suivant le placement, l'authentification sera appliquée à tout le moteur de Servlets, à un hôte particulier ou à une application.

```
<Host name="www.monsite.com" appBase="/usr/local/tomcat/webapps/monsite"
debug="0" unpackWARs="true" autoDeploy="false" liveDeploy="false">

<Alias>monsite.com</Alias>
<Alias>mesites.com</Alias>
<Context path="" docBase="." debug="0" reloadable="false">
<Resource name="jdbc/monsitemysql" auth="Container"
type="javax.sql.DataSource"
username="admin"
password="43tZAE3"
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/monsite"
maxActive="20"
maxIdle="10"
maxWait="10000"
logAbandoned="true"/>

<Realm className="org.apache.catalina.realm.DataSourceRealm"
dataSourceName="jdbc/monsitemysql" localDataSource="true"
userTable="administrateur"
userNameCol="nomadministrateur"
userCredCol="motdepasseadministrateur"
userRoleTable="roles"
roleNameCol="role"
/>

</Context>
</Host>
```

L'élément<*Loader*> définit un chargeur de classe Java. Le rôle de cet élément est de charger les classes Java d'une application Web. Le serveur charge les classes contenues dans le répertoire */WEB-INF/classes* de l'application, les classes contenues dans les fichiers JAR présents dans le répertoire */WEB-INF/lib* et les classes rendues accessibles aux applications par le moteur de Servlets.

L'élément<*Manager*> permet de configurer le gestionnaire de session pour une application Web spécifique. Par défaut, un conteneur Web Java EE stocke les informations de session utilisateur dans la Machine Virtuelle Java. En plus d'être stockées en mémoire, les sessions utilisateurs sont sauvegardées soit dans une base de données soit dans un fichier par la classe d'implémentation *org.apache.catalina.session.PersistentManager*.

L'élément<*Valve*> représente un composant sous forme d'une classe Java. Cet élément peut être considéré comme un filtre de requêtes. Voici les différentes valeurs possibles pour l'attribut *className* et le type de filtre associé :

- *org.apache.catalina.valves.AccessLogValve* : génère un fichier journal des accès au serveur.
- *org.apache.catalina.valves.JDBCAccessLogValve* : génère un fichier journal des accès à une base de données.
- *org.apache.catalina.valves.RemoteAddrValve* : applique une restriction d'accès en fonction des adresses IP des clients.
- *org.apache.catalina.valves.RemoteHostValve* : applique une restriction d'accès en fonction des noms de machines des clients.
- *org.apache.catalina.valves.RequestDumperValve* : génère un fichier journal des requêtes des clients.
- *org.apache.catalina.authenticator.SingleSignOn* : permet une authentification unique entre plusieurs applications.

Il est important de noter que nous pouvons implémenter votre propre filtre en écrivant une classe Java sur le modèle de celles déjà fournies.

Exemple :

Avec *JDBCAccessLogValve*, le code suivant peut être inséré dans le fichier de configuration *server.xml*.

```
<Engine...>
...
<Valve className="org.apache.catalina.valves.JDBCAccessLogValve"
connectionURL="jdbc:mysql://localhost:3306/stattomcat?
user=monutilisateurmysql&amp;password=monmotdepasse"
driverName="com.mysql.jdbc.Driver"
tableName="log"
resolveHosts="false"
pattern="common"/>
....
</Engine>
```

L'attribut *connectionURL* permet de spécifier la base de données et les coordonnées de connexion à la base de données.

L'attribut *driverName* permet de spécifier le pilote JDBC d'accès à la base de données. Celui-ci devra bien sûr être installé dans le répertoire */lib* de Tomcat.

L'attribut *tableName* permet de préciser le nom de la table qui va recevoir les données des journaux.

L'attribut *resolveHosts* permet de remplacer l'adresse IP du client par le nom de machine.

Enfin, l'attribut *pattern* permet de définir le format d'entrée des fichiers journaux.

La base de données doit avoir la syntaxe suivante :

```
CREATE DATABASE "stattomcat";
USE "stattomcat";
CREATE TABLE "log" (
"id" INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
"remoteHost" CHAR(15) NOT NULL DEFAULT "",
"user" CHAR(15),
"timestamp" TIMESTAMP NOT NULL DEFAULT 0,
"virtualHost" VARCHAR(64) NOT NULL DEFAULT "",
"method" VARCHAR(8) NOT NULL DEFAULT "",
"query" VARCHAR(255) NOT NULL DEFAULT "",
```

```
"status" INTEGER UNSIGNED NOT NULL DEFAULT 0,
"bytes" INTEGER UNSIGNED NOT NULL DEFAULT 0,
"referer" VARCHAR(128),
"userAgent" VARCHAR(128),
PRIMARY KEY("id")
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

L'élément <*Listener*> permet de définir un écouteur d'événements sur les éléments <*Server*>, <*Host*> ou <*Engine*>.

Cet élément ne dispose que d'un seul attribut obligatoire nommé *className* qui est la classe Java qui implémente l'écouteur. Tomcat possède par défaut deux écouteurs définis sur l'élément <*Server*>. Ces écouteurs permettent la supervision globale du serveur ainsi que des ressources JNDI. Les objets MBeans de supervision sont définis et utilisés par l'API Java JMX (*Java Management Extension*) que nous verrons dans ce chapitre.

Parmi les autres fichiers de configuration de Tomcat, il existe les fichiers *tomcat-users.xml*, *catalina.policy* et *web.xml*. Ces fichiers sont présents dans le répertoire */conf* de Tomcat. Les fichiers *tomcat-users.xml* et *catalina.policy* sont utilisés pour la sécurité du serveur et le fichier *web.xml* définit les applications Web déployées sur le serveur.

Le fichier de configuration des applications

Le fichier web.xml

Le fichier *web.xml* est un autre fichier de configuration. Un fichier *web.xml* est un descripteur de déploiement d'applications Web Java EE. Le fichier */conf/web.xml* de Tomcat définit les paramètres de configuration utilisés par toutes les applications Web installées sauf, si les applications fournissent leur propre fichier de configuration *web.xml*. Par la suite, toutes les applications déployées utiliseront leur propre fichier *web.xml*.

Ce fichier de configuration commence par la déclaration de Servlets qui sont spécifiques à Tomcat. La Servlet par défaut de Tomcat est définie avec la classe *DefaultServlet* et son attribut *listings* qui permet d'autoriser ou non l'indexation.

La Servlet *InvokerServlet* permet de déclencher des Servlets directement avec des URL http://serveur/application/servlet/nomdelaclasseservlet.

La troisième Servlet est *JspServlet* et permet de transformer des pages JSP en Servlet.

Les sections suivantes de ce fichier de configuration concernent les paramètres Java EE : par exemple, le temps d'expiration des sessions, les types MIME des en-têtes HTTP, les pages d'accueil (ex: index.html, index.jsp...).

Le fichier de configuration des utilisateurs

Le fichier tomcat-users.xml

Ce fichier est utilisé pour les authentifications de Tomcat. Tomcat utilise le système d'authentification basé sur une connexion JNDI. Ce gestionnaire d'authentification est associé au fichier *tomcat-users.xml* contenant les associations identifiant, mot de passe et rôle. La partie manager et administration de Tomcat utilise ce fichier d'authentification.

La déclaration de ce système d'authentification se retrouve dans le fichier de configuration du serveur */conf/server.xml*.

```
<!-- Global JNDI resources -->
  <GlobalNamingResources>

    <!-- Test entry for demonstration purposes -->
    <Environment name="simpleValue" type="java.lang.Integer" value="30"/>

    <!-- Editable user database that can also be used by
         UserDatabaseRealm to authenticate users -->
    <Resource name="UserDatabase" auth="Container"
       type="org.apache.catalina.UserDatabase"
       description="User database that can be updated and saved"
       factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
       pathname="conf/tomcat-users.xml" />

  </GlobalNamingResources>
```

Voici le contenu de ce fichier qui se trouve dans le répertoire : */conf/tomcat-users.xml*.

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
  <user username="admin" password="admin" roles="admin,manager"/>
</tomcat-users>
```

Nous remarquons la définition des rôles et des utilisateurs avec leur mot de passe.

L'application *manager* de Tomcat n'est accessible que par les utilisateurs qui possèdent le rôle *manager*, déclaré dans ce fichier. Il sera possible de modifier ce fichier pour réaliser une authentification qui repose sur une base de données ou un annuaire.

Le fichier de configuration de la sécurité

Le fichier catalina.policy

Le gestionnaire de sécurité de la machine virtuelle Java est appelé *SecurityManager* et permet d'assurer la sécurité des applications exécutées sur le système. Par l'intermédiaire de ce gestionnaire, il est possible de régler l'accès aux ressources que le programme Java peut utiliser. Par défaut, Tomcat est exécuté sans le gestionnaire de sécurité *SecurityManager*. Les applications déployées sur le serveur peuvent donc lire, écrire et supprimer des fichiers, lancer des programmes et des commandes système, ouvrir des flux à travers le réseau, gérer la machine virtuelle, voire l'arrêter...

Pour lancer Tomcat en mode sécurisé, il faut utiliser l'option *-security* sur le script de démarrage (*startup.bat* ou *startup.sh*).

Dans ce cas précis, tout ce qui n'est pas explicitement précisé dans le fichier *catalina.policy* est interdit.

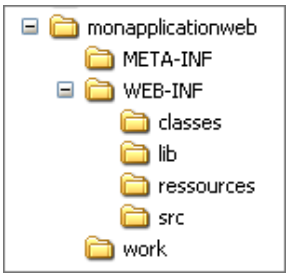
Arborescence d'un projet Tomcat

Une application Web est un ensemble de ressources qui participent au fonctionnement de l'application. Une application est constituée de plusieurs éléments :

- Le composant serveur dynamique : Servlets et JSP.
- Les librairies de classes Java utilitaires.
- Les éléments Web statiques : pages HTML, images, feuilles de style, JavaScript...
- Les composants clients dynamiques : Applets, JWS, JavaBean.
- Un descripteur de déploiement et de configuration de l'application Web sous la forme d'un (ou plusieurs) fichier (s) de configuration au format XML : *web.xml*.

Les spécifications de l'API Servlet indiquent l'organisation des dossiers et fichiers à respecter pour déployer une application Web. L'application est configurée par le fichier XML : */monapplicationweb/WEB-INF/web.xml*.

L'arborescence de l'application Web est la suivante :



Le dossier nommé dans l'exemple *monapplicationweb* représente la racine de l'application Web. N'importe quel nom peut lui être donné, en général c'est le nom de l'application (ex : facturation, boutique...). Une fois en production ce dossier est public, nous retrouvons donc les fichiers HTML, JavaScript...

Le dossier *META-INF* contient le fichier *manifest.mf* qui est généré par l'archiveur *jar.exe* lorsqu'une archive du projet est réalisée pour le déployer. Ce fichier permet de stocker des informations concernant l'archive comme sa version, le contenu, la version du compilateur...

Exemple :

```
Manifest-Version: 1.0
Created-By: 1.5.0_04 (Sun Microsystems Inc.)
```

Le dossier *WEB-INF* représente la partie privée de l'application Web. Il contient le fichier *web.xml* qui représente le descripteur de déploiement.

Le sous-dossier */classes* du dossier *WEB-INF* contient les fichiers *.class* (compilés) de l'application (Servlets, JavaBean, classes utilitaires...).

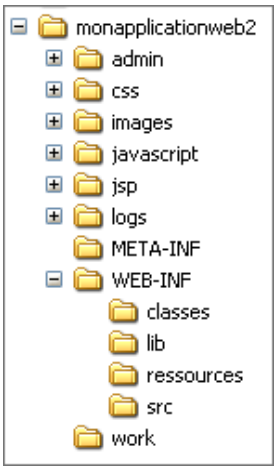
Le sous-dossier */src* du dossier *WEB-INF* contient les fichiers *.java* (sources) de l'application (Servlets, JavaBean, classes utilitaires...).

Le sous-dossier */lib* du dossier *WEB-INF* peut contenir des librairies *.jar* utiles au fonctionnement de l'application Web. Par exemple, la librairie de gestion d'e-mails, les librairies de Struts, les librairies de manipulation de code XML, des fichiers *.dtd* et *.tld*...

Le sous-dossier */ressources* du dossier *WEB-INF* contient les fichiers de configuration pour les applications : par exemple les traductions, les fichiers de gestion de Log4J...

Une application Web utilise fréquemment des fichiers ressources qui doivent être placés dans la même arborescence que les classes, c'est-à-dire dans le répertoire */WEB-INF/classes*. Cependant, ces fichiers ressources seront placés plutôt dans le dossier */WEB-INF/src* sachant qu'Eclipse recopie automatiquement les fichiers de ce répertoire vers le répertoire */WEB-INF/classes*. En effet, lors du rafraîchissement et de la compilation complète des pages, Eclipse vide le répertoire */WEB-INF/classes* ce qui est contraignant.

Une application Web est accessible par défaut à l'adresse : http://nomhote:port/webapp/



Ce second exemple présente une application Java EE plus complexe avec :

- le répertoire */admin* pour l'administration,
- le répertoire */css* pour les feuilles de style,
- le répertoire pour les */images* et pour les */javascript*,
- le répertoire */jsp* qui contient les JSP non compilées,
- le répertoire */work* qui contient les JSP compilées,
- le répertoire */logs* qui permet de stocker les fichiers journaux Log4J.

Bien sûr, il n'est pas nécessaire de connaître par cœur les noms des fichiers, répertoires et l'organisation de l'arborescence pour développer une application Java EE. Des outils de développement évolués comme Eclipse génèrent automatiquement l'arborescence à la suite de la création d'un projet Tomcat/Java EE.

1. Le descripteur de déploiement web.xml

Le fichier *web.xml* contient des informations qui permettent de définir l'environnement d'exécution de l'application Internet et de lier les composants entre eux. Le fichier permet de définir les relations entre les URL et les Servlets/JSP, la page d'accueil, la page d'erreur, les contraintes de sécurité, les authentifications, les ressources pour accéder aux données... Chaque application Web possède son propre fichier de configuration *web.xml* placé dans le répertoire */WEB-INF* de l'application. Ce fichier est appelé le descripteur de déploiement de l'application ou webapp.

Un des avantages de l'utilisation d'un fichier au format XML est le caractère autodescriptif de ce langage. Le fichier *web.xml* commence par déclarer l'encodage et le type de grammaire utilisée pour vérifier sa syntaxe (DOCTYPE ou schéma XML). Le fichier repose sur une DTD, il est donc important de respecter la syntaxe et l'ordre des balises. En réalité, une application simple n'a pas besoin de ce descripteur car le fichier *web.xml* du serveur peut suffire mais c'est une bonne habitude que de toujours définir un fichier de déploiement avec une application.

Tomcat possède un descripteur de déploiement par défaut qui est utilisé pour les applications et qui est placé dans le répertoire */conf*.

Le descripteur de déploiement permet de décrire les éléments suivants :

- les paramètres d'initialisation des Servlets ;
- les variables globales ;
- la session de l'application ;
- les liens entre les URI (*Uniform Resource Identifier*) et les Servlets ;
- les classes de gestion des événements ;

- les déclarations des filtres ;
- les liens entre les types MIME et les applications correspondantes ;
- les pages d'accueil par défaut ;
- les liens entre les codes d'erreur HTTP et les pages d'erreur ;
- les ressources telles que JDBC, JNDI...
- les ressources EJB.

L'élément racine du descripteur de déploiement est la balise <*web-app*>. Tous les autres éléments seront donc des enfants de cette balise. Voici ci-dessous la liste exhaustive ordonnée des balises du fichier de déploiement *web.xml*.

L'élément<*icon*> permet de spécifier le chemin vers une icône qui permet de représenter l'application Web pour des outils de gestion d'applications.

L'élément<*display-name*> : ce nœud permet d'assigner un nom à l'application Web. Ce nom peut être utilisé par un outil de gestion d'applications.

L'élément<*description*> : ce nœud permet de décrire l'application de manière textuelle.

L'élément<*distributable*> est une balise vide qui permet de distribuer l'application sur plusieurs serveurs (exemple : cluster de serveurs Tomcat).

L'élément<*context-param*> : ce nœud permet de définir des paramètres qui seront valables pour l'application. Les informations seront passées sous forme d'une paire nom-valeur dans la *ServletConfig* et donc accessibles depuis des Servlets et pages JSP. Ce principe est très souvent utilisé pour définir des chemins, des fichiers de configuration, des identifiants... Un paramètre peut aussi être passé à une Servlet précise (et non à toutes comme avec <*context-param*>) en utilisant le nœud <*init-param*> des Servlets.

L'élément<*filter*> permet de définir une classe de filtre qui sera applicable avant l'exécution de la Servlet. Le principe de fonctionnement est similaire aux classes Tomcat Valve. Un filtre peut permettre par exemple d'écrire un fichier XSL pour répondre à un contenu XML si la requête se termine par *.xml* ou en HTML si elle se termine par *.html*.

L'élément<*filter-mapping*> : ce nœud permet d'associer un filtre à une Servlet spécifique ou à une URL. Les filtres sont mappés en respectant l'ordre dans lequel ils apparaissent dans le descripteur de déploiement.

L'élément<*listener*> permet de définir un écouteur qui sera appelé suite à un événement particulier.

L'élément<*servlet*> permet de définir une Servlet et les attributs. La Servlet sera définie par son nom et sa classe.

L'élément<*servlet-mapping*> : ce nœud permet de mapper un modèle d'URI vers un nom de Servlet défini dans un nœud <*servlet*>.

L'élément<*session-config*> permet de définir la session de l'application Web avec entre autres le délai d'expiration en minutes des sessions de l'application.

L'élément<*mime-mapping*> : ce nœud mappe une extension au type MIME. C'est une association entre les fichiers publics et des types MIME.

L'élément<*welcome-file-list*> : ce nœud permet de spécifier le ou les fichiers (par ordre de priorité) qui doivent être servis à partir d'un répertoire lorsque seul le nom du répertoire est spécifié (ex : *index.html*, *index.jsp*).

L'élément<*error-page*> permet de définir une page d'erreur qui servira en cas de renvoi d'un code erroné suite à une exception.

L'élément<*taglib*> permet de définir l'emplacement des bibliothèques de balises. Le nœud mappe un modèle d'URI vers un fichier de descripteur de bibliothèques de balises.

L'élément<*ressource-env-ref*> : ce nœud configure une ressource externe qui peut être utilisée par la Servlet.

L'élément<*ressource-ref*> : ce nœud configure également une ressource externe qui peut être utilisée par la Servlet.

L'élément<*login-config*> permet de configurer la méthode d'authentification et le nom du Realm à utiliser pour l'application.

L'élément<*env-entry*> : ce nœud définit le nom d'une ressource accessible grâce à l'interface JNDI.

L'élément<*ejb-ref*> permet de déclarer une référence distante à un Entreprise JavaBean.

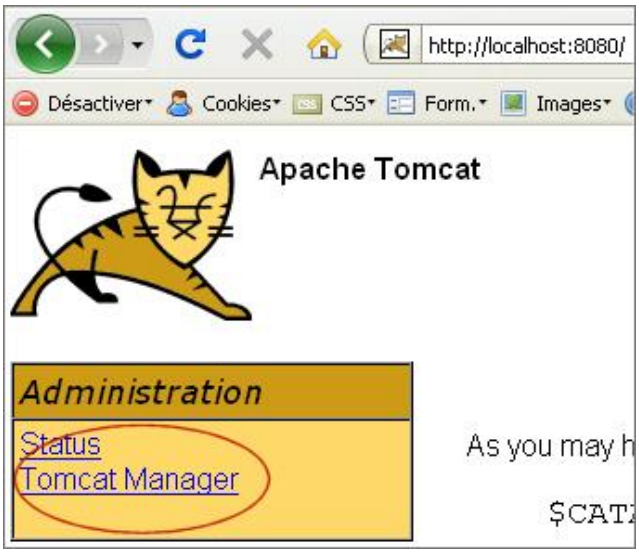
L'élément<*ejb-local-ref*> permet de déclarer une référence locale à un Entreprise JavaBean.

2. Déployer un premier projet

Afin de mettre en application les explications de ce guide, nous allons déployer une première application simple. La configuration des applications Web déployées au sein du serveur Tomcat se fait à l'aide du fichier */conf/server.xml* ou d'un fichier spécifique par application présent dans */conf/Catalina/localhost/monapplication.xml*.

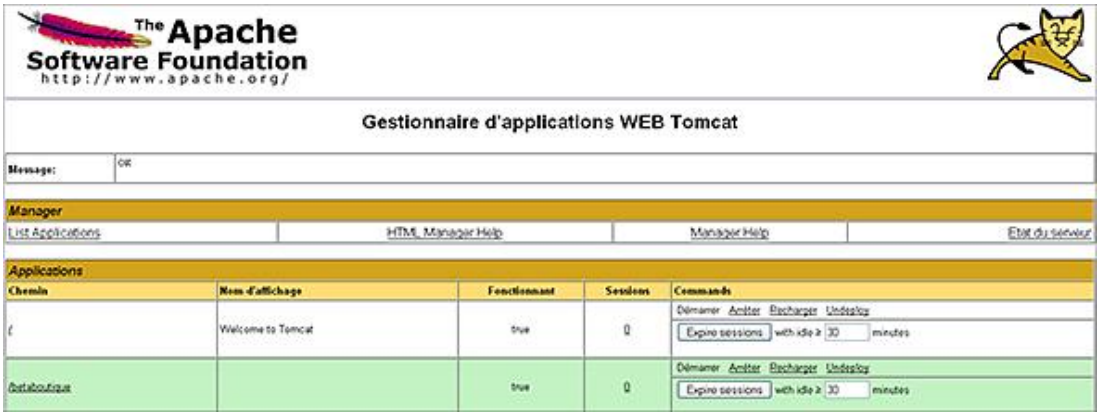
a. Projet simple manuellement

Ces fichiers peuvent être créés à la main car la structure est assez simple mais nous allons dans un premier temps utiliser l'outil *manager* de Tomcat qui permet de déployer une application avec une interface Web.



Le lien **Tomcat Manager** ouvre une fenêtre d'authentification. Les informations de connexion correspondent aux données insérées lors de l'installation du serveur.

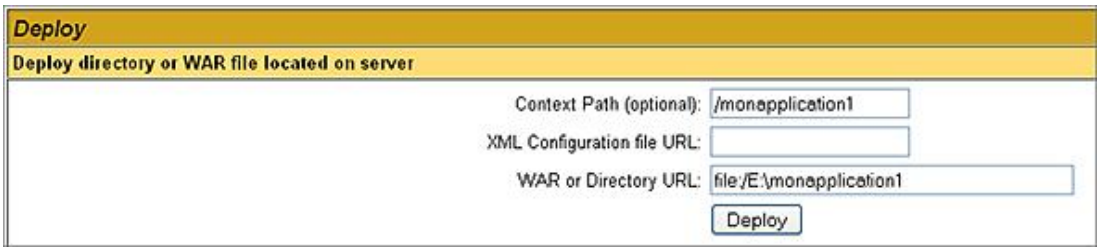
Nous obtenons une page qui liste toutes les applications actuellement déployées sur le serveur.



Pour déployer une nouvelle application, il faut utiliser l'interface en précisant les informations pour le déploiement. La première partie permet de préciser le contexte (chemin d'accès à l'application par URL). Le second paramètre optionnel permet d'indiquer le fichier XML pour la configuration de l'application. Le dernier paramètre correspond au répertoire (ou au fichier *.war*) qui contient l'application. Si nous utilisons un répertoire, celui-ci doit d'abord être créé avant le déploiement de l'application et il doit respecter l'arborescence Java EE (répertoire *WEB-INF*...). Si nous ne précisons pas de fichier de configuration XML, le projet sera déployé dans le répertoire */webapps* de Tomcat et sera donc directement accessible sans fichier de configuration.

Nous utilisons l'application *monapplication1* qui est un simple projet Java EE avec l'arborescence habituelle (*/WEB-INF*, */WEB-INF/classes*, */WEB-INF/lib*, */WEB-INF/src*) et un simple fichier JSP, *index.jsp* qui affiche juste le nom de l'application en HTML et en Java.

```
<h1>monapplication1</h1>
<% out.println("mon application1 en Java"); %>
```



Nous cliquons sur **Deploy**.

Le déploiement a eu pour effet d'ajouter une copie complète du projet dans le répertoire des *webapps* de Tomcat */webapps*. Donc, il n'y a pas d'insertion de la configuration dans le fichier *server.xml* ni de création d'un fichier particulier pour le contexte (*/conf/Catalina/localhost*) car tous les fichiers déployés dans le répertoire des webapps du serveur sont automatiquement exploitables.

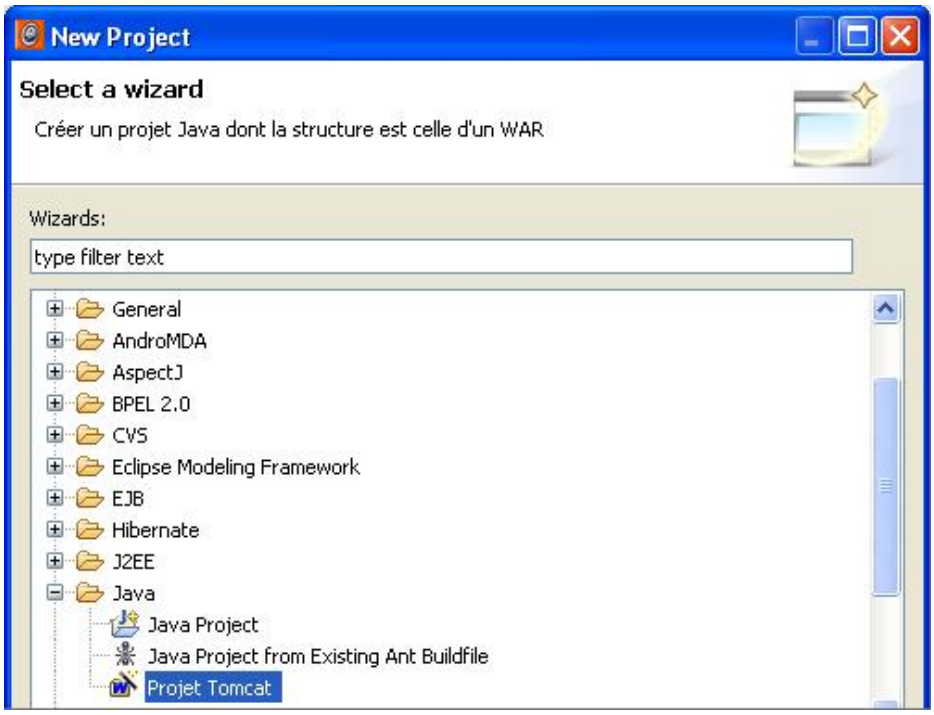
Le projet a été copié dans le répertoire des */webapps* ce qui signifie que désormais, les fichiers devront être modifiés dans ce répertoire */webapps/monapplication1*.

Notre application est alors accessible à l'adresse suivante : http://localhost:8080/monapplication1/

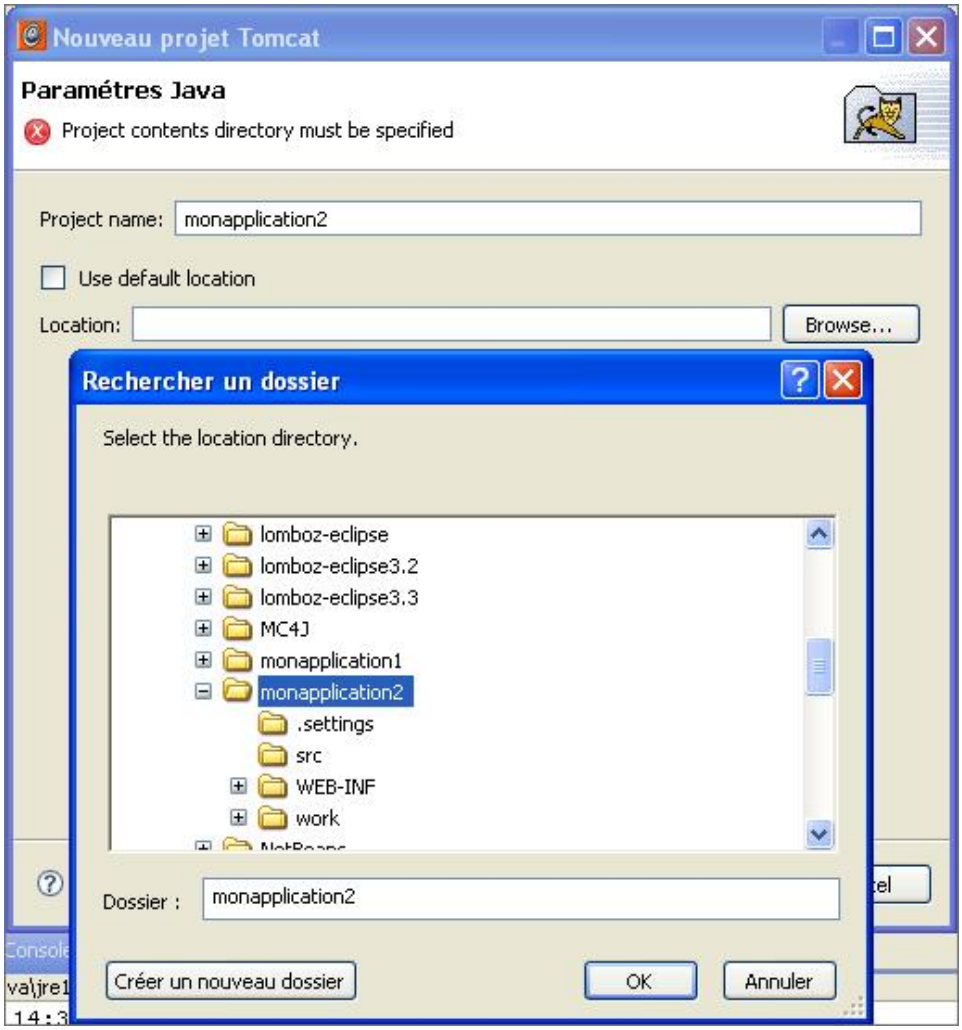
b. Projet simple avec Eclipse

Pour créer un projet Tomcat avec Eclipse, nous allons utiliser la commande **File - New - Project**.

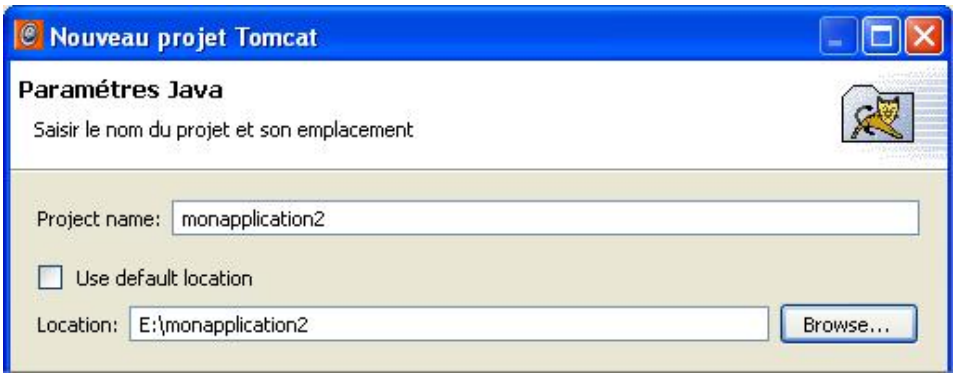
Avant de pouvoir utiliser Tomcat avec Eclipse, il est nécessaire d'installer le plug-in Sysdeo comme précisé au chapitre Objectifs et spécifications de Java EE de ce guide.



L'item **Projet Tomcat** permet de créer un projet Tomcat sans avoir à écrire le déploiement dans le fichier de configuration de Tomcat (*server.xml*) ou un fichier particulier (*/conf/Catalina/localhost/*) et il permet également de créer automatiquement l'arborescence du projet.

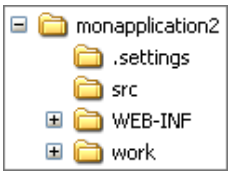


La seconde page nous demande de définir un nom pour le projet Eclipse. Nous précisons le répertoire sur le disque (ce répertoire vide doit être créé avant le déploiement) qui recevra le projet Eclipse et l'intégralité de ses fichiers.



Lors de cette dernière étape, nous indiquons le nom du contexte qui sera accessible à l'adresse : http://localhost:8080/monapplication2.

Eclipse a alors automatiquement créé l'arborescence nécessaire au projet dans le répertoire sélectionné avec *location*.



De même, un fichier nommé *monapplication2.xml* a été créé par Eclipse dans le répertoire */conf/Catalina/localhost*.

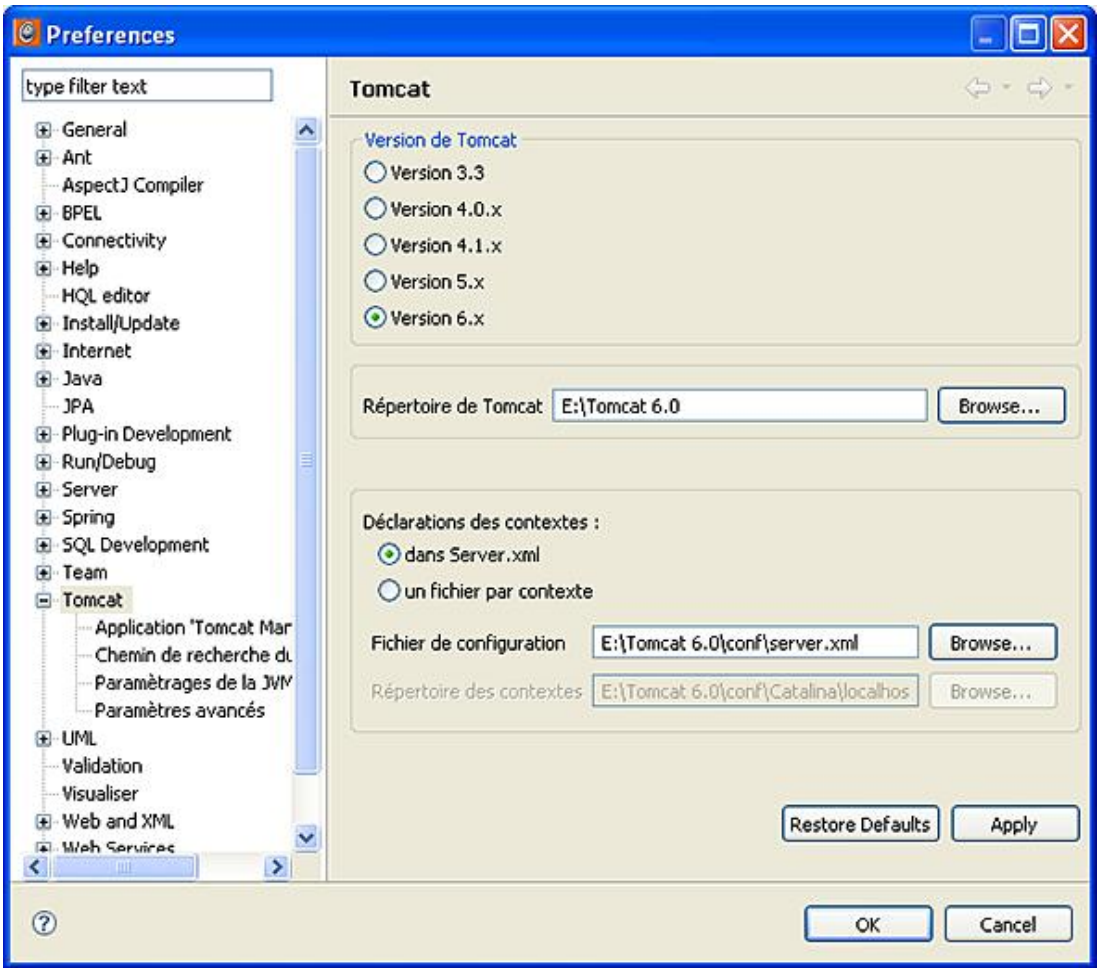
```
<Context path="/monapplication2" reloadable="true"
docBase="E:\monapplication2" workDir="E:\monapplication2\work" />
```

Nous retrouvons le paramétrage courant d'une application Web avec le chemin d'accès à la webapp (le contexte avec le paramètre *path*), l'attribut qui indique que l'application sera rechargée automatiquement (*reloadable="true"*), le répertoire d'accès aux fichiers de l'application (attribut *docBase*) et le répertoire des JSP compilées (attribut *workDir*). L'application *monapplication2* est identique en terme de contenu à *monapplication1*.

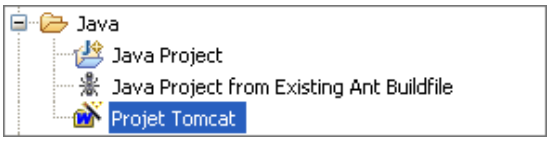
Nous pouvons vérifier le fonctionnement de l'application en demandant le contexte */monapplication2* via l'URL : http://localhost:8080/monapplication2/

Dans une dernière étape, pour bien voir les différentes possibilités de déclaration d'applications, nous allons déployer une application nommée *monapplication3* avec Eclipse mais en déclarant l'application directement dans le fichier de configuration du serveur (*server.xml*) et non dans un fichier spécifique au projet (*/conf/Catalina/localhost/monapplication3.xml*).

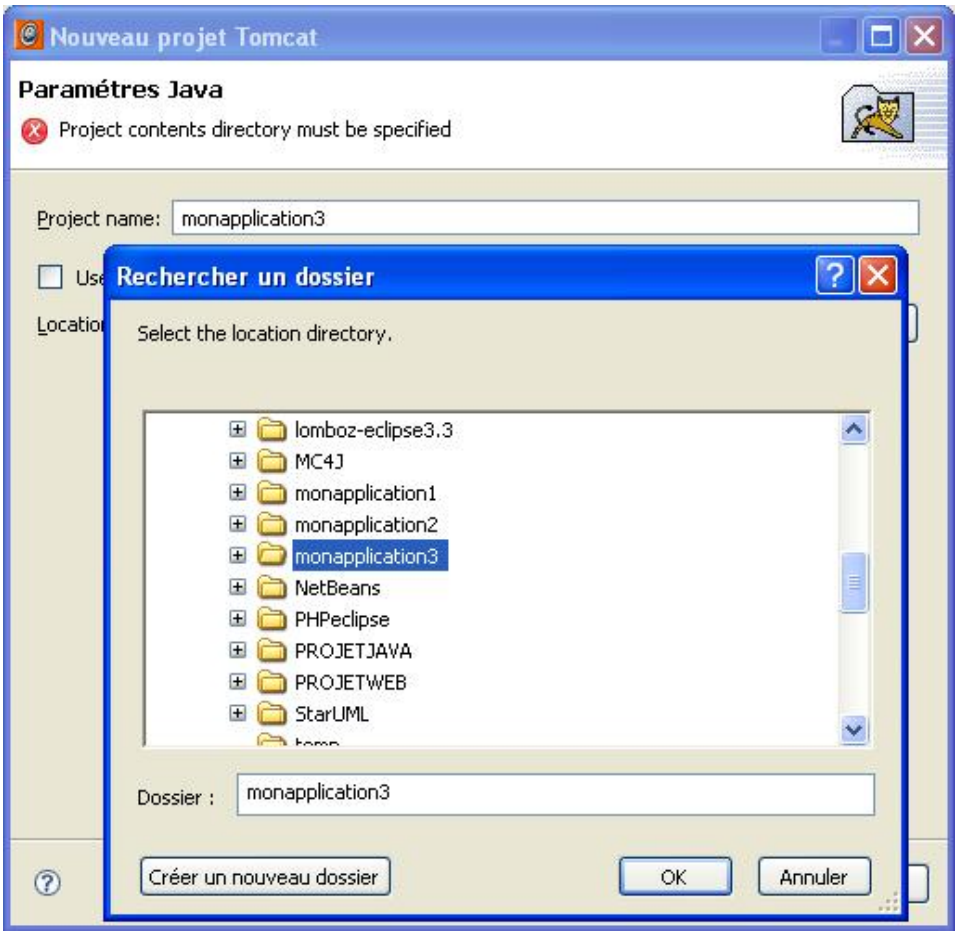
Pour cela, dans le menu d'Eclipse, nous utilisons le menu **Windows - Preferences - Tomcat** et nous modifions la partie **Déclaration des contextes** en précisant désormais le fichier de configuration du serveur : *server.xml*.



Dans le menu d'Eclipse nous réalisons alors un nouveau projet : **File - New - Project**.



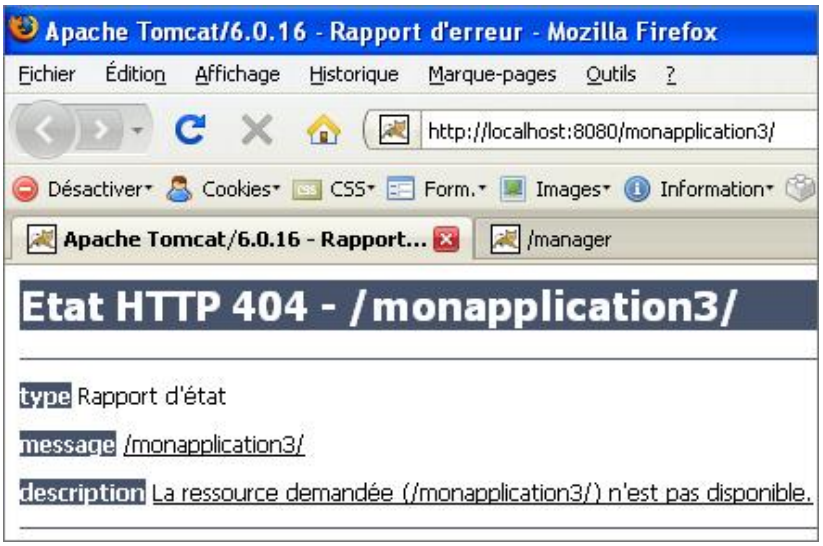
Ensuite, nous indiquons le répertoire (qui est vide mais qui doit déjà exister sur le disque dur) de l'application.



Nous allons créer une simple page *index.jsp* dans le répertoire de *monapplication3*.

```
<h1>monapplication3</h1>
<% out.println("mon application3 en Java"); %>
```

Si nous testons maintenant l'application *monapplication3*, nous remarquons que celle-ci n'est pas accessible à l'adresse suivante : http://localhost:8080/monapplication3/.



En effet, le projet Eclipse *monapplication3* existe et il a été inséré dans le fichier */conf/server.xml* en fin de fichier.

```
...
<Context path="/monapplication3" reloadable="true"
docBase="E:\monapplication3" workDir="E:\monapplication3\work" />
      </Host>
    </Engine>
```

```
  </Service>
</Server>
```

Le problème, c'est que les applications déclarées dans ce fichier nécessitent un redémarrage du serveur. Après cette action, la connexion à l'adresse suivante : http://localhost:8080/monapplication3/ est opérationnelle.

Nous avons vu plusieurs manières de déployer un projet Java EE avec Tomcat. Nous remarquons rapidement que l'utilisation d'un fichier spécifique par contexte (*/conf/Catalina/localhost*) est beaucoup plus souple que la déclaration dans le fichier de configuration du serveur : *server.xml*.

En effet, nous n'avons pas besoin de relancer le serveur, de lire tout le fichier *server.xml* pour la gestion d'une application et nous obtenons un meilleur découpage...

Pour la suite, nous utiliserons la déclaration des applications avec un fichier par application (*/conf/Catalina/localhost*).

Supprimer un projet

L'opération de suppression d'un projet avec Eclipse est très simple, il suffit de faire un clic droit sur le nom du projet et de valider l'action *delete*. Eclipse demande alors si la déclaration du projet doit être supprimée et également le répertoire des fichiers sources associés.

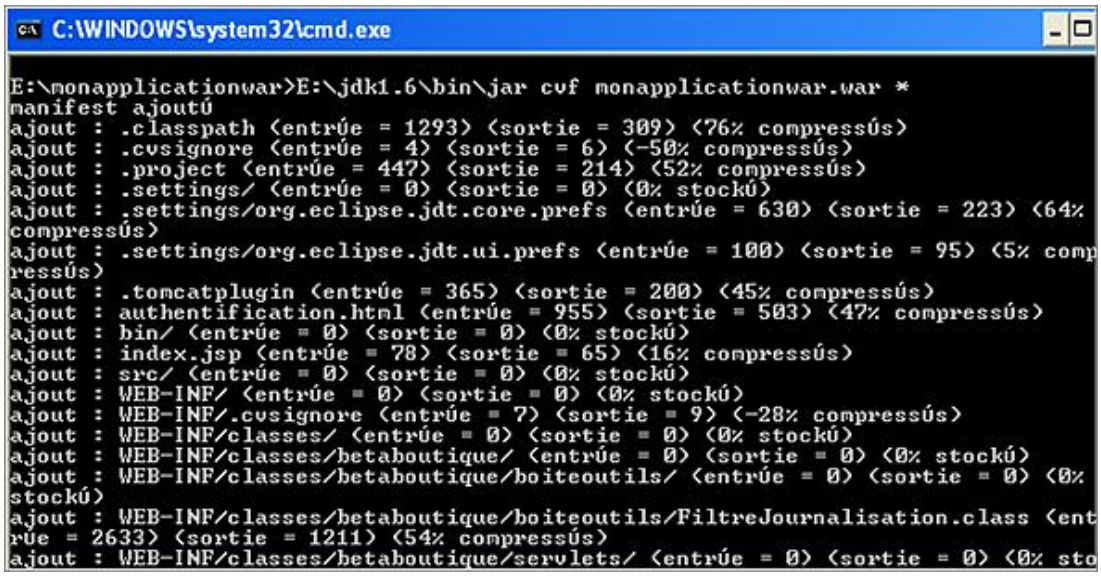
3. Déployer un projet Java EE avec un fichier war

Lors de la mise en production d'un projet, un fichier d'empaquetage *.war* est utilisé pour le déploiement. Dans les gros projets, ce fichier est créé avec l'outil ANT que nous verrons dans un prochain chapitre. La totalité de l'application Web est alors contenue dans un fichier portant l'extension *.war* pour Web ARchive. Ce fichier d'empaquetage est destiné au déploiement de l'application Web au sein du serveur d'applications. Par la suite, c'est le conteneur du serveur d'applications qui s'occupe de la mise en œuvre de l'application.

L'outil *jar.exe* du JDK Java permet de créer l'application Web au format WAR. Pour l'utilisation de cet outil de déploiement, nous allons réaliser une copie du répertoire *monapplication1* sous le nom *monapplicationwar*. Ensuite, nous allons juste changer le contenu de la page *index.jsp* en indiquant le nom du nouveau projet.

```
<h1>monapplicationwar</h1>
<% out.println("monapplicationwar en Java"); %>
```

Désormais, nous possédons une application complète et utilisable. Pour réaliser la création du fichier *.war*, nous ouvrons une console de commande et nous nous plaçons dans le répertoire de l'application afin de respecter l'organisation des fichiers au sein de l'archive. Ensuite, nous lançons la commande *jar* (à partir du chemin complet si la variable d'environnement n'a pas été précisée).



Le paramètre *cvf* permet de créer le fichier *MANIFEST.MF* qui donnera des informations sur la version de l'application, la version de Java... Un fichier nommé *monapplicationwar.war* est alors créé.

Pour déployer ce projet, il ne reste plus qu'à déposer ce fichier dans le répertoire */webapps* de n'importe quel serveur compatible Java EE. Comme indiqué précédemment, toutes les applications du répertoire */webapps* d'une application sont automatiquement déployées. Il faut remarquer l'intérêt de développer avec un langage comme Java qui est portable, plus besoin de se soucier de l'environnement d'exécution final si le standard WAR est respecté.

Nous réalisons juste une copie de notre archive dans le répertoire */webapps* de Tomcat. Nous patientons quelques instants et le serveur déploie automatiquement la nouvelle application en décompressant l'archive dans son répertoire */webapps*. La trace du serveur dans ce cas précis est d'ailleurs observable :

```
8 oct. 2008 15:18:00 org.apache.catalina.startup.HostConfig deployWAR
INFO: Déploiement de l'archive monapplicationwar.war de l'application web
```

La nouvelle application est déployée, le résultat est visible avec l'interface manager de Tomcat. Il est alors possible d'accéder directement à l'application sans déclaration dans le serveur d'applications.

Analyse, monitoring et supervision

1. Présentation

Le système de journalisation de Tomcat et de Java est très complexe et évolué. Tomcat utilise la bibliothèque Jakarta *commons-logging*. Cette bibliothèque fait partie du projet Jakarta de la fondation Apache. Le projet contient un ensemble de bibliothèques de développement Java très variées. La librairie *commons-logging* est semblable à la bibliothèque Log4J et permet la journalisation tout comme l'API *java.util.logging* de Java.

Le fichier de journalisation de Tomcat utilise le fichier de configuration */conf/logging.properties*. C'est le principal fichier de configuration pour le serveur mais chaque application déployée peut fournir son propre fichier *loggin.properties* dans son répertoire */WEB-INF/classes*.

La machine virtuelle de Java est responsable de la réservation de l'espace mémoire nécessaire à l'application. Au démarrage de l'application, celle-ci réserve de la mémoire auprès du système d'exploitation. Si l'application nécessite plus de mémoire que cette valeur limite, alors la machine virtuelle s'arrête en déclenchant l'erreur *java.lang.OutOfMemory-Error*. Cette erreur est très fréquente en Java et doit donc être évitée au maximum.

Il est donc extrêmement important de mesurer et surveiller la consommation mémoire du serveur dans la machine virtuelle pour anticiper ce problème. La configuration de la machine virtuelle peut être visualisée par le biais du manager à cette adresse : http://localhost:8080/manager/status.

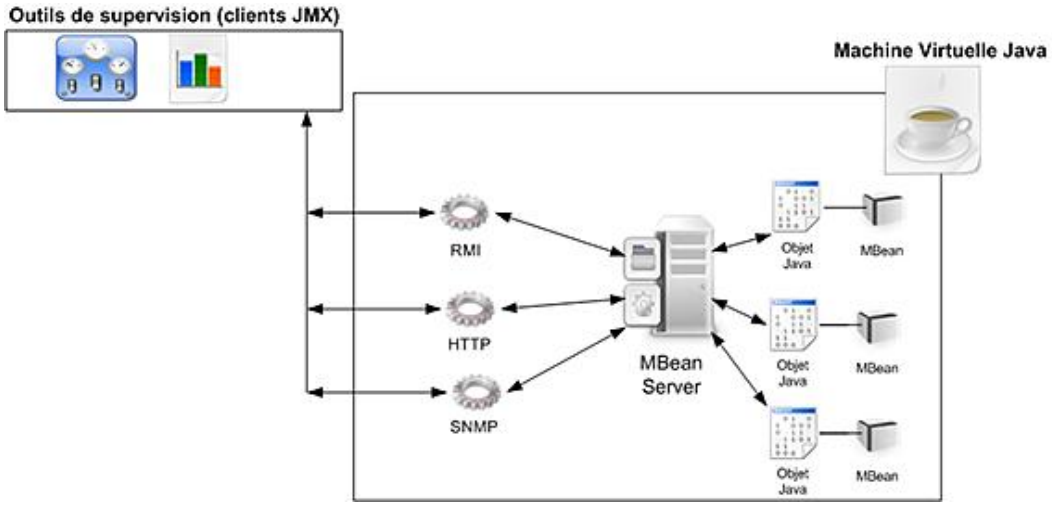
On retrouve la quantité de mémoire disponible (*Free Memory*) dans la machine virtuelle, la mémoire utilisable (*Total Memory*) et la mémoire maximum allouable auprès du système (*Max Memory*).

2. Tester la montée en charge

Nous réaliserons par la suite des tests de montée en charge avec l'outil JMeter. Il est également possible de visualiser le nombre de connexions JDBC disponibles à un instant précis, le nombre de threads occupés, la mémoire consommée, les classes chargées...

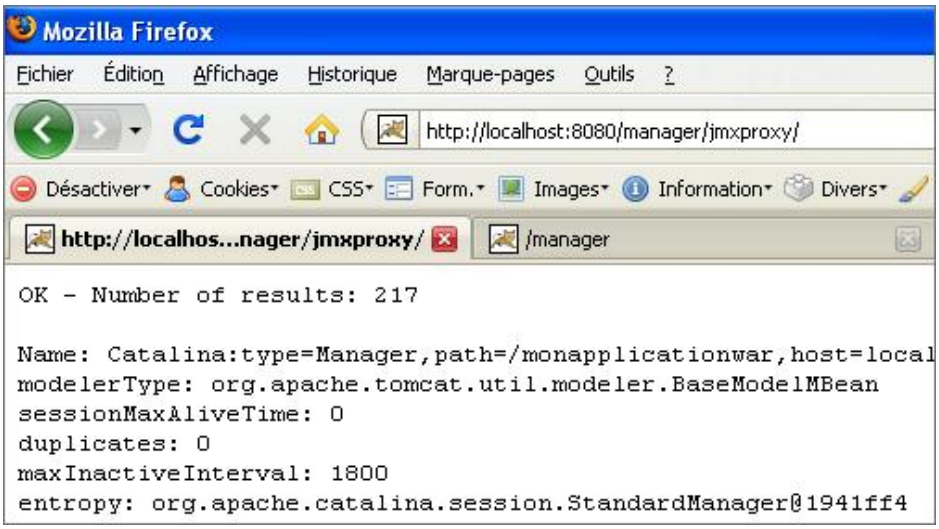
Java Management Extensions JMX est une API Java conçue pour la supervision des applications. JMX est intégrée à partir de la plate-forme Java 1.5. Actuellement, JMX est le standard pour le développement des outils de supervision et d'administration dans les technologies Java.

Dans une machine virtuelle Java, il est possible d'associer aux différents objets Java des composants qui permettent d'obtenir des informations sur l'exécution et le traitement des objets. Ces composants sont appelés des MBeans. Ces MBeans sont accessibles via l'élément central, le serveur MBeans qui est capable d'utiliser des protocoles variés pour la connexion d'outils de supervision.

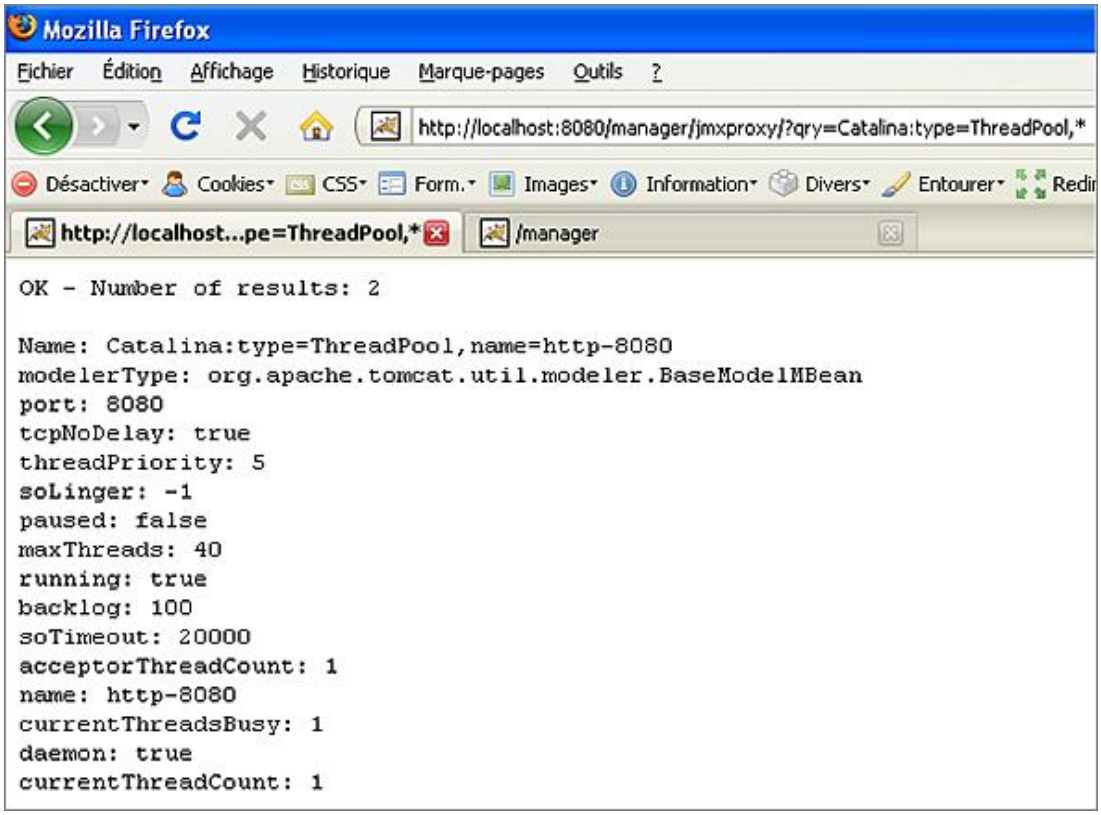


Donc les données des MBeans peuvent être accédées grâce à un client JMX en local ou bien à distance. Tomcat crée des MBeans dynamiquement pendant son exécution, lorsque des applications déployées fonctionnent sur le serveur. Les MBeans dynamiques de Tomcat sont créés grâce aux éléments de configuration appelés *<Listener>* présents dans le fichier *server.xml*.

Les informations JMX sont disponibles à partir du manager à l'URL http://localhost:8080/manager/jmxproxy/. Il est ainsi possible d'obtenir des informations sur les connecteurs, les threads, les classes...



Il est possible d'utiliser des filtres et par exemple d'afficher uniquement les informations sur les objets MBean de type ThreadPool http://localhost:8080/manager/jmxproxy/?qry=Catalina:type=ThreadPool,*.



3. JConsole et MC4J, des consoles JMX

L'ergonomie offerte par le manager de Tomcat est limitée et il est assez difficile de comprendre les paramètres en temps réel. Il existe ainsi plusieurs outils évolués qui permettent d'afficher des rapports détaillés.

a. JConsole

Pour pouvoir obtenir des statistiques en temps réel, il faut utiliser un client lourd JMX. Le logiciel JConsole développé par Sun et livré en standard avec le JDK 1.6 permet d'utiliser JMX. Son utilisation est assez simple et permet la création de graphiques en temps réel. Si JConsole est lancé par défaut sans rien indiquer (*/jdk/bin/jconsole*), nous obtenons des informations sur la machine virtuelle Java installée sur le poste local. Les données ne sont pas ''correctes'' car elles ne concernent pas uniquement la machine virtuelle de Tomcat. Pour cela, il est nécessaire de configurer Tomcat avec un connecteur en lui passant des options au démarrage (port pour se connecter au serveur Mbean...). De même, il est important de sécuriser cette connexion par identifiant et mot de passe afin d'éviter à des

hôtes distants de se connecter à notre machine.

Sous Windows

Avant de pouvoir connecter Tomcat à JConsole, il faut configurer la machine virtuelle Java du serveur pour autoriser l'accès distant de son connecteur. Pour cela, il est nécessaire d'ajouter des options à la machine virtuelle via le script de démarrage de Tomcat. Pour cela, il est nécessaire de configurer la sécurité d'accès au connecteur.

Sécurité

La configuration de la sécurité démarre par la création des fichiers *jmxremote.access* et *jmxremote.password*. Il existe plusieurs possibilités pour la mise en place de ces fichiers. Ces fichiers peuvent être utilisés dans le répertoire */conf* du serveur Tomcat ou dans le répertoire */jdk/jre/lib/management* de l'installation du JDK Java (c'est cette seconde option qui est utilisée dans ce guide).

Le fichier *jmxremote.access* permet d'indiquer les noms d'utilisateurs autorisés à utiliser le connecteur ainsi que les permissions sur le connecteur.

Le fichier *jmxremote.password* contient les mots de passe associés aux utilisateurs.

Le répertoire */jdk/jre/lib/management* contient des fichiers d'exemples qui peuvent être renommés pour l'utilisateur.

Nous allons copier et renommer le fichier *jmxremote.password.template* en *jmxremote.password*. Nous ajoutons ensuite notre utilisateur en fin de fichier.

```
# Following are two commented-out entries.  The "measureRole" role has
# password "QED".  The "controlRole" role has password "R&D".
#
# monitorRole  QED
# controlRole   R&D
moniteurjava monmotdepasse
```

Nous pourrons donc utiliser cet utilisateur avec son mot de passe pour se connecter au serveur MBean mais pour le moment, nous ne pourrons rien faire car l'utilisateur ne possède aucun droit. Le fichier *jmxremote.access* permet de définir les droits en fonction des utilisateurs du fichier *jmxremote.password*.

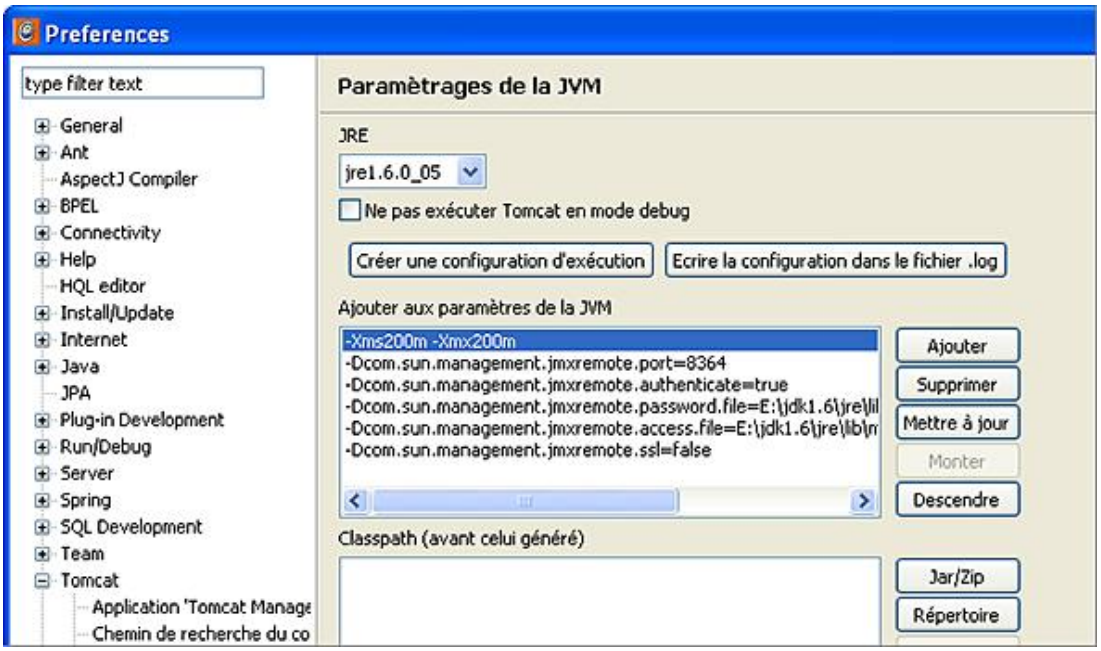
Nous éditons donc ce fichier et ajoutons la ligne suivante :

```
# Default access control entries:
# o The "monitorRole" role has readonly access.
# o The "controlRole" role has readwrite access.
monitorRole   readonly
controlRole   readwrite
moniteurjava readwrite
```

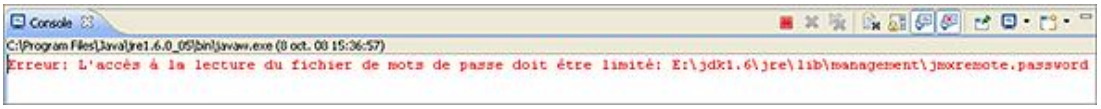
L'option *readwrite* permet d'indiquer qu'il sera possible de réaliser des opérations de lecture sur les objets MBean mais aussi d'écriture ce qui est important par exemple pour vider à distance le garbage collector de la machine virtuelle. Il faut désormais démarrer la machine virtuelle de Tomcat avec un connecteur activé en passant des options au démarrage. Pour cela si le serveur Tomcat est installé avec une archive, le fichier */bin/catalina.bat* peut être modifié.

Par contre, si le serveur Tomcat a été installé en mode service, c'est-à-dire avec l'installation pour Windows, ce fichier n'est pas présent. Il faut dans ce cas ajouter des paramètres dans la console Tomcat ou dans Eclipse pour le démarrage de Tomcat.

```
-Dcom.sun.management.jmxremote.port=8364
-Dcom.sun.management.jmxremote.authenticate=true
-Dcom.sun.management.jmxremote.password.file=
 E:\jdk1.6\jre\lib\management\jmxremote.password
-Dcom.sun.management.jmxremote.access.file=
 E:\jdk1.6\jre\lib\management\jmxremote.access
-Dcom.sun.management.jmxremote.ssl=false
```



Si le serveur est lancé, alors l'erreur suivante est obtenue dans la console d'Eclipse.



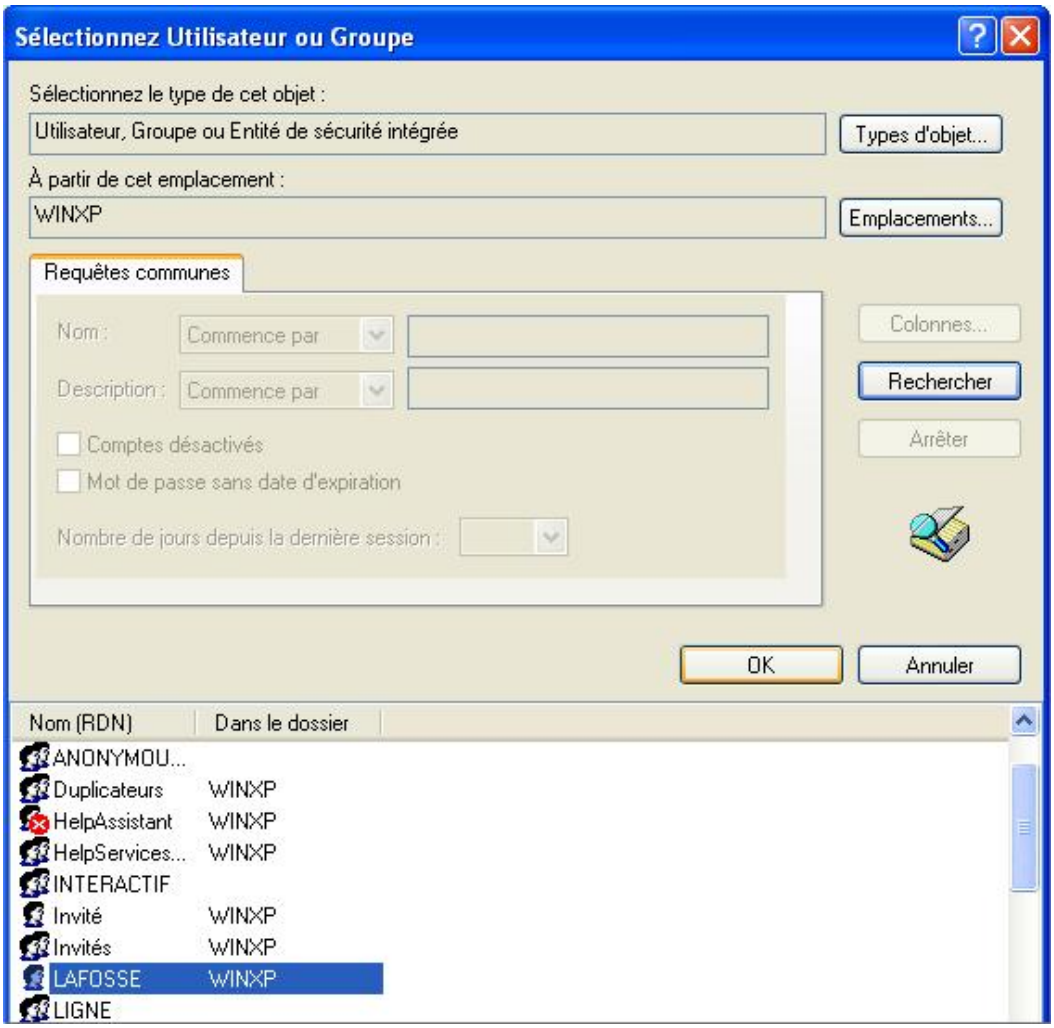
Cette erreur indique que le propriétaire du fichier doit être l'utilisateur qui démarre le serveur.

Sous Windows XP, dans le menu **Outils**, il faut choisir **Options des dossiers**, puis dans la fenêtre cliquer sur l'onglet **Affichage** et décocher **Utiliser le partage de fichiers simple**. Cette action permet de gérer la sécurité sur les fichiers.

Il faut ensuite réaliser un clic droit sur le fichier *jmxremote.password* puis sélectionner l'onglet **Sécurité** et cliquer sur le bouton **Paramètres avancés**. Il faut alors sélectionner l'onglet **Autorisations** et décocher la case **Hérite de l'objet parent les entrées...**

Une boîte de dialogue apparaît alors, il faut répondre **Copier** à la question posée.

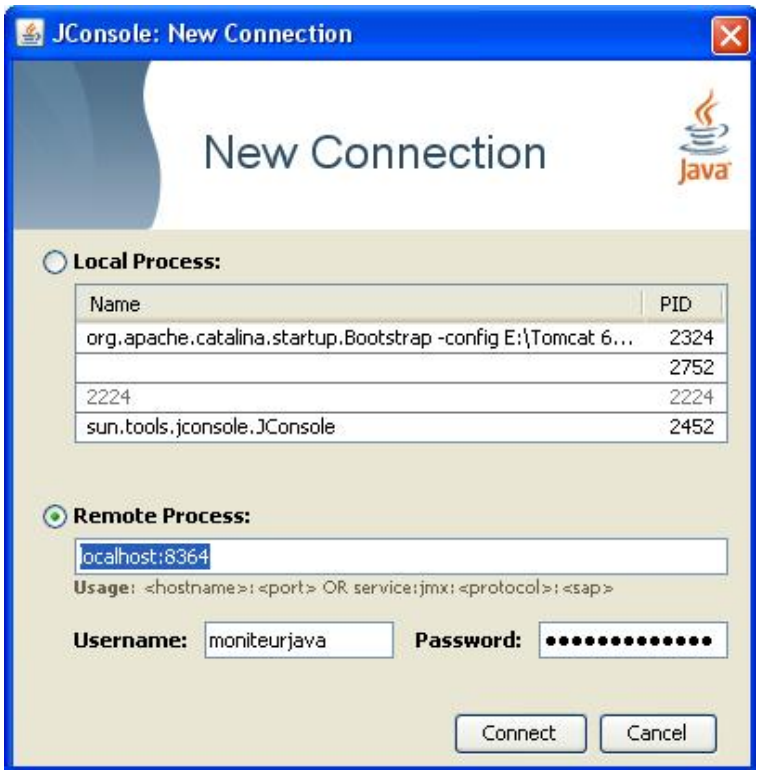
Ensuite, dans la partie **Autorisations**, il faut supprimer tous les utilisateurs autorisés et sélectionner l'utilisateur **Administrateur** du système qui a les droits sur la machine virtuelle Java.



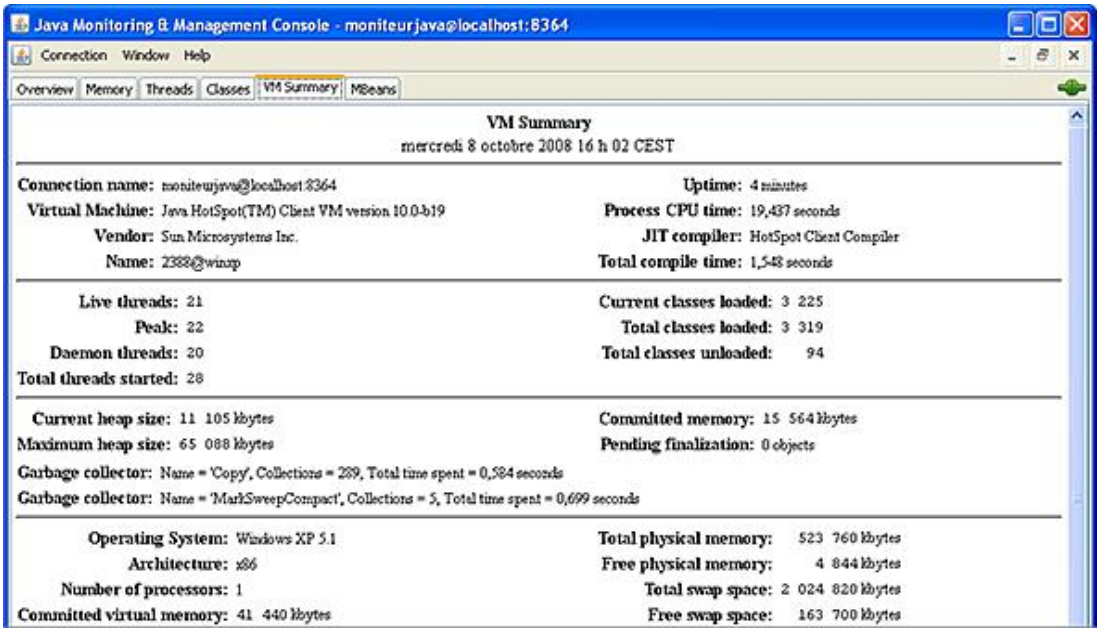
Nous ne devons nous retrouver qu'avec notre utilisateur autorisé.

Le serveur Tomcat peut être lancé, le message d'erreur disparaît. Désormais si nous lançons l'outil JConsole, nous pourrons toujours nous connecter en local mais également à la partie des objets MBean qui contrôlent Tomcat.

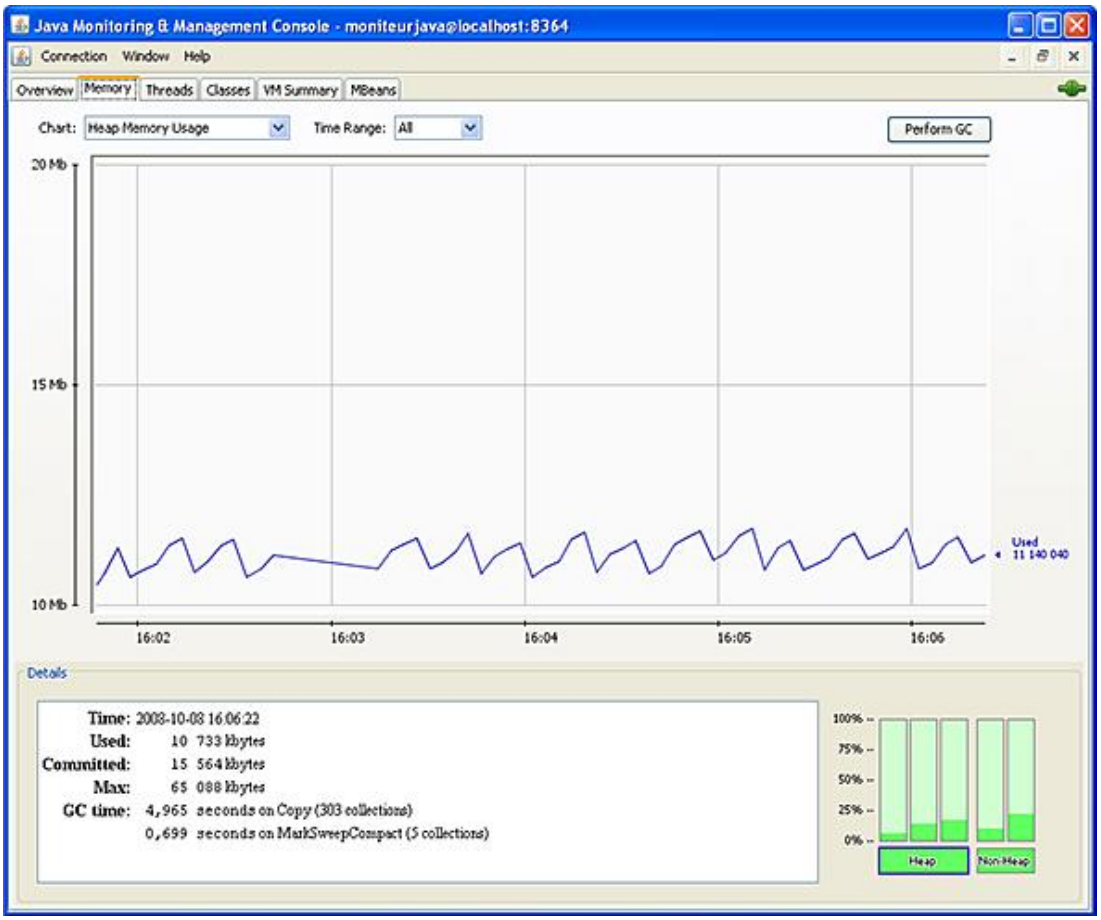
Pour cela, il faut utiliser nos informations de connexion.



Le champ *Remote Process* correspond à l'adresse locale de la machine et au port MBeans. Les champs *Username* et *Password* correspondent aux informations de connexion du fichier *jmxremote.password*.



Dans ce cas de figure, les indications sont correctes et conformes à celles indiquées par le *manager* de Tomcat. Des informations précises sur les applications déployées par le serveur sont obtenues. Dans le premier onglet, il y a un résumé pratique avec les Threads exécutés, la consommation de la mémoire et les classes chargées. Le deuxième onglet permet d'afficher des informations sur la consommation mémoire ce qui est indispensable lors de développement de gros projets afin de vérifier si les connexions JDBC ne sont pas fermées, si des classes ou Servlets consomment trop de mémoire... Les derniers onglets permettent d'avoir des informations sur les Threads, classes, objets MBean et la machine virtuelle.



Nous voyons ici un fonctionnement conforme de la mémoire avec les parties chargement et déchargement.

Sous Linux

Sous Linux l'installation est pratiquement identique. Nous éditons le fichier */usr/local/tomcat/bin/catalina.sh* qui permet de démarrer Tomcat. Nous ajoutons alors uniquement les deux lignes suivantes en milieu de fichier vers la ligne 260 qui est relative à la variable JAVA_OPTS et qui correspond au démarrage.

```
-Dcom.sun.management.jmxremote.port=8364
-Dcom.sun.management.jmxremote.ssl=false
```

Nous procédons ensuite comme sous Windows et nous copions le fichier d'exemple *jmxremote.password.template* en *jmxremote.password*.

```
#cp /usr/local/jdk/jre/lib/management/jmxremote.password.template
/usr/local/jdk/jre/lib/management/jmxremote.password
```

Nous modifions les droits sur ce fichier.

```
#chmod 744 jmxremote.password
```

Comme sous Windows, nous ajoutons notre utilisateur JMX au fichier *jmxremote.password*.

```
# Following are two commented-out entries.  The "measureRole" role has
# password "QED".  The "controlRole" role has password "R&D".
#
# monitorRole  QED
# controlRole   R&D

moniteurjava monmotdepasse
```

Pour la gestion des droits sur le fichier, c'est plus simple que sous Windows, il suffit de positionner les droits suivants sur le fichier.

```
#chmod 600 jmxremote.password
```

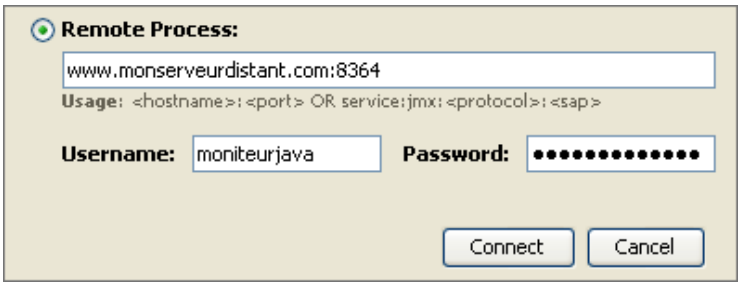
Ensuite, ce qui est important c'est que l'utilisateur qui lance la JVM soit le propriétaire de ce fichier.

```
#chown tomcat:tomcat jmxremote.password
```

Désormais, nous allons déclarer nos droits avec le fichier *jmxremote.access*.

```
# o The "monitorRole" role has readonly access.
# o The "controlRole" role has readwrite access.

monitorRole   readonly
controlRole   readwrite
moniteurjava readwrite
```

L'application de supervision et le connecteur sont désormais opérationnels. Il est donc possible d'utiliser une interface distante pour se connecter au serveur et obtenir des informations de supervision.

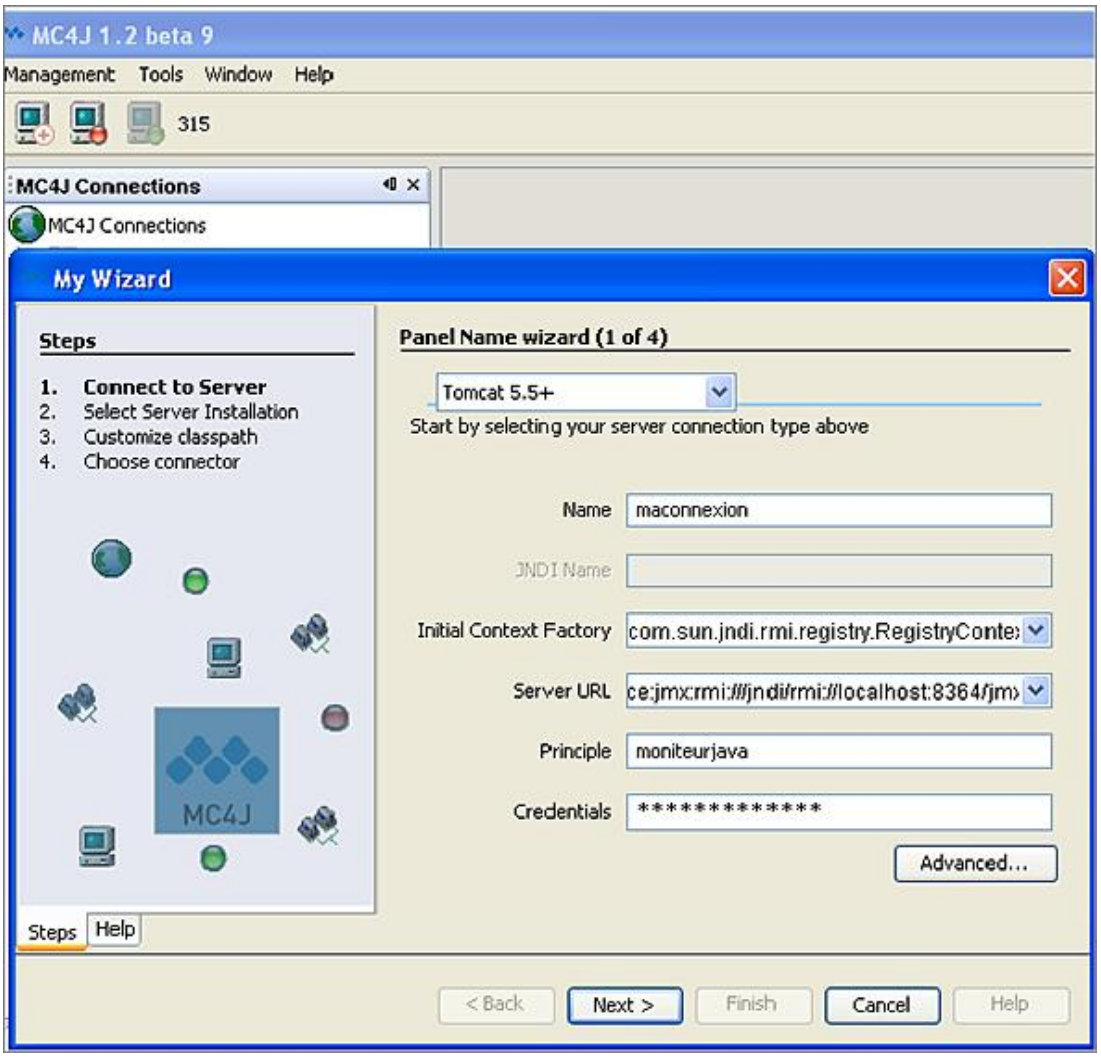


b. MC4J

MC4J est une application JMX encore plus puissante que JConsole. Ce logiciel libre simple d'utilisation est téléchargeable à l'adresse suivante : http://mc4j.org.

Ce logiciel est orienté vers la supervision des serveurs d'applications Java EE. MC4J est par exemple compatible avec Tomcat, JBoss, Weblogic, WebSphere... Il est également disponible pour la majorité des systèmes d'exploitation avec une version pour Windows, Linux et Mac OS X. Après le téléchargement et l'installation du logiciel, il peut être lancé depuis la machine locale ou n'importe quelle machine distante capable d'accéder à la machine locale.

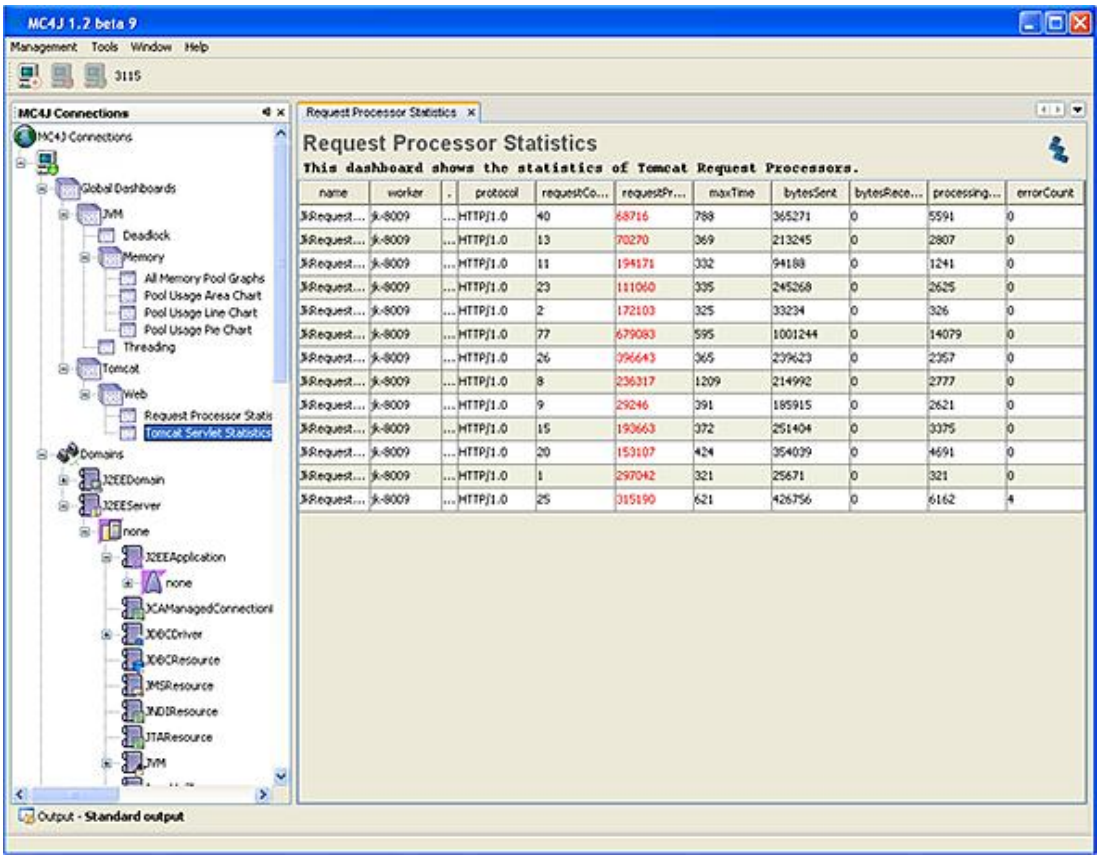
Après le démarrage de MC4J, il faut configurer une connexion au serveur Tomcat en choisissant **Create Server Connection** dans le menu **Management** de l'interface graphique. En premier, le type de serveur auquel se connecter doit être indiqué. Il faut ensuite donner un nom à notre connexion et le numéro de port pour la connexion. Ensuite, si la sécurité est activée sur le serveur d'applications, il faut saisir les informations d'authentification.

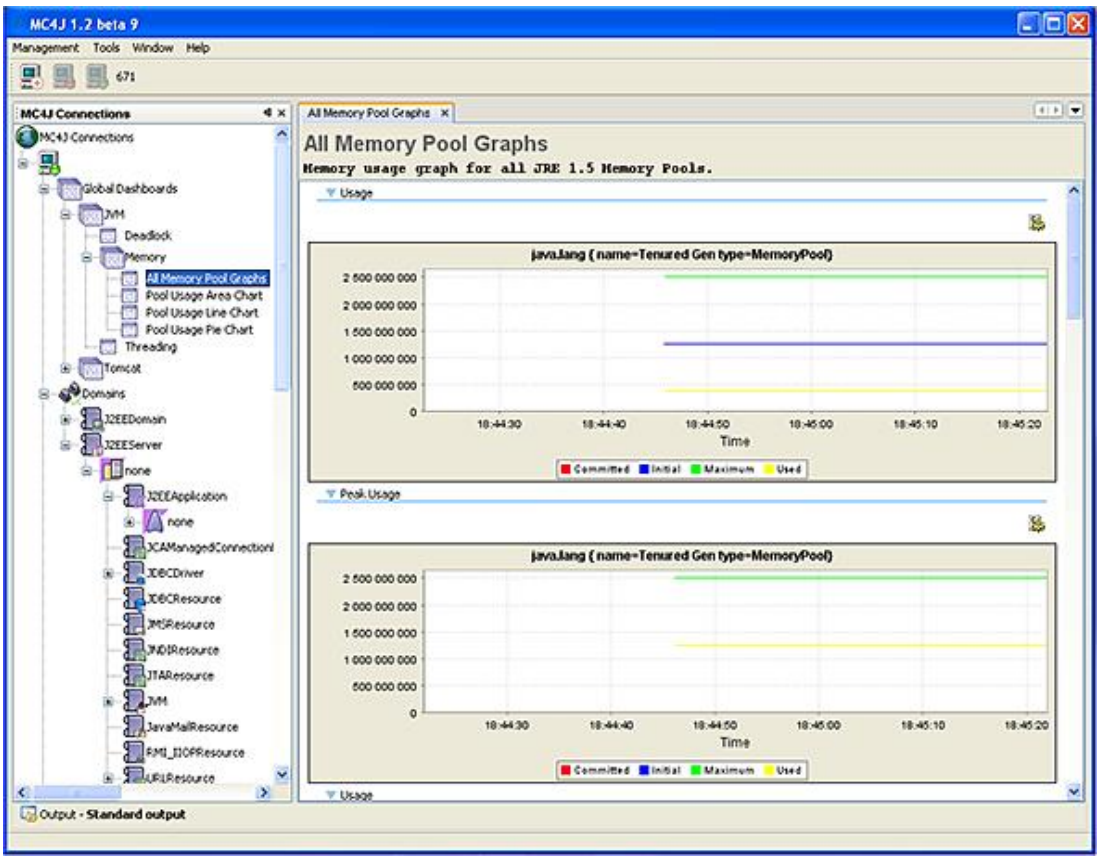


Lors de la deuxième étape, l'assistant demande de préciser le répertoire d'installation de Tomcat. Dans le cas où MC4J et le serveur Tomcat à superviser sont sur des machines différentes, il faudra tout de même installer une version de Tomcat en local (avec une version identique du serveur à superviser) car MC4J utilise les classes de Tomcat pour la supervision.

Une fois la connexion établie, l'écran principal de la console MC4J apparaît avec la liste des MBeans du serveur. Le temps de chargement peut parfois être assez long en fonction de la connexion réseau et de la taille du projet à superviser. Beaucoup d'informations utiles pour la supervision du projet sont alors affichées comme les classes chargées en mémoire, les connexions JDBC, les requêtes des clients.

Avec un outil comme MC4J, il est désormais assez facile de suivre l'évolution en temps réel des ressources internes du serveur pendant un test de montée en charge ou le développement de celui-ci. Cet outil est également très pratique sur un serveur en production, il fait partie des logiciels indispensables d'un administrateur de serveur Java EE.





4. JMeter et les tests de montée en charge

Après avoir correctement configuré le serveur et les outils de supervision, il est essentiel d'installer un outil pour tester la montée en charge du serveur. Il est important d'utiliser ces outils lors de développement Java EE afin de simuler la

charge et d'en mesurer l'impact et les conséquences afin de valider les choix de développement, les pages de code...

Par exemple, lors du développement d'une application Java EE de gestion et de traitement d'images, il sera nécessaire de bien vérifier que lors de l'appel par 100 utilisateurs en simultané, le serveur peut répondre dans un délai raisonnable, que l'application ne va pas saturer la mémoire allouée au serveur... De même, lors d'un choix de conception sur un composant de programmation, l'outil de test pourra nous permettre de valider les librairies, logiciels ou architectures (ex : JDBC ou Hibernate). Il existe plusieurs outils sur le marché pour les tests de montée en charge, tels que l'outil LoadRunner de Mercury et le logiciel libre Apache JMeter.

Apache JMeter

JMeter est un outil écrit en Java et développé par la fondation Apache. JMeter peut générer des tests sur les serveurs Web HTTP et HTTPS, des bases de données avec JDBC, des annuaires et des serveurs FTP. JMeter fournit enfin un ensemble d'outils pour collecter et mesurer les résultats des tests. Du point de vue de la programmation, il pourra analyser plusieurs types de ressources comme des fichiers Java, des Servlets et pages JSP. Apache JMeter est 100 % Java, il requiert donc juste une machine virtuelle Java pour fonctionner et lancer les tests.

JMeter peut être installé sur la même machine que le serveur d'applications, mais afin de ne pas fausser les tests, il vaut mieux éviter d'exécuter JMeter et Tomcat sur les mêmes machines.

L'installation de JMeter est simple, il faut télécharger les fichiers binaires de JMeter sur le site http://jakarta.apache.org/jmeter. Il y a une archive au format *.zip* pour Windows et *.gz* pour Linux. L'installation nécessite simplement la décompression de l'archive dans un répertoire du système. Il est bien sûr indispensable d'avoir installé au préalable un JRE ou JDK correctement avec la variable d'environnement *JAVA_HOME* paramétrée.

Pour lancer JMeter, il faut utiliser le script */bin/jmeter.bat* ou l'exécutable */bin/jmeter.exe* sous Windows et la commande */bin/jmeter* sous Linux.

Il est possible d'ajouter des librairies à JMeter pour tester des connexions JDBC par exemple. Pour cela, il suffit de copier la librairie dans le répertoire */lib* et */lib/ext* de JMeter.

Si JMeter détecte une erreur, celle-ci sera écrite dans le fichier de log appelé */log/jmeter.log*. Ce fichier est créé au lancement de l'application. Le fichier de configuration de JMeter pour SSL, proxy et utilisateurs est */bin/jmeter.properties*.

Utilisation

Au démarrage de l'interface, le plan de tests (*Test Plan*) et le plan de travail (*WorkBench*) sont affichés. Un plan de test permet de décrire la série de tests à exécuter par JMeter. Pour ajouter ou supprimer un plan de tests, il faut faire un clic droit dans l'explorateur. Il est également possible de charger des éléments de tests présents dans un fichier. Un plan de tests est sauvegardé au format *JMX*. Un plan de travail contient les éléments qui ne sont pas utilisés par le plan de tests, il s'agit d'un espace de stockage temporaire.

Avant de commencer l'écriture d'un plan de tests, il est nécessaire de configurer le serveur et les ressources dans des conditions qui soient le plus proche possible d'un environnement en production. Il faut donc paramétrer correctement la mémoire allouée, le serveur Tomcat, les connecteurs aux bases de données...

Un plan de tests consiste à tester la montée en charge du serveur en simulant des accès distants.

Il existe plusieurs étapes essentielles pour la construction d'un plan de test :

- Définir le groupe de threads qui simulent le nombre de requêtes en direction du serveur. Un thread est en fait un utilisateur potentiel qui navigue sur le site en production.
- Écrire la configuration du serveur en précisant le nom de la machine en production, le port à utiliser, le protocole...
- Écrire les requêtes HTTP à invoquer par les utilisateurs.
- Ajouter un ou plusieurs composants de mesure JMeter qui va (ou vont) analyser et traiter les réponses sur la charge.
- Enregistrer le plan de tests et le lancer.

Créer un plan de tests

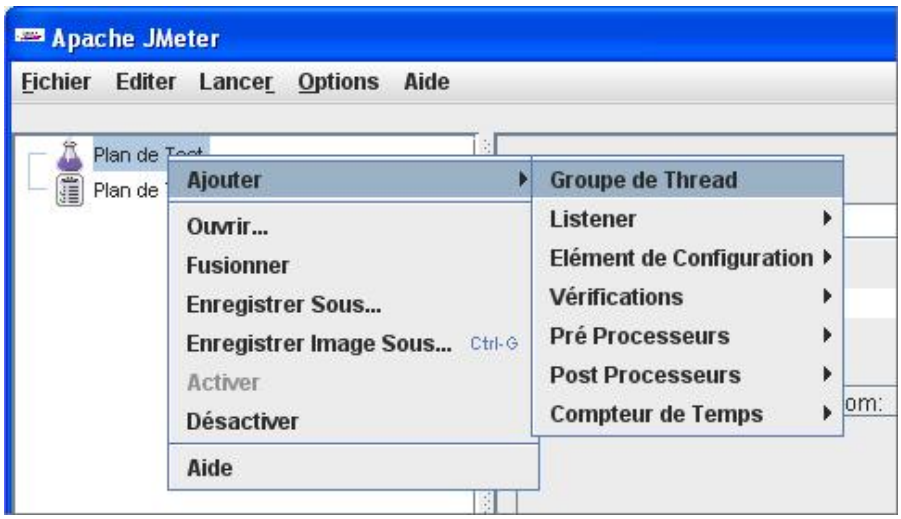
La première étape consiste à créer les utilisateurs (*Threads*) qui vont envoyer des requêtes au serveur. Nous allons créer par exemple 5 utilisateurs qui vont envoyer des requêtes à 2 pages du site, 2 fois de suite. Il y aura donc au

total : 5 utilisateurs * 2 pages * 2 appels = 20 requêtes.

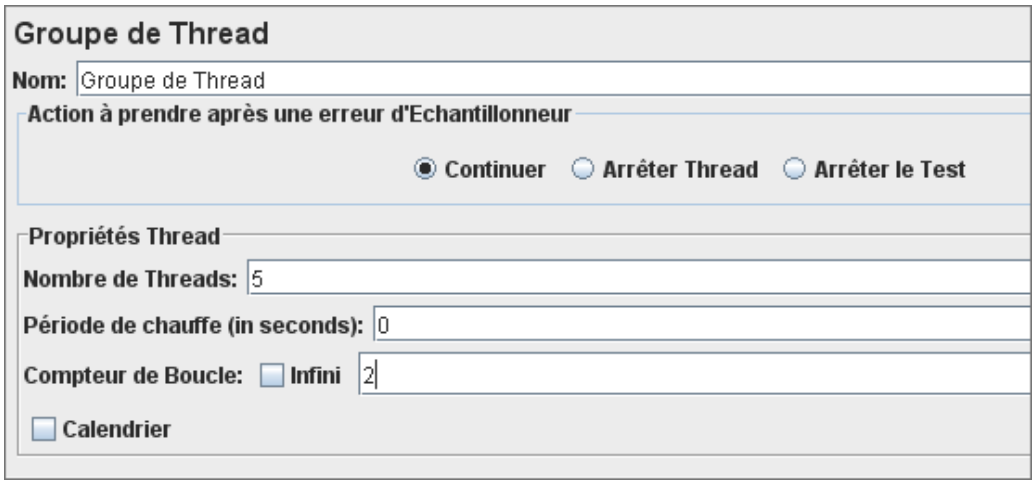
Utilisateur

Il faut ajouter un groupe d'utilisateurs (*ThreadGroup*) à notre plan de tests.

Nous faisons un clic droit sur le plan de tests et l'action **Ajouter - Groupe de Thread**.



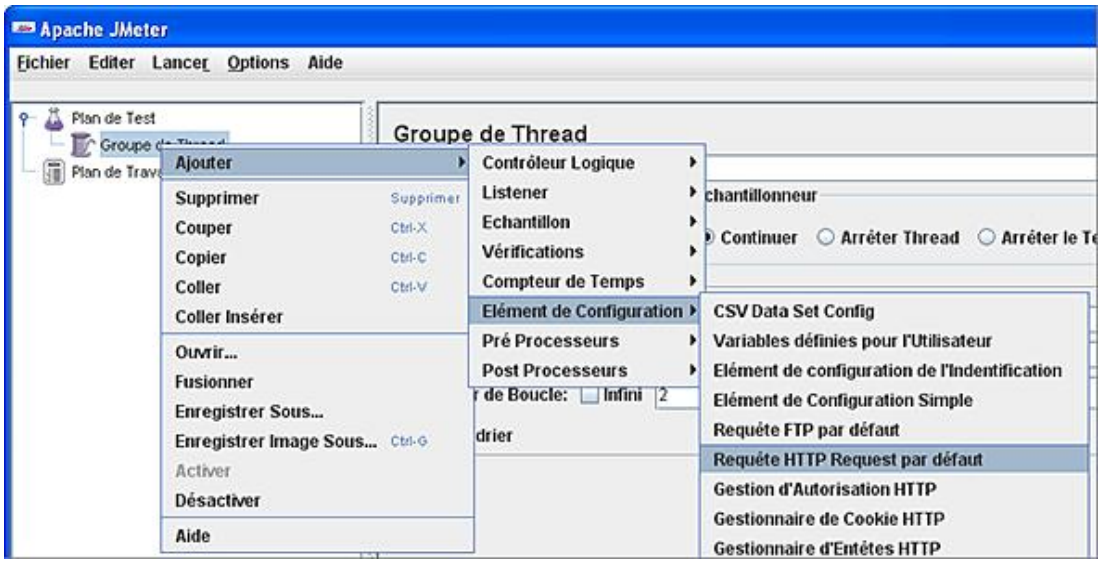
L'écran affiché permet de configurer le nombre de Threads ainsi que l'intervalle de temps pendant lequel ils seront démarrés. La propriété **Nombre de Threads** (*Number of Thread*) correspond au nombre d'utilisateurs, dans notre exemple : 5. La propriété **Période de chauffe** (*Ram-Up*) définit le nombre de secondes entre le déclenchement de chaque utilisateur. En positionnant à 0, JMeter va démarrer tous les utilisateurs en même temps et donc simuler des accès simultanés au serveur d'applications. Enfin, la propriété **Compteur de Boucle** (*Loop-Count*) permet d'indiquer le nombre de boucles à effectuer sur le serveur. Nous positionnons cette propriété à 2 pour effectuer 2 répétitions. Il est aussi possible de rejouer cette séquence à l'infini en cochant la case appropriée.



Configuration du serveur

La seconde étape consiste à préciser le serveur d'applications qui sera utilisé pour les tests.

Il faut d'abord sélectionner le groupe de Thread dans le plan de tests. Puis après un clic droit sur cet élément, les choix **Ajouter (Add) - Elément de Configuration - Requête HTTP Request par défaut** sont sélectionnés.



Les valeurs à préciser sont le protocole utilisé, le nom de domaine du serveur ou son adresse IP et le numéro de port de celui-ci. Des paramètres aux requêtes peuvent être également ajoutés. Dans ce cas, ils seront transmis avec toutes les requêtes en direction du serveur.

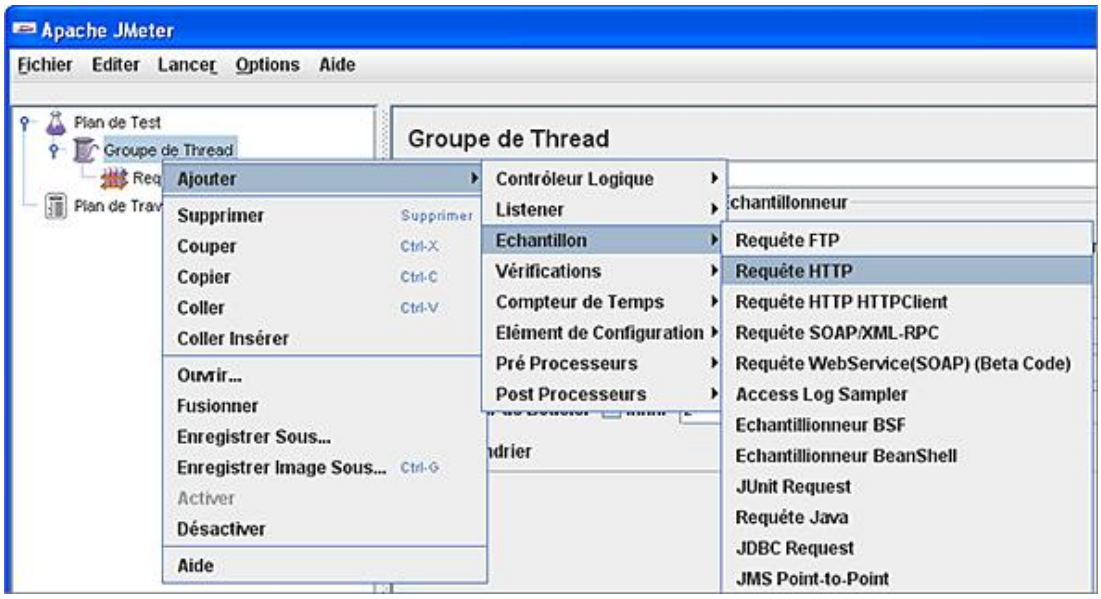
Écrire les requêtes utilisateurs

Lors de cette troisième étape, il est temps d'écrire les requêtes HTTP, l'objectif étant de simuler la navigation d'un utilisateur sur le site en production.

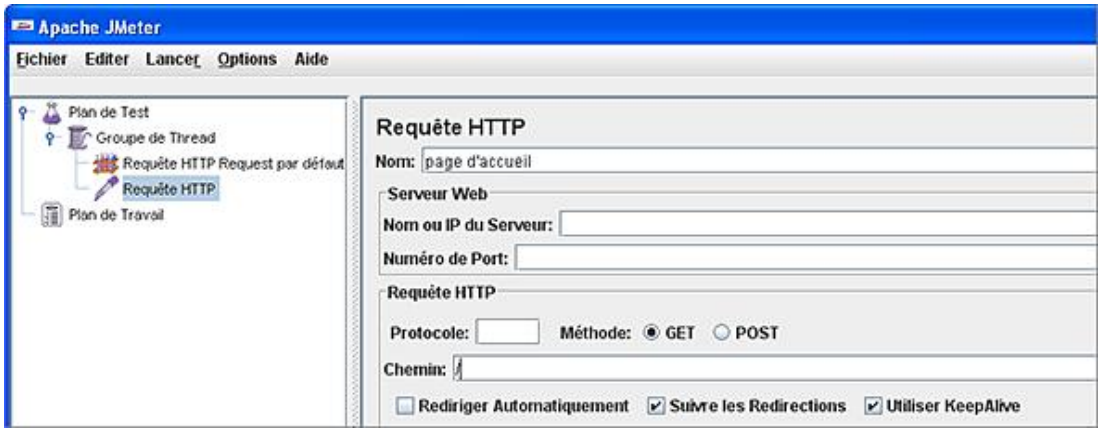
D'après notre plan de tests, il faut utiliser deux requêtes HTTP. La première pour la page d'accueil du site http://127.0.0.1:8080/index.jsp et la seconde vers une page d'aide http://127.0.0.1:8080/aide.jsp.

JMeter va donc envoyer les requêtes dans l'ordre où elles apparaissent dans l'arbre.

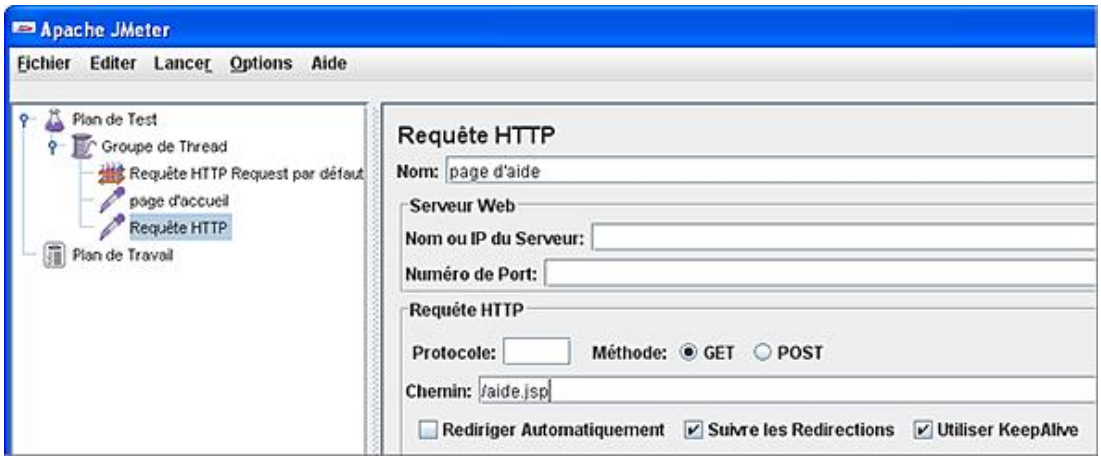
Pour ajouter une page, il faut faire un clic droit sur notre groupe de Threads dans notre plan de travail et ajouter une requête HTTP.



La configuration du serveur a été réalisée précédemment, il n'y a donc que très peu d'éléments à renseigner. Les éléments indispensables sont la méthode HTTP à utiliser (GET ou POST) ainsi que le nom et le chemin de la ressource. Nous donnons un nom explicite à notre requête (page d'accueil) et nous positionnons le chemin d'accès à la page (/ étant donné que c'est la racine du site).



Nous réalisons ensuite la même opération pour notre seconde page/requête.



Ajouter un écouteur/composant de mesure

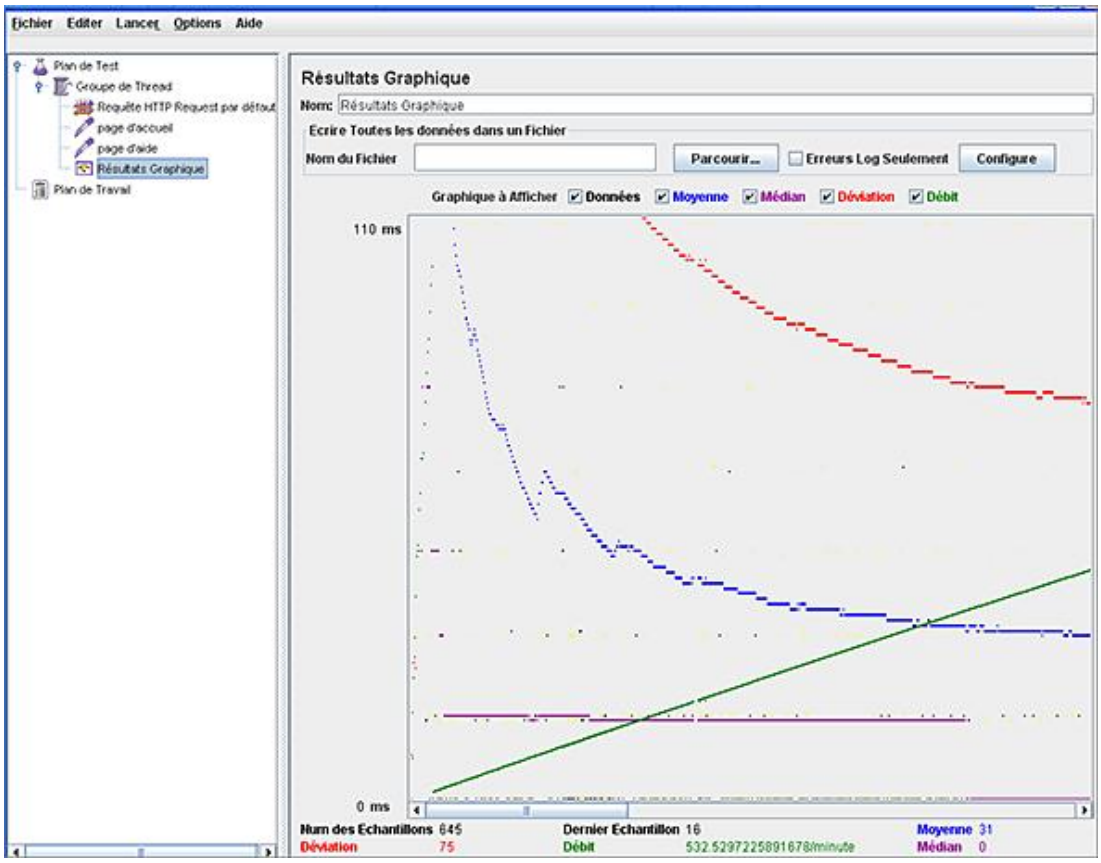
Lors de cette étape, il faut ajouter un ou plusieurs écouteurs pour analyser les tests. Pour ajouter un moniteur de résultats, nous réalisons un clic droit sur le groupe de Thread dans notre plan de tests et nous ajoutons par exemple un écouteur graphique. Nous précisons alors un fichier de sauvegarde pour les résultats en utilisant le bouton **Parcourir**. Le fichier de sortie est au format *.jtl*.

Enregistrer et lancer le plan de tests

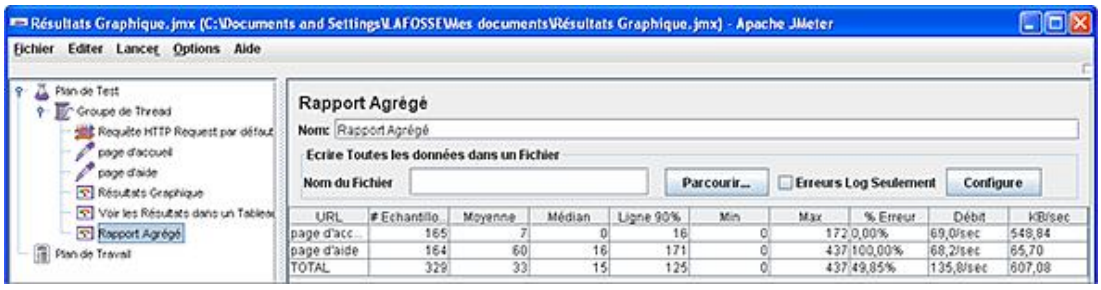
L'enregistrement du plan de tests n'est pas obligatoire mais conseillé. Pour sauvegarder un plan de tests, il faut sélectionner l'item **Enregistrer** du menu et préciser un nom de fichier *.jmx*.

Pour lancer l'exécution du plan de tests, il faut utiliser le menu Démarrer (*Run*). JMeter indique alors avec un petit carré vert lumineux en haut à droite de la fenêtre principale l'exécution des tests. Le rectangle repasse en gris quand les tests sont arrêtés. Après avoir sélectionné *Stop*, le rectangle vert reste allumé jusqu'à la fermeture de tous les Threads.

Il est possible d'ouvrir le même plan de tests avec plusieurs fenêtres d'analyse sans problème. Il est également possible d'utiliser des boucles infinies pour simuler des accès en temps réel sur une plus grande période.



Les résultats d'analyse peuvent être de toute sorte comme des graphiques, des tableaux de données, des arbres...



JMeter est un outil très puissant qui permet d'analyser en temps réel la montée en charge d'un serveur. Les résultats graphiques, textuels et sous forme d'arbres sont très précieux en phases de développement et de production.

Apache-Tomcat et SSL/HTTPS

1. Présentation

Le serveur Tomcat propose dans sa configuration par défaut un connecteur HTTPS préconfiguré. La configuration est en commentaire dans le fichier principal *server.xml*. Il est donc très simple d'activer ce connecteur en réactivant cette ligne et en utilisant des clés de cryptage conformes.

Le connecteur HTTPS de Tomcat utilise l'élément de configuration <*Connector*> avec le port SSL indiqué (8443). Cet élément possède les mêmes attributs que les autres connecteurs mais également les attributs *keystoreFile* (chemin de stockage des clés), *keystorePass* (mot de passe du fichier de stockage des clés), *sslProtocol* (le protocole utilisé par HTTPS), *clientAuth* (certificat client obligatoire ou non), *truststoreFile* (fichier de validation des certificats clients), *truststorePass* (le mot de passe du fichier des certificats clients).

Voici la configuration que l'on peut utiliser avec un certificat auto-signé.

```
<Connector port="8443" maxHttpHeaderSize="8192"
               maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
               enableLookups="false" disableUploadTimeout="true"
               acceptCount="100" scheme="https" secure="true"
               clientAuth="false" sslProtocol="TLS" />
```

Pour la mise en place de Tomcat en mode HTTPS, il est nécessaire de disposer de clés cryptées conformément aux spécifications en vigueur.

Comme pour l'utilisation d'un serveur Tomcat HTTP en production, la configuration idéale en terme de performances et de fiabilité est l'utilisation du moteur HTTP d'un serveur Web à la place de celui de Tomcat. Pour cela, nous utiliserons sur un serveur de production sous Linux, le serveur Apache-SSL. La configuration d'Apache-SSL est semblable à celle d'Apache. Il est cependant nécessaire d'utiliser des clés correctes et une configuration adaptée. Voici ci-dessous un exemple de configuration sur un serveur Linux Debian.

Fichier */etc/apache-ssl/httpd.conf*

```
NameVirtualHost adresseip:443
<VirtualHost adresseip:443>
DocumentRoot /usr/local/tomcat/webapps/monprojet
ServerName monprojet.com
ServerAlias *.monprojet.com
CustomLog /var/log/apache/monprojet-access_log combined
Port 443
SSLRequireSSL
SSLEnable

Alias /javascript /usr/local/tomcat/webapps/monprojet/javascript
JkUnMount /javascript/* worker1
JkAutoAlias /usr/local/tomcat/webapps/monprojet/

 <Location "/WEB-INF">
 AllowOverride None
 deny from all
 </Location>

 <Location "/META-INF">
 AllowOverride None
 deny from all
 </Location>

 <Location "/*.jsp">
 JkMount worker1
 </Location>

 <Location "/*.do">
 JkMount worker1
 </Location>
DirectoryIndex index.html index.htm index.php index.jsp
</VirtualHost>
...
SSLEnable
```

```
...
SSLCertificateFile /etc/apache-ssl/cles/monprojet.cert
SSLCertificateKeyFile /etc/apache-ssl/cles/monprojet.key
```

2. En résumé

Ce chapitre a présenté le serveur d'applications Apache-Tomcat. La première partie a concerné l'installation de Tomcat, la réalisation d'un script de gestion du serveur et la présentation des fichiers journaux.

Dans un deuxième temps, le guide a introduit la mise en place de la partie administration et la gestion de la mémoire allouée au serveur.

La troisième partie plus technique, a précisé comment coupler Tomcat avec le serveur Web Apache.

L'architecture de Tomcat avec l'arborescence et les fichiers de configuration au format XML ont été également détaillés.

Puis un bref rappel XML a été évoqué afin de pouvoir manipuler les fichiers de configuration du serveur Java EE. Cette étape a été suivie d'une présentation de l'arborescence d'un projet/webapp Tomcat avec l'introduction du descripteur de déploiement *web.xml* et le déploiement d'un premier projet simple.

L'avant-dernière partie a concerné la supervision, le monitoring du serveur et de ses applications déployées. L'outil JMX intégré à Tomcat et les outils de manipulation JConsole et MC4J ont été détaillés, suivis de la mise en œuvre de JMeter pour les tests de montée en charge.

Enfin la dernière partie a été axée sur la mise en œuvre de Tomcat avec l'utilisation du protocole HTTPS.

Qu'est-ce qu'une JavaServer Page ?

1. Présentation

Une page JSP est une page HTML qui peut contenir du code Java. D'un point de vue technique, il est possible de faire la même chose avec une page JSP qu'avec une Servlet. La différence est essentiellement au niveau de la structure. Une page JSP est un squelette de page HTML qui contient des morceaux de code Java permettant de réaliser des traitements dynamiques ou d'intégrer des données. Une Servlet est à l'inverse, plutôt composée de code Java et parfois de portions de code HTML pour la gestion de l'affichage.

La technologie JSP permet l'utilisation et la création de balises JSP (taglib) qui ajoutent des fonctionnalités pour étendre les possibilités de développement. Le mécanisme JSP permet de séparer la logique contrôlant la présentation des données de la logique métier, contrôlant la valeur des données. D'un point de vue technique, les JSP sont compilées par le moteur JASPER pour devenir des Servlets Java.

Il existe également des actions JSP (balises XML) qui appellent des fonctions afin de rendre un service spécifique : inclusion de page, redirection...

Le principe de fonctionnement est très proche des langages PHP et ASP mais à l'inverse de ces mécanismes, JSP compile le code au lieu de l'interpréter à chaque fois, ce qui soulage la charge de traitement du serveur.

2. Introduction

Les pages JSP sont exécutées par le serveur d'applications pour répondre aux requêtes des clients. Les JSP ont plusieurs fonctionnalités :

- la création de sites Web dynamiques (pages Web dynamiques) ;
- le travail avec des bases de données ;
- l'amélioration de la sécurité (le code est exécuté par le serveur et donc non accessible par les clients).
- l'utilisation des JavaBeans (simplicité du code, manipulation d'objets simples...) ;
- l'utilisation de balises personnalisées (taglib).

Avec ce modèle de programmation, il est plus aisé de scinder le développement d'une application Web : le graphiste s'occupe de la partie présentation HTML et le développeur s'occupe de la partie logique avec les traitements et l'accès aux données.

Côté serveur, une page JSP est interprétée une seule fois, soit lors du premier appel, soit lors du lancement du conteneur Web. En fait, le conteneur Web crée une Servlet à partir de la page JSP. La Servlet est ensuite compilée, chargée en mémoire (s'il n'y a pas d'erreur) et mise en service pour répondre aux requêtes clients.

Il existe une Servlet système dans le conteneur Java EE qui est configurée pour traiter les ressources concernant les fichiers portant l'extension *.jsp* et *.jspf* (*javax.servlet.jsp* et *javax.servlet.jsp.tagext*) .

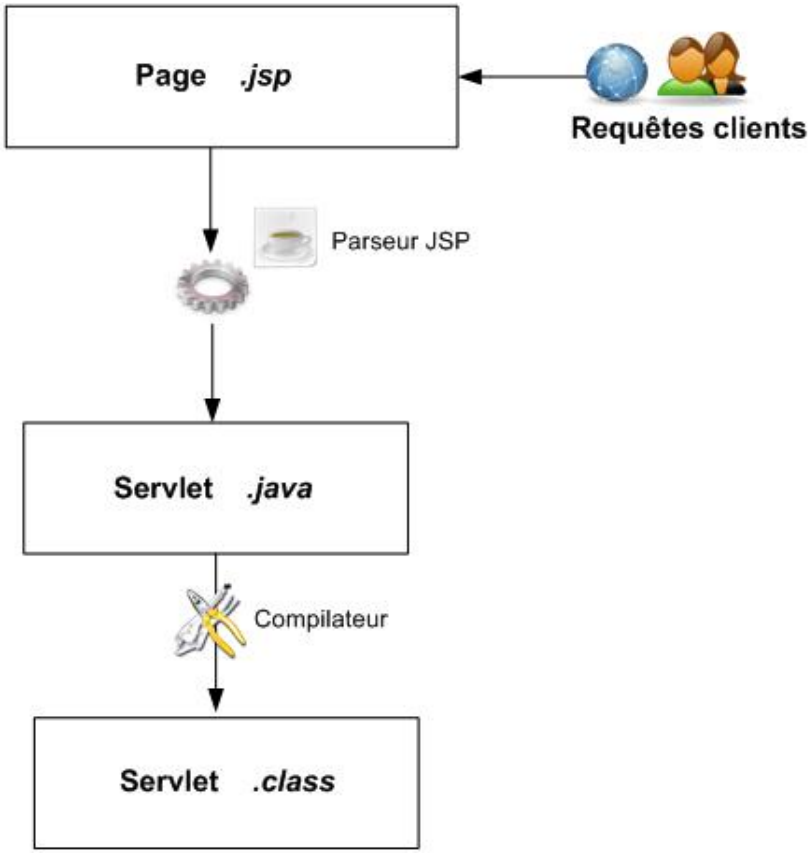
3. JASPER

Les JSP sont compilées par le compilateur JASPER (il en existe d'autres) pour devenir des Servlets JAVA. Tomcat intègre un moteur de Servlets appelé CATALINA. JASPER génère une Servlet source Java (fichier *.java*) qui est à son tour compilée par CATALINA pour devenir une Servlet exécutable (fichier *.class*). La syntaxe propre à Java est encadrée par des balises <% et %>, ce qui explique au compilateur JASPER que le reste du code doit être traité comme du langage HTML.

4. Cycle de vie d'une Servlet

Une JSP est composée de texte brut contenant du code HTML et du code Java. Lors du premier appel de la page par un utilisateur distant, la page JSP est traduite en une Servlet Java par l'intermédiaire du parseur JSP contenu dans le serveur Java EE.

Le compilateur va ensuite traduire ce fichier Java généré, en un fichier *.class*, et ce nouveau fichier *.class* est alors exécuté par la machine virtuelle.

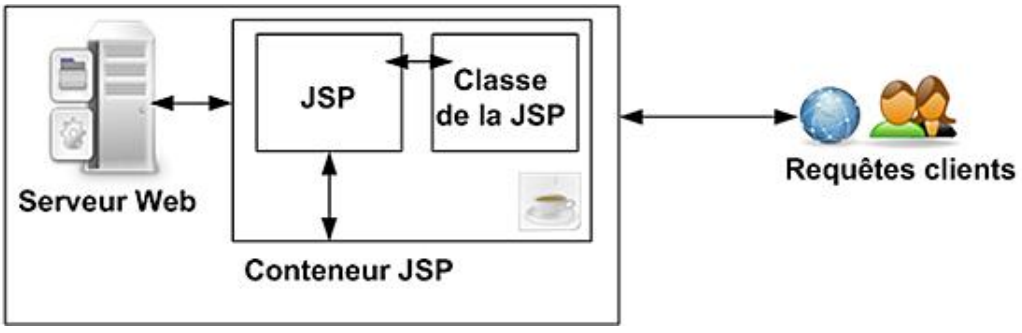


La transformation de la page *.jsp* en Servlet *.class* n'est effectuée qu'une seule fois, lors du premier appel de la page. C'est pour cela, que le premier chargement de la page est beaucoup plus long que les suivants, qui ne nécessitent pas cette étape. Les utilisateurs suivants n'auront pas ce temps d'attente puisque la page est déjà compilée et chargée en mémoire. En cas de modification de la page, celle-ci est à nouveau compilée et mise à jour automatiquement.

Le cycle d'une page JSP comporte quatre phases :

- **Chargement et instanciation** : le serveur localise la classe Java correspondant à la page JSP et la charge dans la machine virtuelle.
- **Initialisation** : la page JSP est initialisée selon le principe d'une Servlet.
- **Traitement des requêtes clients** : la page répond aux requêtes des clients. Aucun code Java n'est envoyé aux clients.
- **Fin du cycle** : toutes les requêtes sont traitées et terminées, les instances de la classe sont détruites.

Lorsqu'un client envoie une requête à une page JSP, le serveur transmet celle-ci au conteneur de JSP et celui-ci détermine la classe d'implémentation qui devra la traiter.



Une page JSP est en fait une Servlet ce qui signifie que tout ce qui est utilisable pour une Servlet (cookies, sessions, context...) peut être également mis en œuvre dans les pages JSP.

Pour résumer, les développeurs et concepteurs graphique utilisent la technologie JSP pour développer rapidement des pages Web dynamiques au contenu riche. Les pages JSP peuvent être créées facilement et entretenues car elles sont basées sur du code HTML et XML. Toute la puissance de Java est alors cachée derrière les pages JSP.

Déclarations, commentaires et scriptlets

1. Présentation

Une page JSP porte l'extension *.jsp* et doit être accessible avec un navigateur dans l'arborescence du serveur Java EE. Les pages sont placées dans le répertoire de l'application (*webapp*) à partir de la racine ou dans un répertoire spécifique (*/jsp, /vues, /vues/articles/*...). Les pages JSP sont conçues pour avoir un comportement dynamique et elles sont chargées de répondre aux requêtes envoyées par les clients. C'est le code Java inséré dans le code HTML qui permet ce comportement dynamique.

2. Les éléments JSP

Nous avons besoin, comme pour d'autres langages de scripts (ex : PHP), d'indiquer au serveur où commence et s'arrête le code HTML et le code Java. Pour cela, la spécification JSP définit des balises qui peuvent être employées pour délimiter le code. Ces balises permettent de définir trois catégories d'éléments :

- **Les directives** : informations sur la page.
- **Les scripts** : placement du code dans la page.
- **Les actions** : inclusion de code basé sur des balises dans la page courante.

3. Les directives

Les directives sont des éléments qui permettent de préciser des informations relatives à la page. Il existe les trois directives suivantes : *page*, *include* et *taglib*. Les directives JSP commencent par <%@ et se terminent par %>. Les directives sont parfois appelées directives de moteur JSP. Elles fournissent des instructions et des paramètres qui déterminent la façon dont une page JSP est traitée. Par contre, une directive n'affecte que la page JSP qui la contient. Chaque directive a des attributs auxquels peuvent être affectées des valeurs spécifiques. L'instruction de directive inclut le nom de la directive, suivi de l'attribut et des paires de valeurs dont l'utilisation est souhaitée.

La directive page

L'élément directive de page permet de définir les attributs de la page JSP, comme les importations de classes ou paquetages, la page invoquée en cas d'erreur, le type de contenu... La directive *page* peut être définie plusieurs fois dans un même document. Il est recommandé de toujours placer cette directive en tout début de document pour une question de logique et de lisibilité.

Nous pouvons développer une page d'erreur générique pour le projet *betaboutique*. Cette page est placée dans un répertoire nommé */vues* afin de bien séparer les éléments de traitements (Servlets) des éléments d'affichage (JSP).

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="java.util.ArrayList" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>ERREURS</title>
</head>
<body>
<%
ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
%>
<h2>Les erreurs suivantes se sont produites</h2>
<ul>
    <%
    for(int i=0;i<erreurs.size();i++)
    {
        out.println("<li>"+(String)erreurs.get(i)+"</li>");
    }
```

```
    %>
</ul>
</body>
</html>
```

Il faut remarquer la définition de la directive *page* qui permet à la première ligne, par l'intermédiaire de ses attributs, d'indiquer le langage utilisé, le type mime et l'encodage.

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

L'attribut *language* permet de préciser le langage utilisé, Java dans tous les cas. L'attribut *contentType* indique le type MIME utilisé par la page JSP. L'attribut *pageEncoding* indique le type d'encodage de caractère utilisé dans la JSP pour la réponse HTTP. Il existe plusieurs autres attributs comme *errorPage* qui permet de spécifier la page JSP à afficher en cas d'erreur. L'attribut *info* permet de faire une description de la JSP et peut être récupéré par l'instruction *getServletInfo()*. L'attribut *session* permet par exemple d'indiquer si le contexte de session HTTP est accessible ou non dans la JSP.

Il est possible de modifier la première ligne de cette page et l'attribut *contentType* pour afficher du texte brut.

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

Il y a également sur la seconde ligne du fichier un autre exemple d'utilisation de la directive *page* avec l'importation de la classe Java *ArrayList* pour la manipulation d'une collection d'objets.

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

De cette manière, les objets *ArrayList*, ainsi que toutes les fonctions de cette classe, pourront être manipulés et utilisés.

La directive include

Nous pouvons également modifier légèrement notre page d'erreur pour utiliser le second type de directive, à savoir la directive *include*. Dans notre exemple, une page HTML statique est utilisée pour inclure un message descriptif en début de la JSP. Une autre page JSP pourra bien entendu être incluse sans problème.

Le projet *betaboutique* utilise une Servlet d'authentification nommée *ServletAuthentification* qui permet de vérifier la syntaxe de l'identifiant et du mot de passe. En cas d'erreur, les données sont transférées à la page JSP *erreurs.jsp*. Cette page est chargée d'afficher les raisons de l'échec de l'authentification. Voici le code de cette page avec l'inclusion d'une page d'en-tête au format HTML.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="java.util.ArrayList" %>
<%@ include file="enteteerreurs.html" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>ERREURS</title>
</head>
<body>
<%
ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
%>
<h2>Les erreurs suivantes se sont produites</h2>
<ul>
    <%
    for(int i=0;i<erreurs.size();i++)
    {
        out.println("<li>"+(String)erreurs.get(i)+"</li>");
    }
    %>
</ul>
</body>
</html>
```

Si aucune directive n'est utilisée dans une page JSP, le moteur JSP du serveur Web utilisera ses propres paramètres par défaut. Par exemple, si la directive *page* avec l'attribut *contentType* est omise, le moteur JSP

affichera automatiquement les informations comme une page WEB (*contentType=”text/html”*).

Chaque page JSP compilée est transformée en une Servlet placée dans le répertoire */work/org/apache/jsp*. Pour cette page nommée *erreurs.jsp*, la Servlet générée est présente sous le nom : */work/org/apache/jsp/erreurs_jsp.jsp*. Son code compilé est présent dans le fichier : */work/org/apache/jsp/erreurs_jsp.class*.

Voici le code de la JSP transformée en Servlet par le moteur JSP :

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.util.ArrayList;

public final class erreurs_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
  private static java.util.List _jspx_dependants;
  static {
    _jspx_dependants = new java.util.ArrayList(1);
    _jspx_dependants.add("/enteteerreurs.html");
  }
  public Object getDependants() {
    return _jspx_dependants;
  }
  public void _jspService(HttpServletRequest request,
HttpServletResponse response)
        throws java.io.IOException, ServletException {
    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;
    try {
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/html; charset=ISO-8859-1");
      pageContext = _jspxFactory.getPageContext(this, request,
                  response, null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;
      out.write('\r');
      out.write('\n');
      out.write("<h2><font color=\"red\">ERREURS SUR LA PAGE</font></h2>");
      out.write("\r\n");
      out.write("\r\n");
      out.write("\r\n");
      out.write("\r\n");
      out.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\" \"http://www.w3.org/TR/html4/loose.dtd\">\n");
      out.write("<html>\n");
      out.write("<head>\n");
      out.write("<meta http-equiv=\"Content-Type\"
content=\"text/html; charset=ISO-8859-1\">\n");
      out.write("<title>ERREURS</title>\n");
      out.write("</head>\n");
      out.write("<body>\r\n");
      out.write("<!-- un commentaire HTML renvoyé au
client -->\r\n");
      out.write('\n');
ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
      out.write("\r\n");
      out.write("<h2>Les erreurs suivantes se sont
```

```
produites</h2>\r\n");
      out.write("<ul>\r\n");
      out.write("\t");
      //commentaire monoligne
      /* commentaire multilignes */
      for(int i=0;i<erreurs.size();i++)
      {
      out.write("\r\n");
      out.write("\t\t<li>");
      out.print( this.miseEnFormeMessage((String)erreurs.get(i)) );
      out.write("</li>\r\n");
      out.write("\t");

      }
      out.write("\r\n");
      out.write("</ul>\n");
      out.write("</body>\n");
      out.write("</html>\r\n");
    } catch (Throwable t) {
      if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          out.clearBuffer();
        if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
      }
    } finally {
      if (_jspxFactory != null)
_jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

La directive taglib

Le dernier type de directive JSP est l'utilisation de librairies externes personnalisées, par le biais de la directive taglib (<%@ *taglib ...*%>), que nous utiliserons dans le chapitre dédié à cette technologie.

4. Les scripts

Les scripts sont des éléments qui permettent de placer le code Java dans les pages JSP. Il existe trois types de scripts :

- **Les déclarations** : pour déclarer et/ou initialiser une variable.
- **Les scriptlets** : pour contenir les instructions Java.
- **Les expressions** : pour renvoyer des informations aux utilisateurs.

Il peut y avoir une partie de script, du code HTML, puis de nouveau du script dans une même page à condition que chaque morceau de script soit bien entouré par les balises Java.

a. Les déclarations

Une déclaration est employée pour introduire et éventuellement initialiser une variable ou une méthode Java, comme dans un programme. Une déclaration permet de définir un bloc pour les variables globales ainsi que pour des méthodes qui pourront ensuite être utilisées dans le reste du document JSP. Ce principe est très pratique pour l'utilisation de méthodes, car en Java tout est objet, et donc nécessite la déclaration de classes. Cependant, cette technique autorise la simple utilisation de méthodes sans classe et objet.

Pour créer une déclaration, il faut placer le code entre le délimiteur d'ouverture <%*!* et le délimiteur de fermeture %>. Les déclarations peuvent être placées à n'importe quel endroit dans une page JSP, mais il est préférable comme pour les directives, de les placer en début de fichier.

Nous allons modifier notre page d'erreur JSP afin d'ajouter une méthode qui permet d'insérer la balise HTML de couleur rouge pour les messages d'erreurs.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ include file="enteteerreurs.html" %>
<%@ page import="java.util.ArrayList" %>
<%!
//retourner le message en couleur rouge
private String miseEnFormeMessage(String message)
{
    return "<font color=\"red\">"+message+"</font>";
}
%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>ERREURS</title>
</head>
<body>
<%
ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
%>
<h2>Les erreurs suivantes se sont produites</h2>
<ul>
    <%
    for(int i=0;i<erreurs.size();i++)
    {
        out.println("<li>"+this.miseEnFormeMessage((String)
erreurs.get(i))+"</li>");
    }
    %>
</ul>
</body>
</html>
```

b. Les scriptlets

Les scriptlets contiennent des instructions Java. Une scriptlet est un bloc de code incorporé dans une page. Le code écrit dans une scriptlet est en Java. De même, les scriptlets peuvent contenir n'importe quel code Java valide. Une page contenant une scriptlet est souvent appelée un modèle. Pour ajouter une scriptlet à une page, il faut placer le code entre le délimiteur d'ouverture <% et le délimiteur de fermeture %>.

À l'intérieur d'une scriptlet l'objet *out* peut être utilisé avec la méthode *print* (ou *println*) pour générer du contenu texte à destination du client. Les visiteurs qui invoquent la page JSP ne pourront pas voir le code Java contenu dans cette page même s'ils affichent le code source de la page avec leur navigateur Web, ce dernier ne contenant que la sortie générée après traitement.

La partie de code présente dans la page d'erreur précédente correspond à une scriptlet.

```
<%
for(int i=0;i<erreurs.size();i++)
{
    out.println("<li>"+this.miseEnFormeMessage((String)
erreurs.get(i))+"</li>");
}
%>
```

Le code d'une scriptlet ressemble beaucoup à l'élément déclaration des scripts, cependant il existe plusieurs différences entre les deux syntaxes :

- Les scriptlets ne peuvent être employées pour définir des méthodes. Seules les déclarations permettent cela.

- Les variables déclarées dans une déclaration sont des variables d'instance (donc accessibles dans toutes les scriptlets de la page).

- Les variables déclarées dans une scriptlet sont locales (donc visibles uniquement à l'intérieur du bloc dans lequel elles sont définies).

c. Les expressions

Les expressions sont utilisées pour renvoyer au client les valeurs d'expressions Java. Elles permettent en effet de générer une sortie sur une page JSP. Souvent, les expressions sont utilisées comme raccourcis pour simplifier le code. Par exemple, le code *out.println(...)* peut être remplacé par l'expression <%= %>. Le serveur Java EE traite le code contenu dans l'expression et convertit le résultat en une chaîne.

Une expression ne peut pas s'achever par un point virgule. Si un point virgule est utilisé dans une expression, une erreur se produira. En effet, l'expression suivante *<%= exp >* est convertie par le compilateur en la scriptlet suivante *<% out.println(exp); %>*.

Une expression peut contenir un appel de méthode, une instruction ou autre. Nous allons modifier notre page d'erreur du projet *betaboutique* afin d'utiliser une expression à la place de l'objet *out*.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ include file="enteteerreurs.html" %>
<%@ page import="java.util.ArrayList" %>
<%!
//retourner le message en couleur rouge
private String miseEnFormeMessage(String message)
{
    return "<font color=\"red\">"+message+"</font>";
}
%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>ERREURS</title>
</head>
<body>
<%
ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
%>
<h2>Les erreurs suivantes se sont produites</h2>
<ul>
    <%
    for(int i=0;i<erreurs.size();i++)
    {
    %>
        <li><%= this.miseEnFormeMessage((String)erreurs.get(i))%></li>
    }
    %>
</ul>
</body>
</html>
```

d. Les commentaires

L'ajout de commentaires permet de clarifier le code HTML et JSP. Il est possible d'utiliser des commentaires HTML dans les pages JSP. Ces commentaires apparaissent dans la page renvoyée au navigateur client. Ils sont alors de la forme suivante : *<!-- un commentaire HTML renvoyé au client -->*.

Ce commentaire n'apparaît pas dans la page du client mais peut être visualisé dans le source HTML avec l'outil du navigateur. Il existe également la possibilité d'ajouter des commentaires dans le code JSP en utilisant les balises suivantes : *<%-- un commentaire JSP qui ne sera pas renvoyé au client --%>*.

L'ensemble des informations et du code placé entre les balises de commentaire JSP sera supprimé avant le traitement de la page JSP sur le serveur Web et ne sera pas renvoyé au client.

Par contre, un commentaire JSP est de la même forme qu'un commentaire HTML, il ne doit pas se placer dans du code Java (une scriptlet). Les scriptlets sont codées en langage Java, les commentaires Java sont donc appliqués. On retrouve les types monoligne // et multilignes /* */.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>COMMENTAIRES</title>
</head>
<body>
<!-- un commentaire HTML renvoyé au client -->
<%-- un commentaire JSP qui ne sera pas renvoyé au client --%>
<%
//commentaire monoligne
/* commentaire multilignes */
%>
</body>
</html>
```

e. Les actions

Les actions dans une page JSP sont écrites sous la forme d'éléments XML et permettent de condenser en une seule ligne un traitement qui serait plus long à écrire à l'aide d'un script Java. Il existe deux types d'actions :

- **Les actions standards**.
- **Les actions personnalisées**.

Les actions standards sont définies par la spécification JSP (d'où le nom standard) et sont les actions de base, accessibles à toutes les pages JSP, car définies dans l'API Java EE. Les éléments XML correspondant, commencent tous par le préfixe *jsp*. Les actions personnalisées viennent enrichir les actions standards en offrant d'autres possibilités. Ces librairies sont déclarées par la directive *taglib* vue précédemment (*<%@ taglib prefix="..." uri="..." %>*). Pour le moment nous allons étudier les librairies standards avant d'aborder par la suite les librairies personnalisées.

La spécification JSP définit les actions standards suivantes :

- <jsp:useBean>
- <jsp:setProperty>
- <jsp:getProperty>
- <jsp:param>
- <jsp:include>
- <jsp:forward>
- <jsp:plugin>, <jsp:params>, <jsp:fallback>
- <jsp:attribute>
- <jsp:body>
- <jsp:invoke>
- <jsp:doBody>

<jsp:useBean>, <jsp:setProperty>, <jsp:getProperty>

Ces actions standards sont certainement les plus importantes. Elles permettent de créer ou d'importer un JavaBean

dans la page et de le manipuler. L'action *<jsp:useBean../>* possède plusieurs attributs mais trois sont essentiels (id, class et scope).

```
<jsp:useBean id="monObjet" class="maClasse" scope="session" />
```

La signification est la suivante : si l'objet *monObjet* existe dans la portée indiquée par la propriété *scope* alors il est importé dans la page et peut être utilisé, sinon cet objet est créé avec la portée indiquée et peut être utilisé. L'élément *id* est en fait le nom utilisé pour accéder au JavaBean dans le reste de la page. Ce nom doit être unique, il s'agit d'un nom de variable référençant l'instance du JavaBean.

Les valeurs possibles sont :

- *request* : pour les attributs de portée requête.
- *session* : pour les attributs de portée session.
- *application* : pour les attributs de portée application.
- *page* : pour les attributs de portée page.

L'élément *class* est le nom pleinement qualifié de la classe du JavaBean. Il est indispensable de créer une référence à un JavaBean à l'aide de *<jsp:useBean.../>* avant de pouvoir utiliser les actions standards *<jsp:setProperty.../>* et *<jsp:getProperty.../>* qui sont les accesseurs du JavaBean.

Dans notre projet *betaboutique*, nous avons utilisé un objet appelé *client1* de la classe JavaBean *Client* qui permet de stocker les informations du client en cas de succès. Nous allons modifier notre Servlet d'authentification afin de stocker cet objet dans la portée requête et récupérer ses informations pour l'affichage.

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import betaboutique.javabean.Client;

public class ServletAuthentification extends HttpServlet {

...

     //vérifier l'égalité des valeurs
     if( (identifiant!=null && identifiant.equals(ident)) &&
(motdepasse!=null && motdepasse.equals(mdp)) )
     {
          if(urlBienvenue==null)
          {
          throw new ServletException("Le paramètre [urlBienvenue]
n'a pas été initialisé");
          }
          else
          {
                 //créer un JavaBean client
                 Client client=new Client();
                 client.setIdentifiant(identifiant);
                 client.setMotdepasse(motdepasse);
        //on déclenche la Servlet qui permet de gérer la partie
Modèle de l'application
                 request.setAttribute("client",client);

    getServletContext().getRequestDispatcher(urlBienvenue).forward(request,
response);
          }
     }
     //authentification incorrecte
     else
```

```
     {
     erreursParametres.add("Les coordonnées de l'utilisateur sont
incorrectes");
     }
...

}
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="betaboutique.javabean.Client" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<%
Client client=(Client)request.getAttribute("client");
if(client==null)
{
    client.setIdentifiant("");
    client.setMotdepasse("");
}
%>
<h2>Bienvenue client : <%= client.getIdentifiant() %>
<%= client.getMotdepasse() %></h2>
</body>
</html>
```

Nous remarquons que la dernière partie du code, qui correspond à la page *bienvenue.jsp*, permet de récupérer l'objet *client* dans la portée *request*, de vérifier sa présence (*client==null*) et d'afficher ses informations. Ce code est un peu lourd à utiliser et nécessite beaucoup de tests. Comme l'objet *client* est un JavaBean, nous pouvons utiliser les actions standards JSP pour le manipuler. Nous allons modifier notre code afin d'utiliser ce principe.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="betaboutique.javabean.Client" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="request"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>
</body>
</html>
```

Nous remarquons que le code est beaucoup plus simple et ”propre”. Le JavaBean est utilisé dans sa portée et ses accesseurs *<jsp:getProperty.../>* pour accéder aux attributs.

Pour résumer, le code suivant :

```
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="request"/>
```

est équivalent au code suivant :

```
<%
Client client=(Client)request.getAttribute("client");
if(client==null)
{
```

```
    client.setIdentifiant("");
    client.setMotdepasse("");
}
%>
```

Voici d'ailleurs la portion de code de la Servlet générée (*/work/org/apache/jsp/bienvenue_jsp.java*) :

```
...
out.write("<body>\n");
      betaboutique.javabean.Client client = null;
      synchronized (session) {
        client = (betaboutique.javabean.Client) _jspx_page_context.
getAttribute("client", PageContext.SESSION_SCOPE);
        if (client == null){
          client = new betaboutique.javabean.Client();
          _jspx_page_context.setAttribute("client", client,
PageContext.SESSION_SCOPE);
        }
      }
      out.write("\r\n");
      out.write("<h2>Bienvenue client : ");
      out.write(org.apache.jasper.runtime.JspRuntimeLibrary.toString
((((betaboutique.javabean.Client)_jspx_page_context.findAttribute
("client")).getIdentifiant())));
      out.write(' ');
      out.write(org.apache.jasper.runtime.JspRuntimeLibrary.toString
((((betaboutique.javabean.Client)_jspx_page_context.findAttribute
("client")).getMotdepasse())));
      out.write("</h2>\n");
      out.write("</body>\n");
...
```

Comment cela fonctionne-t-il ?

La classe *Client*, qui est située dans le paquetage *betaboutique.javabean*, est instanciée. Nous utilisons un objet JavaBean nommé *client* qui est stocké dans la portée *request*. Le test qui permet de savoir si l'objet n'est pas *null*, n'est plus obligatoire avec l'action *<jsp:useBean...>*. La Servlet réalise elle-même le test de façon transparente. Nous pouvons, dans notre exemple, accéder au JavaBean car dans notre Servlet l'objet a été passé avec une portée *request*. Le code est le suivant :

```
...
//créer un JavaBean client
Client client=new Client();
client.setIdentifiant(identifiant);
client.setMotdepasse(motdepasse);
//déclencher la Servlet qui permet de gérer la partie Modèle de l'application
request.setAttribute("client",client);
getServletContext().getRequestDispatcher(urlBienvenue).forward
(request, response);
...
```

Maintenant modifions la portée de notre JavaBean de cette façon :

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="betaboutique.javabean.Client" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>
</body>
```

```
</html>
```

Notre code JavaBean *client* sera alors accessible dans les autres pages JSP de la session de cet utilisateur. Si la portée utilisée est de type *application*, l'objet sera alors visible par toutes les pages du site quel que soit l'utilisateur connecté. Enfin, si la portée est *page*, l'objet pourra être utilisé uniquement dans la page. Dès que l'utilisateur quitte la page courante, l'objet JavaBean est alors détruit.

Nous pouvons modifier notre Servlet afin de manipuler un objet *client* dans la session de l'utilisateur.

```
...
//créer un JavaBean client
Client client=new Client();
client.setIdentifiant(identifiant);
client.setMotdepasse(motdepasse);
//déclencher la Servlet qui permet de gérer la partie Modèle de l'application
HttpSession session=request.getSession();
session.setAttribute("client",client);
getServletContext().getRequestDispatcher(urlBienvenue).forward
(request, response);
...
```

L'objet *client* est désormais stocké dans une portée *session*. Si nous utilisons le code du dernier exemple exécuté, voici le résultat :

Le JavaBean a une portée *request* et ne peut donc pas lire un JavaBean de portée *session*. Il est possible de modifier la portée de notre JavaBean et d'actualiser notre page.

```
...
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
<h2>Bienvenue client : <jsp:getProperty name="client" property=
"identifiant"/> <jsp:getProperty name="client" property="motdepasse"/></h2>
...
```

Le JavaBean est désormais accessible. Il possède une portée *session* ; il peut donc être accédé dans d'autres pages du site (ex : *sessionjavabean.jsp*).

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="betaboutique.javabean.Client" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
<h2>Bienvenue client avec la session : <jsp:getProperty
name="client" property="identifiant"/> <jsp:getProperty
name="client" property="motdepasse"/></h2>
</body>
</html>
```

Pour résumer, le code suivant :

```
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
```

est équivalent au code suivant :

```
<%
Client client=(Client)session.getAttribute("client");
if(client==null)
{
    client.setIdentifiant("");
    client.setMotdepasse("");
}
%>
```

L'action *<jsp:setProperty.../>* permet de modifier la valeur d'une propriété du JavaBean. L'attribut *name* correspond à l'identifiant de l'objet JavaBean et l'attribut *property* est le nom de la propriété à modifier. La valeur de l'attribut *property* peut être * et permet dans ce cas de lier tous les paramètres de la requête au JavaBean.

Voici un exemple de cette utilisation. Le formulaire HTML de la première page (*authentification.html*) appelle directement une page JSP qui renseigne les attributs correspondants et affiche le résultat à l'écran.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Authentification BetaBoutique</title>
</head>
<body>
<h1>Authentification - Client</h1>
<form action="bienvenueformulaire.jsp" method="POST">
<table border="1" cellspacing="0" cellpadding="5">
<tr>
    <td>Identifiant/Login : </td>
    <td><input type="text" name="identifiant" id="identifiant"
value="" size="20"/></td>
</tr>
<tr>
    <td>Mot de passe : </td>
    <td><input type="text" name="motdepasse" id="motdepasse"
value="" size="20"/></td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="submit"
name="valider" id="valider" value="Valider"/></td>
</tr>
</table>
</form>
</body>
</html>
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="betaboutique.javabean.Client" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="request"/>
<jsp:setProperty name="client" property="*"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>
</body>
</html>
```

Si un des attributs n'est pas renseigné dans la requête, l'action *<jsp:setProperty..../>* tente d'utiliser le paramètre de la requête portant le même nom que la propriété. Si l'attribut n'est pas présent, il prend simplement la valeur null. L'action *<jsp:getProperty.../>* permet de lire la valeur d'une propriété du JavaBean. L'attribut *name* correspond à l'identifiant de l'objet JavaBean et le paramètre *property* correspond à un des attributs de l'objet.

Le code suivant :

```
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/></h2>
```

Est équivalent au code suivant :

```
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="request"/>
<h2>Bienvenue client :<%= client.getIdentifiant() %></h2>
```

Nous remarquons que cette technique très puissante permet à un intégrateur Web de réaliser du code et d'utiliser un langage objet dynamique simplement, sans être un développeur Java confirmé. Cette action *<jsp:useBean.../>* est donc très intéressante à utiliser sauf dans le cas d'une portée *page*. Pour utiliser les actions *<jsp:setProperty.../>* et *<jsp:getProperty.../>*, l'attribut à manipuler doit être un JavaBean ce qui oblige a déclarer chaque accesseur (ce qui par contre, n'est pas obligatoire lorsque l'on utilise *request.getParameter(...)*).

Son utilisation est facilement remplaçable par *request.getParameter(...)*, elle est alors intéressante pour des questions de visibilité et pour renseigner une liste complète de paramètres en une seule étape *<jsp:setProperty name="client" property="*"/>*.

<jsp:forward>

Cette action permet de réaliser une redirection et donc de transférer le contrôle à une autre page.

La syntaxe est la suivante :

```
<jsp:forward page="mapage.jsp"/>
```

Elle est équivalente au code suivant :

```
<%
RequestDispatcher disp=request.getRequestDispatcher("mapage.jsp");
disp.forward(request,response);
%>
```

Des paramètres peuvent être également passés à la page appelée en utilisant la balise *<jsp:param>*.

```
<jsp:forward page="mapage.jsp">
  <jsp:param name="identifiant" value="jerome"/>
  <jsp:param name="motdepasse" value="lafosse"/>
</jsp:forward>
```

<jsp:include>

Cette action permet de donner le contrôle à la page JSP courante, puis lorsque cette page est terminée, à une autre page indiquée par cette balise. Cet élément permet d'inclure le contenu d'une ressource Web statique ou dynamique (dans ce cas après son traitement) dans le contenu de la réponse HTTP de la page JSP retournée.

La syntaxe est la suivante :

```
<jsp:include page="mapage.jsp"/>
```

Elle est équivalente au code suivant :

```
<%
RequestDispatcher disp=request.getRequestDispatcher("mapage.jsp");
disp.include(request,response);
%>
```

Des paramètres peuvent être également passés à la page incluse en utilisant la balise *<jsp:param>*.

```
<jsp:include  page="mapage.jsp">
  <jsp:param name="identifiant" value="jerome"/>
  <jsp:param name="motdepasse" value="lafosse"/>
```

```
</jsp:include>
```

Il ne faut pas confondre la directive include (<%@ *include file="mapage.jsp"* %>) et l'action *<jsp:include/>*. La directive est exécutée au moment de la compilation de la page JSP, tandis que l'action est exécutée autant de fois que la page est appelée et peut donc produire un résultat différent à chaque appel (principe d'un paramètre de requête et d'une inclusion de contenu dynamique).

Les objets implicites

1. Présentation

Les objets implicites sont créés automatiquement lors du traitement de la page JSP par le serveur Java EE. Il existe neuf objets implicites. L'avantage des objets implicites est qu'ils permettent d'invoquer directement leurs méthodes sans avoir à les déclarer et initialiser. Par exemple, une page JSP peut accéder directement à la requête par l'intermédiaire d'un objet implicite nommé *request*. Les objets implicites utilisables en JSP sont les suivants :

- *request* : classe *javax.servlet.ServletRequest*.
- *response* : classe *javax.servlet.ServletResponse*.
- *session* : classe *javax.servlet.http.HttpSession*.
- *pageContext* : classe *javax.servlet.jsp.PageContext*.
- *application* : classe *javax.servlet.ServletContext*.
- *config* : classe *javax.servlet.ServletConfig*.
- *exception* : classe *java.lang.Throwable*.
- *out* : classe *javax.servlet.jsp.JspWriter*.
- *page* : classe *java.lang.Object*.

2. Utilisation

L'objet request

L'objet implicite *request* permet de référencer le contexte de la requête HTTP. Cet objet permet d'utiliser les méthodes relatives à la requête courante. Le plus souvent, cet objet est utilisé pour connaître les paramètres de la requête. Les paramètres envoyés avec les méthodes POST et GET sont donc manipulables.

L'objet response

L'objet implicite *response* permet de gérer la réponse renvoyée au client de l'application. Les informations générées par le serveur Web avant qu'elles ne soient envoyées au client sont stockées dans cet objet.

L'objet session

L'objet implicite *session* permet de référencer le contexte de session HTTP associé au client. Cet objet permet ainsi de récupérer et manipuler les objets des sessions utilisateurs.

L'objet pageContext

L'objet implicite *pageContext* permet de référencer le contexte de la page JSP. Cet objet implicite permet de centraliser les différents attributs de la page JSP dans un seul objet et fournit les fonctionnalités associées à ces attributs.

Il est possible par exemple de récupérer un attribut dans la requête ou la session, modifier la valeur d'un attribut, récupérer un attribut du fichier de configuration de l'application (*web.xml*)...

L'objet application

L'objet implicite *application* permet de référencer le contexte de l'application Web. Le principal intérêt de cet objet implicite est de manipuler le fichier de configuration de l'application (*web.xml*) afin de lire et d'écrire des paramètres.

Voici un exemple, qui permet d'afficher l'adresse email du Webmestre présente dans le fichier de configuration de l'application (*web.xml*).

```
<%
out.println("Contactez le Webmestre :
"+application.getInitParameter("emailAdministrateur"));
%>
```

L'objet config

L'objet implicite *config* permet de référencer le contexte de l'application Web. Cet objet permet de manipuler les informations concernant la configuration de l'environnement dans lequel une page JSP est traitée sur un serveur Web.

Il ne faut pas confondre une application et l'environnement de l'application. L'objet implicite *application* permet de manipuler les objets de l'application, donc déclarés dans le fichier de configuration de cette façon :

```
<context-param>
    <param-name>emailAdministrateur</param-name>
    <param-value>admin@betaboutique.fr</param-value>
</context-param>
```

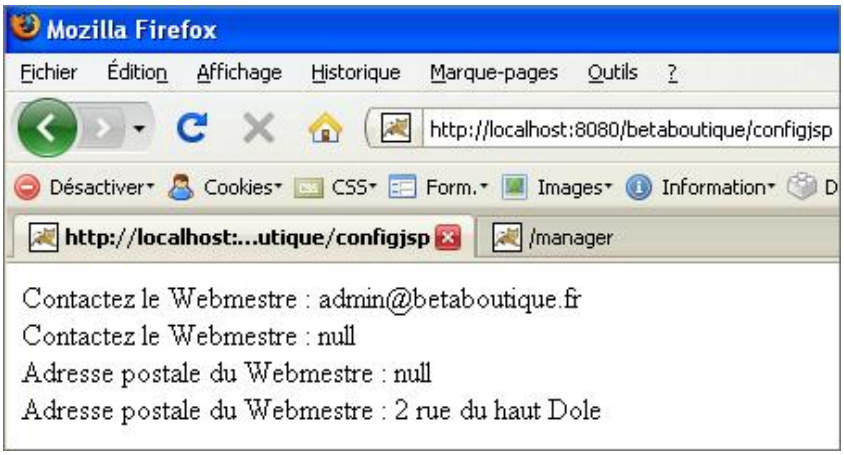
Par contre, si l'objet implicite *config* est utilisé pour accéder à ces attributs cela ne fonctionne pas. L'objet implicite *config* permet de manipuler les informations de l'environnement, soit la déclaration suivante :

```
<servlet-name>servletauthentification</servlet-name>
<servlet-class>betaboutique.servlets.client.Servlet
Authentification</servlet-class>
<init-param>
    <param-name>defautIdentifiant</param-name>
    <param-value>monidentifiant</param-value>
</init-param>
```

Voici un exemple de code de la page *bienvenueformulaire.jsp* du projet *betaboutique* afin de bien comprendre la distinction entre l'objet *application* et l'objet *config*.

```
<%
//afficher les informations avec les objets implicites
out.println("Contactez le Webmestre :
"+application.getInitParameter("emailAdministrateur")+"<br/>");
out.println("Contactez le Webmestre :
"+config.getInitParameter("emailAdministrateur")+"<br/>");
out.println("Adresse postale du Webmestre :
"+application.getInitParameter("adressewebmestre")+"<br/>");
out.println("Adresse postale du Webmestre :
"+config.getInitParameter("adressewebmestre")+"<br/>");
%>
```

```
...
<servlet>
    <servlet-name>configjsp</servlet-name>
    <jsp-file>/bienvenueformulaire.jsp</jsp-file>
    <init-param>
            <param-name>adressewebmestre</param-name>
            <param-value>2 rue du haut Dole</param-value>
    </init-param>
</servlet>
<!-- mapping des servlets -->
<servlet-mapping>
    <servlet-name>configjsp</servlet-name>
    <url-pattern>/configjsp</url-pattern>
</servlet-mapping>
...
```



Cet exemple montre clairement que l'objet implicite *application* correspond à l'application déployée et que l'objet *config* correspond à une partie spécifique de l'environnement de l'application.

L'objet exception

L'objet implicite *exception* est utilisé pour gérer les erreurs qui pourraient se produire lors du traitement de la page JSP. Cet objet est accessible dans une page d'erreur déclarée dans le fichier de configuration par les balises : *<error-page.../>*. Voici un exemple d'appel d'une page JSP et son code.

```
...
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/erreur.jsp</location>
</error-page>
...
```

```
...
<%@page isErrorPage="true" %>
<%= exception.printStackTrace(); %>
...
```

L'objet out

L'objet implicite *out* permet de référencer le flux de sortie des données. Il permet d'envoyer du texte en direction de l'utilisateur Web. Contrairement aux Servlets, il est accessible directement et évite le bloc de code suivant :

```
//flux de sortie
PrintWriter out=response.getWriter();
```

L'objet page

L'objet implicite *page* permet de référencer l'instance courante de la Servlet obtenue après compilation de la JSP. Cet objet est synonyme du mot-clé *this* et n'est pas très utilisé en programmation JSP.

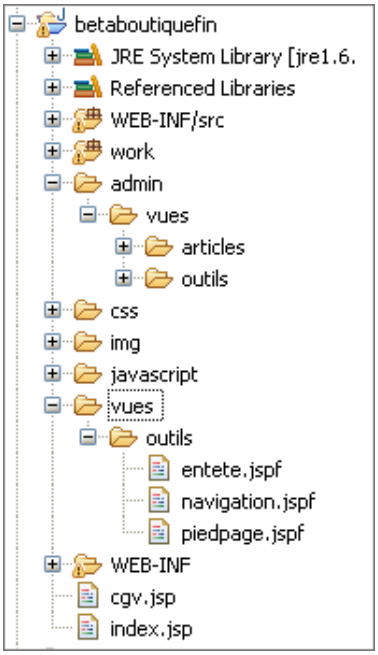
Premières JSP simples

1. Présentation

Nous allons intégrer des premières JSP simples pour la mise en page de la boutique *BetaBoutique*. Les pages JSP utilisées pour de la mise en page, sont appelées des fragments, en référence à des morceaux de carrelage dans le monde du bâtiment.

Les fragments de pages en JSP possèdent l'extension *.jspf*. Pour le développement de ce projet, il est utile de créer une seconde application avec Eclipse (ex : *betaboutiquefin*) afin de pouvoir faire des essais (tester les exemples du guide) avec l'application *betaboutique* et d'utiliser une autre version finale de la boutique.

L'application *betaboutiquefin* possède à cette étape du guide l'arborescence suivante :



Le répertoire */vues* contient les pages *.jsp* et *.jspf* qui seront utilisées pour l'affichage.

Le répertoire */img* contient toutes les images du projet.

Le répertoire */css* contient les feuilles de style du projet.

Le répertoire */javascript* contient les librairies JavaScript utilisées pour les contrôles, effets, auto-complétion...

Le répertoire */admin* contient les pages et répertoires pour la partie administration.

Cette application contient le code définitif mais pas les exemples et tests que nous réalisons.

Le fichier de configuration de l'application est le suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

     <!-- fichier de point de départ de l'application -->
     <!-- il interdit en plus l'exploration de l'arborescence -->
     <!-- attention, chemin toujours relatif -->
     <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
     </welcome-file-list>
</web-app>
```

Pour le moment le fichier de configuration de l'application (*web.xml*) est simple et indique la page d'accueil à utiliser lors de l'appel du site (projet). De cette façon, en utilisant l'URL suivante : http://localhost:8080/betaboutiquefin/, c'est la page *index.jsp* qui sera appelée.

La page *index.jsp* est très simple, elle possède deux directives d'inclusion afin d'utiliser une page fragment pour l'en-tête du site et une autre pour le pied de page.

```
<%@ include file="vues/outils/entete.jspf" %>

<%@ include file="vues/outils/piedpage.jspf" %>
```

Le code utilisé pour la mise en page est donc présent dans les deux pages suivantes */vues/outils/entete.jspf* et */vues/outils/piedpage.jspf*.

Lors de l'utilisation d'inclusions, il faut faire très attention aux chemins des images, librairies, feuilles de style et scripts JavaScript. En effet, dans notre exemple c'est la page *index.jsp* qui est appelée, notre chemin sera donc relatif à cette page. Dans ce cas, au sein de la page */vues/outils/entete.jspf*, il faudra utiliser le répertoire */img* directement sans remonter dans l'arborescence (ex : *../../img*).

Voici un exemple d'utilisation de la feuille de style et du chemin précisé :

```
<!-- feuilles de styles -->
<link rel="stylesheet" type="text/css" href="css/styles.css"
title="defaut" />
```

2. Utilisation

Nous utilisons donc trois pages JSP pour réaliser la mise en page du projet. La page */vues/outils/entete.jspf* utilise elle-même la page */vues/outils/navigation.jspf*. La page */vues/outils/piedpage.jspf* est utilisée pour le bas de page.

Le code de la page */vues/outils/entete.jspf* est présenté ci-dessous. Ce code simple, est essentiellement composé de balises HTML. La seule partie utilisée (et qui nécessite l'utilisation d'une page JSP) en Java est la directive d'inclusion pour le menu de navigation :

```
<html>
<head>
<title>BetaBoutique</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta name="description" content="BetaBoutique">
<!-- feuilles de styles -->
<link rel="stylesheet" type="text/css" href="css/styles.css"
title="defaut" />
<link rel="stylesheet" type="text/css" href="css/styles_panier.css"
title="defaut" />
<!--  javascript -->
<link type="text/css" rel="stylesheet" media="all"
href="javascript/jtip/jtip.css"/>
<script type="text/javascript" src="javascript/jquery/jquery.js"></script>
```

```
<script type="text/javascript" src="javascript/jtip/jtip.js"></script>
</head>
<body>

<div id="global">

     <div id="entete">
         <table border="0" cellspacing="0" cellpadding="0"
width="100%" height="100%">
         <tr>
             <td align="left" valign="top"><a href="/"><img
src="img/bandeau_haut_gauche.jpg" border="0"/></a></td>
             <td align="left" valign="top"><img src=
"img/bandeau_haut_centre.jpg"/></td>
             <td align="left" valign="top" background=
"img/bandeau_haut_droite.jpg" width="222">
                 <table border="0" cellspacing="0"
cellpadding="0" width="100%" height="100%">
                 <tr><td height="50">
                 <form id="formulairerecherche"
name="formulairerecherche" action="" method="post">
                 <div class="titregris">RECHERCHE</div>
                 <input type="text" id="recherche"
name="recherche" value=""/>
                 <input type="image" src="img/ok.gif"
align="absmiddle"/>
                 </form>
                 </td></tr>
                 <tr><td align="center"><span class="liengris">
<a href="">MON COMPTE</a></span>      
<span class="liengris"><a href="">MON PANIER</a></span></td></tr>
                 <tr><td align="right" height="20"><span
class="liengris"><a href="">DECONNEXION</a> </span><br/><br/>
</td></tr>
                 </table>
             </td>
         </tr>
         </table>
     </div>


     <div id="contenu">


     <table border="0" cellspacing="0" cellpadding="0" width="100%">
     <tr>
         <td valign="top" width="200">
          <!-- COLONNE GAUCHE -->
          <%@ include file="navigation.jspf" %>
         </td>


         <!-- COLONNE CENTRE -->
         <td valign="top" width="100%" style="padding-left:5px">
         <ul id="menuhaut">
          <li><a href="/" title="Accueil">Accueil</a></li>
          <li><a href="" title="Promotions">Promotions</a></li>
          <li><a href="" title="Nouveaut&eacute;s">Nouveaut&eacute;s
</a></li>
          <li><a href="" title="CGV">Conditions de vente</a></li>
          <li><a href="" title="Panier">Commander</a></li>
         </ul><br/>
```

La page de navigation */vues/outils/navigation.jspf* est également très simple. Elle permet d'afficher la liste des catégories ainsi que le résumé du panier réalisé en JavaScript. Pour le moment, les catégories sont affichées en dur, mais il est évident que par la suite, ce menu sera généré de manière dynamique (lecture des catégories dans une source de données).

```
<table border="0" cellspacing="0" cellpadding="0" name="menugauche"
```

```
id="menugauche" style="background: url('img/fondmenu.gif')
repeat-x;"  height="500">
          <tr>
            <td width="100%" valign="top">
             <ul class="menucategorie">
                 <li><a href=""><img src="img/fleche.gif"
border="0" align="absmiddle"/> Policier et Thriller</a></li>
                 <li><a href=""><img src="img/fleche.gif"
border="0" align="absmiddle"/> Com&eacute;die</a></li>
                 <li><a href=""><img src="img/fleche.gif"
border="0" align="absmiddle"/> Documentaires</a></li>
                 <li><a href=""><img src="img/fleche.gif"
border="0" align="absmiddle"/> Jeunesse et Famille</a></li>
                 <li><a href=""><img src="img/fleche.gif"
border="0" align="absmiddle"/> S&eacute;ries TV</a></li>
                 <li><a href=""><img src="img/fleche.gif"
border="0" align="absmiddle"/> Spectacles et Humour</a></li>
                 <li><a href=""><img src="img/fleche.gif"
border="0" align="absmiddle"/> Fantastique</a></li>
             </ul>
            </td>
          </tr>
          <tr>
            <td valign="top" align="center" bgcolor="#ffce00"
height="50" class="textepanier">
             <a href="panierresumer.php?width=400"
class="jTip" id="aidedynamique" name="Panier Dynamique"
onclick="javascript:window.location='';">
               <img src="img/panier.gif" border="0"
style="padding-top:10px;"/> <span>Panier</span>
             </a>
             <br/>
</td>
          </tr>
          <tr>
            <td align="center"><img src="img/cartes.gif"
border="0" style="margin-bottom:5px;"/><br/></td>
          </tr>
</table>
```

Le projet *BetaBoutique* est codé en XHTML avec l'utilisation de tableaux, balises * *, **... Il est évident que pour un développement professionnel, les techniques conformes au W3C seraient utilisées de préférence (pas de tableau pour la mise en page, utilisation massive de feuilles de style).

Enfin, le code de la page */vues/outils/piedpage.jspf* est présenté ci-dessous. Il permet de fermer la mise en page et d'afficher des informations sur la boutique.

```
</td>
</tr>
</table>
      <div id="betaboutique">
      BetaBoutique est une boutique de et par la
soci&eacute;t&eacute; BetaBoutique SARL au Capital
10 000 Euros n° siret 111 222 333 444 555
      </div>
</div>
</body>
</html>
```

Nous avons vu comment utiliser des pages JSP simples ainsi que les fragments *.jspf*. La page obtenue à cette étape du projet est présentée ci-dessous. Le code de la page *index.jsp* est le suivant :

```
<%@ include file="vues/outils/entete.jspf" %>
<%@ include file="vues/outils/piedpage.jspf" %>
```

Nous remarquons que chaque page du site devra utiliser l'inclusion de la page d'en-tête *entete.jspf* et de la page de pied de page *piedpage.jspf*.

Cette technique est utilisée par tous les développeurs Web avec tous les langages de programmation. Par contre, Java EE offre une possibilité très souple qui permet de configurer un en-tête et un pied de page directement applicables à toutes les pages du site par l'intermédiaire du fichier de configuration de l'application (*web.xml*).

Pour appliquer notre en-tête et notre pied de page à toutes les pages du site, nous allons ajouter les directives *<include-prelude/>* et *<include-coda/>*. Cette technique nécessite le remplacement de la grammaire XML par un schéma pour le fichier *web.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
         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">
     <!-- fichier de point de départ de l'application -->
     <!-- il interdit en plus l'exploration de l'arborescence -->
     <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
     </welcome-file-list>
         <jsp-config>
            <jsp-property-group>
                <url-pattern>*.jsp</url-pattern>
                include-prelude>/vues/outils/entete.jspf</include-prelude>
                <include-coda>/vues/outils/piedpage.jspf</include-coda>
                </jsp-property-group>
        </jsp-config>
</web-app>
```

Nous allons désormais traiter lors de la prochaine partie les exceptions et erreurs en JSP.

Gérer les exceptions et erreurs en JSP

1. Présentation

L'écriture du code dans un langage informatique doit également prévoir la gestion de diverses erreurs qui ne manqueront pas de survenir. Nous avons déjà abordé la gestion des erreurs et des exceptions lors des premiers exemples de ce chapitre. Nous avons déjà rencontré des bogues lors de nos développements. Parfois, les réponses à ces bogues sont affichées de manière peu sympathique sous la forme de trace ou stack trace.

Les applications Web Java peuvent traiter les exceptions de différentes façons. La manière la plus courante de traiter les exceptions en Java est d'utiliser les blocs *try...catch* ou la directive *throws*.

En Java, un programme déclenche (*throws*) une exception ou une erreur lorsqu'un dysfonctionnement détourne le programme de son exécution normale. La liste des exceptions est recensée dans la documentation javadoc de la classe *Exception*. http://java.sun.com/j2se/1.3/docs/api/java/lang/Exception.html

Cependant, nous avons besoin d'un moyen permettant de traiter des exceptions imprévues. Nous disposons pour cela en JSP de deux solutions :

- La directive *page*.
- Le descripteur de déploiement de l'application (le fichier *web.xml*).

2. La directive page

La directive *page* possède un attribut nommé *errorPage*. Si une exception se produit et qu'elle n'est pas interceptée par notre programme, c'est-à-dire non traitée dans notre code, la page indiquée est alors retournée par le serveur. Si l'attribut *errorPage* commence par la barre oblique (slash), c'est que l'adresse de la page cible est exprimée relativement à la racine du répertoire de l'application Web. Lors de la définition de la page d'erreur, l'attribut *isErrorPage* doit avoir la valeur *true*.

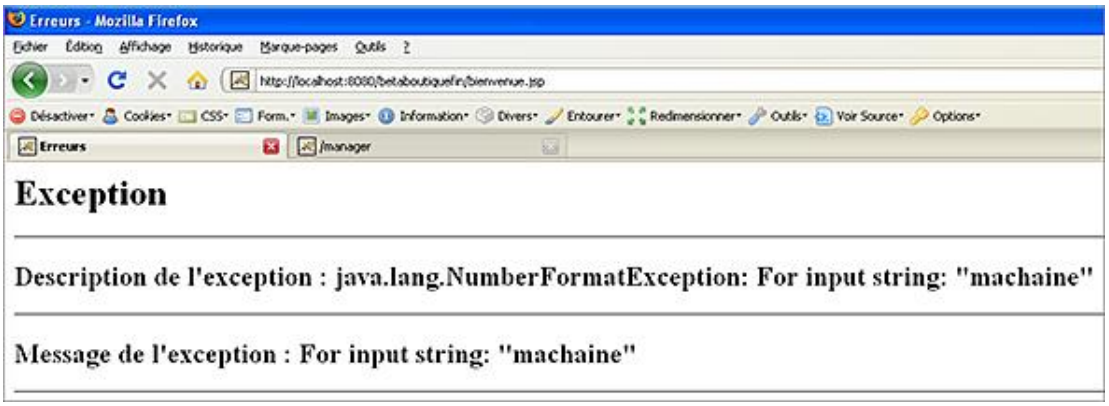
Nous allons utiliser notre application *betaboutique* et modifier la page *bienvenue.jsp* afin de déclencher volontairement des exceptions. Nous allons appeler la page *erreursexception.jsp* qui permet d'afficher des informations sur l'exception qui vient de se déclencher.

```
<%@ page isErrorPage="true" %>
<html>
<head><title>Erreurs</title></head>
<body>
<h1>Exception</h1>
<hr/>
<h2>Description de l'exception : <%= exception.toString() %></h2>
<hr/>
<h2>Message de l'exception : <%= exception.getMessage() %></h2>
<hr/>
</body>
</html>
```

Pour déclencher une première exception nous allons tenter de convertir une chaîne de caractères en entier. Le code JSP est correct mais son exécution aura pour conséquence de déclencher une exception de type *java.lang.NumberFormatException*.

```
<%@ page errorPage="erreursexception.jsp" %>
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="betaboutique.javabean.Client" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>BIENVENUE</title>
</head>
<body>
<%
//declaration d'une chaine de caracteres
```

```
String chaine="machaine";
Integer chaineentier=new Integer(chaine);
%>
</body>
</html>
```



Le message nous indique qu'il s'agit d'une erreur interne à la Servlet. La cause est affichée avec la trace de pile (*stack trace*). Cet affichage de trace de pile est déclenché en réponse à la première exception trouvée et renvoyée par la méthode *exception.printStackTrace()*.

Il faut bien faire la différence entre une erreur de compilation qui empêche la transformation par le moteur de JSP de la page en Servlet et une erreur d'exécution. Les erreurs d'exécution surviennent pendant l'accès à la page par le visiteur. Ces erreurs sont causées par un problème dans la page JSP (bien compilée) ou bien dans le code qui est appelé par la page, comme un JavaBean par exemple ou une inclusion.

Lors du développement d'applications, il est important de suivre les conseils suivants pour corriger les problèmes :

- Il faut commencer par lire le message affiché sur la page d'erreur.
- Si le message affiché ne nous permet pas de corriger le problème, il faut analyser le fichier *.java*.
- Si vraiment dans les cas extrêmes, l'erreur n'est pas trouvée, il faut compiler la Servlet pour voir la trace affichée.

La directive *page* est donc très utile pour désigner une page d'erreur à afficher en cas de problème. Cependant, l'inconvénient de cette méthode est que la même page d'erreur est renvoyée quelle que soit l'exception rencontrée.

3. Le descripteur de déploiement (web.xml)

Le descripteur de déploiement permet de désigner des gestionnaires d'erreurs pour toute l'application. Il est ainsi possible d'avoir des pages d'erreur différentes, en fonction du type d'exception, mais aussi des erreurs relatives au serveur (ex : page non trouvée, problème du serveur...). Il est possible de définir des pages d'erreur pour les exceptions Java et pour les erreurs HTTP.

La définition des pages d'erreur vient immédiatement après l'élément *<welcome-file-list/>* du fichier *web.xml* conformément à la DTD.

Nous allons définir une page en cas d'erreur 404 (page non trouvée sur le serveur) dans le fichier *web.xml*.

```
<error-page>
    <error-code>404</error-code>
    <location>/404.html</location>
</error-page>
```

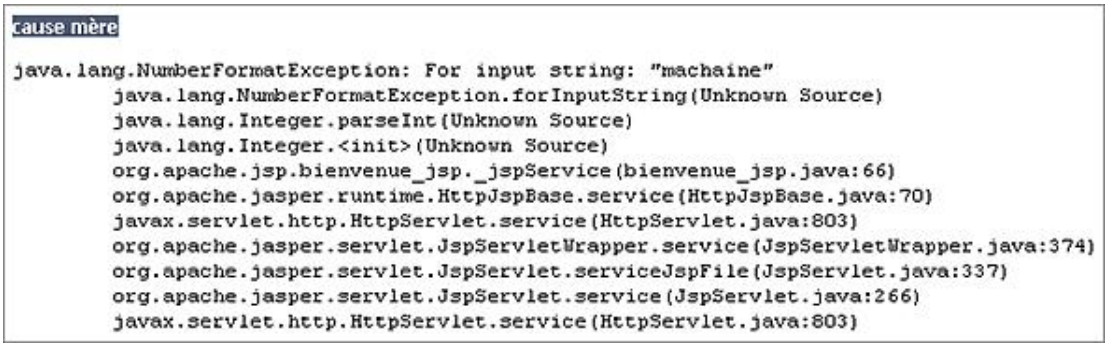
La définition d'une page d'erreur pour une exception Java est identique à celle des erreurs HTTP. Le nom de la classe de l'exception Java et la page d'erreur associée sont indiquées. Dans notre cas, la page affichée en cas de problème de conversion sera */erreurconversion.html*.

Pour tester cette technique, nous allons modifier notre page *bienvenue.jsp* et enlever la directive de la page d'erreur.

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

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<%
//declaration d'une chaine de caracteres
String chaine="machaine";
Integer chaineentier=new Integer(chaine);
%>
</body>
</html>
```

Nous allons déclencher cette page avant de paramétrer le fichier *web.xml* et donc de générer une erreur d'exécution. La trace affichée sans traitement est présentée ci-dessous :



Une astuce de programmation consiste à déclencher volontairement l'erreur et à observer la trace. Dans notre exemple, il est visible que l'exception est de type : *java.lang.NumberFormatException*.

Nous allons donc définir notre exception pour l'erreur de conversion en indiquant la classe précisée.

```
<error-page>
    <exception-type>java.lang.NumberFormatException</exception-type>
    <location>/erreurconversion.html</location>
</error-page>
```

Nous avons modifié le fichier de configuration de l'application, il faut donc recharger l'application pour que nos modifications soient prises en compte. Nous déclenchons à nouveau la page *bienvenue.jsp* et nous observons que la page d'erreur adaptée est alors affichée. Cette technique est donc beaucoup plus puissante que l'utilisation de la directive *page* propre aux JSP.

Une page d'erreur indiquée dans une page JSP a la priorité sur celles indiquées par le descripteur de déploiement.

Nous pouvons donc, avec cette technique gérer plusieurs exceptions Java et erreurs HTTP. La priorité correspond à l'importance de la portée de l'exception, dans le descripteur de déploiement. Nous allons modifier le descripteur de déploiement de la façon suivante :

```
<error-page>
      <exception-type>java.lang.Throwable</exception-type>
      <location>/erreurglobale.html</location>
</error-page>
<error-page>
      <exception-type>java.lang.NumberFormatException</exception-type>
```

```
      <location>/erreurconversion.html</location>
</error-page>
<error-page>
      <error-code>404</error-code>
      <location>/404.html</location>
</error-page>
```

Nous avons ajouté le traitement de l'exception *java.lang.Throwable* qui correspond à toutes les exceptions générées en Java. Autrement dit, dès qu'une exception sera déclenchée (erreur de conversion, calcul, problème de fermeture d'une source de données ou autre) la page *erreurglobale.html* sera appelée. Nous pouvons recharger l'application et déclencher la page *bienvenue.jsp* pour voir le résultat.

Nous remarquons que la page possède une erreur qui déclenche une exception de conversion mais que c'est la page d'erreur globale qui est affichée. En effet, les erreurs sont attrapées par ordre de priorité dans le fichier de définition de l'application (*web.xml*). Comme l'exception *java.lang.Throwable* est plus générale que l'exception *java.lang.NumberFormatException* c'est bien la page *erreurglobale.html* qui est affichée.

Pour un serveur en production, il faudra juste gérer l'exception générale (*java.lang.Throwable*) avec une page d'erreur. L'idéal serait bien entendu une page d'erreur adaptée à la plupart des erreurs rencontrées.

Par contre, lors de l'étape de développement il est important de ne pas mettre une page statique HTML associée à l'exception générale *java.lang.Throwable* car nous n'aurons aucune trace à l'écran en cas d'erreur ce qui est très contraignant pour les débogages.

Bibliothèque de tags JSTL

1. Présentation

Il existe de nombreuses librairies de tags (utilisables à partir de balises XML) proposées pour la technologie JSP Java. Une bibliothèque de balises est composée de grammaires XML (fichier *.tld*) et de classes d'implémentation des fonctionnalités.

Les responsables Java EE se sont aperçus que de nombreux développeurs dépensaient beaucoup d'énergie pour créer de nouvelles balises répondant souvent aux mêmes besoins. Ces actions avaient des syntaxes et des noms différents mais accomplissaient pratiquement la même chose. Le but de JSTL (*Java server page Standard Tag Library*) est de standardiser un certain nombre d'actions. JSTL est donc un ensemble de tags personnalisés développés sous la JSR 052 permettant de réaliser des opérations de structure (conditions, itérations...), gérer les langues, exécuter des requêtes SQL et utiliser le langage XML.

JSTL est actuellement le standard pour l'utilisation de tags, mais il existe de nombreuses autres librairies :

- La librairie de tags Struts : manipulation de JavaBean, HTML, conditions...

  (http://struts.apache.org/1.x/struts-taglib/index.html)

- La librairie Displaytag : gestion de l'affichage de tableaux HTML, XML, Excel...

  (http://displaytag.sourceforge.net/11/)

- La librairie Image taglib : gestion des opérations sur des images.

  (http://jakarta.apache.org/taglibs/sandbox/doc/image-doc/index.html)

- La librairie Upload taglib : gestion du chargement ascendant de fichiers.

  (http://www.servletsuite.com/servlets/uptag.htm)

- La librairie Ajax Upload taglib : gestion du chargement ascendant de fichiers avec la technologie Ajax.

  (http://www.servletsuite.com/servlets/ajaxuploadtag.htm)

2. Utilisation

La mise en place d'une librairie de tags JSP nécessite le chargement de l'archive d'implémentation au format *.jar* dans le projet et l'association *URI/TLD* dans le descripteur de déploiement *web.xml*.

Pour utiliser une librairie de tag, il est nécessaire de procéder de la façon suivante :

- La première étape consiste à télécharger l'archive au format *.jar* qui contient l'implémentation des balises. Puis il faut copier cette archive dans le répertoire des librairies, à savoir */WEB-INF/lib*. Il faut aussi copier tous les fichiers *.tld* (qui sont les grammaires XML des balises) dans le répertoire */WEB-INF/tld* ou */WEB-INF/tlds* (à créer).

- La deuxième étape consiste à créer l'association entre la librairie de tags et notre projet dans le descripteur de déploiement (*web.xml*), par le biais d'une déclaration et d'une URI.

- La dernière étape consiste à ajouter la directive <%@ *taglib.../*> dans chaque page JSP devant utiliser la librairie.

Les fichiers *.jar* contiennent le code Java pour l'utilisation des balises. Par contre, les fichiers *.tld* contiennent la grammaire des balises utilisables avec les archives *.jar*. Ainsi, la déclaration de ces grammaires nous permet de bien utiliser les balises et de bénéficier de messages d'erreurs efficaces lors des utilisations (nombre de paramètres pour la balise, syntaxe des paramètres...).

Dans un premier temps nous allons utiliser la bibliothèque de tags standard Java EE JSTL.

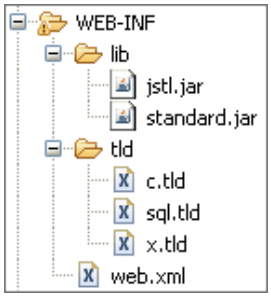
JSTL possède quatre bibliothèques de tags :

- Fonctions de base avec *c.tld* et l'URI http://java.sun.com/jstl/core
- Fonctions de traitement XML avec *x.tld* et l'URI http://java.sun/com/jstl/xml
- Fonctions d'internationalisation (langues) avec *fmt.tld* et l'URI http://java.sun.com/jstl/fmt
- Fonctions de traitement SQL avec *sql.tld* et l'URI http://java.sun.com/jstl/sql

3. Implémentation

Pour utiliser JSTL, il faut copier les librairies *jstl.jar* et *standard.jar* dans notre répertoire de librairies */WEB-INF/lib*. Ces archives sont disponibles sur Internet sur le site Java. Dans un second temps, nous allons copier les fichiers *.tld* (grammaires des balises) dans un répertoire nommé */WEB-INF/tld* que nous allons créer.

L'arborescence de notre projet doit avoir la structure suivante :



Il reste alors une dernière étape après le chargement des librairies *.jar* et la mise en place des grammaires *.tld*, c'est la déclaration des bibliothèques à utiliser dans le descripteur de déploiement du projet (*web.xml*).

Nous plaçons le code suivant en fin de fichier *web.xml* après les balises *<error-page/>*.

```
...
<taglib>
    <taglib-uri>/WEB-INF/tld/c.tld</taglib-uri>
    <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>/WEB-INF/tld/x.tld</taglib-uri>
    <taglib-location>/WEB-INF/tld/x.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>/WEB-INF/tld/sql.tld</taglib-uri>
    <taglib-location>/WEB-INF/tld/sql.tld</taglib-location>
</taglib>
...
```

La balise *<taglib-uri/>* nous indique une URI pleinement qualifiée qui sera par la suite utilisée dans notre directive *<% @taglib/>* des pages JSP. Dans notre cas, la valeur utilisée est le chemin vers la librairie *c.tld*, mais nous pourrions utiliser n'importe quelle URI pleinement qualifiée. Voici un autre exemple d'URI :

```
...
<taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>http://java.sun.com/jstl/xml</taglib-uri>
    <taglib-location>/WEB-INF/tld/x.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>http://java.sun.com/jstl/sql</taglib-uri>
```

```
    <taglib-location>/WEB-INF/tld/sql.tld</taglib-location>
</taglib>
...
```

La seconde balise *<taglib-location/>* permet de préciser le chemin vers la grammaire des librairies que nous venons d'installer. Dans la configuration précédente, nous avons donc paramétré les librairies *c.tld*, *x.tld* et *sql.tld*.

Désormais il ne reste plus qu'à déclarer nos balises de tags dans chaque page JSP qui souhaite les utiliser. Cette opération simple est réalisée par l'intermédiaire de la directive *<%@ taglib.../>*. Il faut préciser le préfixe des balises et le chemin pleinement qualifié identique à celui renseigné dans le descripteur de déploiement : *<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>*.

4. Utilisation de bibliothèques

a. La bibliothèque core

Nous allons utiliser la bibliothèque core : *c.tld*. Cette bibliothèque regroupe les actions fondamentales. Pour cela, nous utilisons notre page *bienvenue.jsp* et nous allons définir les directives pour l'utilisation de nos bibliothèques.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/x.tld" prefix="x" %>
<%@ taglib uri="/WEB-INF/tld/sql.tld" prefix="sql" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>BIENVENUE</title>
</head>
<body>
</body>
</html>
```

Nos bibliothèques de balises sont désormais définies, nous pouvons utiliser de nouvelles balises XML très puissantes pour nos développements.

- Le tag *set* permet de stocker une variable dans une portée particulière (*page*, *request*, *session* ou *application*).
- Le tag *out* permet d'afficher la valeur d'une variable, ce tag est équivalent à *<%= ...%>*.
- Le tag *remove* permet de supprimer une variable.
- Le tag *catch* permet de gérer les exceptions.
- Le tag *if* est utilisé pour réaliser une condition.
- Le tag *choose* est utilisé pour des cas mutuellement exclusifs (équivalent du *switch*).
- Le tag *foreach* est utilisé pour réaliser des itérations.
- Le tag *forTokens* est utilisé pour découper une chaîne selon un ou plusieurs séparateurs.
- Le tag *import* permet d'accéder à une ressource via son URL pour l'inclure ou l'utiliser dans la page JSP.
- Le tag *redirect* permet de réaliser une redirection vers une nouvelle URL.

Grâce à l'utilisation de grammaires XML (fichiers *.tld* pour les taglibs), Eclipse sait gérer l'auto-complétion des balises et les erreurs de déclaration.

Voici un exemple d'utilisation de la bibliothèque *core*.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/x.tld" prefix="x" %>
<%@ taglib uri="/WEB-INF/tld/sql.tld" prefix="sql" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>BIENVENUE</title>
</head>
<body>
<c:set var="maVariable" value="ma valeur avec JSTL" scope="page" />
<c:out value="${maVariable}" />
<br/>
<c:if test="${1==1}">Une conditionnelle opérationnelle</c:if>
<c:if test="${1==2}">Une conditionnelle non affichée</c:if>
<br/>
<c:forEach begin="1" end="4" var="i">
    <c:out value="${i}"/><br/>
</c:forEach>
<br/>
<c:import url="/message.txt" var="message"/>
<c:out value="${message}"/>
</body>
</html>
```

b. La bibliothèque XML

Dans un second temps, nous allons utiliser la bibliothèque xml : *x.tld*.

Pour cela, nous allons créer une nouvelle page *xmltaglib.jsp* et définir la directive pour utiliser nos nouvelles balises. Cette bibliothèque très puissante permet de manipuler des données en provenance d'un contenu XML (document ou contenu généré à la volée par programmation). Nous allons créer une structure de fichier XML dans notre page *xmltaglib.jsp*.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/x.tld" prefix="x" %>
<%@ taglib uri="/WEB-INF/tld/sql.tld" prefix="sql" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>XML</title>
</head>
<body>

<c:set var="xml">

    <?xml version="1.0" encoding="ISO-8859-1"?>
    <dvds>
      <dvd id="1">
        <titre>Taxi 4</titre>
        <image>taxi4.png</image>
      </dvd>
      <dvd id="2">
        <titre>Le choc</titre>
        <image>choc.png</image>
      </dvd>
      <dvd id="3">
        <titre>Mort un dimanche de pluie</titre>
        <image>muDd32D3.png</image>
      </dvd>
    </dvds>
```

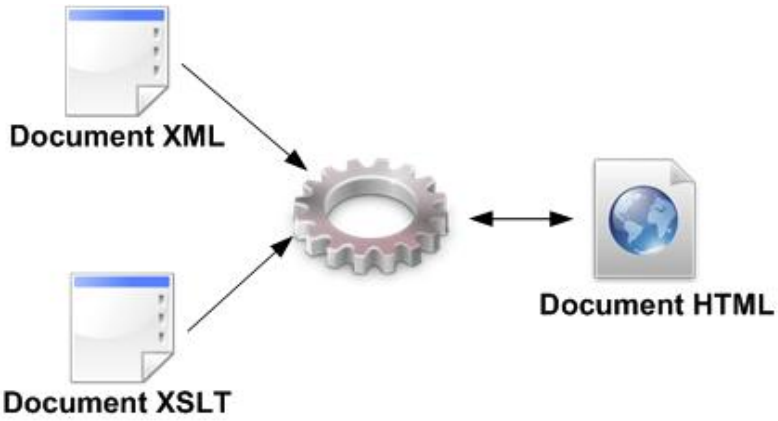
```
</c:set>
<c:out value="${xml}" escapeXml="false"/>
</body>
</html>
```

Cette page affiche le contenu complet XML sous la forme d'un flux textuel. En effet, la variable nommée *xml* stocke le contenu du code XML et la balise *<c:out.../>* permet de l'afficher. Le tag *parse* permet d'analyser le document et de stocker le résultat dans une variable qui pourra être exploitée par la JSP. Le tag *set* est équivalent au tag *set* de la bibliothèque *core*. Il permet d'évaluer l'expression fournie dans l'attribut *select* et de stocker le résultat dans une variable. Le tag *out* est équivalent au tag *out* de la bibliothèque *core*. Il permet d'envoyer le résultat dans le flux de sortie. L'attribut *select* permet de préciser le chemin XPath de l'arbre XML. Le tag *if* est équivalent au tag *if* de la bibliothèque *core* à la différence qu'il évalue une expression XPath. Le tag *choose* est équivalent au tag *choose* de la bibliothèque *core* à la différence qu'il utilise une expression XPath. Le tag *forEach* est équivalent au tag *forEach* de la bibliothèque *core*. Il permet de réaliser des boucles sur des nœuds.

Le tag *transform* permet d'appliquer une transformation XSLT (*eXtensible Stylesheet Language Transformations*) à un document XML. L'attribut *xsl* permet de spécifier la feuille de styles XSL à utiliser.

Le tag *transform* est introduit dans le cadre du projet *betaboutique* afin de réaliser une transformation XSLT sur le document XML qui contient la liste des DVD. Le code précédent permet d'afficher sous forme textuelle le contenu du document XML.

Nous allons donc séparer les données de la mise en page en utilisant la technique suivante :



Un premier document au format XML (ou contenu XML généré à la volée) contient uniquement des métadonnées et ne s'occupe en aucun cas de la présentation. Son contenu peut avoir la structure suivante :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dvds>
 <dvd id="1">
    <titre>Taxi 4</titre>
    <image>taxi4.png</image>
 </dvd>
 <dvd id="2">
    <titre>Le choc</titre>
    <image>choc.png</image>
 </dvd>
 <dvd id="3">
    <titre>Mort un dimanche de pluie</titre>
    <image>muDd32D3.png</image>
 </dvd>
</dvds>
```

Un second document (présent en dur sur le serveur) est composé de balises XML/XPath et HTML, il permet de lire les données du fichier XML et de gérer la mise en forme de ces données. Il s'occupe uniquement d'opérations simples (conditions, boucles) et de la mise en page. Ce document est une feuille de styles XSLT au format *.xsl*.

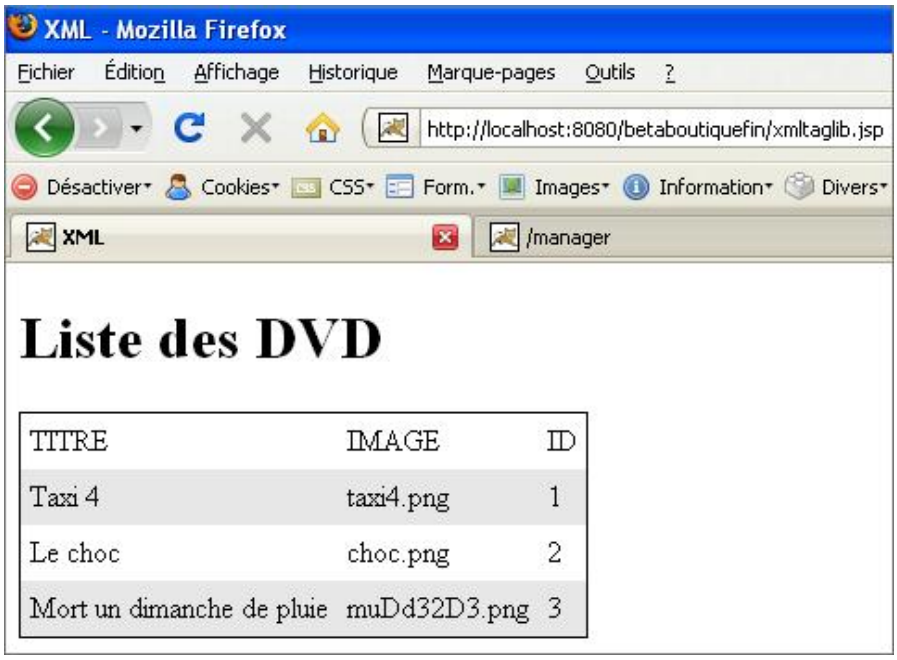
Voici un exemple de son contenu pour l'affichage des DVD :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/
1999/XSL/Transform">
<xsl:output method="html" indent="yes" encoding="ISO-8859-1"/>
<xsl:template match="/">
    <html>
```

```
    <head><title>BETABOUTIQUE</title></head>
    <body>
    <h1>Liste des DVD</h1>
    <table border="0" cellspacing="0" cellpadding="5"
style="border-style:solid;border-width:1px">
    <tr><td>TITRE</td><td>IMAGE</td><td>ID</td></tr>
        <xsl:for-each select="/dvds/dvd">
            <tr>
            <xsl:if test="position() mod 2=1">
                <xsl:attribute name="bgcolor">#eaeaea
</xsl:attribute>
            </xsl:if>
                <td><xsl:value-of select="titre"/></td>
                <td><xsl:value-of select="image"/></td>
                <td><xsl:value-of select="@id"/></td>
            </tr>
        </xsl:for-each>
    </table>
    </body>
    </html>
</xsl:template>
</xsl:stylesheet>
```

Nous avons donc désormais nos données XML, notre fichier de mise en forme *.xsl*, il ne nous reste plus qu'à utiliser notre page JSP qui permet par l'intermédiaire de la balise *<x:transform.../>* de transformer le tout en document HTML.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/x.tld" prefix="x" %>
<%@ taglib uri="/WEB-INF/tld/sql.tld" prefix="sql" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>XML</title>
</head>
<body>
<c:set var="xml">
       <?xml version="1.0" encoding="ISO-8859-1"?>
       <dvds>
         <dvd id="1">
           <titre>Taxi 4</titre>
           <image>taxi4.png</image>
         </dvd>
         <dvd id="2">
           <titre>Le choc</titre>
           <image>choc.png</image>
         </dvd>
         <dvd id="3">
           <titre>Mort un dimanche de pluie</titre>
           <image>muDd32D3.png</image>
         </dvd>
       </dvds>
</c:set>
<!-- utilisation du fichier de mise en forme -->
<c:import url="/feuillexsl.xsl" var="xsl"/>
<x:transform xml="${xml}" xslt="${xsl}"/>
</body>
</html>
```



La composition des deux fichiers une fois la transformation réalisée donne un document au format HTML. Cette technique très puissante permet de complètement séparer la partie *données* de la partie *présentation*. Le document résultant dans notre cas est un fichier au format HTML, mais nous aurions pu générer des documents PDF, des images SVG, des pages 3D VRML... Bien entendu, par la suite, le contenu XML des données sera généré de façon dynamique avec une page JSP ou une Servlet. Les données seront lues dans une source de données de type fichier ou base de données, la feuille de style sera adaptée en conséquence pour l'affichage et la balise *<x:tranform.../>* permettra de générer à la volée la page HTML (ou n'importe quel autre format compatible).

c. La bibliothèque I18n

Nous allons mettre en place la bibliothèque I18n : *fmt.tld*. Cette bibliothèque regroupe les actions pour la gestion de l'internationalisation. L'internationalisation en programmation et développement consiste à gérer les langues et donc à proposer des pages multilingues. Le terme I18n est standardisé pour l'internationalisation et correspond aux 18 caractères qui composent le mot *internationalisation* entre le i et le n.

En programmation la langue est présentée avec une locale. Une locale est composée de deux paramètres : la première partie *lang* et la seconde partie *country* séparées par le caractère underscore. Par exemple :

- le français France est représenté par : *fr_FR*.
- le français canadien est représenté par : *fr_CA*
- l'anglais américain est représenté par : *en_US*

Pour manipuler la bibliothèque de langue, nous ajoutons sa définition dans le fichier de configuration de l'application (*web.xml*).

```
...
<taglib>
    <taglib-uri>/WEB-INF/tld/fmt.tld</taglib-uri>
    <taglib-location>/WEB-INF/tld/fmt.tld</taglib-location>
</taglib>
...
```

Nous ajoutons également la déclaration avec la directive adaptée dans notre page JSP, qui utilise cette bibliothèque de tags.

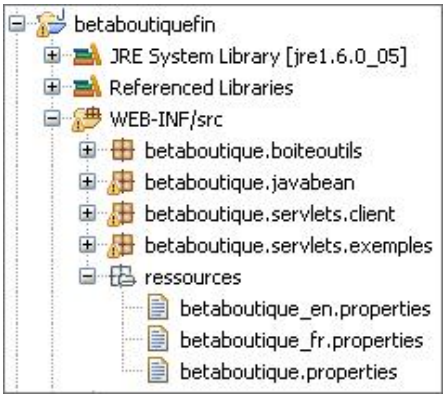
```
<%@ taglib uri="/WEB-INF/tld/fmt.tld" prefix="fmt" %>
```

Pour la mise en œuvre de la localisation des messages en Java, il faut utiliser un ensemble de fichiers appelés *bundle* en anglais. Il faut définir un fichier par langue/locale. Chaque fichier possède un préfixe commun appelé *basename* et doit avoir l'extension *.properties*. Les fichiers pour des langues particulières doivent avoir le même préfixe suivi d'un caractère underscore et du code langue. Pour être accessibles, ces fichiers doivent être inclus dans le classpath, ils

doivent donc être placés dans le répertoire */WEB-INF/classes*.

Ce répertoire est fréquemment effacé par Eclipse lors du développement, lors des mises à jour du projet (compilation des fichiers...). Nos fichiers de langues seront donc effacés à chaque fois. Le plus simple est alors de déposer les fichiers de langues dans un répertoire nommé */WEB-INF/src/ressources* car celui-ci sera copié automatiquement (et à chaque modification) dans le répertoire */WEB-INF/classes*.

Pour la mise en œuvre de l'internationalisation, nous allons créer un répertoire *ressources* dans l'arborescence */WEB-INF/src* et deux fichiers. Le premier fichier nommé *betaboutique.properties* correspond aux données de la langue par défaut (français dans notre cas) et le fichier *betaboutique_en.properties* correspond aux données en anglais. Nous utilisons également un fichier pour le français nommé *betaboutique_fr.properties*. Voici l'arborescence au final :



Un fichier de langue contient des paires de valeurs composées d'une clé unique et de sa valeur associée.

Exemple :

```
dvd.webmestre=Merci de contacter webmestre@betaboutique.com
```

Voici les deux fichiers que nous allons définir. Cet exemple regroupe seulement deux traductions, mais nous pouvons indiquer autant de traductions que souhaitées.

Fichier : *betaboutique.properties*

```
#gestion des dvd
dvd.accueil=Bienvenue sur la page des DVD
dvd.webmestre=Merci de contacter webmestre@betaboutique.com
```

Fichier : *betaboutique_en.properties*

```
#gestion des dvd en anglais
dvd.accueil=Welcome in the DVD page
dvd.webmestre=Please contact webmestre@betaboutique.com
```

La technique d'internationalisation abordée dans ce chapitre est valable pour toute la programmation Java et sera utilisée de la même façon avec des Servlets, Struts, des applications JWS ou des Applets.

- Le tag *setBundle* permet de localiser/paramétrer le *bundle* utilisé par défaut.
- Le tag *message* permet de localiser un message en fonction de sa langue et de l'afficher.
- Le tag *setLocale* permet de positionner une locale (changer de langue).
- Le tag *formatNumber* permet de formater un nombre en fonction de la locale.
- Le tag *parseNumber* permet des conversions.
- Le tag *formatDate* permet de formater la date selon la locale.
- Le tag *setTimeZone* permet de stocker le fuseau horaire dans une variable.

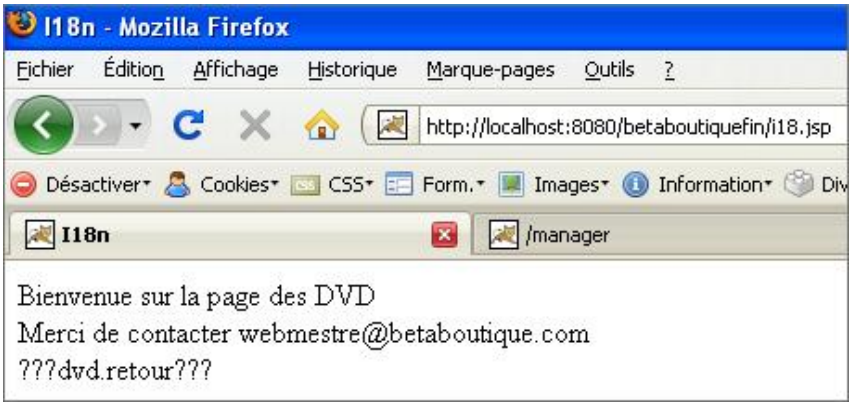
- Le tag *timeZone* permet de préciser le fuseau horaire à utiliser dans le corps de la page.

Avant de tester ces fonctionnalités, nous allons mettre notre navigateur en français. Pour cela, avec Firefox nous utilisons le menu **Outils - Options - Avancé - Langues - Choisir** et nous laissons uniquement les langues *Français[fr]* et *Français/France [fr-fr]*. Pour la prise en compte de ces opérations, il faut fermer le navigateur et ouvrir une nouvelle fenêtre.

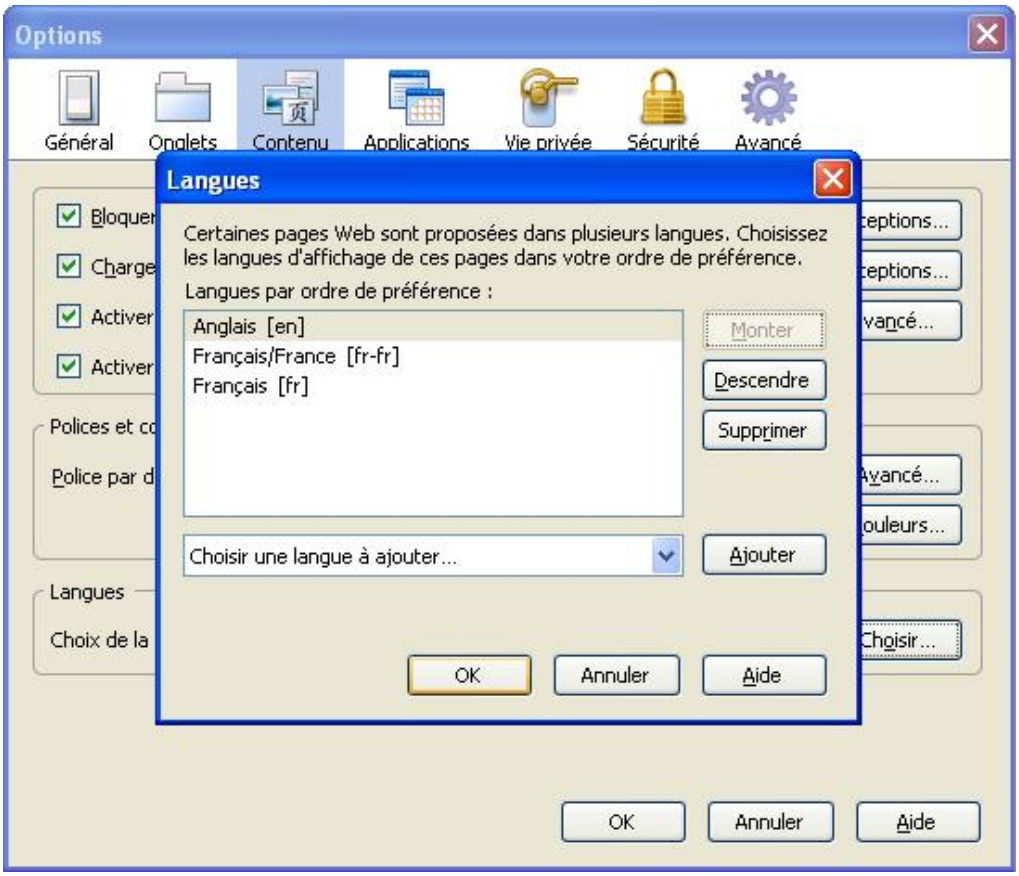
Pour mettre en application l'internationalisation, nous allons créer une page JSP nommée *i18n.jsp* et utiliser le code ci-dessous :

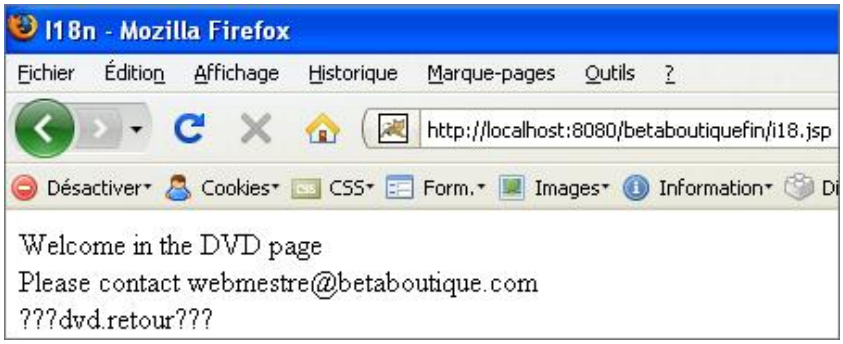
```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/x.tld" prefix="x" %>
<%@ taglib uri="/WEB-INF/tld/sql.tld" prefix="sql" %>
<%@ taglib uri="/WEB-INF/tld/fmt.tld" prefix="fmt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>I18n</title>
</head>
<body>
<!--  préciser le bundle/fichier de langue à utiliser -->
<fmt:setBundle basename="ressources.betaboutique" />
<!--  afficher le message appelé dvd.accueil -->
<fmt:message key="dvd.accueil"/>
<br/>
<!--  afficher le message appelé dvd.webmestre -->
<fmt:message key="dvd.webmestre"/>
<br/>
<!--  afficher un message inexistant -->
<fmt:message key="dvd.retour"/>
</body>
</html>
```

Si aucun couple de valeur n'est trouvé pour la clé fournie, le tag renvoie alors *???xxx???* où *xxx* représente le nom de la clé, comme pour le cas : *dvd.retour*.



Notre page est correctement affichée en français (langue par défaut dans le navigateur). Nous allons paramétrer notre navigateur en anglais selon le même principe que précédemment et relancer le navigateur.





La page est automatiquement traduite dans la langue sélectionnée.

Cette technique de programmation très utile permet de proposer des sites multilingues. Par la suite, nous pourrons proposer un service basé sur une Servlet qui permet de changer la locale à la volée avec des liens HTML. Nous pouvons également modifier une locale avec les taglibs JSP de cette façon (après avoir positionné le navigateur en français) :

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/x.tld" prefix="x" %>
<%@ taglib uri="/WEB-INF/tld/sql.tld" prefix="sql" %>
<%@ taglib uri="/WEB-INF/tld/fmt.tld" prefix="fmt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>I18n</title>
</head>
<body>
<!--  locale en anglais -->
<fmt:setLocale value="en"/>
<!--  préciser le bundle/fichier de langue à utiliser -->
<fmt:setBundle basename="ressources.betaboutique" />
```

```
<!--  afficher le message appelé dvd.accueil -->
<fmt:message key="dvd.accueil"/>
<br/>
<!--  afficher le message appelé dvd.webmestre -->
<fmt:message key="dvd.webmestre"/>
<br/>
<!--  afficher un message inexistant -->
<fmt:message key="dvd.retour"/>
</body>
</html>
```

d. La bibliothèque DataBase

Cette bibliothèque SQL facilite l'accès et la manipulation de bases de données. Elle permet entre autres de créer une connexion vers une base de données, de réaliser des requêtes de sélection, d'encapsuler plusieurs requêtes dans une transaction ou de réaliser des mises à jour. Ces balises sont très pratiques pour des sites qui nécessitent un développement rapide. Par contre, elles sont peu utilisées dans les projets qui séparent la partie Vue de la partie Modèle/Accès aux données. En utilisant de telles balises, en effet le code HTML, le code de traitement et l'accès aux données sont mélangés dans la même page JSP.

L'exemple suivant nommé *sql.jsp* permet de se connecter à la base de données *betaboutique* et de lister la totalité des enregistrements de la table *categorie* afin de générer le menu principal de navigation. Le tag permet de créer un lien vers la base de données à partir d'une connexion simple ou d'un pool de connexion JDBC.

Ce code utilise une connexion JDBC vers une base de données MySQL. Il est donc nécessaire de copier le pilote MySQL approprié dans le répertoire */WEB-INF/lib* de l'application ou */common/lib* du serveur Tomcat ou */lib* du serveur Tomcat 6.X.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/sql.tld" prefix="sql" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>SQL</title>
</head>
<body>
<sql:setDataSource driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/betaboutique"
user="root" password=""/>
<!-- sélectionner toutes les catégories  -->
<sql:query sql="SELECT * FROM categorie ORDER BY nomcategorie
ASC" var="categories"/>
<!-- afficher les catégories  -->
<ul>
<c:forEach var="categorie" begin="0" items="${categories.rows}">
    <li><c:out value="${categorie.nomcategorie}"/></li>
</c:forEach>
</ul>
</body>
</html>
```

Dans le cas d'un pool de connexion JDBC, c'est le tag *<sql:setDataSource.../>* qui change. Le nom de notre connexion JNDI est précisé en paramètre.

```
<sql:setDataSource dataSource="jdbc_betaboutique_MySQL"/>
```

Cette bibliothèque de tag permet également de gérer des paramètres dans les requêtes, les transactions, les rollbacks...

Bibliothèque de balises personnalisées

1. Présentation

Nous avons étudié les balises/taglibs proposées par Java, des communautés diverses et des développeurs. Parfois, pour les besoins d'un projet ou par habitude de développement nous avons besoin d'une bibliothèque de balises qui n'existe pas encore ou qui ne fournit par les services souhaités (balises pour la pagination, le cryptage de données, la transformation de chaînes de caractères). Nous pouvons alors développer notre propre bibliothèque de balises personnalisées en suivant le principe du standard JSTL et de ses taglibs.

De cette façon, nos pages JSP ne contiendront aucun code Java. En fait, il ne s'agit pas vraiment de supprimer le code Java mais plutôt de le masquer de façon qu'il soit invisible pour les graphistes/concepteurs des pages.

Nous avons utilisé plusieurs balises JSP standards comme *<jsp:useBean.../>*, *<jsp:forward.../>*... Ces balises répondent à des besoins courants et sont utilisées par tous les programmeurs JSP. Les bibliothèques de balises servent à répondre à des besoins particuliers pour nos applications. De même, lorsqu'un programmeur a conçu une bibliothèque de balises, les développeurs peuvent ensuite s'en servir dans leurs pages JSP sans connaître le détail de leur implémentation. Ces balises dynamiques s'utilisent de la même manière que les balises HTML, le partage des tâches entre programmeurs et concepteurs de l'interface devient beaucoup plus clair.

2. Actions personnalisées

Le terme action personnalisée fait référence à une balise dans une page JSP. Les actions personnalisées sont employées dans des pages JSP comme des balises ordinaires. Elles sont identifiées par un préfixe et un nom : *<préfixe:nom/>*. Le préfixe permet d'éviter des conflits de noms entre les balises des différentes bibliothèques. Ce préfixe est choisi par le développeur. Le préfixe est suivi du nom de l'action, également choisi par le développeur de la bibliothèque. Les actions peuvent être vides (juste la présence de la balise) mais peuvent également posséder un corps. Enfin, les actions peuvent avoir des attributs qui spécifient les détails de leur comportement.

3. Mise en place

La première étape dans la création d'une balise personnalisée consiste à créer le fichier de classe gestionnaire de balises. Ce gestionnaire de balises stocke les méthodes qui lancent des actions spécifiques lorsque les balises personnalisées sont traitées. Cette classe Java chargée d'implémenter le comportement de l'action doit respecter les spécifications d'un JavaBean et implémenter une des interfaces d'extension de balises.

Il existe plusieurs interfaces pour la gestion des actions :

- *Tag* et *BodyTag* : pour les actions simples avec ou sans corps.
- *IterationTag* : pour gérer les itérations plus facilement.
- *SimpleTag* et *JspFragmentTag* : pour encapsuler le contenu du corps de l'action dans un objet.

Les gestionnaires de balises simples

L'interface *SimpleTag* et la classe *SimpleTagSupport* permettent d'implémenter tous les gestionnaires de balises JSP 2.0, avec ou sans itération et évaluation du corps. Pour utiliser une action personnalisée, il suffit de créer une classe qui étend la classe de base *SimpleTagSupport* et redéfinir les méthodes nécessaires pour produire le comportement souhaité.

Dans la majorité des cas, la méthode *doTag()* est suffisante. Cette méthode gère l'intégralité du comportement de l'action (sans se soucier du début de la balise, du corps et de la fin). Un gestionnaire de balises doit importer les paquetages *javax.servlet.jsp* et *javax.servlet.jsp.tagext*.

Un gestionnaire de balises simple doit avoir une méthode *doStartTag()* qui contient le code exécuté. Cette méthode doit être déclarée en accès public afin d'être accessible en dehors du gestionnaire de balises. La méthode *doStartTag()* doit retourner une valeur pour indiquer que la balise renvoie quelque chose ou la valeur *SKIP_BODY* pour sauter cette étape.

Afin de mettre en place notre bibliothèque de balises, nous allons commencer par créer un nouveau paquetage nommé *taglib* et placer la classe suivante dans ce paquet.

```
package taglib;
```

```
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.*;

public class SimpleTag extends TagSupport {

    public int doStartTag()throws JspException
    {

        return SKIP_BODY;
    }
}
```

La classe du gestionnaire de balises est créée, nous pouvons utiliser l'objet *PageContext* pour gérer le flux de sortie et envoyer des données à la page utilisatrice. La méthode *getOut()* de cet objet permet d'envoyer des informations au client. Cette méthode peut générer une exception qu'il est donc nécessaire de traiter.

```
package taglib;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.*;

public class SimpleTag extends TagSupport {

    public int doStartTag()throws JspException
    {
        try
        {
           pageContext.getOut().print("Webmestre : info@betaboutique.com");
        }
        catch(Exception e)
        {
           throw new JspTagException(e.getMessage());
        }
        return SKIP_BODY;
    }
}
```

4. Mise en place d'un fichier de description

Les fichiers de définition des bibliothèques portent l'extension *.tld* comme par exemple *c.tld*, *x.tld* ou *fmt.tld*. Ces fichiers qui sont des grammaires XML, sont placés dans le répertoire */WEB-INF/tld* de l'application courante.

Une fois le gestionnaire créé avec une classe Java adaptée, nous devons créer le fichier de description au format XML. Le fichier commence par un en-tête XML qui contient des informations sur le fichier avec le prologue et la grammaire utilisée. Nous commençons par créer le fichier suivant : */WEB-INF/tld/betaboutique.tld*.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

- Les balises <*taglib*> et <*taglib/*> sont utilisées pour délimiter le corps principal du fichier de description.
- La balise <*tlib-version/*> correspond au numéro de version de la bibliothèque du concepteur.
- La balise <*jsp-version/*> correspond à la version JSP supportée par la bibliothèque du développeur.
- La balise <*short-name/*> est un nom abrégé de la bibliothèque.
- La balise <*uri/*> est un identificateur optionnel de ressources.
- La balise <*info/*> est une brève description de la bibliothèque.

Ensuite, vient la définition de chacune des balises. Chaque définition commence avec la balise <*tag/*> qui accepte les sous-éléments ci-dessous :

- La balise <*name/*> est le nom officiel de la balise. Ce nom est le même que celui utilisé dans les pages JSP.
- La balise <*tag-class/*> est la désignation de la classe Java supportant la balise.
- La balise <*info/*> est une description de la balise.
- La balise <*body-content/*> stipule le contenu de la balise.

Voici notre fichier de description *betaboutique.tld* dans sa version minimale :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>2.0</jsp-version>
    <short-name>Une premiere balise pour le Webmestre</short-name>
    <tag>
        <name>webmestre</name>
        <tag-class>taglib.SimpleTag</tag-class>
    </tag>
</taglib>
```

5. Configuration de la librairie dans le descripteur web.xml

Nous avons créé le code du gestionnaire de balises ainsi que sa grammaire associée, nous devons maintenant réaliser la dernière étape qui consiste à paramétrer le descripteur de déploiement (*web.xml*) avec la balise <*taglib.../*>. Cette opération est identique à l'utilisation des actions standards JSTL, elle est souvent nommée aiguillage.

Il faut d'abord définir la balise <*taglib-uri.../*> pour préciser une adresse pleinement qualifiée. Puis il faut indiquer simplement la localisation de notre fichier de description des balises sur le serveur et recharger le contexte pour que ces modifications soient prises en compte.

```
...
<taglib>
    <taglib-uri>/WEB-INF/tld/betaboutique.tld</taglib-uri>
    <taglib-location>/WEB-INF/tld/betaboutique.tld</taglib-location>
</taglib>
...
```

6. Utilisation d'une librairie personnalisée

Nous avons suivi les trois étapes nécessaires à la mise en place d'une bibliothèque de balises personnalisées, à savoir :

- la création de la classe Java pour le gestionnaire de balises ;
- le codage du fichier de description des balises au format XML ;
- la configuration du fichier *web.xml*.

Nous pouvons maintenant utiliser notre bibliothèque avec la directive <*%@ taglib .../*> et l'URI correspondante dans une page JSP. Nous allons reprendre notre page *bienvenue.jsp* du projet betaboutique pour mettre en œuvre la librairie.

La directive *taglib* emploie l'attribut *uri* pour spécifier un identifiant relatif au fichier de description des balises. Cet identifiant doit être le même que celui spécifié dans le fichier *web.xml*. Il faut également préciser un préfixe pour référencer la bibliothèque de balises qui contient l'information sur la balise personnalisée. Chaque bibliothèque de

balises a besoin d'un préfixe différent pour éviter des conflits d'actions.

Pour utiliser une balise personnalisée dans une page JSP, nous devons saisir le préfixe et le nom de l'action que nous avons affectés dans le fichier de description, séparés par un symbole deux points. Les balises simples qui ne contiennent pas de corps ou d'information entre la balise de début et de fin peuvent être réduites à une seule balise.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<bb:webmestre/>
OU
<bb:webmestre></bb:webmestre>
</body>
</html>
```

7. Gestionnaire de balises et gestion des attributs

Une balise personnalisée peut contenir des attributs qui seront spécifiés au sein de la page JSP. Dans une balise personnalisée, un attribut est représenté par une variable au sein du fichier de classe du gestionnaire de balises. Cette variable peut avoir une valeur par défaut, qui sera utilisée par la balise si aucune valeur n'est précisée pour l'attribut lors de son utilisation.

Lorsque, dans la page JSP, la balise est utilisée avec un attribut, la valeur donnée pour cet attribut est transmise au gestionnaire de balises. Il faut alors créer une méthode d'affectation (accesseur) pour cet attribut dans le gestionnaire de balises. La méthode doit être en accès public, il n'y a pas de type de retour et le nom de la méthode est le même que celui de l'attribut.

Nous allons créer une seconde balise qui affiche l'adresse email du Webmestre avec un lien *mailto* ou pas suivant le choix du codeur.

```
package taglib;

import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.*;

public class SimpleTag extends TagSupport {

    //lien en mailto ou pas
    private boolean mailto=false;

    //action
    public int doStartTag()throws JspException
    {
        try
        {
            if(mailto)
{
               pageContext.getOut().print("<a
href=\"mailto:info@betaboutique.com\">Webmestre :
info@betaboutique.com</a>");
            }
            else
            {
                pageContext.getOut().print("Webmestre :
info@betaboutique.com");
            }
        }
        catch(Exception e)
```

```
        {
            throw new JspTagException(e.getMessage());
        }
        return SKIP_BODY;
    }

    //action a la fermeture du tag
    public int doEndTag()
    {
        try
        {
            JspWriter out=pageContext.getOut();
            out.print("<hr/>");
        }
        catch(Exception e)
        {

        }
        return SKIP_BODY;
    }

    public boolean isMailto() {
        return mailto;
    }

    public void setMailto(boolean mailto) {
        this.mailto = mailto;
    }
}
```

Ensuite, nous devons modifier notre fichier de description pour gérer les attributs. La balise *<attribute.../>* permet de préciser des détails sur un attribut et doit se situer à la suite de la balise *<tagclass>* dans le fichier de description. Le nom de l'attribut est précisé à l'aide de la balise *<name>*. Le nom donné à cette balise est sensible à la casse et doit être le même que celui utilisé dans les pages JSP. La balise suivante *<required>* indique si l'attribut est indispensable lors de son utilisation dans une page JSP. Si la valeur utilisée est *false*, cet attribut sera optionnel. Si sa valeur est *true*, l'attribut devra être obligatoirement utilisé, sinon la page JSP provoquera une erreur.

Voici le nouveau contenu du fichier *betaboutique.tld*.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>2.0</jsp-version>
    <short-name>Une premiere balise pour le Webmestre</short-name>
    <tag>
        <name>webmestre</name>
            <tag-class>taglib.SimpleTag</tag-class>
            <attribute>
                <name>mailto</name>
                <required>true</required>
        </attribute>
    </tag>
</taglib>
```

Si désormais nous utilisons notre page *bienvenue.jsp* avec le code suivant, la page nous indique que d'après la grammaire de la bibliothèque l'attribut *mailto* est obligatoire.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
```

```
</head>
<body>
<bb:webmestre/>
</body>
</html>
```

Nous devons donc préciser l'attribut *mailto* pour générer une page sans erreur.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>

<bb:webmestre mailto="false"/>
OU
<bb:webmestre mailto="true"/>

</body>
</html>
```

La page est exécutée sans erreur et l'attribut *mailto* est bien pris en compte. Vous remarquez également l'utilité de la fonction *doEndTag()* qui permet d'afficher une barre de séparation HTML (*<hr/>*) après chaque utilisation de la balise.

L'ajout de la balise *<rtexprvalue>true</rtexprvalue>* dans le fichier de description autorise la valeur de l'attribut à être affectée durant l'exécution du code JSP.

8. Gestionnaire de balises et gestion du corps des balises

Le corps d'une balise est l'information comprise entre les balises de début et de fin. Le corps de la balise peut être constitué de texte ou de code JSP. Un gestionnaire de balises peut donc être créé pour utiliser l'information contenue dans le corps de la balise personnalisée.

La méthode *doStartTag()* est toujours utilisée, mais pour une balise avec corps, cette méthode doit retourner la valeur *EVAL_BODY_INCLUDE*, pour que le serveur Web traite l'information contenue dans le corps de la balise. Le gestionnaire de balises doit également inclure une méthode *doEndTag()* lors du traitement du corps. Cette méthode contient pour rappel, le code à exécuter après le traitement du corps.

La méthode *doEndTag()* doit retourner une valeur pour indiquer si le reste de la page JSP doit être traité ou non. Dans la plupart des cas, la valeur retournée est *EVAL_PAGE* pour indiquer que le reste de la page doit être traité par le serveur. Pour que le reste de la page ne soit pas traité, il faut utiliser la valeur *SKIP_PAGE* (pour les sessions par exemple).

```
package taglib;

import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.*;
public class SimpleTag extends TagSupport {

    //lien en mailto ou pas
```

```
    private boolean mailto=false;

    //action
    public int doStartTag()throws JspException
    {
        try
        {
            if(mailto)
            {
                pageContext.getOut().print("<a
href=\"mailto:info@betaboutique.com\">");
            }
            else
            {
                pageContext.getOut().print("Webmestre : ");
            }
        }
        catch(Exception e)
        {
            throw new JspTagException(e.getMessage());
        }
        return EVAL_BODY_INCLUDE;
    }

    //action a la fermeture du tag
    public int doEndTag()
    {
        try
        {
            JspWriter out=pageContext.getOut();
            if(mailto)
            {
                pageContext.getOut().print("</a>");
            }
            out.print("<hr/>");
        }
        catch(Exception e)
        {

        }
        return EVAL_PAGE;
    }

    public boolean isMailto() {
        return mailto;
    }
    public void setMailto(boolean mailto) {
        this.mailto = mailto;
    }
}
```

Après avoir créé un fichier gestionnaire de balises pour une balise personnalisée, il est conseillé d'inclure dans le fichier de description une note indiquant si la balise comprend un corps. La balise <*body-content/*> permet de signaler la présence éventuelle d'un corps. Le type de contenu de la balise personnalisée est alors précisé en insérant une valeur exemple entre les balises <*body-content*> et <*/body-content*>. Si aucun type de contenu n'est spécifié, il prendra la valeur précisée par défaut. Une bonne habitude est d'utiliser le terme JSP pour préciser le type de contenu.

Il est conseillé de toujours inclure la balise <*body-content*> au sein du fichier de description. Dans le cas d'une balise vide il faudra utiliser <*body-content*>*empty*</*body-content*>. Si par contre, des instructions dynamiques autre que du code JSP sont utilisées, il faut insérer la définition suivante : <*body-content*>*tagdependent*</*body-content*>.

Nous pouvons désormais utiliser notre balise avec un corps, avec le fichier de définition suivant :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

```
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>2.0</jsp-version>
    <short-name>Une premiere balise pour le Webmestre</short-name>
    <tag>
        <name>webmestre</name>
        <tag-class>taglib.SimpleTag</tag-class>
        <body-content>JSP</body-content>
        <attribute>
            <name>mailto</name>
            <required>true</required>
        </attribute>
    </tag>
</taglib>
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<bb:webmestre mailto="false">contact@betaboutique.com</bb:webmestre>
<bb:webmestre mailto="true">info@betaboutique.com</bb:webmestre>
</body>
</html>
```

9. Gestionnaire de balises et gestion du contenu du corps

Le principe de l'exemple précédent est très intéressant mais il nous empêche de manipuler le contenu du corps de la balise (adresse email dans notre cas). Pour manipuler le corps d'une balise, le gestionnaire de balises doit dériver de la classe *BodyTagSupport*. La classe *BodyTagSupport* dérive elle-même de la classe *TagSupport* et contient des méthodes permettant ce type de traitement.

Il faut créer une méthode *doAfterBody()* pour traiter le corps d'une balise. Dans cette méthode, l'objet *BodyContent* permet de stocker des informations sur le corps reçu. La méthode *getString()* permet de récupérer le corps de la balise sous la forme d'une chaîne de caractères qui peut alors être manipulée. Enfin, la méthode *getEnclosingWriter()* de l'objet *BodyContent* permet de retourner le résultat à la page JSP.

Le code suivant permet de traiter le corps passé avec la balise et de le mettre en majuscule.

```
package taglib;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.*;

public class SimpleTag extends BodyTagSupport {

    //lien en mailto ou pas
    private boolean mailto=false;

    //action
    public int doAfterBody()throws JspException
    {
        try
        {
            BodyContent bodycontent=this.getBodyContent();
            String bodytext=bodycontent.getString();

            if(mailto)
            {
                bodycontent.getEnclosingWriter().print("<a
href=\"mailto:info@betaboutique.com\">"+bodytext.toUpperCase()+"
```

```
</a><br/><hr/>");
            }
            else
            {
                bodycontent.getEnclosingWriter().print("Webmestre :
"+bodytext.toUpperCase()+"<br/><hr/>");
            }
        }
        catch(Exception e)
        {
            throw new JspTagException(e.getMessage());
        }
        return EVAL_BODY_INCLUDE;
    }

    public boolean isMailto() {
        return mailto;
    }
    public void setMailto(boolean mailto) {
        this.mailto = mailto;
    }
}
```

L'utilisation d'une balise personnalisée qui manipule le contenu d'un corps est identique à l'utilisation de toute autre balise contenant de l'information entre les indicateurs de début et de fin. La seule différence réside dans le fait que le gestionnaire de balises peut modifier cette information avant de la retourner à la page JSP. Si un problème surgit pendant le traitement du gestionnaire de balises, une erreur de serveur sera générée et le reste de la page JSP ne sera pas traité. Le corps de la balise peut contenir directement du texte ou être généré dynamiquement par d'autres méthodes, comme du code Java contenu dans une scriptlet ou une expression.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<bb:webmestre mailto="false">contact@betaboutique.com</bb:webmestre>
<bb:webmestre mailto="true">info@betaboutique.com
<%= new java.util.Date() %></bb:webmestre>
</body>
</html>
```

Cette technique très puissante est couramment utilisée pour accéder à une base de données, manipuler des images, réaliser du formatage (résumés, remplacement de caractères...) ou envoyer un email automatique.

Une fois que la librairie est terminée, c'est-à-dire que l'implémentation du code est réalisée et que la grammaire *.tld* est déclarée, nous pouvons empaqueter la totalité de la librairie dans une archive au format *.jar*. Nous allons illustrer par un nouvel exemple la lecture de paramètre dans le fichier de configuration de l'application *web.xml*. Cette technique est utilisée pour réaliser des liens vers des ressources comme les feuilles de style CSS ou les fichiers JavaScript.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <context-param>
        <param-name>urlApplication</param-name>
        <param-value>http://192.168.0.1:8080/betaboutique/</param-value>
    </context-param>
</web-app>
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
<title>TAGLIB</title>
</head>
<body>
<%
String parametre=pageContext.getServletContext().getInitParameter
("urlApplication");
if(parametre!=null)
{
    out.println(parametre);
}
%>
</body>
</html>
```

Cette technique est très lourde à mettre en place et peu adaptée. Nous allons développer une taglib simple permettant de réaliser ce type de code avec une seule balise. Nous commençons par créer une nouvelle classe nommée *Configuration* dans le paquetage *betaboutique.taglib*.

```
package betaboutique.taglib;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.*;

@SuppressWarnings("serial")
public class Configuration extends TagSupport
{
    private String attribut;

    public int doStartTag() throws JspException
    {
        try
        {
            /* afficher le paramètre contenu dans le contexte
de l'application */
            pageContext.getOut().print(pageContext.getServlet
Context().getInitParameter(attribut));
        }

        catch (Exception e)
        {
            throw new JspTagException(e.getMessage());
        }

        return SKIP_BODY;
    }
    public int doEndTag()
    {
        return SKIP_BODY;
    }
    public void setAttribut(String attribut)
    {
        this.attribut = attribut;
    }
}
```

Ensuite, nous procédons à la définition de la grammaire avec le fichier configuration.tld dans le répertoire */WEB-INF/tld*.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag
Library 1.2//EN" "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>2.0</jsp-version>
    <short-name>Opérations sur les parametres</short-name>
    <tag>
        <name>config</name>
        <tag-class>betaboutique.taglib.Configuration</tag-class>
        <body-content>JSP</body-content>
```

```
        <attribute>
            <name>attribut</name>
            <required>true</required>
        </attribute>
    </tag>
</taglib>
```

Il ne reste maintenant plus que la dernière étape de la configuration, à savoir la déclaration de la librairie dans le fichier de gestion de l'application *web.xml*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <context-param>
        <param-name>urlApplication</param-name>
        <param-value>http://192.168.0.1:8080/betaboutique/
</param-value>
    </context-param>
    <taglib>
        <taglib-uri>/WEB-INF/tld/configuration.tld</taglib-uri>
        <taglib-location>/WEB-INF/tld/configuration.tld</taglib-location>
    </taglib>
</web-app>
```

L'exemple précédent peut maintenant être remplacé par le code simple suivant :

```
<%@ taglib uri="/WEB-INF/tld/configuration.tld" prefix="conf" %>
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head><title>TAGLIB</title></head>
<body>
<conf:config attribut="urlApplication"/>
</body>
</html>
```

Les JavaBeans ou Beans

1. Présentation

Un composant JavaBean, également appelé composant logiciel, est une classe Java conçue pour être réutilisable lors des développements en Java. Un composant JavaBean définit :

- Des propriétés correspondant aux données d'un objet.
- Des événements permettant au JavaBean de communiquer avec d'autres classes.

Une simple classe est un composant JavaBean si :

- Elle est en accès public.
- Elle possède un constructeur public sans paramètre (constructeur par défaut).
- Elle définit des méthodes préfixées par *get* et *set* appelées accesseurs, permettant d'interroger et de modifier des données de la classe. Les propriétés peuvent être éventuellement héritées d'une super-classe.

Les composants JavaBeans permettent une séparation entre le code Java et la gestion de l'affichage des données. En renvoyant le code dans les Beans au lieu de le laisser accessible dans les scriptlets, nous allégeons le code source JSP. En informatique, les standards favorisent la portabilité des programmes d'un système à un autre. Nombreux sont les programmeurs qui ont été forcés de récrire tout ou partie de leurs programmes pour les rendre utilisables sur d'autres plates-formes que celle d'origine.

Un composant JavaBean est un composant logiciel. Un composant logiciel est un bloc de programme élémentaire qui offre des accès via une interface, ce qui permet de masquer la complexité des détails du composant. Le but est de concevoir des applications complexes et volumineuses par combinaison de petits blocs.

Le modèle JavaBean (JCA) propose un standard pour le développement de composants réutilisables et portables en langage Java. Les JavaBeans sont des composants fonctionnels capables de communiquer entre eux de façon standardisée. L'architecture JCA est en théorie apte à être déployée sous n'importe quel système d'exploitation et dans n'importe quel environnement applicatif. Les JavaBeans permettent de masquer leurs détails fonctionnels, cette approche est appelée sous le terme : *encapsulation*. Chaque objet possède une partie privée, inaccessible aux objets qui utilisent ses services et une partie publique, l'interface. Chaque objet est donc capable de travailler avec d'autres, chaque composant simple (article, utilisateur, produit...) est responsable d'une tâche bien précise qui participe à l'ensemble du projet.

2. Utilisation

Techniquement, les JavaBeans permettent d'éviter que le code des pages JSP devienne trop long et difficile à manipuler. Les méthodes publiques qui conservent l'intégrité du principe d'encapsulation servent à lire et modifier les valeurs de certaines variables du JavaBean. Les méthodes qui servent à lire sont des lecteurs (accesseurs) et celles qui permettent de créer ou modifier des valeurs sont appelées des modificateurs (mutateurs). En pratique elles sont souvent appelées getter et setter. Les accesseurs et mutateurs ont des noms standardisés qui commencent par une minuscule et qui sont suivis par le nom de la propriété commençant par une majuscule.

Exemple : un attribut *prix* possède deux méthodes, *getPrix()* et *setPrix(param)*.

Pour notre projet *betaboutique*, nous avons créé un premier JavaBean *client* avec les attributs associés. Par la suite, nous aurons une classe JavaBean pour les articles de la boutique, les commandes et les catégories.

Le JavaBean de la classe *Client* possède la structure suivante :

```
package betaboutique.javabean;

public class Client implements java.io.Serializable {

    private String identifiant=null;
    private String motdepasse=null;

    //Constructeur par défaut (sans paramètre)
    public Client()
    {

    }

    public String getIdentifiant() {
```

```
        return identifiant;
    }

    public void setIdentifiant(String identifiant) {
        this.identifiant = identifiant;
    }

    public String getMotdepasse() {
        return motdepasse;
    }

    public void setMotdepasse(String motdepasse) {
        this.motdepasse = motdepasse;
    }
}
```

Cette classe est bien un JavaBean car elle est sérialisable, elle possède un constructeur par défaut sans paramètre et tous ses attributs conservent l'encapsulation en proposant uniquement un accès par l'intermédiaire des méthodes.

Si nous reprenons notre page d'authentification (*authentification.html*), la Servlet (*ServletAuthentification*) de vérification et la page de bienvenue (*bienvenue.jsp*), nous pouvons instancier un objet client en cas de succès et le lire dans la page JSP.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Authentification BetaBoutique</title>
</head>
<body>
<h1>Authentification - Client</h1>
<form action="authentificationclient" method="POST">
<table border="1" cellspacing="0" cellpadding="5">
<tr>
    <td>Identifiant/Login : </td>
    <td><input type="text" name="identifiant" id="identifiant"
value="" size="20"/></td>
</tr>
<tr>
    <td>Mot de passe : </td>
    <td><input type="text" name="motdepasse" id="motdepasse"
value="" size="20"/></td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="submit"
name="valider" id="valider" value="Valider"/></td>
</tr>
</table>
</form>
</body>
</html>
```

```
...
//vérifier l'égalité des valeurs
if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
{
        if(urlBienvenue==null)
        {
            throw new ServletException("Le paramètre
[urlBienvenue] n'a pas été initialisé");
        }
        else
        {
            //créer un JavaBean client
            Client client=new Client();
            client.setIdentifiant(identifiant);
            client.setMotdepasse(motdepasse);
            request.setAttribute("client",client);
        getServletContext().getRequestDispatcher
(urlBienvenue).forward(request, response);
        }
}
```

```
...
```

La page de bienvenue accède au JavaBean *client* de façon simple et intuitive.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>

<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="request"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>

</body>
</html>
```

Pour rappel, la balise *<jsp:useBean.../>* est obligatoire et permet par le biais de ses attributs de manipuler l'objet de portée indiquée par l'attribut *scope*. L'attribut *id* correspond au nom du JavaBean à créer ou à récupérer dans la portée (variable qui servira à manipuler l'objet), l'attribut *class* correspond à la classe du JavaBean et l'attribut *scope* correspond à la portée où le JavaBean va être créé ou lu. Si l'objet est présent dans la portée, il est lu, sinon un nouvel objet de nom indiqué par l'attribut *id* est créé.

La portée *scope* peut avoir les valeurs suivantes :

- *request* : JavaBean valable uniquement dans la requête ;
- *session* : JavaBean valable tout au long de la session de l'utilisateur ;
- *page* : JavaBean valable uniquement pour la page en cours ;
- *application* : JavaBean partagé par l'ensemble des pages de l'application.

Suite à cette déclaration, le moteur JSP sait que l'objet désigné est un composant JavaBean, ce qui permet d'exploiter les caractéristiques particulières à ce genre d'objet.

Les JavaBeans et les Entreprises JavaBeans (EJB) sont deux choses différentes, mais en raison de quelques similarités, ils partagent le même nom. Les JavaBeans sont des composants écrits en Java manipulés par des Servlets et pages JSP. Les EJB sont des composants spéciaux, fonctionnant sur serveur Java EE et utilisés pour construire la logique applicative et d'accès aux données.

Les balises JavaBean les plus utilisées sont : *<jsp:setProperty.../>* qui permet d'assigner une valeur à une propriété (ou toutes les valeurs automatiquement avec le signe *) et *<jsp:getProperty.../>* qui permet de lire la valeur d'un attribut. La balise *<jsp:setProperty.../>* possède un attribut *param* qui permet de lire un attribut dans la requête avec le nom indiqué par le champ param et de l'affecter directement au JavaBean.

La pseudo-valeur * force toutes les propriétés d'un composant JavaBean à prendre les valeurs qui ont été transmises au serveur dans le flux de la requête. Si cette technique est utilisée au retour d'une Servlet ou suite à une saisie dans un formulaire HTML, les noms des composants de saisie du formulaire doivent être les mêmes que les propriétés correspondantes dans le composant JavaBean.

Les JavaBeans peuvent être créés dans une page JSP et être utilisés dans d'autres pages JSP de la même application par exemple, en fonction de la portée déclarée. Nous allons montrer ce principe en utilisant un lien sur la page de bienvenue *bienvenue.jsp* pour aller sur la page *sessionjavabean.jsp* et afficher les valeurs du JavaBean.

```
...
//vérifier l'égalité des valeurs
    if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
    {
        if(urlBienvenue==null)
        {
            throw new ServletException("Le paramètre
[urlBienvenue] n'a pas été initialisé");
        }
```

```
        else
        {
        //créer un JavaBean client
        Client client=new Client();
        client.setIdentifiant(identifiant);
        client.setMotdepasse(motdepasse);
        HttpSession session=request.getSession();
        session.setAttribute("client",client);
session.setAttribute("identifiantutilisateur",
"un autre identifiant envoyé avec la requête");
        getServletContext().getRequestDispatcher
(urlBienvenue).forward(request, response);
        }
    }
...
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" <%@ page language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>
<jsp:setProperty name="client" property="identifiant"
value="changement de l’identifiant après affichage"/>
<a href="sessionjavabean.jsp">Lire le JavaBean dans la session</a>
</body>
</html>
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="betaboutique.javabean.Client" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head><meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
<h2>Bienvenue client avec la session : <jsp:getProperty
name="client" property="identifiant"/> - <jsp:getProperty
name="client" property="motdepasse"/></h2>
</body>
</html>
```

L'instruction *<jsp:useBean.../>* permet de créer un JavaBean d'un type spécifié puis de le lier au nom fourni par l'attribut *id*. En fait, l'instruction *<jsp:useBean.../>* se comporte ainsi seulement si aucun composant JavaBean de ce type et de ce nom n'a été détecté dans la portée déclarée. S'il en existe un (créé par une autre page JSP ou une Servlet), c'est le JavaBean existant qui sera employé. De plus, si le nom précisé par *id* est trouvé, l'action est réalisée avec succès, dans le cas contraire il en résulte une exception *ClassCastException*.

Le code placé entre la balise d'ouverture et de fermeture *<jsp:useBean...>... </jsp:useBean>* n'est exécuté que si le JavaBean n'existe pas. Si un JavaBean convient, le code est ignoré.

Dans l'exemple suivant, si le JavaBean n'existe pas, il utilise d'autres valeurs pour l'identifiant et le mot de passe. Dans le cas contraire, ce code n'est pas exécuté.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>
<jsp:setProperty name="client" property="identifiant"
value="changement de l'identifiant après affichage"/>
<a href="sessionjavabean.jsp">Lire le JavaBean dans la
session</a><br/><br/>

<jsp:useBean id="client2" class="betaboutique.javabean.Client"
scope="request">
    <jsp:setProperty name="client2" property="identifiant"
value="identifiant client 2"/>
    <jsp:setProperty name="client2" property="motdepasse"
value="motdepasse client 2"/>
    <%= "Le JavaBean client2 n'est pas présent dans la
requête, il a été créé et initialisé" %>
</jsp:useBean>
</body>
</html>
```

3. Librairies de gestion des JavaBeans

Les JavaBeans étant très utilisés dans le monde de la programmation Java, plusieurs sociétés et organismes ont développé des librairies spécifiques de leur gestion et manipulation.

Nous allons utiliser la librairie *commons-beansutils* développée par la communauté Apache (http://commons.apache.org/beanutils/). Cette librairie très pratique, est désormais intégrée en standard dans de très nombreux outils compatibles Java EE comme le framework Struts.

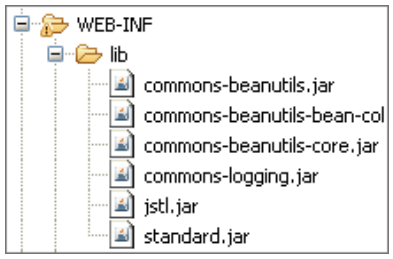
Le modèle (ou *pattern* en anglais) JavaBean est utilisé en standard, mais il manque parfois des méthodes aux librairies de base Java pour manipuler ces objets. Cette adresse fournit tous les composants nécessaires à l'utilisation de la librairie proposée par la communauté Apache :

(http://commons.apache.org/downloads/download_beanutils.cgi).

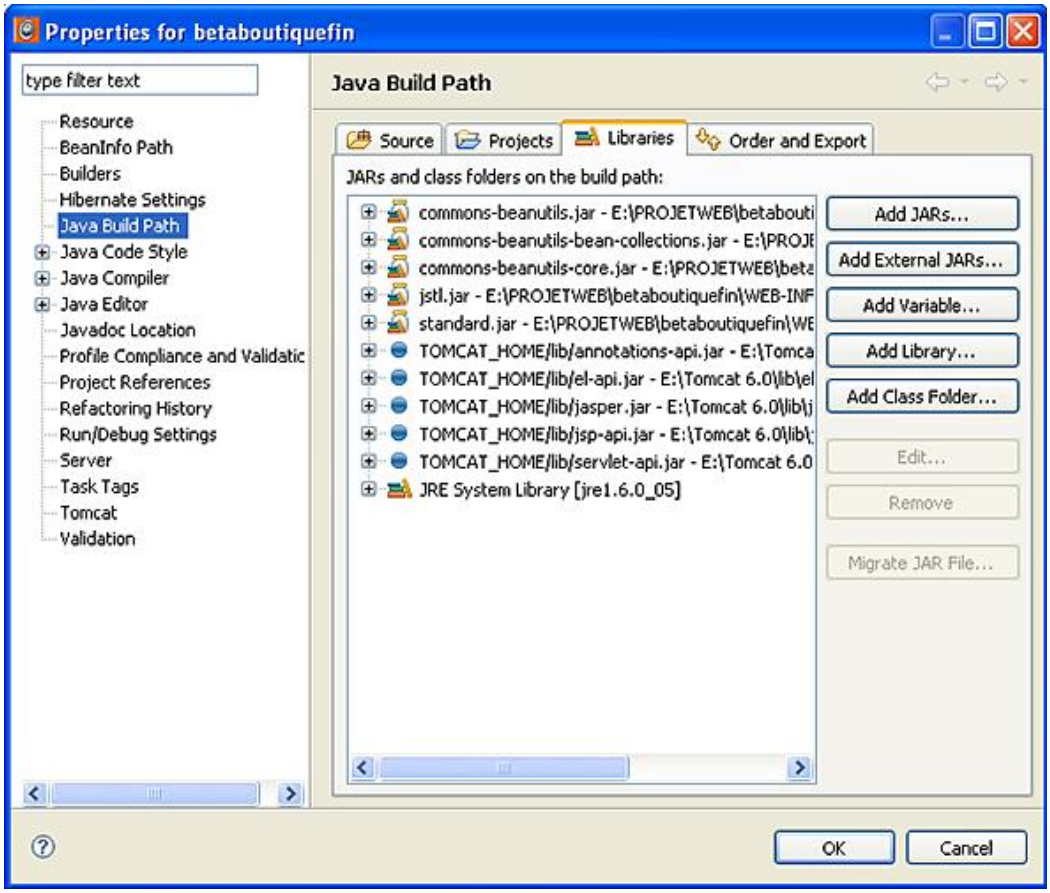
La documentation de l'API est présente à cette adresse :

(http://commons.apache.org/beanutils/apidocs/org/apache/commons/beanutils/packagesummary.html#package_description)

Après avoir téléchargé la librairie au format adapté, nous pouvons copier les archives *.jar* dans le répertoire */WEB-INF/lib* de notre projet et renseigner le classpath Java. L'arborescence de notre projet est alors la suivante :



Pour insérer les nouvelles librairies dans le projet/classpath, il faut utiliser le menu **Project - Properties - Java Build Path - Add External JARs** et ajouter les librairies.



Nous allons aborder la manipulation des JavaBeans avec la librairie *common-beans* et une nouvelle Servlet nommée *ServletCommonBean*. Cette Servlet sera utilisée pour manipuler le JavaBean *client* inséré dans la session par la Servlet d'authentification. Le JavaBean *client* est plus compliqué et possède désormais un attribut adresse qui est lui-même un objet.

```
package betaboutique.javabean;

public class Client implements java.io.Serializable {
    private String identifiant=null;
```

```
    private String motdepasse=null;
    private Adresse adresse=new Adresse();

    //Constructeur par défaut (sans paramètre)
    public Client()
    {

    }
    public String getIdentifiant() {
        return identifiant;
    }

    public void setIdentifiant(String identifiant) {
        this.identifiant = identifiant;
    }

    public String getMotdepasse() {
        return motdepasse;
    }

    public void setMotdepasse(String motdepasse) {
        this.motdepasse = motdepasse;
    }

    public Adresse getAdresse() {
        return adresse;
    }

    public void setAdresse(Adresse adresse) {
        this.adresse = adresse;
    }
}
```

```
package betaboutique.javabean;

public class Adresse implements java.io.Serializable {

    private String adresse=null;
    private String rue=null;

    //Constructeur par défaut (sans paramètre)
       public Adresse()
       {

       }

    public String getAdresse() {
        return adresse;
    }

    public void setAdresse(String adresse) {
        this.adresse = adresse;
    }

    public String getRue() {
        return rue;
    }

    public void setRue(String rue) {
        this.rue = rue;
    }
}
```

Nous allons réaliser une authentification par le biais de la Servlet adaptée qui va stocker un objet JavaBean *client* dans la session et nous rediriger sur la page de bienvenue. Sur cette page de retour, nous allons utiliser un lien HTML pour déclencher notre nouvelle Servlet de manipulation du JavaBean présent dans la session.

```
...
//vérifier l'égalité des valeurs
if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
{
            if(urlBienvenue==null)
```

```
            {
                throw new ServletException("Le paramètre
[urlBienvenue] n’a pas été initialisé");
            }
            else
            {
                //créer un JavaBean client
                Client client=new Client();
                client.setIdentifiant(identifiant);
                client.setMotdepasse(motdepasse);
                Adresse adresseclient=new Adresse();
                adresseclient.setAdresse("39800 POLIGNY");
                adresseclient.setRue("2 rue du haut");
                client.setAdresse(adresseclient);

                HttpSession session=request.getSession();
                session.setAttribute("client",client);
session.setAttribute("identifiantutilisateur", "un
autre identifiant envoyé avec la requête");
getServletContext().getRequestDispatcher
(urlBienvenue).forward(request, response);
            }
}
...
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>
<br/>
<a href="librairiecommonbean">Utiliser la librairie commonbean</a>
</body>
</html>
```

```
package betaboutique.servlets.client;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import org.apache.commons.beanutils.PropertyUtils;
import betaboutique.javabean.Client;

@SuppressWarnings("serial")
public class ServletCommonBean extends HttpServlet {


    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer la session
        HttpSession session=request.getSession();
        //créer un JavaBean client
        Client client=new Client();
        //récupérer notre JavaBean dans la session
        client=(Client)session.getAttribute("client");

        //afficher l’identifiant du client
        System.out.println("identifiant : "+client.getIdentifiant());
```

```
        //afficher les infos du client avec commonbean
        try
        {
            System.out.println("identifiant commonbean :
"+PropertyUtils.getSimpleProperty(client,"identifiant"));
            //changer une propriété du JavaBean à la volée
            PropertyUtils.setSimpleProperty(client,
"identifiant","ma nouvelle valeur");
            System.out.println("identifiant commonbean :
"+PropertyUtils.getSimpleProperty(client,"identifiant"));
            //lire une propriété objet du JavaBean
            System.out.println("adresse commonbean :
"+PropertyUtils.getNestedProperty(client,"adresse.rue"));
        }
        catch(Exception e)
        {
            System.out.println("Erreur de lecture");
        }
    }


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

}
```

La classe *PropertyUtils* a été utilisée car elle permet de manipuler des JavaBeans en programmation Java (donc dans une Servlet ou page JSP). Il est possible d'accéder à l'objet ainsi qu'à l'ensemble de ses propriétés.

La classe *PropertyUtils* permet de manipuler un JavaBean existant (instancié à partir d'une classe JavaBean). La classe *DynaBean* permet de manipuler des JavaBeans créés à partir d'une classe ou des JavaBeans créés à la volée, sans avoir une classe correspondante. Par exemple, toutes les données d'un formulaire HTML pour l'inscription d'une personne seront stockées dans un *DynaBean* et manipulées comme un JavaBean. La classe *WrapDynaBean* permet de réaliser la transformation d'un JavaBean vers un *DynaBean*.

Nous allons modifier notre classe pour créer un objet *DynaBean* à partir d'un JavaBean et afficher ses valeurs.

```
package betaboutique.servlets.client;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import org.apache.commons.beanutils.DynaBean;
import org.apache.commons.beanutils.WrapDynaBean;
import betaboutique.javabean.Adresse;
import betaboutique.javabean.Client;

@SuppressWarnings("serial")
public class ServletCommonBean extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer la session
        HttpSession session=request.getSession();
        //créer un JavaBean client
        Client client=new Client();
        //récupérer notre JavaBean dans la session
        client=(Client)session.getAttribute("client");

        try
        {
            //transformer l'objet client JavaBean en DynaBean
            DynaBean dynaBeanclientcopie = new WrapDynaBean(client);
            //lecture de ses valeurs
            System.out.println("dynaclient recopié identifiant :
"+dynaBeanclientcopie.get("identifiant"));
            Adresse adresse=(Adresse)(dynaBeanclientcopie.get("adresse"));
```

```
            System.out.println("dynaclient recopié adresse :
"+adresse.getRue());
        }
        catch(Exception e)
        {
            System.out.println("Erreur de lecture");
        }

    }

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

La classe *LazyDynaBean* permet de créer directement des *DynaBeans*, afin de les manipuler par la suite dans nos pages et de bénéficier de la puissance de ces objets.

```
package betaboutique.servlets.client;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.beanutils.DynaBean;
import org.apache.commons.beanutils.LazyDynaBean;

@SuppressWarnings("serial")
public class ServletCommonBean extends HttpServlet {


    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        try
        {
            //créer un dynabean dynamiquement
            DynaBean dynaBeanclient = new LazyDynaBean();
            dynaBeanclient.set("age","12 ans");
            dynaBeanclient.set("profession","vendeur");
            System.out.println("dynaclient créé :
"+dynaBeanclient.get("age"));
            System.out.println("dynaclient créé :
"+dynaBeanclient.get("profession"));
        }
        catch(Exception e)
        {
            System.out.println("Erreur de lecture");
        }

    }

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

}
```

Une autre classe très utile de manipulation des JavaBeans est la classe *BeanUtils*. Nous allons utiliser cette classe avec un formulaire HTML simple. Le formulaire nommé : *article.html* permet de saisir le nom de l'article, sa référence et son prix.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Authentification BetaBoutique</title>
</head>
```

```
<body>
<h1>Article</h1>
<form action="article" method="POST">
<table border="1" cellspacing="0" cellpadding="5">
<tr>
    <td>Nom :</td>
    <td><input type="text" name="nomarticle" id="nomarticle"
value="" size="20"/></td>
</tr>
<tr>
    <td>Référence : </td>
    <td><input type="text" name="refarticle" id="refarticle"
value="" size="20"/></td>
</tr>
<tr>
    <td>Prix : </td>
    <td><input type="text" name="prixarticle" id="prixarticle"
value="" size="20"/></td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="submit"
name="valider" id="valider" value="Valider"/></td>
</tr>
</table>
</form>
</body>
</html>
```

Cette page va déclencher la Servlet *ServletArticle* qui permet de renseigner directement un JavaBean *article* et d'aller sur une page d'affichage de l'article (*article.jsp*).

```
...
<servlet>
    <servlet-name>servletarticle</servlet-name>
    <servlet-class>betaboutique.servlets.client.ServletArticle
</servlet-class>
</servlet>
...
<servlet-mapping>
    <servlet-name>servletarticle</servlet-name>
    <url-pattern>/article</url-pattern>
</servlet-mapping>
...
```

```
package betaboutique.servlets.client;
import java.io.IOException;
import java.util.Enumeration;
import java.util.HashMap;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.beanutils.BeanUtils;
import betaboutique.javabean.Article;

@SuppressWarnings("serial")
public class ServletArticle extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
          //attention, pas de contrôle sur les saisies !
        //création d'un JavaBean article
        Article article=new Article();

        //création d'une collection
        HashMap map = new HashMap();
        Enumeration names = request.getParameterNames();
        while (names.hasMoreElements())
        {
          // chaque paramètre de la requête est mis dans une collection
          String name = (String) names.nextElement();
          map.put(name,request.getParameterValues(name));
```

```
        }

        try
        {
            //insérer toutes nos données dans le Bean
article en une seule étape
            BeanUtils.populate(article, map);
            //appeler la page d'affichage de l'article,
insérer l'article dans une base ou autre...
            System.out.println("Article : "+article.getNomarticle());
            //l'objet JavaBean article est mis dans la requête
            request.setAttribute("article", article);
            //redirection sur la page d'affichage
            getServletContext().getRequestDispatcher
("/article.jsp").forward(request, response);
        }
        catch(Exception e)
        {

        }
    }

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

}
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>FICHE ARTICLE</title>
</head>
<body>
<jsp:useBean id="article" class="betaboutique.javabean.Article"
scope="request"/>
<h2>ARTICLE</h2><br/>
Nom : <jsp:getProperty name="article"
property="nomarticle"/><br/>
Référence : <jsp:getProperty name="article"
property="refarticle"/><br/>
Prix : <jsp:getProperty name="article"
property="prixarticle"/><br/>
<br/>
</body>
</html>
```

Cette technique est très puissante. Elle permet de renseigner directement un objet à partir de données saisies. Cet objet pourra être ensuite manipulé avec toutes les librairies qui fonctionnent avec les JavaBeans et DynaBeans, comme la sérialisation d'objet, la transformation d'objets en données XML, PDF...

En une seule ligne, grâce au code suivant : *BeanUtils.populate(article, map);* tout notre objet a été renseigné.

En plus, si une propriété est envoyée et copiée dans notre JavaBean mais qu'elle n'appartient pas à notre JavaBean, cela ne gêne pas son fonctionnement. Par exemple, si nous insérons dans notre formulaire HTML un nouveau champ pour l'image de l'article mais que le JavaBean ne le possède pas, lors de la copie, cela ne provoquera pas d'erreur. L'attribut sera simplement ignoré.

Nous allons prendre un dernier exemple plus complexe avec la classe *Article* qui possède un attribut de type collection (ensemble de valeurs) pour stocker les acteurs associés à un article/DVD.

Le principe serait le même avec une propriété qui serait un objet d'une autre classe (ex : *personne* et l'instance de la classe *Adresse*).

```
package betaboutique.javabean;

import java.util.ArrayList;
```

```
public class Article implements java.io.Serializable {

    private String nomarticle=null;
    private String refarticle=null;
    private float prixarticle=0;
    private ArrayList<String> acteurs=new ArrayList<String>();

    //Constructeur par défaut (sans paramètre)
    public Article()
    {

    }

    public String getNomarticle() {
        return nomarticle;
    }

    public void setNomarticle(String nomarticle) {
        this.nomarticle = nomarticle;
    }

    public float getPrixarticle() {
        return prixarticle;
    }

    public void setPrixarticle(float prixarticle) {
        this.prixarticle = prixarticle;
    }

    public String getRefarticle() {
        return refarticle;
    }

    public void setRefarticle(String refarticle) {
        this.refarticle = refarticle;
    }

    public ArrayList<String> getActeurs() {
        return acteurs;
    }

    public void setActeurs(ArrayList<String> acteurs) {
        this.acteurs = acteurs;
    }
}
```

Le formulaire HTML permet de saisir les articles avec des champs *<input.../>*.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Authentification BetaBoutique</title>
</head>
<body>
<h1>Article</h1>
<form action="article" method="POST">
<table border="1" cellspacing="0" cellpadding="5">
<tr>
    <td>Nom :</td>
    <td><input type="text" name="nomarticle"
id="nomarticle" value="" size="20"/></td>
</tr>
<tr>
    <td>Référence : </td>
    <td><input type="text" name="refarticle"
id="refarticle" value="" size="20"/></td>
</tr>
<tr>
    <td>Prix : </td>
    <td><input type="text" name="prixarticle"
id="prixarticle" value="" size="20"/></td>
```

```
</tr>
<tr>
    <td>Acteur  1: </td>
    <td><input type="text" name="acteur1"
id="acteur1" value="" size="20"/></td>
</tr>
<tr>
    <td>Acteur  2: </td>
    <td><input type="text" name="acteur2"
id="acteur2" value="" size="20"/></td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="submit"
name="valider" id="valider" value="Valider"/></td>
</tr>
</table>
</form>
</body>
</html>
```

La Servlet est un peu plus complexe et permet de recomposer la collection des acteurs et de la stocker dans le JavaBean. La page d'affichage permet de parcourir la collection contenue dans le Bean.

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.HashMap;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.beanutils.BeanUtils;
import betaboutique.javabean.Article;

@SuppressWarnings("serial")
public class ServletArticle extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //création d'un JavaBean article
        Article article=new Article();
        //liste des acteurs
        ArrayList<String> acteurs=new ArrayList<String>();


        //création d'une collection
        HashMap map = new HashMap();
        Enumeration names = request.getParameterNames();
        while (names.hasMoreElements())
        {
          //chaque paramètre de la requête est mis dans une collection
          String name = (String) names.nextElement();

          //premier acteur
          if(name.equals("acteur1"))
          {
            acteurs.add((String)request.getParameter(name));
          }
          //second acteur
          else  if(name.equals("acteur2"))
          {
            acteurs.add((String)request.getParameter(name));
          }
          //paramètre habituel
          else
          {
              map.put(name,request.getParameterValues(name));
          }
        }
```

```
        //fin de la boucle, mettre la liste des acteurs dans le JavaBean
        map.put("acteurs",acteurs);

        try
        {
            //insérer toutes nos données dans le Bean
article en une seule étape
            BeanUtils.populate(article, map);
            //l'objet JavaBean article est mis dans la requête
            request.setAttribute("article", article);
            //redirection sur la page d'affichage
            getServletContext().getRequestDispatcher
("/article.jsp").forward(request, response);
        }
        catch(Exception e)
        {
            System.out.println("Erreur dans la Servlet ServletArticle");
        }
    }

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

}
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld"
prefix="bb" %>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>FICHE ARTICLE</title>
</head>
<body>
<jsp:useBean id="article" class="betaboutique.javabean.Article"
scope="request"/>
<h2>ARTICLE</h2><br/>
Nom : <jsp:getProperty name="article" property="nomarticle"/><br/>
Référence : <jsp:getProperty name="article" property="refarticle"/><br/>
Prix : <jsp:getProperty name="article" property="prixarticle"/><br/>
Acteur : <jsp:getProperty name="article" property="acteurs"/><br/>
Acteur :
<ul>
<c:forEach var="acteur" items="${article.acteurs}">
    <li><c:out value="${acteur}"/></li>
</c:forEach>
</ul>
<br/>
</body>
</html>
```

4. Sérialiser des JavaBeans

Les JavaBeans sont des standards et peuvent donc être sérialisés. La sérialisation (*sérialization* en anglais) permet d'encoder un objet (et son état) sous la forme d'une suite d'octets. Cette suite d'octets peut ensuite être utilisée pour sauvegarder l'objet (persistance) sur le disque, dans une base de données ou permettre son transport sur le réseau.

Il existe ensuite un processus symétrique qui permet de décoder la suite d'octets pour créer une copie conforme de l'objet d'origine. Cette activité est appelée désérialisation.

Dans de nombreux cas, il est nécessaire de sauvegarder l'état complet d'un objet. La solution qui vient immédiatement à l'esprit est de sauvegarder sous forme de chaînes de caractères chacun des attributs. Cette solution, s'avère rapidement très complexe pour les attributs de type tableau, collection ou même objet. La sauvegarde d'éléments de type *int* ou *String* par exemple ne pose pas de problème, mais par contre, la sauvegarde sous forme de chaînes de caractères d'éléments issus d'autres classes est complexe. De plus, la lecture par la suite de l'objet sérialisé en vue de la sauvegarde de l'objet est

très complexe à gérer. La solution offerte par Java est la sérialisation d'objets.

Ensuite, il sera possible de lire l'état complet de cet objet depuis un fichier binaire et de ré-instancier un objet dont l'état sera le même que celui de l'objet précédemment sauvegardé.

Pour rendre une classe sérialisable, c'est-à-dire pour que les objets de cette classe soient transformables en flux d'octets, il suffit de réaliser une implémentation de l'interface *Serializable* lors de la définition de cette classe. L'interface *Serializable* ne comprend aucune méthode, elle ne sert qu'à déclarer la capacité de sérialisation des objets issus de la classe.

Un objet est sérialisable à condition que tous ses composants soient eux-mêmes sérialisables. Les méta-éléments ainsi que les chaînes de caractères sont des éléments sérialisables. Par contre, les éléments qui sont déjà des flux d'octets ne sont pas sérialisables (images, threads, descripteurs de flux...). Si une classe sérialisable possède un attribut non sérialisable (une image par exemple), il faut alors déclarer cet attribut comme *transient*. La classe sera alors opérationnelle mais cet attribut ne sera pas pris en compte lors de la sérialisation.

---

a. Sérialiser un objet

Nous avons des classes sérialisables (JavaBean) dans notre projet, nous pouvons donc exploiter des flux d'objets sérialisés. Nous allons donc modifier notre Servlet de gestion d'articles afin d'enregistrer notre objet *article* dans un fichier binaire. La classe *ObjectOutputStream* permet la sérialisation d'objets vers un flux. L'écriture d'un objet est réalisée avec la méthode *writeObject(...)* et la purge du buffer avec la méthode *flush()*. La Servlet *ServletArticle* permet de stocker sous la forme d'un flux binaire l'objet *article* dans un fichier nommé *article.dat* présent à la racine du disque *C:*. Le nom du fichier binaire ainsi que son extension n'ont pas d'importance. Après exécution de la Servlet, nous pouvons observer la création d'un fichier *C:\article.dat* avec un flux d'octets.

```
package betaboutique.servlets.client;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.HashMap;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.beanutils.BeanUtils;
import betaboutique.javabean.Article;

@SuppressWarnings("serial")
public class ServletArticle extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //création d'un JavaBean article
        Article article=new Article();
        //liste des acteurs
        ArrayList<String> acteurs=new ArrayList<String>();

        //création d'une collection
        HashMap map = new HashMap();
        Enumeration names = request.getParameterNames();
        while (names.hasMoreElements())
        {
          //chaque paramètre de la requête est mis dans une collection
          String name = (String) names.nextElement();

          //premier acteur
          if(name.equals("acteur1"))
          {
              acteurs.add((String)request.getParameter(name));
          }
          //second acteur
          else  if(name.equals("acteur2"))
          {
          acteurs.add((String)request.getParameter(name));
          }
          //paramètre habituel
          else
          {
```

```
          map.put(name,request.getParameterValues(name));
          }
        }

        //fin de la boucle, mettre la liste des acteurs
dans le JavaBean map.put("acteurs",acteurs);

    try
        {
            //insérer toutes nos données dans le Bean
article en une seule étape
            BeanUtils.populate(article, map);

            try
            {
            //sérialiser notre objet article dans un
fichier binaire nommé article.dat
        ObjectOutputStream f=new ObjectOutputStream(new
FileOutputStream("C:\\article.dat"));
                //écriture
                f.writeObject(article);
                f.flush();
                f.close();
            }
            catch(Exception e)
            {
                System.out.println("Erreur dans la Servlet ServletArticle");
            }
        }
        catch(Exception e)
        {
            System.out.println("Erreur dans la Servlet ServletArticle");
        }
    }


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

b. Désérialiser un objet

La classe *ObjectInputStream* permet de lire un flux sérialisé en entrée et donc de désérialiser un objet depuis un flux. La désérialisation permet de lire un objet (ou plusieurs) et de créer de nouvelles instances de cet objet à partir de l'état de celui sauvegardé. La méthode *readObject()* retourne une référence de type *Object*. Il y aura donc juste à réaliser un transtypage (cast) de l'objet pour l'utiliser.

Nous allons créer une page JSP nommée *deserialisationarticle.jsp* qui permet de lire le JavaBean et de l'afficher en direct. Cette page sera appelée directement et sera fonctionnelle tant que l'objet sérialisé *article.dat* sera présent sur le disque de la machine.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ page import="java.io.ObjectInputStream" %>
<%@ page import="java.io.FileInputStream" %>
<%@ page import="betaboutique.javabean.Article" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>FICHE ARTICLE</title>
</head>
<body>
<%
//lecture de l'objet sérialisé
ObjectInputStream f = new ObjectInputStream(new
FileInputStream("C:\\article.dat"));
```

```
Article articleserialise = (Article) f.readObject();
%>
<!--  renseigner le JavaBean -->
<jsp:useBean id="article" class="betaboutique.javabean.Article">
    <jsp:setProperty name="article" property="nomarticle"
value="<%= articleserialise.getNomarticle() %>"/>
    <jsp:setProperty name="article" property="refarticle"
value="<%= articleserialise.getRefarticle() %>"/>
    <jsp:setProperty name="article" property="prixarticle"
value="<%= articleserialise.getPrixarticle() %>"/>
    <jsp:setProperty name="article" property="acteurs"
value="<%= articleserialise.getActeurs() %>"/>
</jsp:useBean>

<h2>ARTICLE</h2><br/>
Nom : <jsp:getProperty name="article" property="nomarticle"/><br/>
Référence : <jsp:getProperty name="article" property="refarticle"/><br/>
Prix : <jsp:getProperty name="article" property="prixarticle"/><br/>
Acteur : <jsp:getProperty name="article" property="acteurs"/><br/>
Acteur :
<ul>
<c:forEach var="acteur" items="${article.acteurs}">
    <li><c:out value="${acteur}"/></li>
</c:forEach>
</ul>
<br/>
</body>
</html>
```

Cette technique très intéressante permet des manipulations d'objets. Nous retrouvons bien notre objet d'origine avec ses attributs simples (chaînes de caractères) et complexes (collection). Cette méthode est un moyen efficace de réaliser de la persistance d'objet, mais du fait de sa lenteur lors de nombreux accès, ne permet pas de gérer la persistance complète d'une application.

c. Sérialisation et désérialisation en XML

La sérialisation précédente sous forme de flux d'octets est intéressante mais empêche la lecture du fichier avec d'autres langages que Java. Il sera donc obligatoire de désérialiser chaque objet pour pouvoir le relire. Pour répondre à ce problème, Java propose des classes qui permettent de réaliser la sérialisation et la désérialisation à partir de données XML.

Nous allons reprendre notre exemple précédent (Servlet et page JSP) pour réaliser la même étape avec un flux XML.

```
...

     try
     {
         //on insère toutes nos données dans le Bean article en une seule
étape
         BeanUtils.populate(article, map);

         try
         {
              //encoder l'objet en XML
                  XMLEncoder e = new XMLEncoder(new BufferedOutputStream
(new FileOutputStream("C:\\article.xml")));
              e.writeObject(article);
              e.close();
         }
         catch(Exception e)
         {
              System.out.println("Erreur dans la Servlet ServletArticle");
         }
     }
     catch(Exception e)
     {
        System.out.println("Erreur dans la Servlet ServletArticle");
     }
...
```

L'objet *article* est désormais sérialisé sous la forme d'un fichier XML nommé *article.xml* qui possède la structure suivante :

```
- <java version="1.6.0_05" class="java.beans.XMLDecoder">
  - <object class="betaboutique.javabean.Article">
    - <void property="acteurs">
      - <void method="add">
          <string>jo</string>
        </void>
      - <void method="add">
          <string>eastwo</string>
        </void>
      </void>
    - <void property="nomarticle">
        <string>le nom</string>
      </void>
    - <void property="prixarticle">
        <float>8.0</float>
      </void>
    - <void property="refarticle">
        <string>la référence </string>
      </void>
    </object>
  </java>
```

Du fait de cette sérialisation avec le standard XML, il sera possible d'exploiter l'objet avec n'importe quelle technologie compatible comme par exemple PHP, ASP, Flash ou JavaScript.

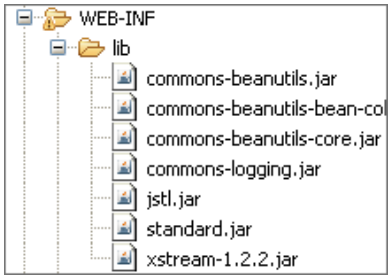
La page JSP suivante permet alors d'afficher l'objet sérialisé à partir du fichier XML.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ page import="java.beans.XMLDecoder" %>
<%@ page import="java.io.BufferedInputStream" %>
<%@ page import="java.io.FileInputStream" %>
<%@ page import="betaboutique.javabean.Article" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>FICHE ARTICLE</title>
</head>
<body>
<%
//lecture de l'objet sérialisé
XMLDecoder e = new XMLDecoder(new BufferedInputStream(new
FileInputStream("C:\\article.xml")));
Article articleserialise = (Article) e.readObject();
%>
<!--  renseigner le JavaBean -->
<jsp:useBean id="article" class="betaboutique.javabean.Article">
    <jsp:setProperty name="article" property="nomarticle"
value="<%= articleserialise.getNomarticle() %>"/>
    <jsp:setProperty name="article" property="refarticle"
value="<%= articleserialise.getRefarticle() %>"/>
    <jsp:setProperty name="article" property="prixarticle"
value="<%= articleserialise.getPrixarticle() %>"/>
    <jsp:setProperty name="article" property="acteurs"
value="<%= articleserialise.getActeurs() %>"/>
</jsp:useBean>

<h2>ARTICLE</h2><br/>
Nom : <jsp:getProperty name="article" property="nomarticle"/><br/>
Référence : <jsp:getProperty name="article" property="refarticle"/><br/>
Prix : <jsp:getProperty name="article" property="prixarticle"/><br/>
```

```
Acteur : <jsp:getProperty name="article" property="acteurs"/><br/>
Acteur :
<ul>
<c:forEach var="acteur" items="${article.acteurs}">
    <li><c:out value="${acteur}"/></li>
</c:forEach>
</ul>
<br/>
</body>
</html>
```

La technique précédente bien que très facile d'utilisation nécessite le passage par un fichier ''temporaire''. De plus, cette API est très lente avec de nombreuses connexions et n'est donc pas envisageable pour la persistance massive de données dans un environnement en production.

L'API XStream (http://xstream.codehaus.org/) simple d'utilisation, permet de sérialiser des objets Java, mais surtout, elle est très rapide et ne consomme pas beaucoup de mémoire. L'API XStream est disponible sous la forme d'une librairie *xstream-1.2.2.jar* que nous devons inclure à notre projet (*/WEB-INF/lib* et classpath).



Un des avantages de cette librairie est qu'elle permet de travailler avec des objets sérialisables, c'est-à-dire qui implémentent l'interface *Serializable* mais également avec des objets non sérialisables. Nous allons modifier notre Servlet *ServletArticle* afin d'afficher l'article dans la console Java sous la forme d'un flux XML.

```
...
    try
    {
       //on insère toutes nos données dans le Bean article en une seule étape
       BeanUtils.populate(article, map);

       try
       {
              //instanciation de la classe XStream
              XStream xstream = new XStream(new DomDriver());
              //conversion du contenu de l'objet article en XML
              String xml=xstream.toXML(article);
              //affichage de la conversion
              System.out.println("XML : "+xml);
       }
       catch(Exception e)
       {
             System.out.println("Erreur dans la Servlet ServletArticle");
          }
    }
    catch(Exception e)
    {
          System.out.println("Erreur dans la Servlet ServletArticle");
    }
  }
...
```

Le résultat de cette exécution produit le contenu XML ci-dessous. Nous remarquons deux éléments très important : la librairie a traité elle-même les caractères spéciaux en entité XML (les apostrophes du titre) et la structure XML est la même que celle de la classe. Les balises ont les mêmes noms et sont facilement manipulables.

Article

Nom :	l'homme de l'ombre
Référence :	référence
Prix :	12
Acteur 1:	malkowitch
Acteur 2:	eastwood
	Valider

```
XML : <betaboutique.javabean.Article>
  <nomarticle>l&apos;homme de l&apos;ombre</nomarticle>
  <refarticle>référence</refarticle>
  <prixarticle>12.0</prixarticle>
  <acteurs>
    <string>eastwood</string>
    <string>malkowitch</string>
  </acteurs>
</betaboutique.javabean.Article>
```

Nous pouvons reprendre notre exemple précédent et sérialiser l'objet *article* dans un fichier et le lire en le désérialisant avec la page *deserialiserarticle.jsp*.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ page import="com.thoughtworks.xstream.XStream" %>
<%@ page import="com.thoughtworks.xstream.io.xml.DomDriver" %>
<%@ page import="java.io.File" %>
<%@ page import="java.beans.XMLDecoder" %>
<%@ page import="java.io.BufferedInputStream" %>
<%@ page import="java.io.FileInputStream" %>
<%@ page import="betaboutique.javabean.Article" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head><meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>FICHE ARTICLE</title>
</head>
<body>
<%
//lecture de l'objet sérialisé
XStream xstream = new XStream(new DomDriver());
FileInputStream fis = new FileInputStream(new File
("C:\\article.xml"));
Article articleserialise =null;
try
{
    articleserialise = (Article) xstream.fromXML(fis);
}
catch(Exception e)
{

}
%>
<!--  renseigner le JavaBean -->
<jsp:useBean id="article" class="betaboutique.javabean.Article">
    <jsp:setProperty name="article" property="nomarticle"
```

```
value="<%= articleserialise.getNomarticle() %>"/>
    <jsp:setProperty name="article" property="refarticle"
value="<%= articleserialise.getRefarticle() %>"/>
    <jsp:setProperty name="article" property="prixarticle"
value="<%= articleserialise.getPrixarticle() %>"/>
    <jsp:setProperty name="article" property="acteurs"
value="<%= articleserialise.getActeurs() %>"/>
</jsp:useBean>


<h2>ARTICLE</h2><br/>
Nom : <jsp:getProperty name="article" property="nomarticle"/><br/>
Référence : <jsp:getProperty name="article" property="refarticle"/><br/>
Prix : <jsp:getProperty name="article" property="prixarticle"/><br/>
Acteur : <jsp:getProperty name="article" property="acteurs"/><br/>
Acteur :
<ul>
<c:forEach var="acteur" items="${article.acteurs}">
    <li><c:out value="${acteur}"/></li>
</c:forEach>
</ul>
<br/>
</body>
</html>
```

Nous allons modifier notre application pour mettre en œuvre un modèle à plusieurs étapes et facilement maintenable. Un formulaire de saisie permet d'insérer un nouvel article. Cette saisie est transformée en JavaBean. L'objet JavaBean est retourné à la page JSP qui permet de transformer simplement le JavaBean en flux XML. Ce principe pourrait être également envisagé pour une lecture d'informations dans une base de données. La page JSP affiche alors les données au format XML.

```
...
   try
   {
       //on insère toutes nos données dans le Bean article en une seule étape
       BeanUtils.populate(article, map);
       //stocker le JavaBean dans la requête
       request.setAttribute("article", article);
       //redirection sur la page d'affichage
       getServletContext().getRequestDispatcher("/article.jsp").forward
(request, response);
   }
   catch(Exception e)
   {
       System.out.println("Erreur dans la Servlet ServletArticle");
   }
...
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<%@ page language="java" contentType="text/xml;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="com.thoughtworks.xstream.XStream" %>
<%@ page import="com.thoughtworks.xstream.io.xml.DomDriver" %>
<%@ page import="betaboutique.javabean.Article" %>
<%
Article article=(Article)request.getAttribute("article");
if(article!=null)
{
    //instanciation d'un objet xstream
    XStream xstream = new XStream(new DomDriver());
    //afficher le flux XML
    out.println(xstream.toXML(article));
}
%>
```

La saisie du formulaire pour un article entraîne automatiquement une réponse au format XML. Cette technique peut être particulièrement utilisée pour afficher des listings d'articles, le panier en cours...

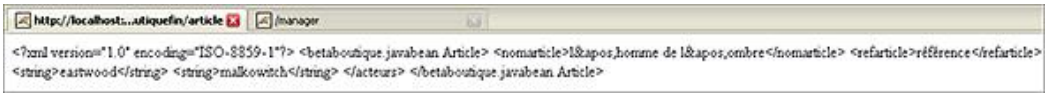
Il faut remarquer que la page JSP commence par le prologue XML et que le contenu de la page (attribut *contentType*) est en XML. Désormais cette page peut être parsée avec le langage Flash ou JavaScript. Nous allons afficher notre page avec une feuille XSLT pour présenter nos résultats dans un tableau.

Voici le contenu de la nouvelle page JSP qui permet de stocker le flux XML dans une variable et de le transformer en HTML pour l'affichage par l'intermédiaire de la feuille XSLT *articlexsl.xsl*.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="com.thoughtworks.xstream.XStream" %>
<%@ page import="com.thoughtworks.xstream.io.xml.DomDriver" %>
<%@ page import="betaboutique.javabean.Article" %>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/x.tld" prefix="x" %>
<%
String buffer=null;
Article article=(Article)request.getAttribute("article");
if(article!=null)
{
    //instanciation d'un objet xstream
    XStream xstream = new XStream(new DomDriver());
    //stocker le flux XML dans un buffer
    buffer=xstream.toXML(article);
}
%>
<!--  stocker le flux XML dans une variable pour
le traiter par la suite -->
<c:set var="xml">
    <?xml version="1.0" encoding="ISO-8859-1"?>
    <%= buffer %>
</c:set>
<c:out value="${xml}"/>
```

Cette première partie de code permet de stocker le flux XML dans une variable et de l'afficher. Dans une application professionnelle, nous pourrions utiliser un paramètre pour savoir si nous voulons en retour la transformation ou le contenu XML d'origine. Cette étape serait également nécessaire pour le développement de la feuille XSLT afin de bien analyser la structure générée par XStream pour les données (nom de la racine, nom des nœuds, nom des attributs...).

Le résultat présenté dans le navigateur est le suivant :



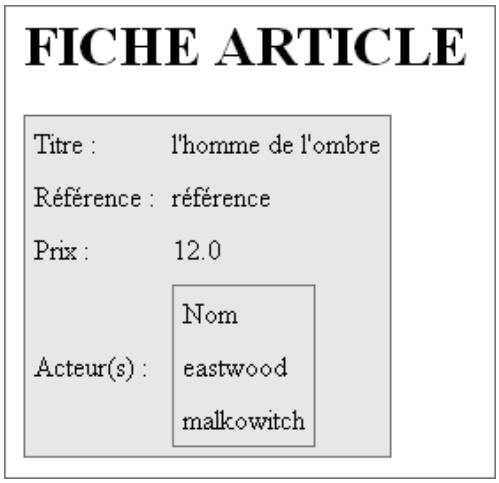
Maintenant, nous allons déclencher la transformation de ce flux XML en HTML grâce à notre feuille de style XSLT.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- sortie -->
<xsl:output method="html" indent="yes"
encoding="ISO-8859-1"/>
<xsl:template match="/">
<html><head><title>BETABOUTIQUE</title></head>
<body>
    <h1>FICHE ARTICLE</h1>
    <table border="0" cellspacing="0" cellpadding="5"
style="border-style:solid;border-width:1px;background-
color:#eaeaea">
    <tr><td>Titre : </td><td><xsl:value-of select=
"/betaboutique.javabean.Article/nomarticle"/></td></tr>
    <tr><td>Référence : </td><td><xsl:value-of select="
/betaboutique.javabean.Article/refarticle"/></td></tr>
         <tr><td>Prix : </td><td><xsl:value-of select=
"/betaboutique.javabean.Article/prixarticle"/></td></tr>
    <tr><td>Acteur(s) : </td>
    <td>
        <table border="0" cellspacing="0" cellpadding="5"
style="border-style:solid;border-width:1px">
        tr><td>Nom</td></tr>
        xsl:for-each select="/betaboutique.javabean.Article/
acteurs/string">
        <tr>
            <xsl:if test="position() mod 2=1">
            <xsl:attribute name="bgcolor">#eaeaea</xsl:attribute>
            </xsl:if>
            <td><xsl:value-of select="."/></td>
        </tr>
        </xsl:for-each>
        </table>
```

```
        </td>
        </tr>
    </table>
    </body>
    </html>
</xsl:template>
</xsl:stylesheet>
```

La page JSP de transformation du contenu XML en contenu HTML est la suivante :

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="com.thoughtworks.xstream.XStream" %>
<%@ page import="com.thoughtworks.xstream.io.xml.DomDriver" %>
<%@ page import="betaboutique.javabean.Article" %>
<%@ taglib uri="/WEB-INF/tld/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/x.tld" prefix="x" %>
<%
String buffer=null;
Article article=(Article)request.getAttribute("article");
if(article!=null)
{
    //instanciation d'un objet xstream
    XStream xstream = new XStream(new DomDriver());
    //stocker le flux XML dans un buffer
    buffer=xstream.toXML(article);
}
%>
<!--  stocker le flux XML dans une variable pour le traiter par la suite -->
<c:set var="xml">
    <?xml version="1.0" encoding="ISO-8859-1"?>
    <%= buffer %>
</c:set>
<!--  transformer le contenu XML en HTML par le biais de la feuille XSLT -->
<c:import url="/articlexsl.xsl" var="xsl"/>
<x:transform xml="${xml}" xslt="${xsl}"/>
```



Les données sont correctement affichées avec les bonnes entités HTML, la structure conservée et les éléments composés (comme les acteurs) sont également bien traités. Grâce à cette technique, nous avons totalement séparé la partie Données de la partie Mise en page. Ce type de programmation permet une meilleure maintenance et surtout de manipuler les données avec différents langages selon les besoins du commanditaire (Java, PHP, Flash ou JavaScript).

Pour résumer, la technologie JavaBean est un standard de programmation, elle permet donc d'utiliser des librairies très puissantes de manipulation comme la sérialisation, la transformation d'objets en XML, les transformations en PDF, en VRML... Cette technologie permet de générer des pages 100% compatibles XML afin de les parser avec une (ou plusieurs) feuille(s) XSLT, du code JavaScript ou une application graphique Java évoluée.

Transfert de contrôle

1. Présentation

Il est très fréquent lors de développements d'applications, d'échanger des informations entre une Servlet et une page JSP puis de transférer le contrôle de la JSP vers une Servlet. Nous allons introduire un exemple de chaque contrôle afin de réaliser des redirections, des appels de pages... Cette technique a été présentée tout au long des exemples, mais il est important de bien comprendre son fonctionnement.

Nous allons utiliser l'exemple de l'authentification avec la Servlet *ServletAuthentification* pour présenter les différents types de redirections.

2. Utilisation

a. Transfert du contrôle d'une Servlet à une page JSP

Dans un premier temps, nous réalisons une simple redirection entre une page HTML ou JSP et une Servlet. L'utilisateur réalise une authentification (*authentification.html*), la Servlet (*ServletAuthentification*) vérifie l'intégrité des données et appelle une page de bienvenue (*bienvenue.jsp*).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Authentification BetaBoutique</title>
</head>
<body>
<h1>Authentification - Client</h1>
<form action="authentificationclient" method="POST">
<table border="1" cellspacing="0" cellpadding="5">
<tr>
    <td>Identifiant/Login : </td>
    <td><input type="text" name="identifiant" id="identifiant"
value="" size="20"/></td>

</tr>
<tr>
    <td>Mot de passe : </td>
    <td><input type="text" name="motdepasse" id="motdepasse"
value="" size="20"/></td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="submit"
name="valider" id="valider" value="Valider"/></td>
</tr>
</table>
</form>
</body>
</html>
```

```
...
//vérifier l'égalité des valeurs
if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
{
    if(urlBienvenue==null)
    {
        throw new ServletException("Le paramètre
[urlBienvenue] n'a pas été initialisé");
    }
    else
    {
```

```
      //créer un JavaBean client
      Client client=new Client();
      client.setIdentifiant(identifiant);
      client.setMotdepasse(motdepasse);
      //insérer le JavaBean dans la requête
      request.setAttribute("client", client);
      //redirection simple
      response.sendRedirect("bienvenue.jsp");
    }
}
...
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>
</body>
</html>
```

Dans cet exemple, la page *bienvenue.jsp* ne peut pas afficher le JavaBean *client*. En effet, l'action *response.sendRedirect("bienvenue.jsp")* réalise une redirection forcée et perd les données présentes dans la requête. Cette technique permet de vider, la totalité de la requête et donc d'éviter par exemple de réaliser plusieurs insertions d'un même article dans un panier en cas de rafraîchissement de la page par l'utilisateur.

Avec cette méthode, il est quand même possible de forcer le passage de paramètre uniquement en GET avec les appels HTTP. Voici un second exemple avec le passage d'un message dans la redirection forcée.

```
...
//vérifier l'égalité des valeurs
if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
{
    if(urlBienvenue==null)
    {
        throw new ServletException("Le paramètre
[urlBienvenue] n'a pas été initialisé");
    }
    else
    {
        //créer un JavaBean client
        client=new Client();
        client.setIdentifiant(identifiant);
        client.setMotdepasse(motdepasse);
        //insérer le JavaBean dans la requête
        request.setAttribute("client", client);
        //redirection simple
        response.sendRedirect("bienvenue.jsp?message=bienvenue");
    }
}
...
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/betaboutique.tld" prefix="bb" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="session"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>
<% out.println("Message : "+request.getParameter("message"))
; %>
</body>
</html>
```

Pour réaliser des transferts de contrôle entre une Servlet et une page JSP sans perdre les données, nous devons utiliser le *RequestDispatcher*. Cette classe permet de positionner des attributs dans la requête (*session*, *context*...) et de les lire dans une page JSP appelée.

La Servlet précédente est modifiée avec l'utilisation de la classe *RequestDispatcher*.

```
...
//vérifier l'égalité des valeurs
if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            if(urlBienvenue==null)
            {
                throw new ServletException("Le paramètre
[urlBienvenue] n'a pas été initialisé");
            }
            else
            {
                //créer un JavaBean client
                Client client=new Client();
                client.setIdentifiant(identifiant);
                client.setMotdepasse(motdepasse);
                //insérer le JavaBean dans la requête
                request.setAttribute("client", client);

    //redirection simple
    getServletContext().getRequestDispatcher("/bienvenue.jsp?
message=bienvenue").forward(request, response);
            }
        }
...
```

Il est possible d'améliorer l'exemple précédent en passant le paramètre *message* dans la requête et non dans le nom de la page. Par contre, dans ce cas, il faut modifier la page *bienvenue.jsp* afin d'utiliser la méthode *getAttribute(...)* et non *getParameter(...)*.

```
...
//vérifier l'égalité des valeurs
if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            if(urlBienvenue==null)
            {
                throw new ServletException("Le paramètre
[urlBienvenue] n'a pas été initialisé");
            }
            else
            {
                //créer un JavaBean client
                Client client=new Client();
                client.setIdentifiant(identifiant);
                client.setMotdepasse(motdepasse);
                //insérer le JavaBean dans la requête
                request.setAttribute("client", client);
    //insérer le message dans la requête
    request.setAttribute("message", "bienvenue sur le site");
```

```
    //redirection simple
    getServletContext().getRequestDispatcher
("/bienvenue.jsp").forward(request, response);
            }
        }
...
```

b. Transfert du contrôle d'une page JSP à une Servlet

Lors du développement d'une application, il est très fréquent qu'une page JSP transfère le contrôle à une Servlet avec des informations associées.

Pour réaliser cette opération, il faut utiliser la directive *<jsp:forward page="nompage"/>*. Cette directive permet également de passer des paramètres à la Servlet avec la directive *<jsp:param .../>*. Par exemple, dans une page JSP qui permet de faire un calcul ou de charger un fichier, après le bloc de traitement, l'utilisateur est automatiquement redirigé vers une Servlet d'insertion dans une base de données avec le code suivant :

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<jsp:forward page="ServletInsertionImage">
    <jsp:param name="insertion" value="OK"/>
</jsp:forward>
```

Bien que cette technique soit très simple d'utilisation, elle n'est pas très souvent utilisée étant donné que dans la plupart des cas c'est l'utilisateur lui-même qui déclenche, par l'intermédiaire d'un lien ou d'un formulaire HTML, la Servlet appelée par la page JSP.

c. Transfert du contrôle d'une page JSP à une autre page JSP

Une page JSP peut donner le contrôle à une autre page JSP en utilisant deux méthodes :

- En utilisant la directive *<jsp:forward.../>* de la première page vers la seconde.
- En utilisant le bouton d'un formulaire avec les données associées, ou un lien avec paramètres.

Pour illustrer ces techniques, nous allons de nouveau utiliser le formulaire d'authentification qui cette fois-ci va appeler directement la page JSP *bienvenueformulaire.jsp*. Nous utilisons donc un formulaire avec passage de paramètre en POST et un lien avec passage de paramètre en GET. Nous donnons en même temps, l'utilisation des différents accès en fonction de la méthode HTTP utilisée (*request.getAttribute(...)* en POST et *request.getParameter(...)* en GET).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Authentification BetaBoutique</title>
</head>
<body>
<h1>Authentification - Client</h1>
<form action="bienvenueformulaire.jsp" method="POST">
<table border="1" cellspacing="0" cellpadding="5">
<tr>
    <td>Identifiant/Login : </td>
    <td><input type="text" name="identifiant" id="identifiant"
value="" size="20"/></td>
</tr>
<tr>
    <td>Mot de passe : </td>
    <td><input type="text" name="motdepasse" id="motdepasse"
value="" size="20"/></td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="submit"
name="valider" id="valider" value="Valider"/></td>
</tr>
```

```
</table>
<br/>
<a href="bienvenueformulaire.jsp?identifiant=monidentifiant&
motdepasse=monmotdepasse">Passage du controle à une
autre page JSP avec paramètres</a>
</form>
</body>
</html>
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="betaboutique.javabean.Client" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>

<!--  affichage des paramètres en POST -->
<jsp:useBean id="client" class="betaboutique.javabean.Client"
scope="request"/>
<jsp:setProperty name="client" property="*"/>
<h2>Bienvenue client : <jsp:getProperty name="client"
property="identifiant"/> <jsp:getProperty name="client"
property="motdepasse"/></h2>

<!--  affichage des paramètres en GET -->
<%
out.println("Identifiant : "+request.getParameter("identifiant")+"<br/>");
out.println("Mot de passe : "+request.getParameter("motdepasse")+"<br/>");
%>
</body>
</html>
```

Travailler avec des fichiers et répertoires

1. Présentation

L'API Java est extrêmement complète en ce qui concerne la manipulation de répertoires et fichiers. Les pages JSP sont ainsi très souvent utilisées pour créer un fichier, lire le contenu d'un fichier, réaliser une copie d'un fichier image...

Nous allons étudier les différentes techniques, classes et méthodes proposées par Java pour gérer les fichiers et répertoires.

2. Travailler avec les répertoires

Nous allons créer une page JSP nommée *repertoire.jsp* avec un ensemble de commandes pour gérer les répertoires en Java. Il est nécessaire de vérifier l'existence d'un répertoire avant toute opération le concernant. Pour vérifier l'existence d'un répertoire, nous utilisons un objet issu de la classe *File* située dans le paquetage *java.io*.

Cette bibliothèque est très complète pour manipuler des fichiers et répertoires (existence, gestion des droits ou renommage).

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="java.io.File" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>REPERTOIRE</title>
</head>
<body>
<%
//nom du fichier présent sur le disque dur
String nomFichier="C:"+java.io.File.separatorChar+"article.xml";
//instancier un objet file
File file=new File(nomFichier);

//présence du fichier sur le disque
if(file.exists() && file.isFile())
{
    out.println("Le fichier "+nomFichier+" existe sur le système<br/>");
}
else
{
    out.println("Le fichier "+nomFichier+" n'existe pas
sur le système<br/>");}

//gestion des droits
if(file.canRead())
{
    out.println("Le fichier "+nomFichier+" est accessible en lecture<br/>");
}
if(file.canWrite())
{
    out.println("Le fichier "+nomFichier+" est accessible en écriture<br/>");
}

//propriétés
out.println("Le chemin absolu est : "+file.getAbsoluteFile()+"<br/>");
out.println("Le chemin absolu est : "+file.getAbsolutePath()+"<br/>");
out.println("Le nom du fichier est : "+file.getName()+"<br/>");
out.println("Sa date de dernière modification est :
"+file.lastModified()+"<br/>");
out.println("Sa taille est : "+file.length()+" octets<br/>");
%>
```

```
</body>
</html>
```

Cet exemple nous confirme que Java est bien multi plate-forme. En effet, pour la lecture du fichier *article.xml*, l'instruction *java.io.File.separatorChar* permet de mettre le chemin dans le bon sens : *C:\article.xml* sous Windows et */article.xml* sous Linux.

Voici un deuxième exemple qui permet d'afficher l'arborescence des racines de fichiers et le parcours d'un répertoire.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="java.io.File" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>REPERTOIRE</title>
</head>
<body>
<%
//nom du fichier présent sur le disque dur
String nomFichier="C:"+java.io.File.separatorChar+"unzipped";
//instancier un fichier
File fichier=new File(nomFichier);
//listing du répertoire
File[] listing=fichier.listFiles();

//afficher l'arborescence
out.println("<ul>");
for(int i=0;i<listing.length;i++)
{
    out.println("<li>"+listing[i]+"</li>");
}
out.println("</ul>");


//créer un répertoire nommé essai dans nomFichier
File nouveauRepertoire=new File(nomFichier+java.io.File.
separatorChar+"essai");
nouveauRepertoire.mkdir();


//afficher la liste des racines
File[] listeRacine=File.listRoots();
out.println("<ul>");
for(int i=0;i<listeRacine.length;i++)
{
    out.println("<li>"+listeRacine[i]+"</li>");
}
out.println("</ul>");
%>
</body>
</html>
```

3. Travailler avec les fichiers

Nous allons créer une page JSP nommée *fichier.jsp* avec un ensemble de commandes pour gérer les fichiers en Java. Les opérations de base seront utilisées, avec l'écriture de données et la lecture d'informations.

La première étape consiste à créer un objet de type *File*, qui servira à spécifier le chemin et le nom du fichier que nous souhaitons lire. Ensuite, un objet de type *FileReader* utilisant l'objet de type *File*, doit être créé.

La lecture des informations en provenance d'un fichier sera meilleure si les informations sont stockées au fur et à mesure dans un tampon. La méthode *readLine()* de l'objet *BufferedReader* permet de lire une ligne dans un fichier avec le caractère de rupture indiquant une nouvelle ligne.

```
<%@ page language="java" contentType="text/html;
```

```
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="java.io.File" %>
<%@ page import="java.io.FileReader" %>
<%@ page import="java.io.BufferedReader" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>FICHIER</title>
</head>
<body>
<%
//nom du fichier présent sur le disque dur
String nomFichier="C:"+java.io.File.separatorChar+"article.xml";
//instancier un objet file
File file=new File(nomFichier);
//instancier un file reader
FileReader filereader=new FileReader(file);
//instancier un buffer
BufferedReader buffreader=new BufferedReader(filereader);

//lecture du contenu du fichier
String ligne=null;
while((ligne=buffreader.readLine())!=null)
{
    out.println("Ligne  : "+ligne+"<br/>");
}
%>
</body>
</html>
```

L'API Java possède de très nombreuses classes et méthodes pour la manipulation des fichiers, avec toujours le souci de respecter la portabilité. Il sera facile par exemple de connaître la liste des racines de l'arborescence, parcourir des répertoires, supprimer un fichier ou répertoire, renommer ou déplacer un fichier, créer un fichier temporaire...

4. En résumé

Ce chapitre a présenté le mécanisme de JSP en Java. La première partie a introduit le principe des JSP et le cycle de vie d'une page d'exemple. Dans un second temps, les déclarations, commentaires et scriptlets ont été expliqués à partir des pages du projet *BetaBoutique*. Le chapitre suivant a présenté les objets implicites JSP qui sont utilisables directement sans instanciation ni importation. La partie associée a permis de mettre en œuvre une première JSP simple et de présenter la notion de fragment JSPF. Ensuite, la gestion des exceptions et des erreurs en JSP a été largement détaillée dans un chapitre dédié.

Le chapitre suivant a présenté la notion de bibliothèques de tags ou taglibs Java. Ces bibliothèques permettent de réaliser des opérations de programmation, de manipuler des objets simples, de gérer des données XML, de manipuler les langues... Ces bibliothèques très utiles, sont parfois limitées et nécessitent donc le développement de balises personnalisées, comme cela a été expliqué au chapitre suivant. Ces balises personnalisées sont basées sur des classes Java et des fichiers de descriptions au format XML.

Les JavaBeans ont été ensuite présentés avec des exemples du projet *BetaBoutique*. Des manipulations simples ont été d'abord réalisées et ont été agrémentées de manipulations plus complexes à partir de flux d'octets et XML. Les techniques de sérialisation et désérialisation sous formes de données brutes ou XML ont aussi été détaillées à partir d'exemples du projet.

L'avant-dernier chapitre a permis de lister et détailler le transfert de contrôle en Java EE, qui est un élément important de la programmation MVC. Enfin, le dernier chapitre a été une introduction à la manipulation de fichiers et répertoires en JSP/Java.

Qu'est-ce qu'une Servlet ?

1. Présentation

Les Servlets sont la base de la programmation Java EE. La conception d'un site Web dynamique en Java repose sur ces éléments. Une Servlet est un composant Web conçu à partir d'une classe Java qui est déployée au sein d'une application. Les Servlets sont chronologiquement le deuxième élément de Java EE après JDBC.

Une Servlet interagit avec un client Web par l'intermédiaire du protocole HTTP via le mécanisme de requête/réponse. Bien que la totalité du traitement puisse être effectué dans la Servlet, celle-ci fait souvent appel à des classes utilitaires pour la logique métier. Elles apportent un contenu dynamique en réponse à des requêtes client.

En général, une Servlet n'est pas appelée directement, elle ne l'est qu'à travers une URL. Par exemple, la Servlet *ServletMonApplication.java* (compilée en *ServletMonApplication.class*) est appelée lorsque la page *monapplication.htm* est invoquée. La page *monapplication.htm* n'existe pas sur le serveur elle ne sert qu'à faire le lien avec la Servlet.

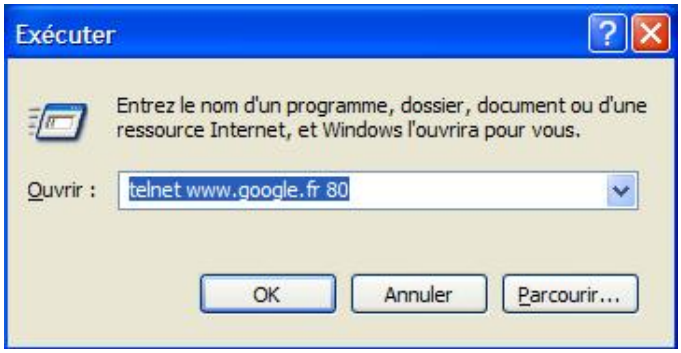
Les Servlets ont de nombreux avantages :

- Elles sont portables et évolutives.
- Elles sont performantes et rapides car chargées en mémoire dès le premier appel.
- Elles sont disponibles car elles s'exécutent dans un contexte multitâche (une seule instance créée).

Bien que les Servlets aient été conçues pour travailler avec tous les types de serveurs (HTTP, FTP, SMTP...) elles ne sont employées en pratique qu'avec des serveurs Web HTTP. L'API Servlet contient pour cela une classe nommée `HttpServlet` afin de gérer le protocole HTTP et les méthodes GET et POST .

2. Requêtes HTTP

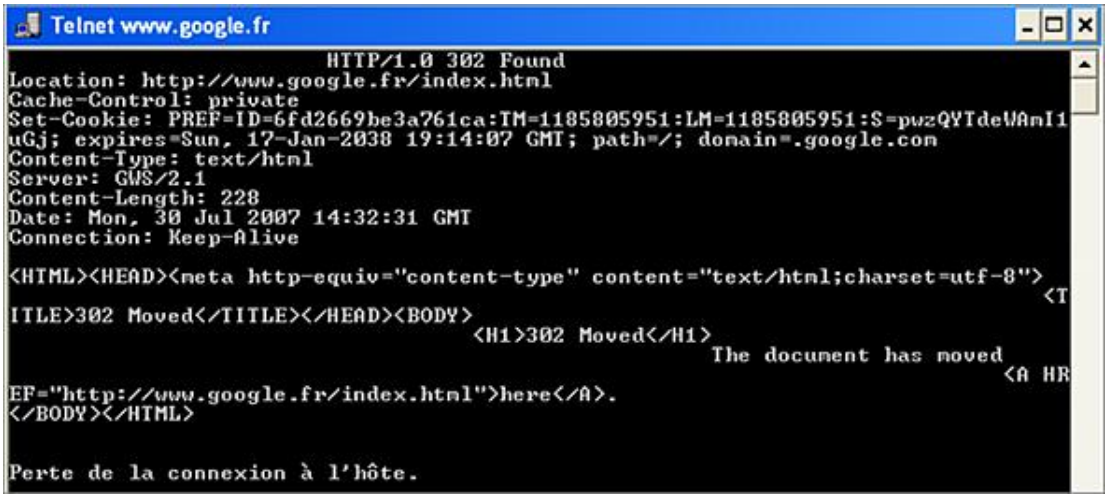
Pour comprendre le fonctionnement de l'exécution de Servlets, il est nécessaire d'expliquer au préalable le protocole HTTP. Sous Windows ou Linux, un client Telnet peut être lancé avec la commande suivante :



Ensuite, une commande HTTP peut être lancée pour récupérer le contenu de la page d'accueil.

```
GET /index.html HTTP/1.0
```

La réponse est alors la suivante :



La ligne de commande Telnet comporte le nom du programme Telnet suivi du nom de l'ordinateur hôte et le port. La requête est composée de son type (GET) suivi de l'URI relative à la ressource concernée puis de l'identificateur HTTP et du numéro de version supporté par le programme Telnet.

Une première pression sur la touche [Entrée] termine la requête et une seconde indique au serveur que l'en-tête est terminé. La première ligne de la réponse correspond à la ligne d'état. Elle indique la version HTTP utilisée, le code de la réponse et le message. Ensuite viennent les en-têtes de réponse : chemin, utilisation du cache, le cookie utilisé pour la session, le type de contenu, la longueur de la réponse, la date...

Lorsqu'un logiciel/programme Web est utilisé comme un navigateur, c'est le navigateur lui-même qui génère ces requêtes quand des pages Internet sont demandées. L'utilisation du protocole HTTP est alors transparente pour le client. Dans notre exemple, nous remarquons que la page : http://www.google.fr/index.html n'existe pas et que le serveur renvoie une page d'erreur adaptée en conséquence. Lorsqu'une requête est envoyée en utilisant la méthode POST, elle peut comporter un corps que le client va transmettre à la ressource demandée. Le plus souvent les requêtes POST sont envoyées par des formulaires affichés dans les navigateurs Web. La méthode GET peut envoyer des données dans l'URL alors que la méthode POST utilise le corps de la requête pour placer les données.

Voici un exemple de méthode POST avec deux paramètres (identifiant et mot de passe) :

```
POST /servlet/personnes HTTP/1.0
Content-type: application/x-www-form-urlencoding
Content-length : 39
identifiant=monidentifiant&motdepasse=monmotdepasse
```

a. Comment le serveur va répondre aux requêtes des clients ?

Avec une requête et la méthode GET, le serveur localise la ressource correspondante à l'URI indiquée et retourne cette ressource dans le corps du message HTTP envoyé au navigateur client. Le navigateur du client analyse celui-ci et affiche la page. La ressource demandée peut être un programme. Dans ce cas, le serveur doit décoder l'URI et appeler le programme spécifique. Le programme peut être écrit en C, Perl...

Ce type de programme appelé CGI est exécuté dans un processus différent de celui du serveur et nécessite la création d'un nouveau programme en cours d'exécution (thread/processus) à chaque appel. Les Servlets Java offrent de nombreux avantages sur ces programmes CGI. Elles peuvent être exécutées dans le même processus que le serveur ce qui économise énormément de ressources, elles sont donc beaucoup plus rapides et sont portables. En effet, les programmes écrits en C doivent être recompilés pour chaque système d'exploitation utilisé.

Le projet BetaBoutique

1. Présentation

Tout au long du guide, nous allons développer un projet de boutique en ligne afin de tester et de mettre en application les parties théoriques et les exemples.

Le développement d'une boutique permet :

- de gérer des utilisateurs/personnes pour les clients, les administrateurs (création, modification, suppression, authentification) ;
- de gérer des articles/produits (création, modification, suppression, tri) ;
- de gérer un panier dynamique (création, modification, suppression, session, listes...) ;
- de gérer des commandes et réservations (transactions, mails, tri...) ;
- de gérer des langues (articles en plusieurs langues) ;
- de manipuler le stock d'articles/produits avec un client lourd (application JWS).

Dans un premier temps, le diagramme des cas d'utilisation de la boutique sera présenté et nous utiliserons ensuite le service Personne/Clients pour développer les premières Servlets.

L'entreprise fictive *BetaBoutique* vend des articles en rapport avec le cinéma, principalement des DVD. Elle exerce son métier avec des fiches papier, des commandes par fax et par chèque bancaire envoyé par courrier postal puis envoie la commande au client. Dès que le chèque est encaissé, elle utilise les services de La Poste pour expédier les colis. La société *BetaBoutique* n'arrive plus à gérer manuellement son expansion et souhaite utiliser un système d'Information pour lui permettre de répondre à cette croissance. Elle attend plusieurs services du Système d'Information comme la vente en ligne, la gestion du catalogue d'articles (essentiellement des DVD) et la base de données des clients.

2. Expression des besoins

Pour modéliser l'expression des besoins de la société *BetaBoutique*, nous allons utiliser UML (*Unified Modeling Language*). Dans un premier temps, le diagramme des cas d'utilisation qui permet de représenter les fonctionnalités du système du point de vue utilisateur sera présenté.

Le diagramme des cas d'utilisation se compose :

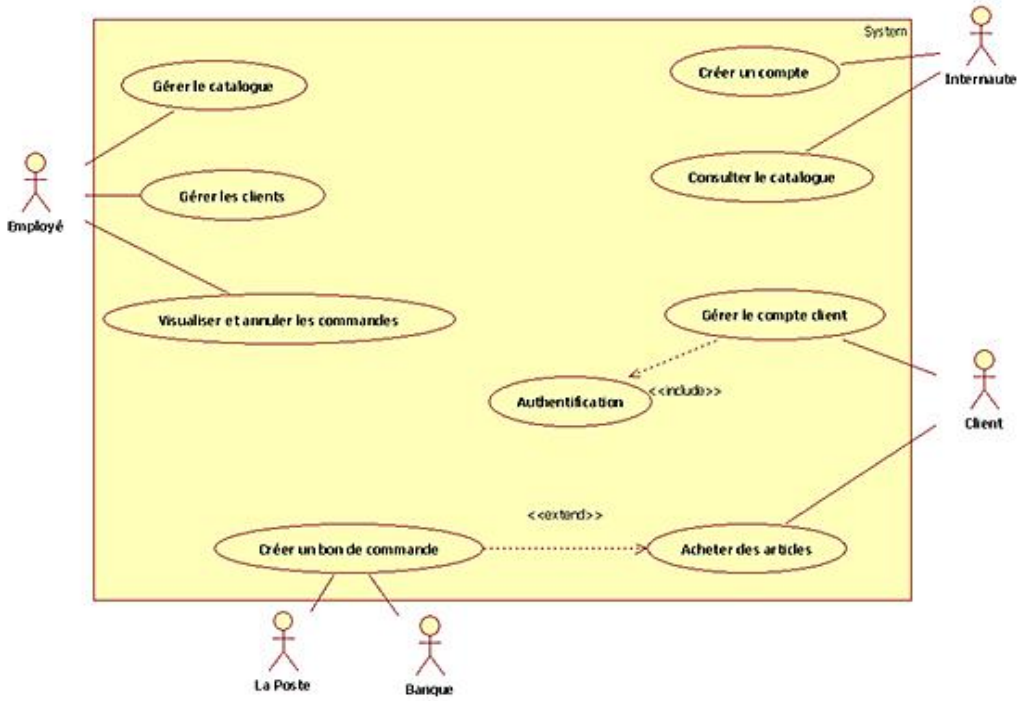
- d'acteurs (entités externes humaines ou robot/matériel qui utilisent le système) ;
- de cas d'utilisation (fonctionnalités proposées par le système).

Les acteurs utilisant le système sont :

- **Employé** : les employés mettent à jour le catalogue d'articles et vérifient les commandes et la liste des clients.
- **Internaute** : les personnes visitant le site peuvent consulter le catalogue.
- **Client** : les clients peuvent visualiser le catalogue, gérer leurs coordonnées et acheter des articles en ligne.

Il est également possible de mentionner comme systèmes externes La Poste qui permet de gérer les livraisons et le système bancaire pour l'encaissement des achats par carte.

Dans un diagramme des cas d'utilisation, le rectangle qui englobe les cas représente le système étudié. Les acteurs sont représentés par une icône appelée stick man et les cas d'utilisation sont représentés par une forme ovale. Les cas d'utilisation concernent les besoins des utilisateurs. Les cas d'utilisation sont donc très souvent réalisés sur la base d'interviews et d'entretiens avec le personnel.



Les acteurs "La Poste" et "Banque" représentent respectivement le service de livraison par la poste et le paiement en ligne, associés au bon de commande (ex : n° de colissimo inséré).

Le diagramme des cas d'utilisation peut être lu comme ceci : Un employé peut gérer les articles du catalogue, administrer les clients, visualiser les commandes.

Un internaute peut créer un compte et consulter le catalogue. Un client peut s'authentifier pour accéder à son compte. Il peut gérer son compte et acheter des articles après avoir complété son panier pour créer un bon de commande. Les cas présentés dans le diagramme peuvent être accompagnés d'un texte explicatif avec le nom du cas d'utilisation, un résumé du service, les acteurs impliqués...

3. Maquettes de la plate-forme

Les maquettes d'écran facilitent la compréhension des cas d'utilisation. Les utilisateurs se repèrent facilement grâce à ces schémas visuels et peuvent ainsi valider les choix d'analyse.

Les Internautes et les clients visualisent le contenu du catalogue à partir d'un navigateur. Sur la colonne de gauche qui représente le menu, sont affichées les catégories d'articles vendus par la société *BetaBoutique*. En cliquant sur une catégorie, le visiteur est redirigé vers une page qui affiche tous les produits de la catégorie sélectionnée.

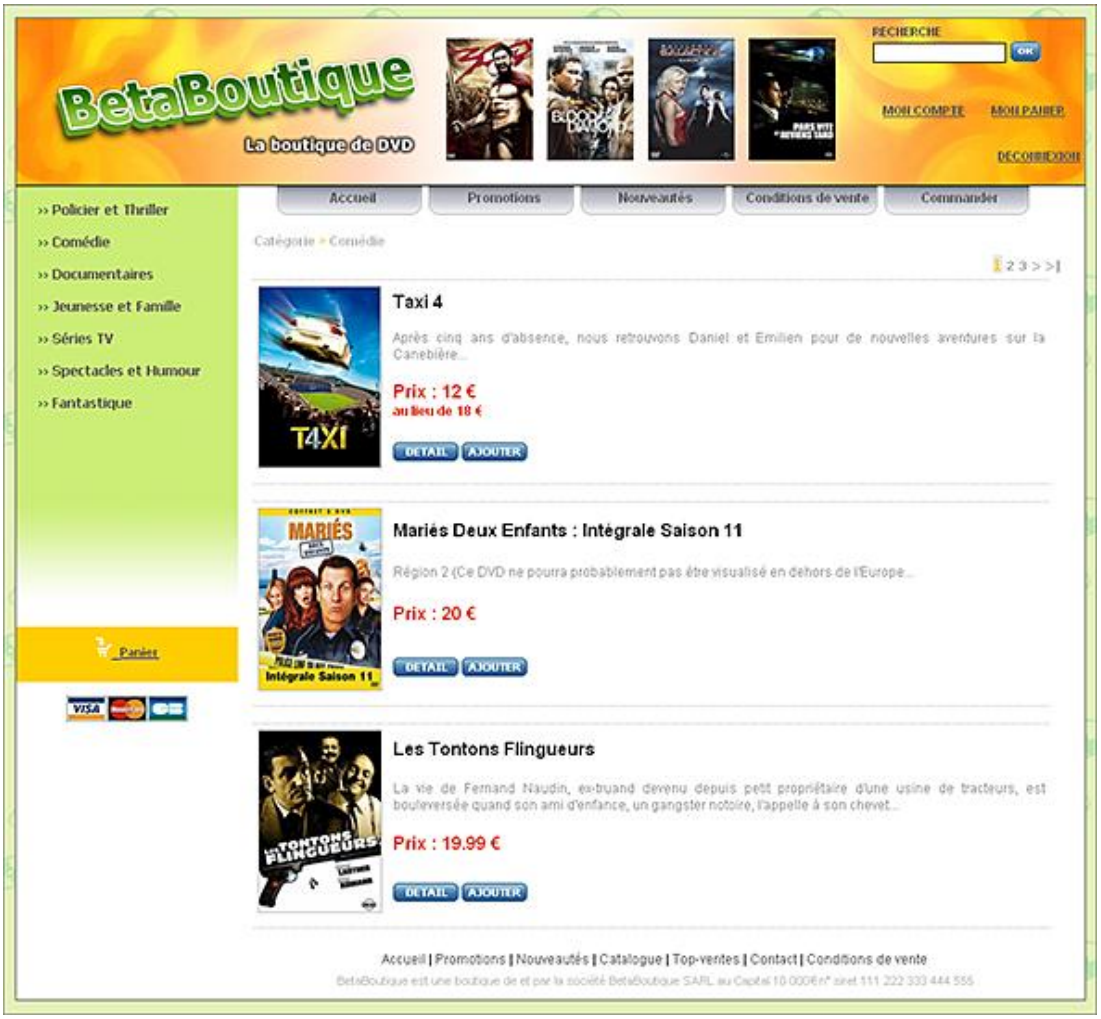
a. Découpage utilisé

La boutique *BetaBoutique* sera découpée selon le modèle suivant avec un en-tête, un menu de navigation et un pied de page. Les fichiers *.jspf* sont des pages JSP qui sont utilisées pour définir des fragments de pages et qui sont incluses dans d'autres pages (comme pour les menus, en-têtes...).

Les fichiers *.jsp* sont utilisés pour des fichiers sources ''complets'' et les fichiers *.jspf* sont utilisés pour des fragments ou segments de fichiers sources (menu, formulaire de recherche...).

b. Catalogue des articles

La page de catalogue des articles, permet d'afficher la liste paginée des produits par catégorie. Un résultat paginé est un découpage des réponses par page (exemple 5 résultats par page et 3 pages pour afficher 15 articles).



c. Fiche article

La fiche article permet d'afficher le détail du produit avec le nom, la note, la description, le prix et un bouton pour ajouter cet article au panier.

d. Rechercher un article

La zone de saisie présente en haut à droite du site permet de rechercher un article à partir de son nom ou de sa description. Les visiteurs peuvent ainsi rechercher des articles à partir d'une chaîne de caractères. La recherche ne tient pas compte de la casse (Majuscules/Minuscules). Si aucun article ne correspond à la recherche, un message informatif est affiché en conséquence.

e. Authentification

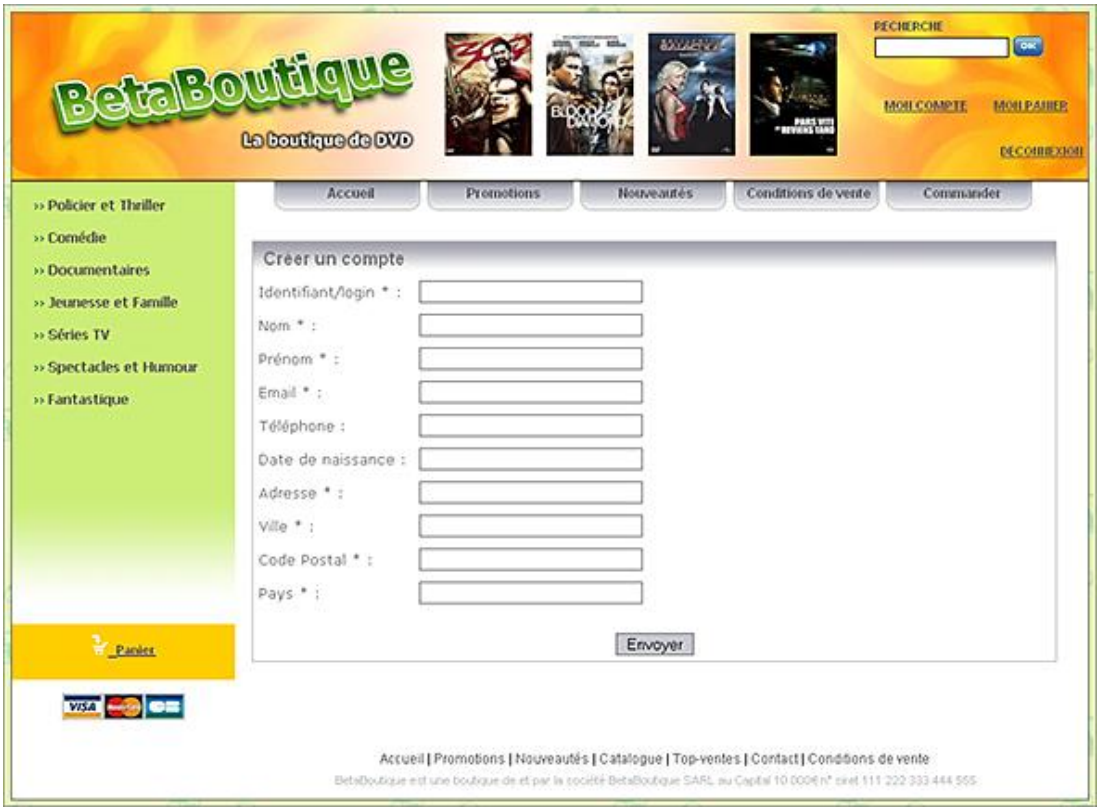
Cette page permet au client de se connecter et se déconnecter du système. Le client doit avoir créé un compte auparavant. Il saisit alors son identifiant et son mot de passe. Si l'authentification est correcte, l'utilisateur est redirigé sur la page d'accueil et un message d'invitation est alors affiché sur la totalité des pages du site jusqu'à la prochaine déconnexion. Ce message permet d'afficher l'identifiant de l'utilisateur actuellement connecté. Lorsque le client se déconnecte, il redevient Internaute/visiteur simple.

f. Créer un compte

Cette page permet au visiteur de se créer un compte dans le système et de devenir ainsi un client. Pour créer un compte, l'Internaute doit saisir un identifiant et un mot de passe avec une confirmation. L'identifiant doit être unique dans le système. Si ce n'est pas le cas, l'Internaute doit être averti et doit recommencer sa saisie avec un autre identifiant.

Les deux mots de passe (mot de passe et sa confirmation) doivent être identiques.

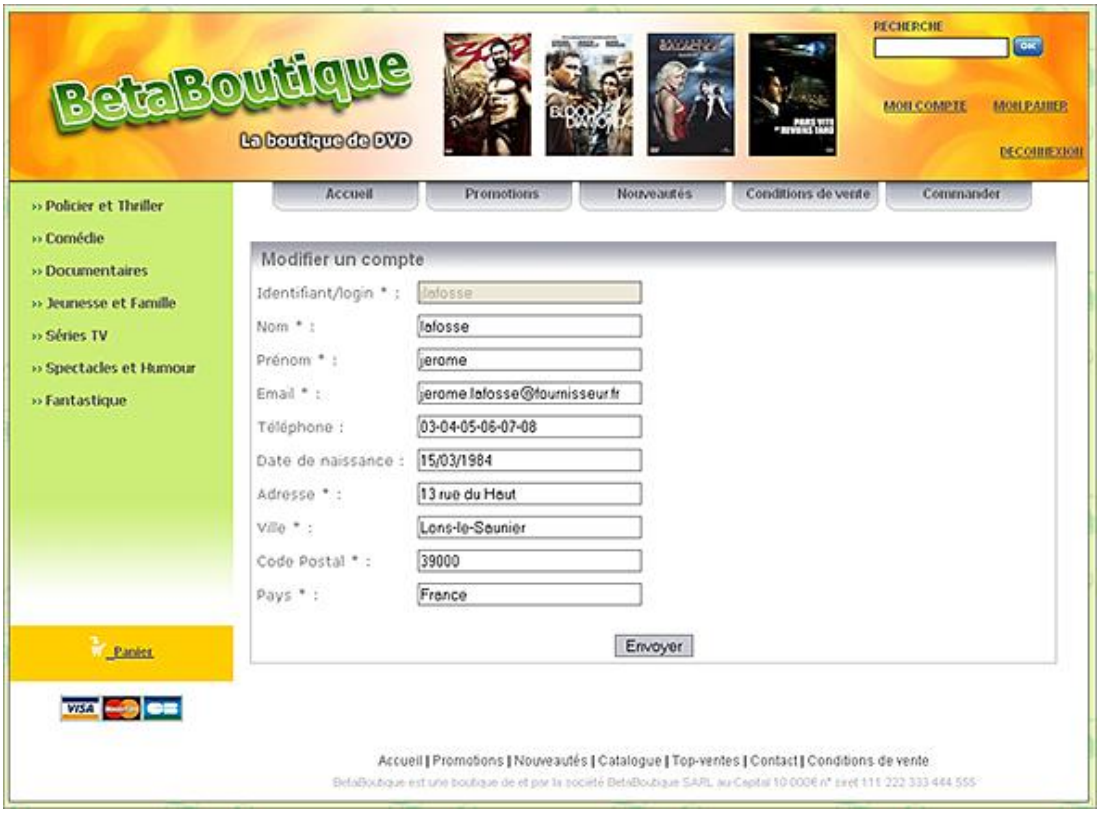
Si les informations sont correctes, le visiteur est alors invité à compléter ses informations personnelles (nom, prénom, email, adresse...).



g. Gérer le compte client

Chaque client peut consulter et mettre à jour ses informations personnelles au sein du système. Pour cela, le client doit se connecter au système et cliquer sur le lien **MON COMPTE** (après authentification). La page affiche en consultation ses données.

Le client peut passer en mode édition en cliquant sur **Modifier**.



h. Acheter des articles

Les clients (visiteurs authentifiés) peuvent acheter des articles dans le système. Le client visualise le catalogue par catégorie d'articles ou réalise une recherche. Lorsqu'il est intéressé par un article, il clique sur le lien adapté pour l'ajouter à son panier. Le client a ensuite la possibilité de modifier la quantité désirée pour chaque article ou supprimer un ou plusieurs articles. Le client peut visualiser à tout moment le contenu de son panier pendant toute la

durée de sa session. Lorsque le caddie est vide, un message informatif est affiché.

Le panier affiche la liste des articles avec le nom, la photo, la quantité, le prix unitaire et le sous-total (prix*quantité). Le montant total du panier est également affiché en bas du panier. Lorsque le client est satisfait et qu'il souhaite valider sa commande, il peut saisir les informations de sa carte bancaire ainsi que son adresse de livraison. Une fois toutes les données validées, un bon de commande est créé et le panier électronique est automatiquement vidé.

i. Gérer les commandes

Les employés de la société *BetaBoutique* peuvent visualiser et supprimer les commandes dans le système. Pour chaque commande, les employés peuvent visualiser le contenu et les articles associés.

j. Gérer les articles

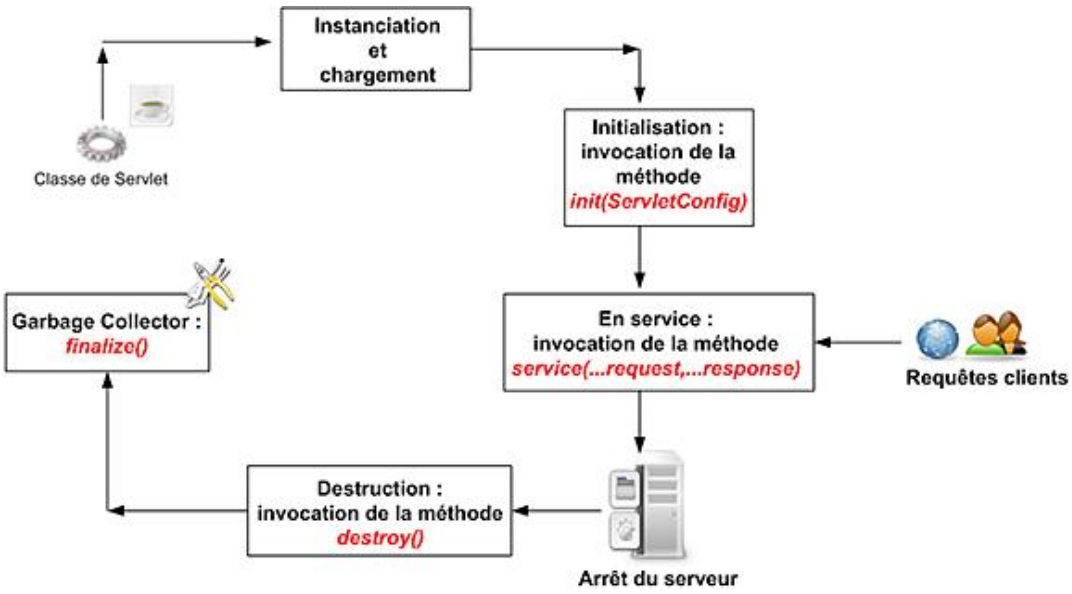
Les employés de la société *BetaBoutique* peuvent visualiser et supprimer les articles dans le système.

Première Servlet

1. Cycle de vie d'une Servlet

Chaque Servlet déployée sur un serveur d'applications possède un cycle de vie bien précis.

- La classe Java représentant la Servlet est déployée au sein du conteneur Web.
- Le conteneur Web va ensuite créer une instance de la classe de la Servlet et la charger en mémoire.
- Le conteneur Web passe ensuite à la phase d'initialisation de la Servlet en invoquant la méthode *init (ServletConfig)*. Cet objet permet par l'intermédiaire de l'objet *ServletConfig* de récupérer les paramètres d'initialisation définis dans le descripteur de déploiement *web.xml* de l'application Web.
- La Servlet passe ensuite dans un état en service ou de manière plus précise, en attente de réception de requêtes clients. Le conteneur gère ensuite une file d'attente de Threads. Chaque Thread invoque la méthode *service(...)* de la Servlet qui exécutera la méthode HTTP adaptée (*doPost(...), doGet(...)*...).
- L'instance de la Servlet reste chargée en mémoire tant que le conteneur Web s'exécute. Lors de l'arrêt du conteneur Web, la méthode *destroy()* est invoquée sur l'instance de la Servlet pour indiquer qu'elle n'est plus dans l'état en service.
- Une Servlet est un objet Java. Une instance non utilisée est donc supprimée de la mémoire par le Garbage Collector Java (déclenchement automatique de la méthode *finalize()*).



2. Fonctionnement d'une Servlet (la classe HttpServlet)

Lorsqu'un client (généralement par l'intermédiaire d'un navigateur mais pas nécessairement) envoie une requête HTTP au serveur, la méthode *service(...)* est invoquée sur la Servlet par le conteneur Web. La méthode reçoit deux paramètres de type *javax.servlet.ServletRequest* et *javax.servlet.ServletResponse* qui permettent d'effectuer des traitements sur la requête émise par le client et sur la réponse qui sera renvoyée.

Un objet de type *javax.servlet.ServletRequest* fournit des méthodes permettant de récupérer ou d'ajouter des données dans la requête ainsi que des méthodes pour les métadonnées de la requête (infos clients, taille, type de contenu, protocole...).

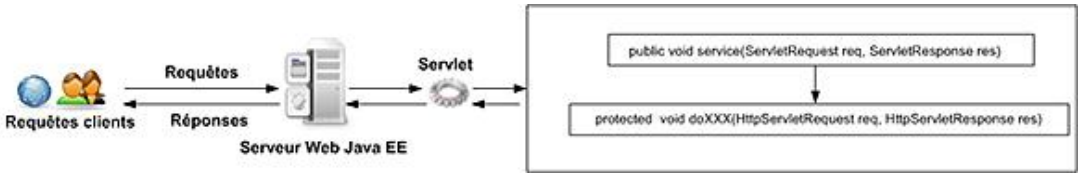
Un objet de type *javax.servlet.ServletResponse* fournit des méthodes pour insérer des données dans la réponse renvoyée au client.

L'interface *javax.servlet.http.HttpServletRequest* hérite de l'interface *javax.servlet.ServletRequest* et l'interface

javax.servlet.http.HttpServletResponse hérite de l'interface *javax.servlet.ServletResponse*.

Par défaut, la méthode *service(...)* invoque la méthode correspondant à la méthode HTTP utilisée par le client (principalement *doGet(...)* et *doPost(...)*) en lui transmettant les paramètres de type *javax.servlet.http.HttpServletRequest* et *javax.servlet.http.HttpServletResponse*. Le développeur n'a plus qu'à implémenter les traitements au sein de la méthode *doXXX(...)*.

Lorsque le conteneur Web reçoit une requête, il analyse l'URI, les en-têtes, le corps et stocke toutes les données dans un objet implémentant l'interface *javax.servlet.ServletRequest*. Il crée ensuite une instance d'un objet implémentant l'interface *javax.servlet.ServletResponse*. Cet objet encapsule la réponse qui sera envoyée au client. Le conteneur appelle ensuite une méthode de la classe de la Servlet, en lui passant les objets pour la requête et pour la réponse. La Servlet traite alors la requête et renvoie la réponse au client par l'intermédiaire du serveur.



a. La méthode service()

L'interface Servlet définit un nombre limité de méthodes. Les méthodes *init()* et *destroy()* ne gèrent pas les requêtes. La seule méthode concernée par cette tâche est *service()*. Les *HttpServlet* sont conçues pour répondre aux requêtes HTTP. Elles doivent donc traiter des requêtes GET, POST, HEAD... La méthode *doGet(...)* traite les requêtes de type GET et la méthode *doPost(...)* les requêtes de type POST. Il existe autant de méthodes *doXXX(...)* qu'il existe de type de requêtes HTTP.

Le rôle du programmeur est donc d'écrire une implémentation de la classe *HttpServlet* en redéfinissant les méthodes dont il a besoin. Dans la majorité des cas il s'agira des méthodes *doGet(...)* et *doPost(...)*.

Pour résumer, une Servlet étend la classe *HttpServlet* et redéfinit la méthode *service()* pour traiter les requêtes HTTP. Toutefois, la classe *HttpServlet* implémente déjà la méthode *service()*. Il est donc préférable de ne pas le faire et de redéfinir (définir) uniquement *doPost(...)* et *doGet(...)*.

Pour rappel, lorsque le conteneur de Servlets reçoit des requêtes HTTP, il associe chaque requête à une Servlet. Il appelle ensuite automatiquement la méthode *service()* de celle-ci. La méthode *service()* détermine le type de requête HTTP et appelle la méthode *doXXX(...)* adaptée. Si par contre, la Servlet redéfinit la méthode *service(),* c'est cette méthode qui sera exécutée.

Il peut parfois être utile de redéfinir la méthode *service()* de la classe *HttpServlet* dans nos Servlets si nous voulons effectuer un même traitement pour des requêtes émises par différentes méthodes HTTP (GET, POST...).

b. La méthode init()

L'interface *javax.servlet.Servlet* fournit des méthodes qui correspondent au cycle de vie de la Servlet : initialisation, en service, destruction.

Lorsque la Servlet est chargée en mémoire suite à une instanciation par le conteneur Web, la méthode *init(...)* de la Servlet est exécutée. Par défaut, cette méthode ne fait rien du tout. Il est cependant conseillé de redéfinir cette méthode dans le code des Servlets pour charger des ressources utilisées dans le reste de la Servlet (connexion JDBC, ouverture d'un fichier...), ou bien pour récupérer des paramètres d'initialisation renseignés dans le descripteur de déploiement (fichier *web.xml* de l'application) par l'intermédiaire de l'objet *ServletConfig* passé en paramètre de la méthode.

c. La méthode destroy()

Lors de l'arrêt du conteneur Web, les instances de Servlets chargées en mémoire sont détruites (par le Garbage Collector) mais avant la réalisation de cette étape, le conteneur exécute la méthode *destroy()* de chaque Servlet chargée.

Par défaut, cette méthode ne fait rien du tout. Il est cependant conseillé de redéfinir dans le code de Servlets cette méthode pour fermer proprement les ressources précédemment ouvertes (connexions JDBC, fichiers...).

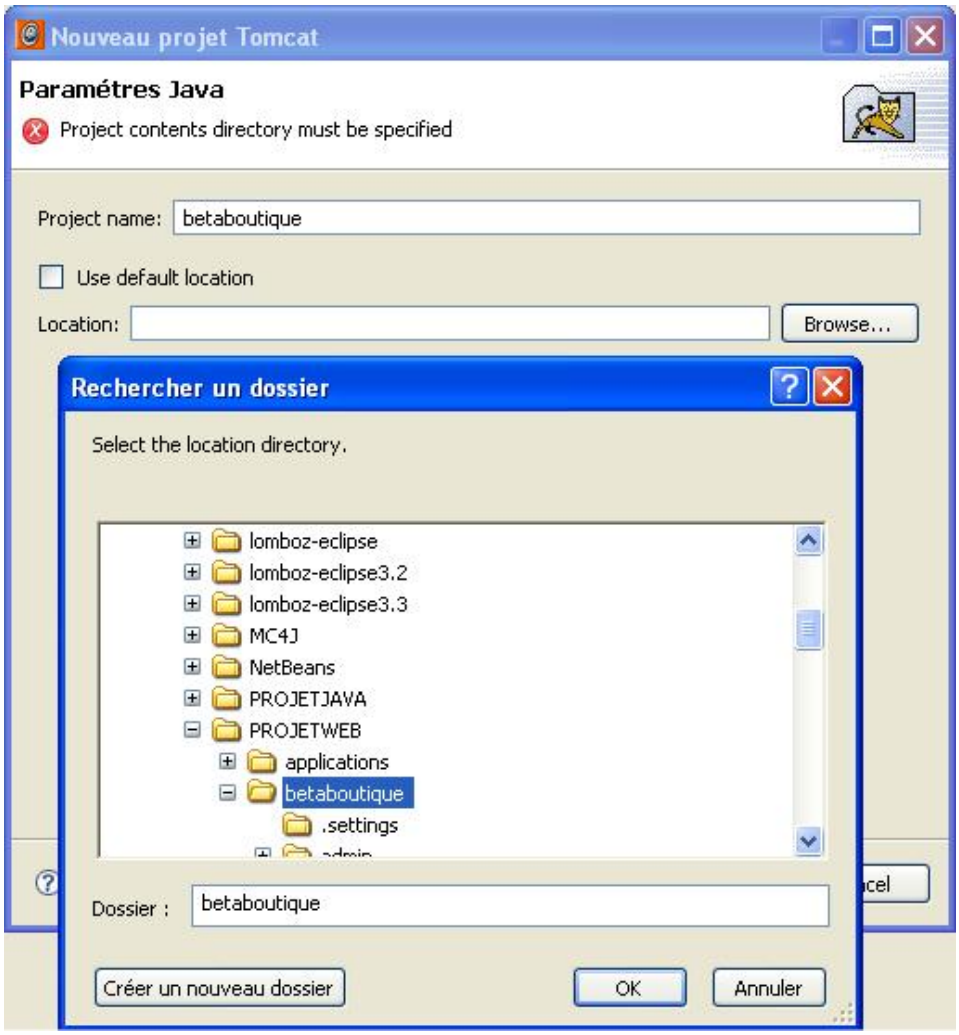
3. Invocation d'une Servlet

La manière la plus naturelle de communiquer avec une Servlet est d'utiliser un client Web et le protocole HTTP. La méthode *doGet(...)* d'une Servlet est invoquée principalement lorsque son URL est saisie directement dans la barre d'adresse du navigateur ou lorsqu'il y a un clic sur un lien hypertexte qui pointe sur l'URL de la Servlet.

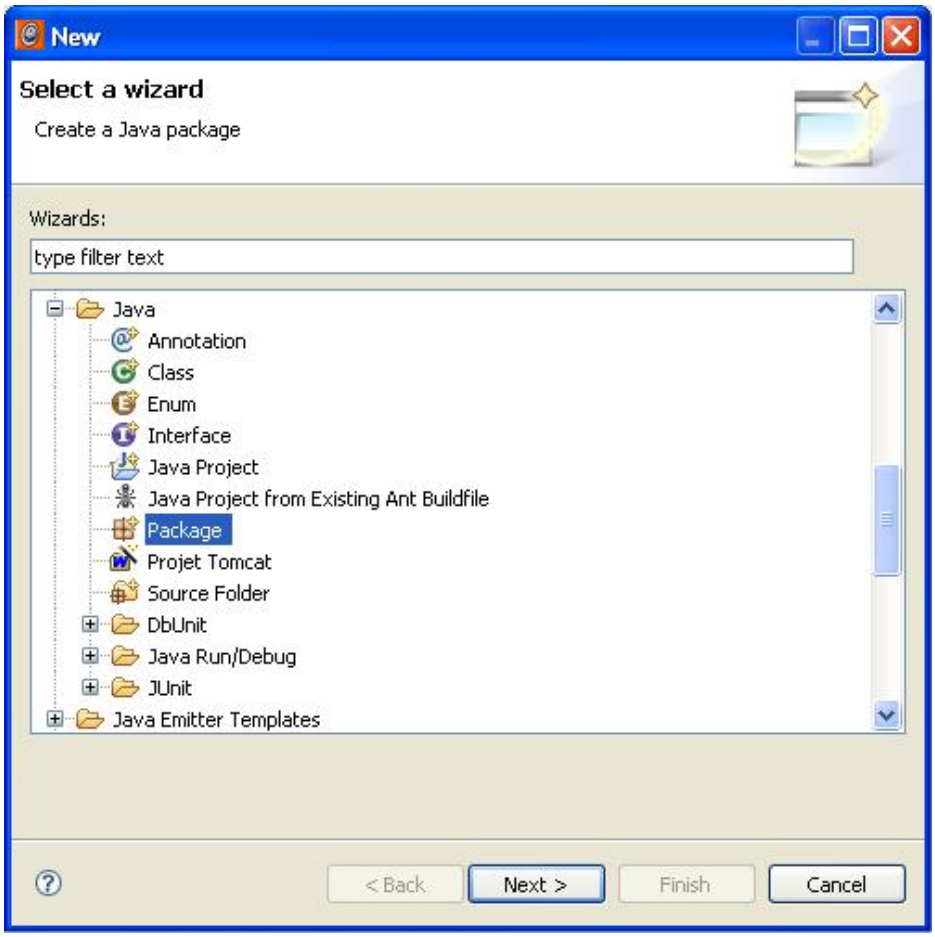
Chaque méthode *doGet(...)* et *doPost(...)* prend deux paramètres. L'objet *HttpServletRequest* encapsule la requête envoyée au serveur et contient toutes les données de la requête. L'objet *HttpServletResponse* encapsule la réponse à retourner au client.

Nous allons créer nos premières Servlets afin de montrer le fonctionnement des méthodes évoquées dans ce guide. Pour cela, sous Eclipse, nous allons créer un nouveau projet Tomcat sous le nom : *betaboutique*.

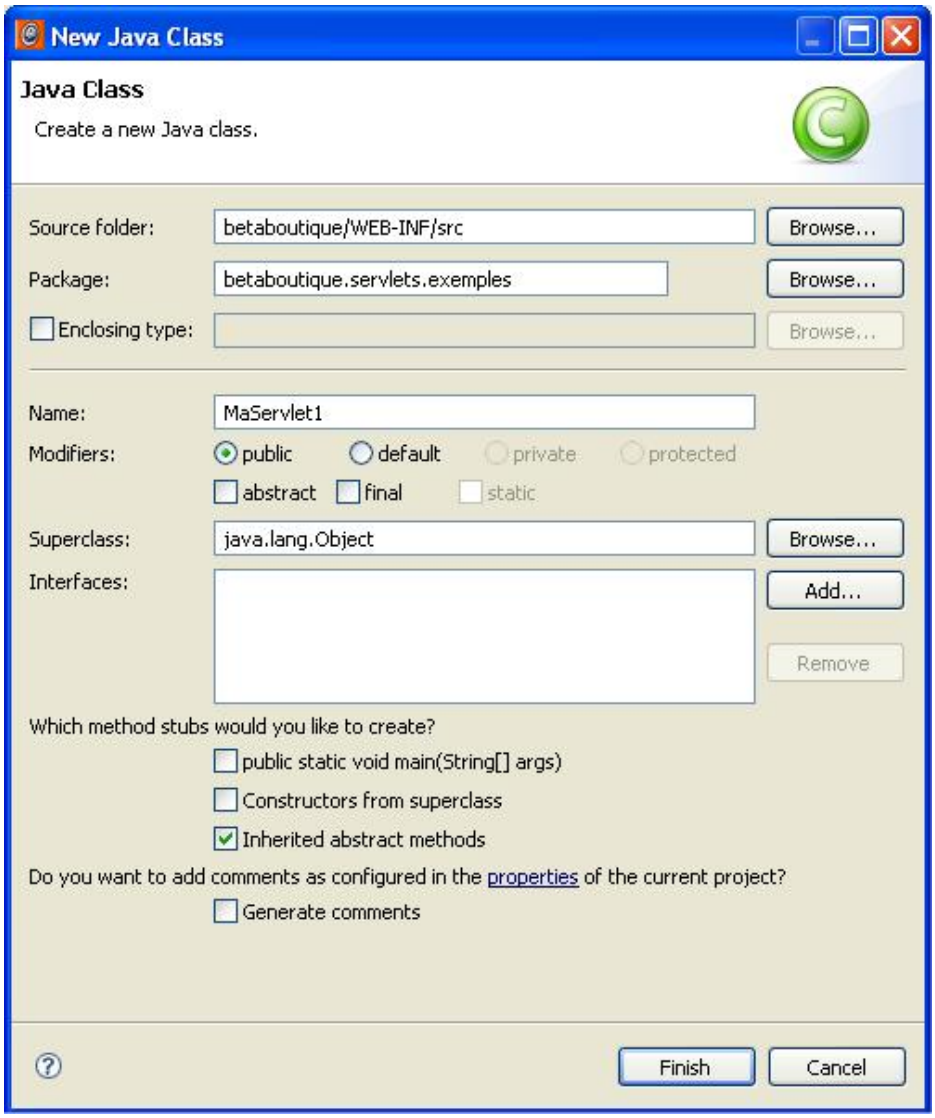
Attention, avant de créer le projet *betaboutique*, un répertoire de ce nom doit être créé sur le disque dur du système.



Nous allons créer au sein de ce projet un paquetage nommé : *betaboutique.servlets.exemples*.



Dans ce paquetage, nous ajoutons une nouvelle classe Java nommée : *MaServlet1.java*.



Toutes ces opérations peuvent bien sûr être réalisées à la main en utilisant un éditeur de texte simple et en enregistrant le fichier *MaServlet1.java* dans le répertoire *WEB-INF/src/* du projet.

Le contenu du fichier *MaServlet1.java* est le suivant :

```
package betaboutique.servlets.exemples;
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;

public class MaServlet1 extends HttpServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse res)throws
ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter out=res.getWriter();
        out.println("<html><body>");
        out.println("<h1>MaServlet1 en GET</h1>");
        out.println("</body></html>");
        out.flush();
        out.close();
    }
}
```

Cette Servlet génère un warning *The serializable class MaServlet1 does not declare a static final serialVersionUID field of*

type long. En effet, une classe sérialisable (comme les JavaBeans) utilise un attribut *serialVersionUID* qui permet d'affecter un numéro de version à la classe. Ce numéro doit être changé et mis à jour lorsque un champ sérialisable (non transient) est ajouté ou supprimé de la classe. C'est donc au développeur de gérer ce numéro de version, mais si ce numéro est absent, le compilateur génèrera un numéro automatiquement. Le champ *serialVersionUID* est utilisé lors de la désérialisation, afin de vérifier que les versions des classes Java sont correctes. Il est donc conseillé de gérer le champ *serialVersionUID* pour les classes sérialisables et de changer cette valeur lors d'un changement sur les champs de la classe. Pour définir l'attribut *serialVersionUID*, après la définition de la classe, le code suivant est utilisé :

```
private static final long serialVersionUID=1L;
```

Ce numéro (1L) n'a pas d'importance mais par convention, on utilise 1L, 2L... Sa valeur n'a pas d'importance du moment qu'elle est mise à jour lors de changements sur les champs sérialisables. Java 5.0/6.0 a introduit de nombreux warnings qui sont des avertissements mais qui n'empêchent pas l'exécution du code. Il est possible d'utiliser la notation suivante `@SuppressWarning` afin de supprimer un warning spécifique. Ainsi les warnings de version peuvent être supprimés avec cette syntaxe (attention, il n'y a pas de point virgule en fin d'instruction).

```
@SuppressWarnings("serial")
public class ServletAuthentification extends HttpServlet...}
```

Il est également possible de cacher un warning sur une méthode avant la déclaration de celle-ci, mais il est important de limiter au maximum la portée de l'annotation `@SuppressWarnings`. En effet, devant une méthode, la portée concerne la méthode elle-même mais devant une classe, elle concerne la totalité de cette dernière.

Cette première Servlet est compilée automatiquement par Eclipse (**Projet - Build Automatically**). Elle est déployée dans le répertoire : */WEB-INF/classes/betaboutique/servlets/exemples/MaServlet1.class*.

Nous pouvons créer une page HTML simple (nommée *maservlet1.html*) avec un lien hypertexte pour déclencher cette Servlet. Le code suivant est placé à la racine de l'application (dans le répertoire du projet *betaboutique*).

```
<html>
<head><title>MaServlet1</title></head>
<body>
<a href="maservlet1">Invoquer la Servlet MaServlet1</a>
</body>
</html>
```

a. Comment effectuer le lien entre la Servlet et l'URL?

Le fichier *web.xml* permet de paramétrer les Servlets de l'application. Chaque Servlet de l'application doit être décrite dans le fichier *web.xml* et le lien de la Servlet avec l'URL y figure également.

La déclaration d'une Servlet se fait dans l'élément XML <*servlet*> :

- <*servlet-name*> est le nom interne de la Servlet, il l'identifie de façon unique.
- <*servlet-class*> est la classe Java associée à la Servlet (notre classe Java).

Le lien entre l'URL/URI et la Servlet s'effectue grâce à l'élément <*servlet-mapping*> :

- <*servlet-name*> est le nom interne donné à la Servlet qui doit être identique à celui indiqué dans l'élément <*servlet*>.
- <*url-pattern*> est l'URL permettant de faire le lien avec la Servlet.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

    <!--  servlets -->
    <servlet>
        <servlet-name>servletmaservlet1</servlet-name>
        <servlet-class>betaboutique.servlets.exemples.MaServlet1
</servlet-class>
    </servlet>
    <!-- mapping des servlets -->
    <servlet-mapping>
```

```
        <servlet-name>servletmaservlet1</servlet-name>
        <url-pattern>/maservlet1</url-pattern>
    </servlet-mapping>
</web-app>
```

À chaque Servlet doit correspondre un élément *<servlet>* dans le fichier de configuration *web.xml* sinon, la Servlet n'existera pas pour le serveur d'applications. Le fichier */WEB-INF/web.xml* n'est pas créé par défaut par Eclipse. Nous pouvons le créer à partir d'un ancien projet qui fonctionne en réalisant un simple copier/coller ou avec les commandes suivantes d'Eclipse : clic droit sur le répertoire */WEB-INF*, **New - Other - XML**.

Désormais si nous tapons l'adresse suivante dans le navigateur : http://localhost:8080/betaboutique/maservlet1.html, nous pouvons accéder à la page HTML qui va déclencher la Servlet par l'intermédiaire d'un lien. Nous pouvons également déclencher directement la Servlet à cette adresse : http://localhost:8080/betaboutique/maservlet1.

Ceci ne peut fonctionner que si le mapping de la Servlet avec l'URL a été correctement effectué dans le fichier *web.xml*.

b. Comment cela fonctionne-t-il ?

Ce fonctionnement est très puissant et permet ainsi de déclencher des Servlets suivant des modèles d'URL. Dans l'exemple précédant l'appel à une URL du type : http://localhost:8080/betaboutique/maservlet1 déclenche la Servlet de nom *servletmaservlet1* qui est liée au fichier *MaServlet1.class*.

Il est possible, par exemple, de modifier ce fonctionnement en déclenchant la Servlet suite à l'appel d'une "fausse" page nommée *maservlet1html.html*.

```
...
    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletmaservlet1</servlet-name>
        <url-pattern>/maservlet1html.html</url-pattern>
    </servlet-mapping>
...
```

Attention, il est important de bien recharger le serveur après chaque modification du fichier de configuration *web.xml* avec le manager de Tomcat par exemple : http://localhost:8080/manager/html/.

Ce mécanisme très puissant est utilisé par les serveurs du monde entier. Ce système peut être amélioré en précisant que toutes les URL qui se terminent par exemple par *.spsf* déclencheront notre Servlet. Ceci pourra être utilisé pour un service particulier de l'application (télécharger le catalogue, une fiche article en PDF...).

```
...
    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletmaservlet1</servlet-name>
        <url-pattern>*.spsf</url-pattern>
    </servlet-mapping>
...
```

c. L'objet response

Dans l'exemple de la Servlet précédente, l'objet *response* permet d'envoyer des données au navigateur client. Cet objet permet d'envoyer la réponse au format TEXT, HTML, XML, sous forme de tableau d'octets... L'exemple précédent utilise un en-tête de réponse au format HTML *response.setContentType("text/html")*. Voici ci-après quatre exemples différents.

Le premier permet de renvoyer au navigateur un contenu au format HTML.

```
package betaboutique.servlets.exemples;
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;

public class MaServlet1 extends HttpServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse
res)throws ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter out=res.getWriter();
        out.println("<html><body>");
        out.println("<h1>MaServlet1 en GET</h1>");
        out.println("</body></html>");
        out.flush();
        out.close();
    }
}
```

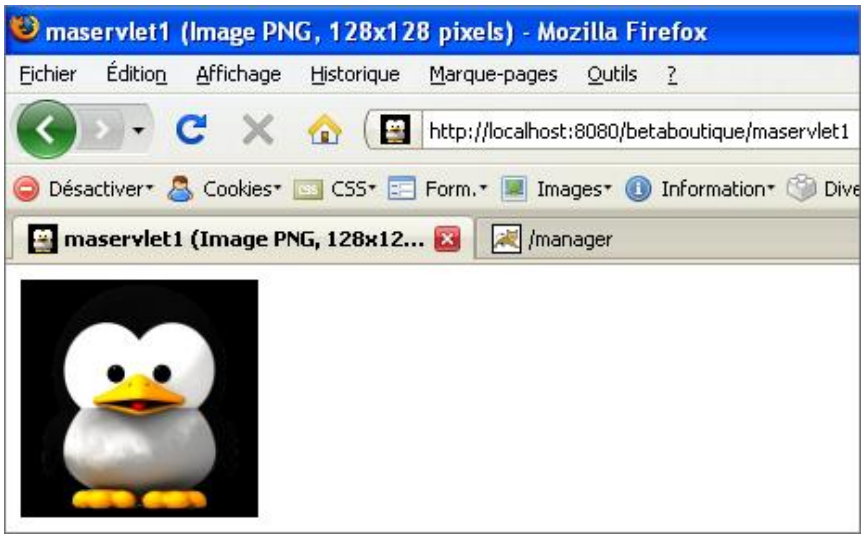
Le second exemple permet de renvoyer un contenu au format XML.

```
package betaboutique.servlets.exemples;
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;

public class MaServlet1 extends HttpServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse
res)throws ServletException, IOException
    {
        res.setContentType("text/xml");
        PrintWriter out=res.getWriter();
        out.println("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>");
        out.println("<racine>");
        out.println("<dvd1>Garde à vue</dvd1>");
        out.println("</racine>");
```

```
        out.flush();
        out.close();
    }
}
```

Le troisième permet de renvoyer un contenu dynamique sous la forme d'un flux d'octets (une image dans ce cas).

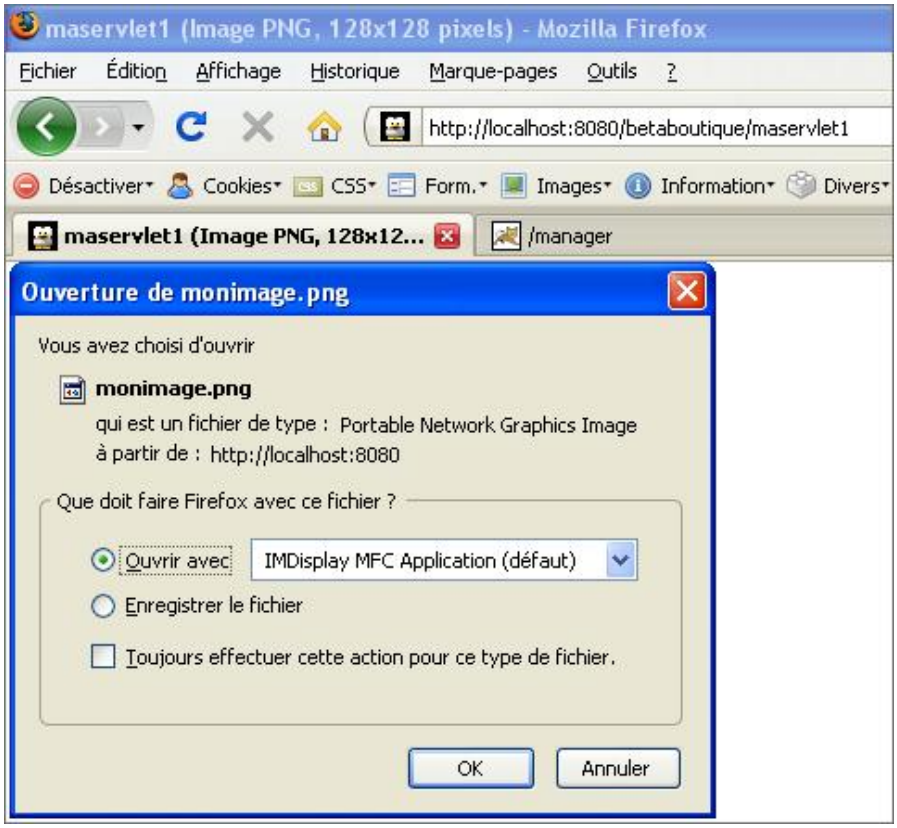
```
package betaboutique.servlets.exemples;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.image.BufferedImage;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.swing.ImageIcon;

public class MaServlet1 extends HttpServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse
res)throws ServletException, IOException
    {
        //acquerir l'image si elle existe
        Image image=null;
        File source=new File("E:\\PROJETWEB\\betaboutique\\monimage.png");
        if(source.exists())
        {
            //acquisition de l'image
            java.awt.Toolkit toolkit=java.awt.Toolkit.getDefaultToolkit();
            image=toolkit.getImage("E:\\PROJETWEB\\
betaboutique\\monimage.png");
            //transformer l'image en BufferedImage
            image=new ImageIcon(image).getImage();
            BufferedImage bufferedImage=newBufferedImage(image.getWidth(null),
image.getHeight(null),BufferedImage.TYPE_INT_RGB);
            Graphics2D g2=bufferedImage.createGraphics();
            g2.drawImage(image,0,0,null);
            g2.dispose();
            //sortie de l'image dans le navigateur
            response.reset();
            response.setContentType("image/png");
            response.setHeader("Pragma","no-cache");
            response.setHeader("Cache-Control","no-cache");
            response.setDateHeader("Expires",0);
            response.setHeader("Content-Disposition","filename=\"monimage\"");
            ByteArrayOutputStream fluxout=new ByteArrayOutputStream();
            try
            {
                ImageIO.write(bufferedImage,"png",fluxout);
                fluxout.writeTo(response.getOutputStream());
                fluxout.close();
            }
            catch(Exception e){}
            finally
            {
                if(fluxout!=null) fluxout.close();
                if(bufferedImage!=null)bufferedImage=null;
            }
        }
        else{
            System.out.println("Image non trouvée sur le disque dur");
        }
    }
}
```



Enfin, le dernier exemple permet de forcer un téléchargement et d'envoyer un fichier du serveur vers le client.

```
package betaboutique.servlets.exemples;

import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MaServlet1 extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
    response.setContentType("application/download");
      response.setHeader("Content-Disposition",
      "attachment;filename=\"monimage.png\"");

       ServletOutputStream out=response.getOutputStream();
        File file=null;
        BufferedInputStream from=null;
        try
        {
          file=new File("E:\\PROJETWEB\\betaboutique\\monimage.png");
          response.setContentLength((int) file.length());
          int bufferSize=64 * 1024;

          try
          {
            from=new BufferedInputStream(new FileInputStream(file),
bufferSize * 2);
            byte[] bufferFile = new byte[bufferSize];
            for (int i=0; ;i++)
            {
              int len=from.read(bufferFile);
              if (len < 0) break;
              out.write(bufferFile, 0, len);
            }
            out.flush();
          }
          finally
          {
            try {from.close();}catch (Exception e){}
            try {out.close();}catch (Exception e){}
          }
```

```
        }
        catch (Exception e)
        {
          return;
        }
    }
}
```



d. L'objet request

L'objet *request* est également très puissant et permet de gérer les données envoyées avec la requête à la Servlet. Par exemple les paramètres utilisateur (*getParameter(...)*) et les paramètres à valeurs multiples comme les champs *<select multiple...>* (*getParameterValues(...)*) peuvent être récupérés. La méthode *getParameterNames(...)* retourne une énumération des noms des paramètres de la requête. La méthode *getParameterMap(...)* retourne une énumération de tous les paramètres stockés dans un objet de type *Map (collection)*.

e. La méthode doPost()

La méthode *doPost(...)* d'une Servlet est invoquée principalement lors de l'envoi de données saisies dans un formulaire HTML (validation par un bouton de type *submit* par exemple). Pour résumer, le développeur doit coder le plus souvent deux méthodes importantes qui sont *doGet(...)* et *doPost(...)*.

Il est judicieux que l'une de ces méthodes (par exemple *doPost(...)*) appelle l'autre pour que la Servlet fonctionne correctement avec la plupart des cas d'utilisation.

```
package betaboutique.servlets.exemples;
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;

public class MaServlet1 extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
```

```
    {
        PrintWriter out=response.getWriter();
        out.println("Invocation de la MaServlet1 en GET ou POST");
    }

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

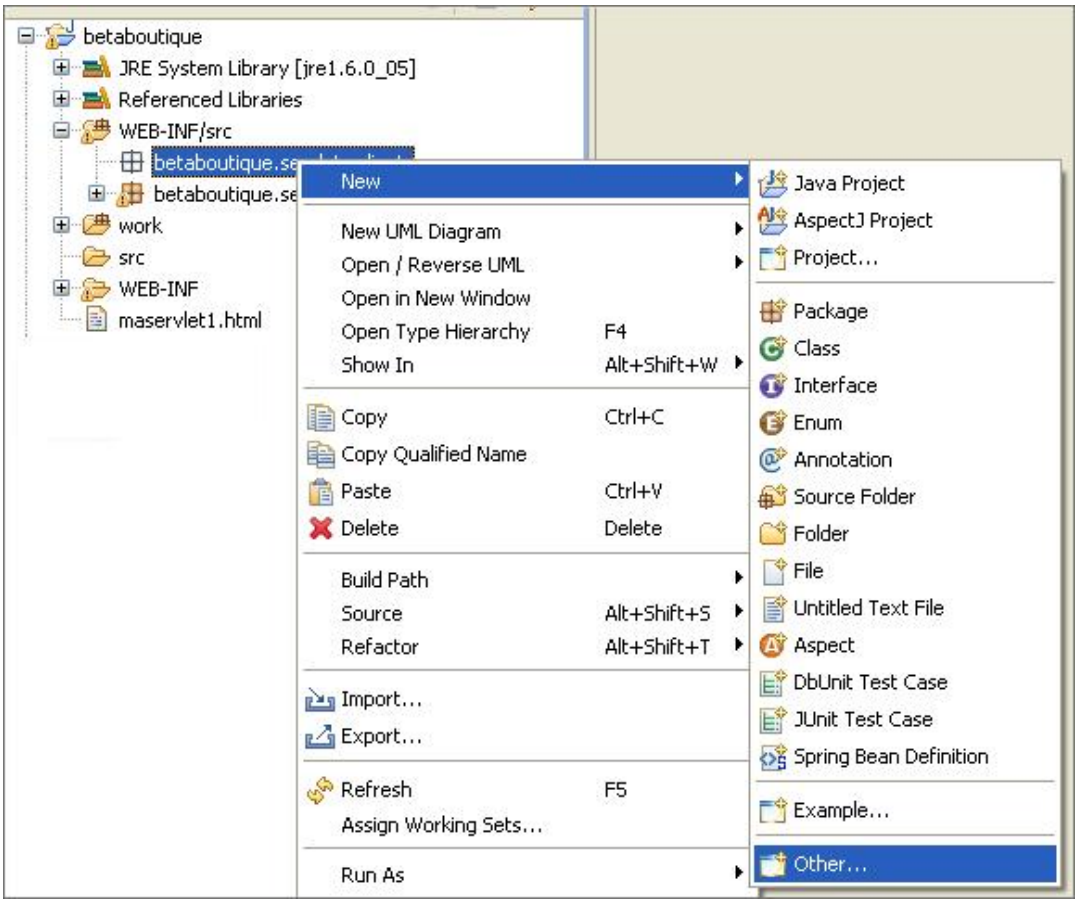
Servlet authentification

1. Gérer l'authentification client

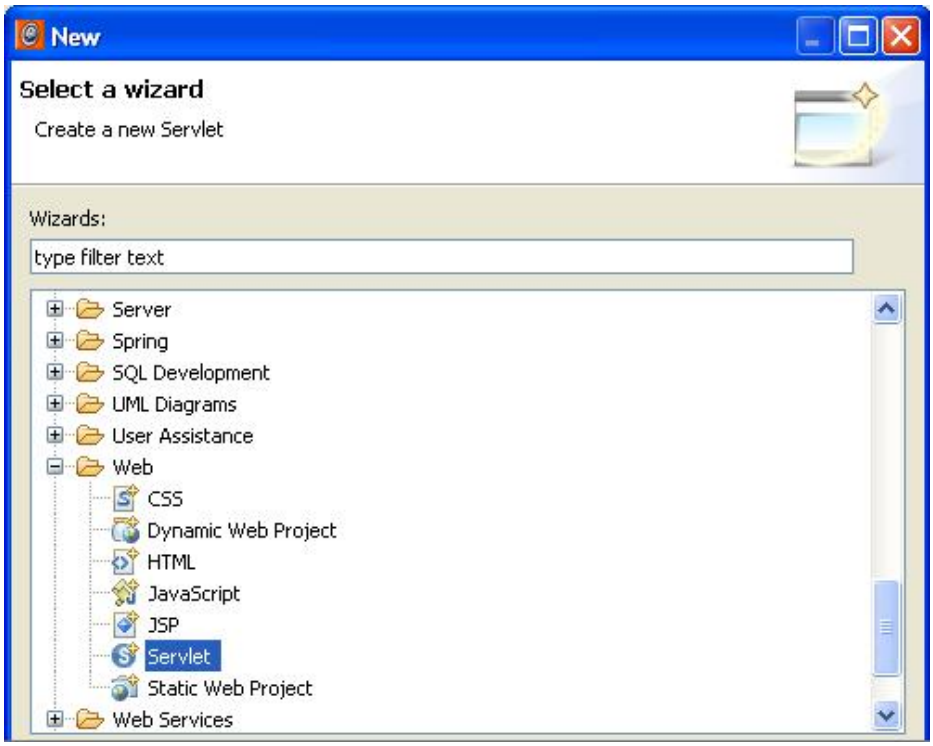
Nous allons mettre en application l'utilisation de Servlets pour le projet *BetaBoutique*. Pour cela, nous allons créer la Servlet et les pages associées pour le service d'authentification client. Dans un premier temps, un seul client pourra s'authentifier par identifiant et mot de passe avec une vérification des données dans la Servlet. Pour cela, nous commençons par créer un nouveau paquetage *betaboutique.servlets.client*.

Sous Eclipse, il est facile d'insérer une Servlet sans écrire tout le code nécessaire pour les méthodes HTTP. Le développeur doit juste surcharger les méthodes *doGet(...)* et *doPost(...)*.

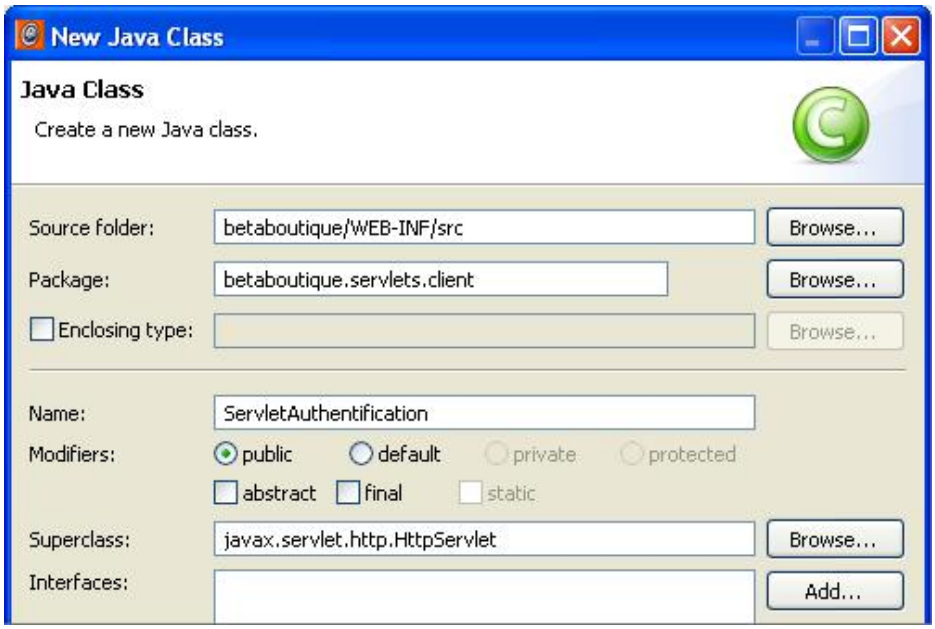
Nous faisons un clic droit sur le paquetage concerné (*betaboutique.servlets.client*), nous sélectionnons les étapes **New - Other**.



Nous sélectionnons alors l'onglet **Web** puis **Servlet** afin de créer notre nouvelle Servlet dans le paquetage adapté.



Nous pouvons également créer une nouvelle Servlet simplement en utilisant les étapes suivantes : clic droit sur le paquetage concerné, **New - class**.



Comme indiqué dans la maquette du projet *BetaBoutique*, pour l'authentification, l'utilisateur doit saisir son identifiant/login et son mot de passe. Pour cela nous allons utiliser un formulaire HTML simple (*authentification.html*) qui va invoquer la Servlet d'authentification (*ServletAuthentification*).

Une nouvelle page HTML peut être insérée à la racine, **New - Other - Web - HTML**.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Authentification BetaBoutique</title>
</head>
<body>
<h1>Authentification - Client</h1>
<form action="authentificationclient" method="POST">
<table border="1" cellspacing="0" cellpadding="5">
```

```
<tr>
    <td>Identifiant/Login : </td>
    <td><input type="text" name="identifiant" id="identifiant"
value="" size="20"/></td>
</tr>
<tr>
    <td>Mot de passe : </td>
    <td><input type="text" name="motdepasse" id="motdepasse"
value="" size="20"/></td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="submit"
name="valider" id="valider" value="Valider"/></td>
</tr>
</table>
</form>
</body>
</html>
```

Désormais, il faut réaliser le mapping dans le fichier de configuration pour que l'appel *<form action="authentificationclient"...>* déclenche notre Servlet : *ServletAuthentification*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!--  servlets -->
    <servlet>
        <servlet-name>servletmaservlet1</servlet-name>
        <servlet-class>betaboutique.servlets.exemples.MaServlet1
</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>servletauthentification</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
Authentification</servlet-class>
    </servlet>
    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletmaservlet1</servlet-name>
        <url-pattern>/maservlet1</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>servletauthentification</servlet-name>
        <url-pattern>/authentificationclient</url-pattern>
    </servlet-mapping>
</web-app>
```

Après le rechargement de Tomcat, l'appel de l'URL *authentificationclient*, déclenchera l'action *servletauthentification* qui est associée à la Servlet *betaboutique.servlets.client.ServletAuthentification*.

Il ne nous reste donc plus qu'à créer le code de la Servlet elle-même.

```
package betaboutique.servlets.client;

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;

public class ServletAuthentification extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //identifiant et mot de passe en dur
        String ident="monidentifiant";
        String mdp="monmotdepasse";
        //récupération de l'identifiant/login dans la requête
```

```
        String identifiant=request.getParameter("identifiant");
        //récupération du mot de passe dans la requête
        String motdepasse=request.getParameter("motdepasse");
        //flux de sortie
        PrintWriter out=response.getWriter();
        //pas d'identifiant
        if(identifiant==null)
        {
            out.println("Authentification incorrecte !");
        }
        //pas de mot de passe
        if(motdepasse==null)
        {
            out.println("Authentification incorrecte !");
        }
        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            out.println("Authentification correcte,
bienvenue : "+identifiant);
        }
        else
        {
        out.println("Authentification incorrecte, mauvaise
saisie des informations !");
        }
    }
    public void doPost(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        doGet(request, response);
    }
}
```

a. Comment cela fonctionne-t-il ?

Lorsqu'une Servlet est appelée pour la première fois, c'est sa méthode *init()* qui est déclenchée. C'est le seul cas où elle est appelée. Si la Servlet a été appelée par la méthode HTTP GET, la méthode *doGet(...)* de la Servlet traite la requête du client. Si la Servlet a été appelée par la méthode HTTP POST, la méthode *doPost(...)* de la Servlet traite la requête du client. La Servlet est dans ce cas la classe *betaboutique.servlets.client.ServletAuthentification*. Si elle n'est pas déjà chargée, elle le sera. Elle restera alors en mémoire pour les futures requêtes.

Dans notre Servlet (et dans la plupart des cas), la méthode *doPost(...)* renvoie à la méthode *doGet(...)* le contenu entier (requête et réponse). Le client pourra donc envoyer indifféremment ses paramètres par un POST ou un GET. Pour s'en convaincre, nous pouvons déclencher la Servlet avec cette URL : http://localhost:8080/betaboutique/authentificationclient?identifiant=monidentifiant&motdepasse= monmotdepasse

La méthode *doGet(...)* reçoit donc les deux paramètres d'authentification que sont *identifiant* et *motdepasse*. La méthode *getParameter(param)* de l'objet *request* sert à récupérer dans la requête du client la valeur du paramètre de nom *param*. Le contenu des champs est vérifié. S'ils ne sont pas à l'état *null*, l'authentification est vérifiée par rapport aux données initialisées dans la Servlet.

Dans cet exemple, c'est une page statique (HTML) qui déclenche une page dynamique (Java). Il n'existe donc aucun moyen d'indiquer à la page HTML de manière dynamique le résultat de l'exécution. En effet, il ne sera possible à aucun moment d'afficher le message d'erreur, de succès ou même les saisies de l'utilisateur dans la page HTML. Pour éviter cela, toute la page HTML peut être codée dans la Servlet.

```
package betaboutique.servlets.client;
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;

public class ServletAuthentification extends HttpServlet {

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

```
    {
        //identifiant et mot de passe
        String ident="monidentifiant";
        String mdp="monmotdepasse";

        //l'identifiant/login est récupéré dans la requête
        String identifiant=request.getParameter("identifiant");
        //le mot de passe est récupéré dans la requête
        String motdepasse=request.getParameter("motdepasse");

        //flux de sortie
        PrintWriter out=response.getWriter();

        //envoi du formulaire
        if(request.getParameter("valider")!=null)
        {
            //pas d'identifiant
            if(identifiant==null)
            {
                out.println("Authentification incorrecte !");
            }
            //pas de mot de passe
            if(motdepasse==null)
            {
                out.println("Authentification incorrecte !");
            }

            //vérifier l'égalité des valeurs
            if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
            {
                out.println("Authentification correcte,
bienvenue : "+identifiant);
            }
            else
            {
                out.println("Authentification incorrecte,
mauvaise saisie des informations !");
            }
        }

        //générer le code HTML pour le formulaire
d'authentification
        response.setContentType("text/html");
        out.println(
                "<html>"+
                "<head>"+
                "<meta http-equiv=\"Content-Type\" content=\"text/html;
charset=ISO-8859-1\">"+
                "<title>Authentification BetaBoutique</title>"+
                "</head>"+
                "<body>"+
                "<h1>Authentification - Client</h1>"+
                "<form action=\"authentificationclient\" method=\"POST\">"+
                "<table border=\"1\" cellspacing=\"0\" cellpadding=\"5\">"+
                "<tr>"+
                    "<td>Identifiant/Login : </td>"+
                    "<td><input type=\"text\" name=\"identifiant\"
id=\"identifiant\" value=\""+identifiant+"\" size=\"20\"/></td>"+
                "</tr>"+
                "<tr>"+
                    "<td>Mot de passe : </td>"+
                    "<td><input type=\"text\" name=\"motdepasse\"
id=\"motdepasse\" value=\""+motdepasse+"\" size=\"20\"/></td>"+
                "</tr>"+
                "<tr>"+
                    "<td colspan=\"2\" align=\"center\">
<input type=\"submit\" name=\"valider\" id=\"valider\"
value=\"Valider\"/></td>"+
                "</tr>"+
```

```
                "</table>"+
                "</form>"+
                "</body>"+
                "</html>"
        );
    }

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

}
```

Cette page est désormais totalement dynamique. Un message peut éventuellement être affiché en fonction de l'authentification (échec ou succès) et les saisies de l'utilisateur sont conservées dans les champs du formulaire.

Toutefois, nous remarquons immédiatement que la Servlet est mal adaptée pour générer du code HTML. Le fonctionnement était beaucoup plus simple et clair avec une page HTML pour le déclenchement et une Servlet pour la logique applicative.

Par la suite, c'est là qu'interviendront les pages JSP.

- Les pages d'appels et de réponses seront des documents dynamiques JSP.
- La logique de traitement des requêtes sera assurée par des Servlets.

Interface ServletConfig

1. Présentation

L'objet de type *javax.servlet.ServletConfig* représente les informations de configuration d'une Servlet au sein d'une application Web.

Le conteneur Web va créer un objet de type *javax.servlet.ServletConfig* pour chaque élément <*servlet*> déclaré dans le fichier de configuration de l'application *web.xml*.

Pour déclarer une Servlet dans le fichier de configuration *web.xml*, nous retrouvons le nom de la Servlet, la classe Java associée ainsi qu'une éventuelle/optionnelle liste de paramètres de la forme nom/valeur. Ces informations de configuration peuvent ensuite être récupérées par la Servlet de préférence dans la méthode *init(...)*.

Le fichier de configuration peut avoir zéro ou plusieurs éléments <*init-param*> par Servlet. Chaque élément <*init-param*> correspond à un paramètre représenté par une paire nom/valeur avec <*param-name*> et <*param-value*>.

Nous allons modifier notre précédente Servlet pour définir l'identifiant et le mot de passe dans le fichier de configuration plutôt que dans la Servlet elle-même.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!--  servlets -->
    <servlet>
        <servlet-name>servletauthentification</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
Authentification</servlet-class>
        <init-param>
            <param-name>defautIdentifiant</param-name>
            <param-value>monidentifiant</param-value>
        </init-param>
        <init-param>
            <param-name>defautMotDePasse</param-name>
            <param-value>monmotdepasse</param-value>
        </init-param>
    </servlet>

    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletauthentification</servlet-name>
        <url-pattern>/authentificationclient</url-pattern>
    </servlet-mapping>

</web-app>
```

2. Initialisation d'une Servlet

Après son chargement en mémoire, le conteneur Web passe en phase d'initialisation de la Servlet et la méthode *init(...)* est invoquée. La méthode *init(...)* est très souvent surchargée pour récupérer des informations de configuration, charger des ressources utiles à la Servlet... L'interface *ServletConfig* permet ensuite de manipuler les paramètres du fichier de configuration de l'application *web.xml*. La méthode *getInitParameter(...)* permet de récupérer la chaîne de caractères associée à un paramètre dans le fichier de configuration (ou la valeur *null* si le paramètre n'existe pas).

La méthode *getInitParameterNames()* permet de récupérer sous la forme d'une énumération l'ensemble des paramètres déclarés dans le fichier de configuration.

La méthode *getServletName()* permet de récupérer le nom de la Servlet déclarée au sein du descripteur de déploiement.

L'exemple suivant est une amélioration de la Servlet d'authentification afin de lire l'identifiant et le mot de passe dans le fichier de configuration pour perfectionner la maintenabilité de l'ensemble.

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.io.PrintWriter;
```

```
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletAuthentification extends HttpServlet {

    //variables de classe
    String ident=null;
    String mdp=null;

    public void init()
    {
        //récupération des paramètres d'initialisation
de la Servlet dans le fichier web.xml
        ServletConfig config=getServletConfig();
        ident=(String)config.getInitParameter("defautIdentifiant");
        mdp=(String)config.getInitParameter("defautMotDePasse");
    }

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupération de l'identifiant/login dans la requête
        String identifiant=request.getParameter("identifiant");
        //récupération du mot de passe dans la requête
        String motdepasse=request.getParameter("motdepasse");

        //flux de sortie
        PrintWriter out=response.getWriter();

        //pas d'identifiant
        if(identifiant==null)
        {
            out.println("Authentification incorrecte !");
        }
        //pas de mot de passe
        if(motdepasse==null)
        {
            out.println("Authentification incorrecte !");
        }

        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            out.println("Authentification correcte,
bienvenue : "+identifiant);
        }
        else
        {
            out.println("Authentification incorrecte, mauvaise
saisie des informations !");
        }
    }

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

Interface ServletContext

1. Présentation

La précédente interface *ServletConfig* est très intéressante pour gérer des paramètres qui sont propres à une Servlet comme un mot de passe, une clé de cryptage ou un chemin vers un fichier du disque dur.

À l'inverse de l'objet *javax.servlet.ServletConfig* qui représente les informations de configuration d'une Servlet au sein d'une application Web, un objet de type *javax.servlet.ServletContext* représente les informations de configuration d'une application Web totale. Chaque Servlet de la même application Web aura donc accès à ces paramètres.

Une autre propriété très importante de l'objet *javax.servlet.ServletContext* est qu'il peut également interagir avec le conteneur Web de l'application. Il y aura donc un accès en lecture mais aussi un autre accès en écriture dans le fichier de configuration. Le but étant alors de créer, lire et supprimer des attributs de façon dynamique, ce qui permet alors le partage de ressources entre Servlets d'une même application Web.

2. Utilisation

Un objet de type *javax.servlet.ServletContext* est obtenu en invoquant directement la méthode *getServletContext()* dans les méthodes des Servlets (objet *servlet*).

De la même manière que la déclaration spécifique de paramètre par Servlet, il est possible de déclarer des paramètres globaux pour toute l'application. Par exemple, il est possible de déclarer l'adresse email du Webmestre, de l'administrateur, une URL complète, une adresse IP fixe, un répertoire, une base de données, le pilote JDBC... Des objets pour des pools de connexions et des objets partagés peuvent être aussi déclarés.

L'élément <*web-app*>, racine du fichier de configuration *web.xml*, peut contenir zéro ou plusieurs éléments <*context-param*> qui permettent de déclarer des paramètres de l'application Web. Chaque élément <*context-param*> correspond à un paramètre représenté par une paire nom/valeur avec les éléments <*param-name*> et <*param-value*>.

Pour notre projet *Betaboutique*, nous allons ajouter l'email de l'administrateur à contacter lors de l'authentification.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!--  paramètres globaux -->
    <context-param>
        <param-name>emailAdministrateur</param-name>
        <param-value>admin@betaboutique.fr</param-value>
    </context-param>
    <!--  servlets -->
    <servlet>
        <servlet-name>servletauthentification</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
Authentification</servlet-class>
        <init-param>
            <param-name>defautIdentifiant</param-name>
            <param-value>monidentifiant</param-value>
        </init-param>
        <init-param>
            <param-name>defautMotDePasse</param-name>
            <param-value>monmotdepasse</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
            <servlet-name>servletauthentification</servlet-name>
        <url-pattern>/authentificationclient</url-pattern>
    </servlet-mapping>
</web-app>
```

3. Récupérer des paramètres

Pour récupérer les paramètres présents dans le fichier de configuration, il existe plusieurs méthodes. La première méthode *getInitParameter(param)* permet de récupérer une chaîne de caractères contenant la valeur d'un paramètre

ou *null* si le paramètre n'existe pas. La seconde méthode *getInitParameterNames()* permet de retourner l'énumération de l'ensemble des paramètres déclarés dans le fichier de configuration.

La Servlet d'authentification est modifiée afin d'afficher l'email de l'administrateur du site en cas d'erreur. Il faut remarquer l'utilisation de l'objet de type *javax.servlet.ServletContext*.

```
package betaboutique.servlets.client;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletAuthentification extends HttpServlet {
    //variables de classe
    String ident=null;
    String mdp=null;
    String emailadministrateur=null;
    public void init()
    {
        //récupération des paramètres d'initialisation
de la Servlet dans le fichier web.xml
         config=getServletConfig();
        ident=(String)config.getInitParameter("defautIdentifiant");
        mdp=(String)config.getInitParameter("defautMotDePasse");
        //récupérer l'email de l'administrateur
        ServletContext servletContext=getServletContext();
        emailadministrateur=servletContext.getInitParameter
("emailAdministrateur");
    }
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupération de l'identifiant/login dans la requête
        String identifiant=request.getParameter("identifiant");
        //récupération du mot de passe dans la requête
        String motdepasse=request.getParameter("motdepasse");
        //flux de sortie
        PrintWriter out=response.getWriter();
        //pas d'identifiant
        if(identifiant==null)
        {
            out.println("Authentification incorrecte !");
        }
        //pas de mot de passe
        if(motdepasse==null)
        {
            .println("Authentification incorrecte !");
        }
        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
        out.println("Authentification correcte,
bienvenue : "+identifiant);
        }
        else
        {
        out.println("Authentification incorrecte, mauvaise
saisie des informations !");
        out.println("Veuillez contacter : "+emailadministrateur);
        }
    }
    public void doPost(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        doGet(request, response);
```

```
    }}
```

4. Ajouter des paramètres

L'avantage de l'utilisation de l'interface *ServletContext* est la possibilité d'ajouter des attributs à la volée de manière logicielle (en programmation). Le contexte est accessible par l'ensemble des Servlets/JSP de l'application Web et peut donc stocker, récupérer et détruire des attributs. Contrairement à l'interface *ServletConfig*, des objets divers peuvent être gérés et pas seulement des chaînes de caractères.

Ce procédé très utile permet par exemple de déclarer au lancement de l'application un objet pour la connexion à la base de données et de le stocker à la volée dans le fichier de configuration comme variable globale. La connexion sera alors disponible pour l'ensemble des classes Java EE (Servlets et JSP). Ces attributs sont des paires clé/valeur, la clé étant une chaîne de caractères et la valeur étant un objet de n'importe quel type.

Les attributs d'une application Web sont donc des variables globales appelées également variables d'application des éléments participants à son fonctionnement et qui peuvent être partagées simultanément par plusieurs Servlets et pages JSP.

La méthode *setAttribute(nom, objet)* est utilisée pour positionner un attribut qui sera visible dans toute l'application (portée application). Si le nom de l'attribut existe déjà, la valeur existante est remplacée par la nouvelle.

La méthode *getAttribute(nom)* est utilisée pour récupérer la valeur d'un attribut de type quelconque dans l'application ou la valeur *null* si l'attribut n'existe pas.

La méthode *getAttributesNames()* permet de récupérer le nom de tous les attributs actuellement stockés dans le contexte de l'application Web.

Enfin, la méthode *removeAttribute(nom)* permet la suppression de l'attribut indiqué dans le contexte de l'application.

5. Mise en application

Afin de mettre en application l'interface *ServletContext*, nous allons créer une classe JavaBean *Client* afin de gérer un objet *client* suite à une authentification correcte. L'objet *client* sera alors enregistré dans le contexte de l'application et lu par une seconde *Servlet (ServletLectureAuthentification)*.

Pour cela nous allons créer un nouveau paquetage appelé *betaboutique.javabean*.

Ce paquetage permet de stocker toutes les classes POJO (*Plain Old Java Object*, utilisé pour faire référence à des classes simples composées d'accesseurs).

L'acronyme POJO est utilisé pour faire référence à la simplicité d'utilisation d'un objet Java en comparaison avec la lourdeur d'utilisation d'un composant EJB (*Entreprise Java Bean*). Les JavaBeans (à ne pas confondre avec les EJB) sont des composants logiciels simples réutilisables et manipulables. Pour être une classe JavaBean, celle-ci devra respecter certaines conventions pour son utilisation, sa réutilisation et sa connexion JavaBean : la classe doit être sérialisable (pour les sauvegardes et lectures), la classe doit avoir un constructeur par défaut (sans argument), les propriétés des méthodes doivent être accessibles via des méthodes (accesseurs). La seule différence réelle entre un POJO et un JavaBean est la possibilité de gérer des événements pour les JavaBeans.

```
package betaboutique.javabean;

public class Client implements java.io.Serializable {

    private String identifiant=null;
    private String motdepasse=null;

    //Constructeur par défaut (sans paramètre)
    public Client() {

    }
    public String getIdentifiant() {
        return identifiant;
    }
    public void setIdentifiant(String identifiant) {
        this.identifiant = identifiant;
    }
    public String getMotdepasse() {
        return motdepasse;
    }
```

```
    public void setMotdepasse(String motdepasse) {
        this.motdepasse = motdepasse;
    }
}
```

Le nouveau code de la Servlet permet de sauvegarder l'objet client correctement authentifié dans le contexte de l'application.

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import betaboutique.javabean.Client;

public class ServletAuthentification extends HttpServlet {

    //variables de classe
    String ident=null;
    String mdp=null;
    String emailadministrateur=null;

    public void init()
    {
        //récupérer les paramètres d'initialisation
de la Servlet dans le fichier web.xml
        ServletConfig config=getServletConfig();
        ident=(String)config.getInitParameter("defautIdentifiant");
        mdp=(String)config.getInitParameter("defautMotDePasse");

        //récupérer l'email de l'administrateur
        ServletContext servletContext=getServletContext();
        emailadministrateur=servletContext.getInitParameter
("emailAdministrateur");
    }

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
...
        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            out.println("Authentification correcte,
bienvenue : "+identifiant);
            //créer l'objet JavaBean Client
            Client client1=new Client();
            client1.setIdentifiant(identifiant);
            client1.setMotdepasse(motdepasse);

            //sauvegarder l'objet client dans le contexte de l'application
            ServletContext servletContext=getServletContext();
            servletContext.setAttribute("client1", client1);
        }
        ...
    }

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

}
```

Après avoir appelé la Servlet *ServletAuthentification* par le biais de l'URL http://localhost:8080/betaboutique/authentificationclient, nous pouvons déclencher la Servlet ci-dessous en respectant le code et la configuration du fichier *web.xml*.

Nous remarquons alors que notre objet *client1* stocké dans la variable globale est accessible pour toute l'application. Lors du redémarrage (ou de l'arrêt) du serveur, le contexte sera détruit ainsi que les objets présents dans ce contexte.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <context-param>
        <param-name>emailAdministrateur</param-name>
        <param-value>admin@betaboutique.fr</param-value>
    </context-param>
    <!--  servlets -->
    <servlet>
        <servlet-name>servletauthentification</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
Authentification</servlet-class>
        <init-param>
            <param-name>defautIdentifiant</param-name>
            <param-value>monidentifiant</param-value>
        </init-param>
        <init-param>
            <param-name>defautMotDePasse</param-name>
            <param-value>monmotdepasse</param-value>
        </init-param>
    </servlet>
    <servlet>
        <servlet-name>servletlectureauthentification</servlet-name>
        <servlet-class>betaboutique.servlets.client.ServletLecture
Authentification</servlet-class>
    </servlet>
    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletauthentification</servlet-name>
        <url-pattern>/authentificationclient</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>servletlectureauthentification</servlet-name>
        <url-pattern>/lectureauthentificationclient</url-pattern>
    </servlet-mapping>

</web-app>
```

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import betaboutique.javabean.Client;

public class ServletLectureAuthentification extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //flux de sortie
        PrintWriter out=response.getWriter();
        //lecture de l'objet dans le contexte de l'application
        ServletContext servletContext=getServletContext();
        Client client1=(Client)servletContext.getAttribute("client1");
        if(client1!=null)
        {
```

```
            out.println("Le client dans le contexte est : ");
            out.println("identifiant : "+client1.getIdentifiant());
            out.println("mot de passe : "+client1.getMotdepasse());
        }
    }

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

Traitement des requêtes

1. Présentation

Les méthodes *doGet(...)* et *doPost(...)* possèdent en premier paramètre un objet de type *javax.servlet.http.HttpServletRequest*. Cet objet est créé par le conteneur Web juste avant le déclenchement d'une des méthodes et correspond à la requête du client. Il existe de nombreuses méthodes dédiées au traitement des requêtes utilisateurs comme la récupération de paramètres, la gestion des flux...

a. Récupérer des paramètres transmis par le client

Les paramètres permettent à la Servlet de réaliser les traitements adaptés. La méthode *getParameter(nom)* permet de récupérer sous la forme d'une chaîne de caractères la valeur d'un paramètre nommé passé en argument. Si le paramètre n'existe pas, la valeur *null* est retournée.

```
...
public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer l'identifiant/login dans la requête
        String identifiant=request.getParameter("identifiant");
...
```

La méthode *getParameterNames()* permet de récupérer sous la forme d'une énumération l'ensemble des noms des paramètres contenus dans la requête.

```
...
public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
Enumeration enume=request.getParameterNames();
while(enume.hasMoreElements())
{
    String nomParametre=(String)enume.nextElement();
    String valeurParametre=request.getParameter(nomParametre);
    out.println("parametre : nom="+nomParametre+" - valeur :
"+valeurParametre);
}
...
```

La méthode *getParameterValues(nom)* permet de récupérer sous la forme d'un tableau de chaînes de caractères un ensemble de valeurs ou la valeur *null* si le paramètre n'existe pas. Cette méthode est utilisée pour les listes <*select*> multiples ou les cases à cocher <*checkbox*>. La méthode *getParameterMap()* permet de récupérer sous la forme d'un objet de type *Map* l'ensemble des paramètres contenus dans la requête.

b. Gérer les attributs du contexte de la requête

L'objet *request* permet de créer un attribut dans la requête. Si l'attribut existe déjà, la valeur existante est remplacée par la nouvelle. Cette technique est très utilisée pour envoyer des paramètres à une page dynamique pour par exemple l'affichage des erreurs, des messages de succès, du numéro d'enregistrement dans la base de données... La méthode *setAttribute(nom,objet)* permet de créer un attribut dans le contexte de la requête. La méthode *getAttribute(nom)* permet de récupérer la valeur d'un attribut dans le contexte de la requête. La méthode *getAttributeNames()* permet de récupérer le nom de tous les attributs actuellement stockés dans le contexte de la requête sous la forme d'une énumération. La méthode *removeAttribute(nom)* permet de supprimer un attribut du contexte de la requête.

c. Récupérer des informations sur l'URL de la requête

Il existe plusieurs méthodes qui permettent d'obtenir des informations sur l'URL de la requête HTTP comme : le protocole utilisé, les paramètres passés, le chemin vers la Servlet...

La méthode *getScheme()* permet de retourner le nom du protocole utilisé HTTP, HTTPS ou FTP .

Ex : http://localhost:8080/betaboutique/authentificationclient dans ce cas retourne : http

La méthode *getContextePath()* permet de retourner sous la forme d'une chaîne de caractères la partie de l'URL qui correspond au nom du contexte de l'application.

Ex : http://localhost:8080/betaboutique/authentificationclient dans ce cas retourne : */betaboutique*.

La méthode *getMethod()* retourne le nom de la méthode HTTP utilisée (GET, POST, DELETE...).

Ex : http://localhost:8080/betaboutique/authentificationclient dans ce cas retourne : GET avec un lien.

La méthode *getQueryString()* retourne la chaîne de requête contenue dans l'URL après le chemin de la ressource invoquée ou la valeur *null* s'il n'en existe pas.

Ex : http://localhost:8080/betaboutique/authentificationclient?param1=1&param2=2 dans ce cas retourne : param1=1&param2=2

La méthode *getRequestUrl()* retourne la partie de l'URL contenue entre le nom du protocole (http dans ce cas) et la chaîne de requête.

Ex : http://localhost:8080/betaboutique/authentificationclient?param1=1&param2=2 dans ce cas retourne : http://localhost:8080/betaboutique/authentificationclient.

d. Récupérer des informations sur le client

Il existe également plusieurs méthodes pour obtenir des informations sur le client qui a émis la requête HTTP. La méthode *getRemoteAddr()* permet d'obtenir l'adresse IP du client. La méthode *getRemoteHost()* permet d'obtenir le nom complet du client. La méthode *getRemoteUser()* retourne le nom de l'utilisateur qui a envoyé la requête.

e. Récupérer des informations sur le serveur

Les deux méthodes suivantes sont très utilisées pour renseigner des URL "en dur" ; dans le cas de redirection, d'invocation de méthodes avec chemin complet... Ces deux méthodes permettent de construire la base de l'URL d'invocation des ressources.

La méthode *getServerName()* retourne le nom d'hôte du serveur.

Ex : http://localhost:8080/betaboutique/authentificationclient?param1=1&param2=2 dans ce cas retourne : localhost.

La méthode *getServerPort()* retourne le numéro du port d'écoute du serveur.

Ex : http://localhost:8080/betaboutique/authentificationclient?param1=1&param2=2 dans ce cas retourne : 8080

Traitement des réponses

Le deuxième paramètre des méthodes *doGet(...)* et *doPost(...)* est un objet de type *javax.servlet.http.HttpServletResponse*. Cet objet est créé par le conteneur Web avant l'invocation de ces méthodes et correspond à la réponse qui sera retournée au client.

Type et taille du contenu de la réponse

Il est possible d'indiquer le type MIME de la réponse HTTP ainsi que la taille des données contenues dans le corps de la réponse. La méthode *setContentType(type)* permet de spécifier le contenu du corps de la réponse (*text/html*, *text/plain*, *text/xml*, *application/pdf*...). La méthode *setContentLength(taille)* permet de spécifier la taille du contenu de la réponse HTTP.

Gérer le flux de sortie

L'interface *javax.servlet.ServletResponse* définit deux méthodes et objets qui permettent d'écrire dans le contenu de la réponse HTTP. La méthode *getOutputStream()* permet d'écrire des données au format binaire dans le corps de la réponse. Il est conseillé de valider l'envoi de la réponse avec la méthode *flush()*. La méthode *getWriter()* permet d'écrire des données au format texte dans le corps de la réponse. Comme pour la méthode *getOutputStream()*, il est conseillé d'utiliser la méthode *flush()* pour valider l'envoi de la réponse.

Encoder les URL

Il est parfois nécessaire d'encoder les URL afin d'inclure l'identifiant de session pour la navigation par état (session), pour conserver des accents sur les paramètres, pour l'encodage de la langue dans les paramètres... La méthode *encodeURL(url)* permet d'encoder l'URL passée en paramètre en incluant l'identifiant de session. La méthode *encodeRedirectURL(url)* permet d'encoder l'URL passée en paramètre en incluant l'identifiant de session mais avec l'utilisation de la méthode *sendRedirect(...)*.

Redirection d'URL et états HTTP

La technique suivante est très utilisée en développement et permet d'envoyer au navigateur du client un ordre de redirection sur une autre ressource de l'application (ou non). L'URL de la ressource est passée en paramètre et peut être relative ou absolue. Nous allons utiliser cette méthode dans notre Servlet d'authentification afin de rediriger l'utilisateur sur la page HTML suivante en cas de succès.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Authentification</title>
</head>
<body>
<h1>Utilisateur correctement authentifié</h1>
</body>
</html>
```

```
package betaboutique.servlets.client;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class ServletAuthentification extends HttpServlet {

    //variables de classe
    String ident=null;
    String mdp=null;
    String emailadministrateur=null;

    public void init()
    {
        //récupérer les paramètres d'initialisation
de la Servlet dans le fichier web.xml
```

```
        ServletConfig config=getServletConfig();
        ident=(String)config.getInitParameter("defautIdentifiant");
        mdp=(String)config.getInitParameter("defautMotDePasse");

        //récupérer l'email de l'administrateur
        ServletContext servletContext=getServletContext();
        emailadministrateur=servletContext.getInitParameter
("emailAdministrateur");
    }

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer l'identifiant/login dans la requête
        String identifiant=request.getParameter("identifiant");
        //récupérer le mot de passe dans la requête
        String motdepasse=request.getParameter("motdepasse");

        //flux de sortie
        PrintWriter out=response.getWriter();

        //pas d'identifiant
        if(identifiant==null)
        {
            out.println("Authentification incorrecte !");
        }
        //pas de mot de passe
        if(motdepasse==null)
        {
            out.println("Authentification incorrecte !");
        }

        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            //redirection vers la page de succès
            response.sendRedirect("authentificationcorrecte.html");
        }
        else
        {
            out.println("Authentification incorrecte, mauvaise
saisie des informations !");
            out.println("Veuillez contacter : "+emailadministrateur);
        }
    }

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

La méthode *sendStatus(statut)* permet d'appliquer un code d'état à la réponse HTTP quand il n'y a pas d'erreur comme par exemple OK(200), CONTINUE(100)...

La méthode *sendError(...)* permet d'envoyer un code d'erreur HTTP au client comme par exemple NOT FOUND(404), SERVICE UNVAILABLE(503)...

Synchronisation des traitements

Une Servlet fonctionne dans un environnement multitâches. À chaque requête reçue par une Servlet, le conteneur Web va créer un thread qui va exécuter la méthode de service (*doGet(...)* ou *doPost(...)*).

Ce principe de fonctionnement peut parfois poser des problèmes si la méthode de service travaille avec des variables de la Servlet. Chaque thread peut modifier la valeur de ces variables. L'interface *javax.servlet.SingleThreadModel* permet d'isoler le fonctionnement de chaque thread. Le conteneur Web prend en charge le fait qu'une instance de la Servlet ne peut être exécutée que par un seul thread à la fois.

Ex : déclaration d'une Servlet qui implémente l'interface SingleThreadModel

```
import javax.servlet.SingleThreadModel;
public class ServletAuthentification extends HttpServlet
implements SingleThreadModel{...}
```

Même si cette solution est fréquemment recommandée, c'est un mauvais conseil. D'ailleurs, avec le compilateur JDK6.0, la méthode est ''barrée'' et donc notée comme dépréciée. L'interface *SingleThreadModel* ne fait que signaler au conteneur qu'un seul thread doit être autorisé pour la Servlet à un instant donné. Ceci ne garantit pas que la Servlet soit protégée contre les accès concurrents. Des variables statiques par exemple sont partagées par toutes les instances d'une classe. Une raison supplémentaire de ne pas utiliser l'interface *SingleThreadModel* est que cette option ne tient pas la charge. Le nombre de Servlets que peut créer le conteneur est limité. Il est bien moins coûteux de créer un nouveau thread qu'un nouvel objet pour la Servlet.

L'utilisation de la synchronisation peut convenir pour des exemples simples mais dans une application réelle, cette solution est inefficace. Le temps de réponse augmente alors proportionnellement avec le nombre de requêtes.

La synchronisation des méthodes *doGet(...)* et *doPost(...)* est également une mauvaise idée. La méthode *service()* appelle systématiquement une de ces deux méthodes, le résultat est donc le même que dans le cas de la synchronisation de la méthode *service()*.

Pour résumer, il est parfois nécessaire de synchroniser des portions de code (comme lors d'utilisation de threads dans des IHM) pour gérer les accès concurrents. Il faut cependant veiller à ce que la synchronisation concerne le plus petit bloc possible (méthode *synchronized(object)*). La réduction du nombre d'instructions synchronisées augmente la vitesse d'exécution du code.

Il reste cependant un point délicat, l'accès concurrentiel aux données. Soit par exemple deux clients souhaitant s'enregistrer par l'intermédiaire d'un formulaire HTML. Ils envoient leurs données en même temps à destination d'une Servlet dont le rôle est le traitement des données et l'enregistrement dans la table des clients. Que se passe-t-il si deux instances de la Servlet effectuent le même traitement en même temps ?

Dans le cas d'une clé sans autoincrément, il y aura une erreur de doublon de l'id d'enregistrement. Un seul des deux clients sera donc enregistré dans la base de données des clients.

Ce problème sera géré par la suite par un mode transactionnel d'accès aux données (propre au système de stockage) qui est plus puissant que la synchronisation de Servlet.

État des clients

Le protocole HTTP est sans état. La technologie utilise un mode déconnecté (connexion pour la requête en cours puis déconnexion). Chaque requête est considérée comme provenant d'un client différent à chaque fois.

Il est important dans une application Web de pouvoir suivre la navigation du client sur le site et de garder certaines informations qui permettent de l'identifier de manière unique et fiable.

Pour notre projet *betaboutique*, nous devons pouvoir garder les articles du client déposés dans son panier afin de retrouver son contenu à tout moment. Sans gestion de cookie et/ou session , tout client qui envoie une requête est considéré comme un nouveau client.

1. Les cookies

Un cookie est un élément de données qui peut être envoyé par le serveur à destination du navigateur client. Le cookie est stocké soit en mémoire (expiration rapide) soit sur le système de fichiers du client (expiration longue). L'avantage réside dans le fait qu'à chaque connexion du client, le navigateur transmet également le cookie (ou les cookies) que notre serveur a précédemment déposé(s). Il existe cependant un inconvénient majeur. L'utilisateur peut configurer son navigateur pour refuser les cookies ou les supprimer directement manuellement. Un cookie est composé d'une paire nom/valeur et de plusieurs propriétés pour renseigner le domaine, la durée de vie, la sécurité... L'API Servlet fournit la classe *javax.servlet.http.Cookie* qui permet de manipuler les cookies en Java. Le constructeur *Cookie(nom,valeur)* permet de créer un cookie. La méthode *getName()* permet de récupérer le nom d'un cookie.

La méthode *getValue()* permet de récupérer la valeur du cookie alors que la méthode *setValue(valeur)* permet de spécifier une valeur à un cookie.

Les méthodes *getDomaine()* et *setDomaine(valeur)* permettent de gérer le domaine lié au cookie.

Les méthodes *getPath()* et *setPath(valeur)* permettent de gérer le contexte de l'application Web.

Les méthodes *getMaxAge()* et *setMaxAge(valeur)* permettent de gérer la durée de vie d'un cookie.

Les méthodes *getSecure()* et *setSecure(valeur)* permettent de gérer la sécurité d'un cookie (*http* ou *https*).

Les méthodes *getComment()* et *setComment(valeur)* permettent de gérer les commentaires associés à un cookie.

Les méthodes *getVersion()* et *setVersion(valeur)* permettent de gérer les versions des cookies.

a. Envoyer des cookies dans la réponse

Nous allons modifier notre Servlet d'authentification afin d'envoyer un cookie en cas de succès d'authentification.

```
...
if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
 {
    //envoyer un cookie au client
    Cookie cookie1=new Cookie("cookie1","authentification correcte !");
    response.addCookie(cookie1);
    //redirection vers la page de succès
    response.sendRedirect("authentificationcorrecte.html");
 }
...
```

b. Récupérer des cookies dans la requête

Nous allons créer une nouvelle Servlet appelée *ServletLectureCookies* qui permet de lire les cookies. L'interface *javax.servlet.http.HttpServletRequest* définit une méthode qui permet de récupérer sous la forme d'un tableau la liste des cookies pour un domaine. Cette méthode retourne *null* si aucun cookie n'a été envoyé. Nous pouvons donc tester la Servlet et la gestion des cookies en réalisant une authentification correcte et en appelant ensuite la page : http://localhost:8080/betaboutique/lecturecookies.

```
package betaboutique.servlets.client;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletLectureCookies extends HttpServlet {


    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //lire les cookies du domaine
        Cookie[] cookies=request.getCookies();
        if(cookies!=null)
        {
            for(int i=0;i<cookies.length;i++)
            {
            Cookie cookie=cookies[i];
            String nomCookie=cookie.getName();
            String valeurCookie=cookie.getValue();
            System.out.println("Cookie : nom="+nomCookie+" -
valeur="+valeurCookie);
            }
        }
    }

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

```
...
    <servlet>
        <servlet-name>servletlecturecookies</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
LectureCookies</servlet-class>
    </servlet>
...
    <servlet-mapping>
        <servlet-name>servletlecturecookies</servlet-name>
        <url-pattern>/lecturecookies</url-pattern>
    </servlet-mapping>
...
```

Le cookie utilisé dans cet exemple n'a pas de durée de vie, il sera donc juste accessible le temps de la session de l'utilisateur (si fermeture et ouverture à nouveau du navigateur en déclenchant la Servlet de lecture des cookies, le cookie est perdu).

Pour utiliser un cookie durable, il faut préciser la durée de celui-ci en secondes.

```
...
    if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
    {
        //envoyer un cookie au client
        Cookie cookie1=new Cookie("cookie1","authentification correcte !");
        //cookie d'une journée
        cookie1.setMaxAge(24*60*60);
        response.addCookie(cookie1);
        //redirection vers la page de succès
        response.sendRedirect("authentificationcorrecte.html");
    }
...
```

Sous Windows avec le navigateur InternetExplorer, les cookies sont stockés par défaut dans le répertoire *C:\Documents and Settings\UTILISATEUR\Cookies*. Nous pouvons remarquer que suite à une authentification correcte, un cookie nommé : *utilisateur@betaboutique[1]* est alors créé.

c. Supprimer un cookie

Pour supprimer un cookie présent sur la machine du client, il faut lui envoyer un nouveau cookie avec les paramètres suivants : nom identique, valeur vide, durée de vie égale à -1. Le cookie sera alors détruit immédiatement après la fermeture du navigateur car considéré comme obsolète.

```
...
//envoyer un cookie au client
Cookie cookie1=new Cookie("cookie1","");
cookie.setMaxAge(-1);
response.addCookie(cookie1);
...
```

2. Les sessions

L'utilisation de sessions HTTP est plus souple que les cookies et offre des possibilités plus intéressantes car nous pouvons stocker des chaînes de caractères mais aussi des objets. Une session permet d'identifier un utilisateur tout au long de sa navigation dans l'application Web. Une session est unique car elle est identifiée par un ID. Contrairement au cookie, une session HTTP n'est pas persistante par défaut. Chaque fois que l'utilisateur visite le site (ouverture et fermeture du navigateur), une nouvelle session est créée. Par contre, elle permet de conserver des données complexes au sein d'un fichier enregistré du côté serveur.

La durée de vie d'une session est paramétrable au niveau du fichier de configuration *web.xml* du serveur Java. En règle générale, la valeur par défaut est de 30 minutes. C'est-à-dire qu'au bout d'une demi-heure d'inactivité sur le site, la session du client est détruite.

Le temps de session peut être paramétré dans le fichier *web.xml* de la façon suivante :

```
...
<!--  définir le temps des sessions (temps en minutes) -->
    <session-config>
        <session-timeout>60</session-timeout>
    </session-config>
...
```

Attention, pour rappel, il est important de bien respecter l'ordre des balises dans le fichier *web.xml*. Les balises relatives aux sessions sont placées après les tags *<servlet-mapping>*.

a. Obtenir une session

L'interface *javax.servlet.http.HttpServletRequest* définit deux méthodes qui permettent d'obtenir une session HTTP. La méthode *getSession()* retourne la session courante et la méthode *getSession(param)* retourne une nouvelle session si la requête ne contient pas déjà de session.

b. Travailler avec une session

L'interface *javax.servlet.http.HttpSession* définit plusieurs méthodes pour manipuler les sessions. La méthode *setAttribute(nom,objet)* permet de stocker un attribut dans le contexte de la session HTTP. Si le nom de l'attribut existe déjà, la valeur existante est remplacée par la nouvelle.

La méthode *getAttribute(nom)* permet de récupérer dans le contexte de la session la valeur d'un attribut ou *null* si l'attribut n'existe pas. La méthode *removeAttribute(nom)* permet de supprimer un attribut dans le contexte de la session. La méthode *isNew()* permet de savoir si la session est nouvelle ou non. Une session est nouvelle tant qu'il n'y a pas eu d'accès. La méthode *invalidate()* permet de détruire immédiatement la session courante et l'ensemble de ses attributs. La méthode *getId()* permet de retourner l'identifiant de la session HTTP.

Nous allons modifier notre Servlet d'authentification pour ajouter dans la session les informations de l'utilisateur et créer une nouvelle Servlet (*ServletLectureSession*) pour lire le contenu des informations de la session.

```
...
        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            //session
```

```
            HttpSession session=request.getSession();
            //si pas de session, destruction et création d'une nouvelle
            if(!session.isNew())
            {
                session.invalidate();
                session=request.getSession();
            }
            //stocker les paramètres de l'utilisateur dans la session
            session.setAttribute("identifiant", identifiant);
            session.setAttribute("motdepasse", motdepasse);

            //redirection vers la page de succès
            response.sendRedirect("authentificationcorrecte.html");
        }
...
```

```
...
    <servlet>
        <servlet-name>servletlecturesession</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
LectureSession</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>servletlecturesession</servlet-name>
        <url-pattern>/lecturesession</url-pattern>
    </servlet-mapping>
...
```

Nous pouvons désormais déclencher l'URL suivante après une authentification correcte afin de lire les données enregistrées dans la session. http://localhost:8080/betaboutique/lecturesession.

```
package betaboutique.servlets.client;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class ServletLectureSession extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //lire les informations de la session
        HttpSession session=request.getSession();
        if(session==null)
        {
            System.out.println("Pas de session");
        }
        else
        {
            System.out.println("Identifiant :
"+session.getAttribute("identifiant"));
            System.out.println("Mot de passe :
"+session.getAttribute("motdepasse"));
        }
    }

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

c. Sessions et réécriture d'URL

Par défaut le mécanisme de sessions utilisateurs utilise un cookie pour identifier l'utilisateur courant. Lorsqu'une session est créée sur le serveur, celui-ci envoie dans la réponse HTTP un cookie avec l'ID du fichier (de la session) créé sur le serveur. Ensuite, à chaque requête envoyée par le client, le cookie est inclus, permettant au serveur de lier la session à un utilisateur précis.

Si l'utilisateur interdit les cookies, la session ne fonctionnera pas car le serveur ne pourra pas récupérer l'ID de la session courante. Il faut donc utiliser une alternative au cookie : la réécriture d'URL.

La technique de réécriture d'URL n'est pas toujours utilisée. Ce procédé est lourd à gérer et impose du travail supplémentaire aux développeurs. Dans la majorité des projets, seule la gestion de session par cookie est utilisée.

Le principe de réécriture consiste à ajouter un paramètre dont la valeur correspond à l'ID de session HTTP du client sur chaque URL (les formulaires HTML, les liens hypertextes, les redirections). Pour cela il est nécessaire d'utiliser les fonctions d'encodage des URL. L'interface *javax.servlet.http.HttpServlet.Response* définit deux méthodes qui permettent d'encoder des URL afin que le serveur puisse les écrire correctement. La méthode *encodeURL(url)* permet d'encoder l'URL passée en paramètre en incluant l'identifiant de session si cela est nécessaire. Cette méthode permet d'ajouter l'identifiant de session au cas où le navigateur ne supporte pas ou n'autorise pas les cookies. La méthode *encodeRedirectURL(url)* est identique et permet la mise en forme lors de redirections avec la méthode *sendRedirect(...)*.

Nous allons modifier notre Servlet d'authentification afin de réaliser un affichage HTML avec un lien vers la Servlet de lecture en conservant l'ID de session dans l'URL.

```
...
        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            //session
            HttpSession session=request.getSession();
            //si pas de session, destruction et création d'une nouvelle
            if(!session.isNew())
            {
                session.invalidate();
                session=request.getSession();
            }
            //stocker les paramètres de l'utilisateur dans la session
            session.setAttribute("identifiant", identifiant);
            session.setAttribute("motdepasse", motdepasse);

            //lien HTML avec ID session
            response.setContentType("text/html");
            out.println("<h1><a href=\""+response.encodeURL("lecturesession")
+"\">Lire le contenu de la session</a></h1>");
        }
...
```

Le serveur d'application détecte automatiquement si le navigateur accepte ou non les cookies. Si le navigateur autorise les cookies, le lien ne sera pas encodé. Par contre, si les cookies sont désactivés (avec la barre de développement Firefox par exemple), le lien est encodé avec l'ID de la session. http://localhost:8080/betaboutique/lecturesession;jsessionid=0C7DD398A9367555D0D4A15F32B5

Les filtres

1. Présentation

Les filtres permettent de donner à une application une structure modulaire. Ils permettent d'encapsuler différentes tâches qui peuvent être indispensables pour traiter des requêtes. La principale fonction d'une Servlet est de recevoir les requêtes et de répondre aux clients concernés. Par contre, il est très souvent nécessaire de réaliser une fonction identique pour chaque Servlet en rapport avec les requêtes et réponses HTTP.

Par exemple, nous voulons stocker dans une base de données pour des statistiques chaque accès à des Servlets du serveur ou router (transporter) un paramètre dans toutes les requêtes sans être obligé d'écrire le code pour chaque Servlet. L'interface *Filter* apparue avec l'API Servlet 2.3 permet de résoudre ce type de problème.

Les filtres permettent ainsi de traiter :

- Les requêtes qui viennent des clients avant qu'elles ne soient traitées par les Servlets.
- Les réponses venant des Servlets avant qu'elles ne soient envoyées aux clients.

Il est possible par exemple de :

- décrypter des requêtes envoyées aux Servlets, traiter les données avec les Servlets et crypter les réponses pour les clients ;
- gérer l'authentification des clients ;
- convertir des formats d'images, appliquer des transformations XSLT sur des données XML...

2. Utilisation

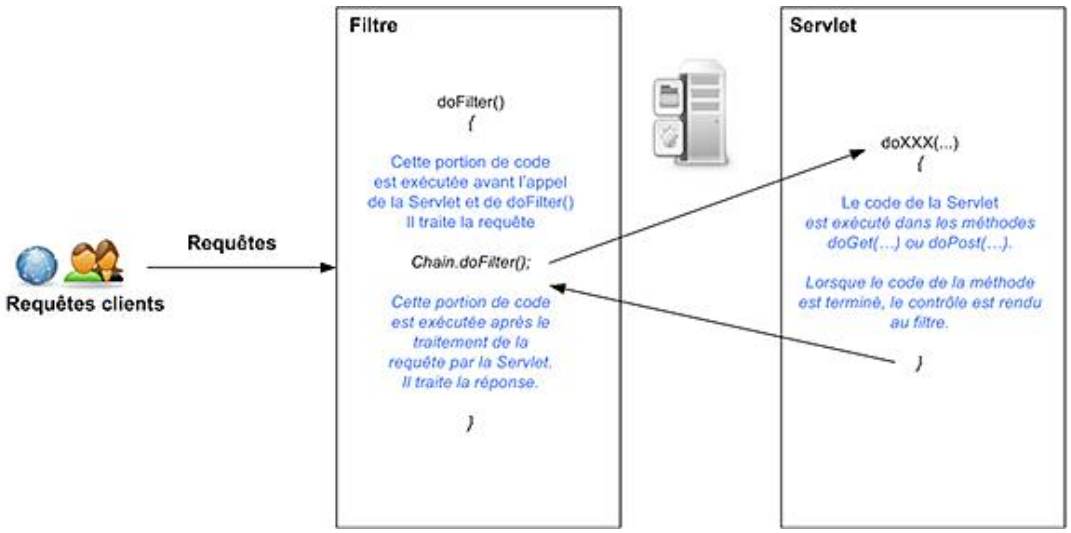
Pour utiliser un filtre il est nécessaire de réaliser deux opérations. La première consiste à écrire une classe qui implémente l'interface *Filter*. La seconde consiste à modifier le descripteur de déploiement (fichier *web.xml*) de l'application pour indiquer au conteneur d'utiliser le filtre.

Nous allons utiliser un filtre pour tracer nos actions au sein de notre application *betaboutique*.

Lorsqu'un filtre est créé, le conteneur appelle sa méthode *init(...)*. Dans cette méthode, nous pouvons accéder aux paramètres d'initialisation avec l'interface *FilterConfig*. Lors du traitement de la requête, le conteneur appelle la méthode *doFilter(...)*. Avant de détruire le filtre, le conteneur appelle sa méthode *destroy(...)*.

Lorsque le filtre appelle *chain.doFilter()*, le filtre suivant dans la chaîne est exécuté. Le code placé avant *chain.doFilter()* est exécuté avant le traitement de la Servlet. Toute modification que le filtre doit apporter avant l'exécution de la requête doit être effectuée avant cet appel. Le code placé après cet appel est exécuté après le traitement de la Servlet. Il est tout à fait possible de chaîner les filtres et d'utiliser un filtre par traitement spécifique.

Le schéma suivant présente le fonctionnement d'un filtre avant et après traitement par la Servlet invoquée.



a. La déclaration du filtre

Le descripteur de déploiement est utilisé pour indiquer le (ou les) filtre(s) qu'il doit appeler pour chaque Servlet ou URL de l'application. Le premier élément *<filter>* permet de déclarer la classe associée au filtre. Cet élément doit être placé en début de fichier de configuration après la déclaration des variables globales au contexte (*<context-param>*).

Dans notre cas, nous définissons un filtre qui sera associé à la classe *FiltreJournalisation* permettant de gérer les accès aux pages du site.

```
...
<!-- definition du filtre -->
<filter>
   <filter-name>filtrejournalisation</filter-name>
   <filter-class>betaboutique.boiteoutils.FiltreJournalisation</filter-class>
</filter>
...
```

Le second élément nécessaire est *<filter-mapping>*. Il permet comme pour les Servlets de gérer le mapping (relations) entre un nom et une Servlet ou une URL. Par exemple, ici le filtre est appliqué uniquement à la Servlet *servletauthentification*.

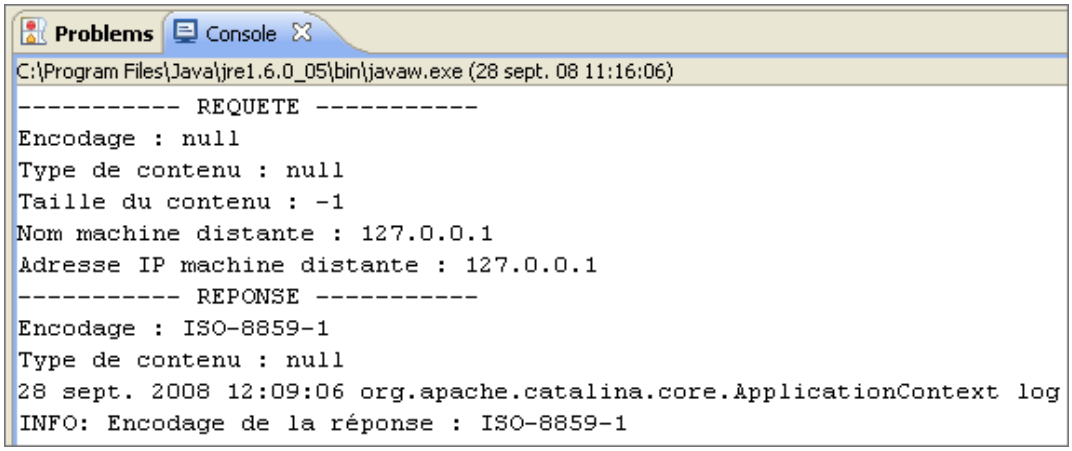
```
...
<filter>
    <filter-name>filtrejournalisation</filter-name>
    <filter-class>betaboutique.boiteoutils.FiltreJournalisation
</filter-class>
<filter-mapping>
    <filter-name>filtrejournalisation</filter-name>
    <servlet-name>servletauthentification</servlet-name>
</filter-mapping>
...
```

Pour ce second exemple, le filtre est appliqué pour l'accès à toutes les URL de l'application.

```
...
<filter>
    <filter-name>filtrejournalisation</filter-name>
    <filter-class>betaboutique.boiteoutils.FiltreJournalisation
</filter-class>
<filter-mapping>
    <filter-name>filtrejournalisation</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
...
```

Le code suivant correspond au filtre *betaboutique.boiteoutils.FiltreJournalisation* et permet d'afficher les informations de journalisation avant et après exécution des pages. Nous pouvons tester le fonctionnement avec l'accès à n'importe quelle URL de l'application étant donné que le paramètre *<url-pattern>* est positionné pour tout écouter (/*).

```
package betaboutique.boiteoutils;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class FiltreJournalisation implements Filter{

    private FilterConfig filterConfig=null;
    // action d'initialisation du filtre
    public void init(FilterConfig filterConfig)
    {
       this.filterConfig=filterConfig;
       System.out.println("Initialisation du filtre :
"+this.filterConfig.getFilterName());
    }
    //action déclenchée lors du traitement d'une requete
    public void doFilter(ServletRequest request,
ServletResponse response, FilterChain chain)
    {
       //traitement de la requête avant la Servlet
       System.out.println("----------- REQUETE -----------");
       System.out.println("Encodage : "+request.getCharacterEncoding());
       System.out.println("Type de contenu : "+request.getContentType());
       System.out.println("Taille du contenu : "+request.getContentLength());
       System.out.println("Nom machine distante : "+request.getRemoteHost());
       System.out.println("Adresse IP machine distante :
"+request.getRemoteAddr());

       try
       {
            //déclencher la servlet appropriée
            chain.doFilter(request, response);
       }
       catch(Exception e)
       {
           //erreur dans le filtre
           System.out.println("Erreur dans le filtre FiltreJournalisation");
       }
       //traitement de la requête avant la Servlet
       System.out.println("----------- REPONSE -----------");
       System.out.println("Encodage : "+response.getCharacterEncoding());
       System.out.println("Type de contenu : "+response.getContentType());
       //exemple d'écriture dans le fichier de log de l'application
       ServletContext context=this.filterConfig.getServletContext();
       context.log("Encodage de la réponse :
"+response.getCharacterEncoding());
    }
    //action qui permet de détruire le filtre
    public void destroy()
    {
       System.out.println("Destruction du filtre :
"+this.filterConfig.getFilterName());
    }
//fin de la classe
}
```



Interface RequestDispatcher

1. Présentation

La plupart du temps, la programmation de Servlets consiste à récupérer des données en paramètre, à traiter ces données et à renvoyer une réponse adaptée au client.

Nous avons utilisé précédemment la méthode *sendRedirect(...)* de l'objet *response* qui permet de rediriger l'utilisateur vers une page précise. Par contre, cette redirection ne permet pas d'envoyer des données directement dans le corps du message (sauf en passant des paramètres à l'URL. Ex : *response.sendRedirect("mapage.jsp?param=1")*).

Un objet de type *javax.servlet.RequestDispatcher* est créé par le conteneur Web pour chaque application et agit comme un routeur de requêtes et réponses HTTP. Un objet de type *javax.servlet.RequestDispatcher* est donc utilisé pour :

- effectuer un traitement continu vers une autre ressource Web ;
- inclure le contenu d'une autre ressource Web dans la réponse HTTP.

Cette interface est la base de l'architecture MVC (Modèle, Vue, Contrôleur) que nous verrons plus tard. La syntaxe de la méthode *getRequestDispatcher(chemin)* doit commencer par un signe / qui permet de faire référence au contexte de l'application Web courante. Ce chemin est complété ensuite vers la ressource statique (HTML, XML, PDF...) ou dynamique (page JSP, une autre Servlet...). Une fois l'objet *javax.servlet.RequestDispatcher* obtenu, il est possible d'invoquer la méthode qui permet de déléguer le traitement de la requête HTTP à la ressource ou la méthode permettant d'inclure du contenu dans la réponse HTTP.

a. Déléguer et transmettre

La délégation consiste à transmettre la requête HTTP courante à une autre ressource Web qui sera chargée de son traitement et qui peut également transmettre la requête à une autre ressource. Pour cela, la méthode *forward(...)* permet de transmettre le contenu complet de la requête (donc avec les paramètres) vers une autre page.

b. Inclure des données

L'inclusion consiste à incorporer le contenu d'une autre ressource Web dans la réponse HTTP qui sera envoyée au client. Cette inclusion peut être utilisée pour l'en-tête ou le pied de page du site en HTML par exemple (ou en JSP s'il y a du contenu dynamique). Pour cela, la méthode *include(...)* permet de réaliser cette opération.

2. Utilisation

Nous allons modifier notre application d'authentification pour le projet *betaboutique*. Dans un premier temps l'utilisateur saisit son identifiant et son mot de passe dans une page HTML simple (*authentification.html*). Si l'authentification est incorrecte ou s'il y a un problème de saisie, l'utilisateur est redirigé vers une page dynamique d'erreur en JSP (le langage JSP sera présenté au chapitre suivant). Si l'authentification est correcte, l'utilisateur est redirigé vers une page dynamique JSP lui indiquant la bienvenue sur la boutique.

a. Explications

L'application proposée est plus complexe que précédemment. La page HTML statique *authentification.html* permet de déclencher la Servlet *ServletAuthentification* par le biais de l'URL *authentificationclient*.

Cette Servlet vérifie la syntaxe de l'identifiant et du mot de passe de façon plus précise avec la valeur *null*, le test de présence et une expression régulière pour la syntaxe. En cas d'erreur, une collection est renseignée puis retournée vers la page JSP dynamique qui affiche simplement le contenu de cette collection comprenant sous forme de chaînes de caractères la liste des erreurs rencontrées.

Le déclenchement de la fonction *forward(...)* de l'objet *javax.servlet.RequestDispatcher* entraîne une redirection (un *return*) et donc le code suivant cette instruction n'est pas exécuté.

En cas de succès de l'authentification, les attributs *identifiant* et *motdepasse* sont passés à la requête qui déclenche le routage vers la page dynamique de bienvenue (*bienvenue.jsp*). Celle-ci va afficher les informations du client. La Servlet dépose donc deux paramètres dans la requête à l'aide de la méthode *setAttribute(...)* et passe le contrôle à

la page *bienvenue.jsp*. La page dynamique *bienvenue.jsp* récupère les attributs dont elle a besoin pour afficher l'identifiant et le mot de passe de l'utilisateur correctement authentifié.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

    <!--  paramètres globaux -->
    <context-param>
        <param-name>emailAdministrateur</param-name>
        <param-value>admin@betaboutique.fr</param-value>
    </context-param>
    <!--  servlets -->
    <servlet>
        <servlet-name>servletauthentification</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
Authentification</servlet-class>
        <init-param>
            <param-name>defautIdentifiant</param-name>
            <param-value>monidentifiant</param-value>
        </init-param>
        <init-param>
            <param-name>defautMotDePasse</param-name>
            <param-value>monmotdepasse</param-value>
        </init-param>
    </servlet>
    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletauthentification</servlet-name>
        <url-pattern>/authentificationclient</url-pattern>
    </servlet-mapping>
</web-app>
```

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletAuthentification extends HttpServlet {

    //variables de classe
    String ident=null;
    String mdp=null;

    public void init()
    {
        //récupérer les paramètres d'initialisation
de la Servlet dans le fichier web.xml
        ServletConfig config=getServletConfig();
        ident=(String)config.getInitParameter("defautIdentifiant");
        mdp=(String)config.getInitParameter("defautMotDePasse");
    }

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer l'identifiant/login dans la requête
        String identifiant=request.getParameter("identifiant");
        //récupérer le mot de passe dans la requête
        String motdepasse=request.getParameter("motdepasse");

        //déclaration d'une liste/collection d'erreurs
        ArrayList<String> erreursParametres=new ArrayList<String>();
```

```
        //pas d'identifiant
        if(identifiant==null)
        {
            erreursParametres.add("Le paramètre [identifiant] est null");
        }
        //identifiant vide
        if(identifiant.equals(""))
        {
            erreursParametres.add("Le paramètre [identifiant] est vide");
        }
        //identifiant inférieur à 5 caractères
        if(!identifiant.matches("^[0-9a-zA-Z]{5,}$"))
        {
            erreursParametres.add("Le paramètre [identifiant]
doit avoir au moins 5 caractères");
        }
        //pas de mot de passe
        if(motdepasse==null)
        {
            erreursParametres.add("Le paramètre [mot de passe] est null");
        }
        //mot de passe vide
        if(motdepasse.equals(""))
        {
            erreursParametres.add("Le paramètre [mot de passe] est vide");
        }
        //motdepasse inférieur à 5 caractères
        if(!motdepasse.matches("^[0-9a-zA-Z]{5,}$"))
        {
            erreursParametres.add("Le paramètre [mot de passe]
doit avoir au moins 5 caractères");
        }

        //en cas d'erreur, redirection avec le RequestDispatcher
vers la page d'erreur dynamique
        if(erreursParametres.size()>0)
        {
            request.setAttribute("erreurs",erreursParametres);
            getServletContext().getRequestDispatcher
("/erreurs.jsp").forward(request, response);
        }

        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            //injecter dans la requête nos paramètres pour
qu'elle puisse les traiter
            request.setAttribute("identifiant",identifiant);
            request.setAttribute("motdepasse",motdepasse);
            //authentification correcte, redirection
vers la page de bienvenue
            getServletContext().getRequestDispatcher
("/bienvenue.jsp").forward(request, response);
        }
    }

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

}
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="java.util.ArrayList" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
```

```
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>ERREURS</title>
</head>
<body>
<%
ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
%>
<h2>Les erreurs suivantes se sont produites</h2>
<ul>
    <%
    for(int i=0;i<erreurs.size();i++)
    {
        out.println("<li>"+(String)erreurs.get(i)+"</li>");
    }
    %>
</ul>
</body>
</html>
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="java.util.ArrayList" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE</title>
</head>
<body>
<%
String identifiant=(String)request.getAttribute("identifiant");
if(identifiant==null)
{
    identifiant="inconnu";
}
String motdepasse=(String)request.getAttribute("motdepasse");
if(motdepasse==null)
{
    motdepasse="inconnu";
}
%>
<h2>Bienvenue client : <%= identifiant %> <%= motdepasse %></h2>
</body>
</html>
```

Le fonctionnement est plus clair désormais. Une Servlet contient la logique applicative pour le traitement et délègue à une ou plusieurs pages dynamiques (JSP, PHP...) la mise en forme des données (police, images, styles). Ceci est la base de la programmation MVC (Modèle Vue Contrôleur) qui repose sur le Design Pattern ou modèle MVC.

Introduction au modèle MVC

1. Présentation

Dans les exemples précédents du projet *betaboutique*, les requêtes HTTP sont gérées par des composants Web qui reçoivent les requêtes, créent les réponses et les retournent aux clients. Il y a donc un seul composant responsable de la logique d'affichage, de la logique métier et de la logique de persistance. Il existe une autre architecture appelée MVC ou Modèle Vue Contrôleur qui permet de séparer clairement les trois activités des composants impliqués.

Dans l'architecture précédente, l'affichage et la manipulation des données sont mélangés dans un seul composant Servlet. Cela peut largement convenir pour un service spécifique, non évolutif et simple, mais cela devient un problème quand le système se développe. Cette architecture conduit à placer du code Java et du code HTML dans les Servlets ou JSP.

Il existe plusieurs solutions à ce problème. La plus simple correspond à l'apparition des pages JSP et consiste à créer des fichiers d'en-tête, de pied de page, de traitement... et d'inclure le tout dans une page générale.

L'architecture MVC sépare la logique métier de l'affichage. Dans ce modèle, un composant est chargé de recevoir les requêtes (Servlets), un autre traite les données (Classes) et un troisième gère l'affichage (JSP). Si l'interfaçage entre ces trois composants est clairement défini, il devient plus simple de modifier un composant sans toucher aux deux autres.

- **Modèle** : le modèle englobe la logique métier et les données sur lesquelles il opère. Toute classe Java qui manipule les données peut jouer le rôle du modèle et de nombreuses applications Web utilisent uniquement des Servlets ordinaires (ou des classes avec JDBC pour manipuler les bases de données).

- **Vue** : dès que la requête courante est traitée, la logique de présentation est réalisée par une vue spécifique.

- **Contrôleur** : le composant (ou les composants) contrôleur reçoi(ven)t les requêtes des clients, les traite(nt) et les transmet(tent) aux composants chargés de traiter les données. Les Servlets sont les composants dont la structure est la plus adaptée. Une Servlet est conçue pour recevoir les requêtes des clients et leur retourner une réponse, ce qui est précisément le rôle du contrôleur.

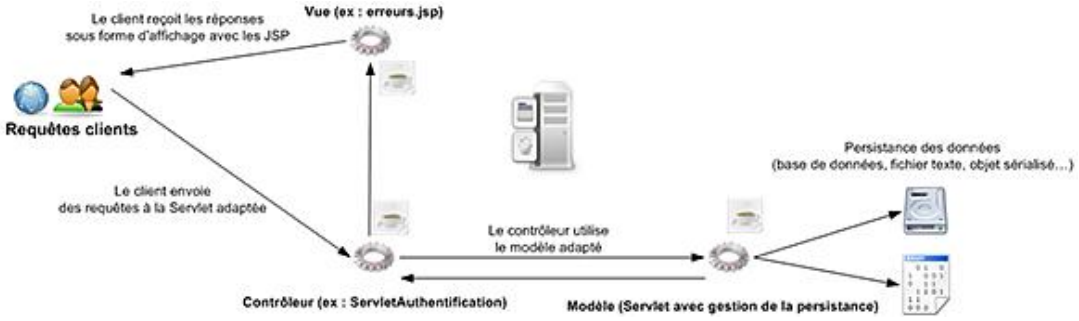
Dans une application comme le projet *betaboutique*, la logique en MVC est la suivante :

Le client émet des requêtes au serveur. Chaque action précise correspond à une Servlet qui redirige les requêtes vers une page JSP adéquate, ou réalise un traitement, ou accède à des données et dans ce cas, déclenche une autre Servlet qui sera chargée de répondre à la demande de l'utilisateur courant.

Le schéma suivant présente une structure de type MVC avec l'utilisation de Servlets et pages JSP.

Pour le moment, nous n'utilisons pas de partie modèle car nous ne manipulons pas de données persistantes comme des enregistrements d'une base de données ou des fichiers.

Notre modèle MVC sera donc limité aux parties : clients, Contrôleur/Action avec les Servlets et Affichage/Vue avec les JSP.



Les Servlets qui jouent le rôle de contrôleur dans une application MVC doivent disposer d'un moyen pour transmettre les requêtes aux composants chargés de l'affichage. Ce moyen est fourni par l'objet *RequestDispatcher*. Ce composant permet de faire suivre une requête d'un composant vers un autre.

Un objet *RequestDispatcher* peut être obtenu avec la méthode *getServletContext()*. À partir de cet objet, il est possible d'obtenir un *RequestDispatcher* à l'aide des méthodes suivantes : *getNamedDispatcher(nom)* ou *getRequestDispatcher (chemin)*.

La méthode *getRequestDispatcher(...)* fonctionne avec un chemin qui commence par la barre oblique et qui est relatif au contexte de l'application. La méthode *getNamedDispatcher(...)* correspond et doit être identique à un sous-élément

<*servlet-name*> d'un élément <*servlet-mapping*> du descripteur de déploiement *web.xml*.

2. Utilisation

Nous allons modifier notre service d'authentification afin de respecter le standard MVC.

a. Les spécifications

Le client utilise une page statique ou dynamique pour s'authentifier (*authentification.html*). Cette page déclenche le contrôleur qui est la Servlet : *ServletAuthentification*. Le contrôleur réalise un traitement simple (vérification de l'authentification) et réalise le routage de la réponse vers la page *bienvenue.jsp* ou vers la page d'erreur *erreurs.jsp* qui sont les vues.

Tout le routage (chemins) est paramétré dans le fichier de configuration de l'application afin de faciliter la maintenance et l'évolutivité du projet.

```
...
<!--  servlets -->
    <servlet>
        <servlet-name>servletauthentification</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
Authentification</servlet-class>
        <init-param>
            <param-name>defautIdentifiant</param-name>
            <param-value>monidentifiant</param-value>
        </init-param>
        <init-param>
            <param-name>defautMotDePasse</param-name>
            <param-value>monmotdepasse</param-value>
        </init-param>
        <!--  url  -->
        <init-param>
            <param-name>urlAuthentification</param-name>
            <param-value>/authentification.html</param-value>
        </init-param>
        <init-param>
            <param-name>urlErreurs</param-name>
            <param-value>/erreurs.jsp</param-value>
        </init-param>
        <init-param>
            <param-name>urlBienvenue</param-name>
            <param-value>/bienvenue.jsp</param-value>
        </init-param>
    </servlet>
...
```

Les routes ou chemins sont définis par des paramètres de la Servlet, cela permet de changer leurs noms sans avoir à recompiler l'application.

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletAuthentification extends HttpServlet {

    //variables de classe
    String ident=null;
    String mdp=null;
    //routes
    String urlAuthentification=null;
    String urlErreurs=null;
```

```
    String urlBienvenue=null;

    public void init()
    {
        //récupérer les paramètres d'initialisation
de la Servlet dans le fichier web.xml
        ServletConfig config=getServletConfig();
        ident=(String)config.getInitParameter("defautIdentifiant");
        mdp=(String)config.getInitParameter("defautMotDePasse");
        //récupérer les routes
        urlAuthentification=(String)config.getInitParameter
("urlAuthentification");
        urlErreurs=(String)config.getInitParameter("urlErreurs");
        urlBienvenue=(String)config.getInitParameter("urlBienvenue");
    }

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer l'identifiant/login dans la requête
        String identifiant=request.getParameter("identifiant");
        //récupérer le mot de passe dans la requête
        String motdepasse=request.getParameter("motdepasse");

        //déclaration d'une liste/collection d'erreurs
        ArrayList<String> erreursParametres=new ArrayList<String>();

        //pas d'identifiant, retour sur la page d'authentification
        if(identifiant==null)
        {
            if(urlAuthentification==null)
            {
                throw new ServletException("Le paramètre
[urlAuthentification] n'a pas été initialisé");
            }
            else
            {

    getServletContext().getRequestDispatcher(urlAuthentification).forward
(request, response);
            }
        return;
        }
        //identifiant vide
        if(identifiant.equals(""))
        {
            erreursParametres.add("Le paramètre [identifiant] est vide");
        }
        //identifiant inférieur à 5 caractères
        if(!identifiant.matches("^[0-9a-zA-Z]{5,}$"))
        {
            erreursParametres.add("Le paramètre [identifiant]
doit avoir au moins 5 caractères");
        }

        //pas de mot de passe
        if(motdepasse==null)
        {
            if(urlAuthentification==null)
            {
                throw new ServletException("Le paramètre
[urlAuthentification] n'a pas été initialisé");
            }
            else
            {

    getServletContext().getRequestDispatcher(urlAuthentification).forward
(request, response);
            }
        return;
```

```
        }
        //mot de passe vide
        if(motdepasse.equals(""))
        {
                erreursParametres.add("Le paramètre [mot de passe]
est vide");
        }
        //motdepasse inférieur à 5 caractères
        if(!motdepasse.matches("^[0-9a-zA-Z]{5,}$"))
        {
            erreursParametres.add("Le paramètre [mot de passe]
doit avoir au moins 5 caractères");
        }

        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            if(urlBienvenue==null)
            {
                throw new ServletException("Le paramètre
[urlBienvenue] n'a pas été initialisé");
            }
            else
            {
                //injecter dans la requête nos paramètres
pour qu'elle puisse les traiter
                request.setAttribute("identifiant",identifiant);
                request.setAttribute("motdepasse",motdepasse);
                //authentification correcte, redirection vers
la page de bienvenue

    getServletContext().getRequestDispatcher(urlBienvenue).forward
(request, response);
            }
        }
        //authentification incorrecte
        else
        {
            erreursParametres.add("Les coordonnées de
l'utilisateur sont incorrectes");
        }

        //en cas d'erreur, redirection avec le
RequestDispatcher vers la page d'erreur dynamique
        if(erreursParametres.size()>0)
        {
            if(urlErreurs==null)
            {
                throw new ServletException("Le paramètre
[urlErreurs] n'a pas été initialisé");
            }
            else
            {
                request.setAttribute("erreurs",erreursParametres);
                getServletContext().getRequestDispatcher(urlErreurs).forward
(request, response);
            }
        }
    }

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

}
```

Désormais, nous pouvons appeler directement la page : http://localhost:8080/betaboutique/authentificationclient.

Le mot de passe et l'identifiant ne sont pas présents dans la requête, le contrôleur va rediriger l'utilisateur vers la page : *urlAuthentification*.

```
...
//pas d'identifiant, retour sur la page d'authentification
    if(identifiant==null)
    {
        if(urlAuthentification==null)
        {
            throw new ServletException("Le paramètre
[urlAuthentification] n'a pas été initialisé");
        }
        else
        {

    getServletContext().getRequestDispatcher(urlAuthentification).forward
(request, response);
        }
    return;
    }
...
```

Si l'utilisateur réalise une mauvaise authentification suite à une erreur de saisie ou de syntaxe ou des coordonnées incorrectes, il est alors redirigé vers la page d'erreur (vue) adaptée.

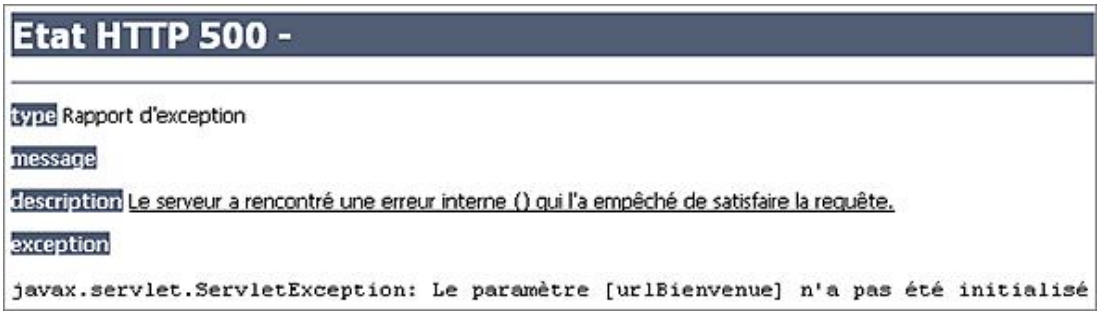
```
...
//en cas d'erreur, redirection avec le RequestDispatcher vers
la page d'erreur dynamique
        if(erreursParametres.size()>0)
        {
            if(urlErreurs==null)
            {
                throw new ServletException("Le paramètre
[urlErreurs] n'a pas été initialisé");
            }
            else
            {
                request.setAttribute("erreurs",erreursParametres);
                getServletContext().getRequestDispatcher
(urlErreurs).forward(request, response);
            }
        }
...
```

Si l'utilisateur réalise une authentification correcte, il est redirigé vers la page de bienvenue.

```
...
//vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            if(urlBienvenue==null)
            {
                throw new ServletException("Le paramètre
[urlBienvenue] n'a pas été initialisé");
            }
            else
            {
                //injecter dans la requête nos paramètres
pour qu'elle puisse les traiter
                request.setAttribute("identifiant",identifiant);
                request.setAttribute("motdepasse",motdepasse);
                //authentification correcte, redirection vers
la page de bienvenue

    getServletContext().getRequestDispatcher(urlBienvenue).forward
(request, response);
            }
        }
```

```
...
```

Nous pouvons essayer de tester notre application avec tous les cas, notamment avec une mauvaise page de routage afin de déclencher volontairement l'exception.



Notre architecture est plus complexe mais respecte le modèle MVC. La logique de traitement (vérification de l'authentification) est réalisée par une Servlet. Le contrôleur qui réalise le routage est également réalisé par une Servlet et l'affichage des données est exclusivement réservé à des pages JSP (code HTML, CSS, JavaScript...).

Nous allons réaliser une dernière modification afin de mettre en œuvre l'architecture MVC au complet avec la partie Modèle. Le modèle permet de gérer la persistance des données, dans notre cas, la Servlet contrôleur *ServletAuthentification* va déclencher une autre Servlet nommée *ServletAuthentificationModele* qui va sauvegarder le client dans la session.

Dans un système évolué, le modèle utilise une base de données avec la technologie JDBC, des fichiers sérialisés ou un framework de persistance comme Hibernate ou JPA.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!--  paramètres globaux -->
    <context-param>
        <param-name>emailAdministrateur</param-name>
        <param-value>admin@betaboutique.fr</param-value>
    </context-param>

    <!--  servlets -->
    <servlet>
        <servlet-name>servletauthentification</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
Authentification</servlet-class>
        <init-param>
            <param-name>defautIdentifiant</param-name>
            <param-value>monidentifiant</param-value>
        </init-param>
        <init-param>
            <param-name>defautMotDePasse</param-name>
            <param-value>monmotdepasse</param-value>
        </init-param>
        <!--  url  -->
        <init-param>
            <param-name>urlAuthentification</param-name>
            <param-value>/authentification.html</param-value>
        </init-param>
        <init-param>
            <param-name>urlErreurs</param-name>
            <param-value>/erreurs.jsp</param-value>
        </init-param>
        <init-param>
            <param-name>urlBienvenue</param-name>
            <param-value>/bienvenue.jsp</param-value>
        </init-param>
        <init-param>
            <param-name>urlAuthentificationModele</param-name>
            <param-value>/authentificationclientmodele</param-value>
        </init-param>
    </servlet>
```

```
    <servlet>
        <servlet-name>servletauthentificationmodele</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
AuthentificationModele</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>bienvenuesession</servlet-name>
        <jsp-file>/bienvenuesession.jsp</jsp-file>
    </servlet>
    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletauthentification</servlet-name>
        <url-pattern>/authentificationclient</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>servletauthentificationmodele</servlet-name>
        <url-pattern>/authentificationclientmodele</url-pattern>
    </servlet-mapping>
</web-app>
```

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletAuthentification extends HttpServlet {

    //variables de classe
    String ident=null;
    String mdp=null;
    //routes
    String urlAuthentification=null;
    String urlErreurs=null;
    String urlBienvenue=null;
    String urlAuthentificationModele=null;

    public void init()
    {
        //récupérer les paramètres d'initialisation
de la Servlet dans le fichier web.xml
        ServletConfig config=getServletConfig();
        ident=(String)config.getInitParameter("defautIdentifiant");
        mdp=(String)config.getInitParameter("defautMotDePasse");
        //récupérer les routes
        urlAuthentification=(String)config.getInitParameter
("urlAuthentification");
        urlErreurs=(String)config.getInitParameter("urlErreurs");
        urlBienvenue=(String)config.getInitParameter("urlBienvenue");

    urlAuthentificationModele=(String)config.getInitParameter
("urlAuthentificationModele");
    }

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

        //vérifier l'égalité des valeurs
        if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
        {
            if(urlBienvenue==null)
            {
```

```
                throw new ServletException("Le paramètre
[urlBienvenue] n’a pas été initialisé");
            }
            else
            {
                //déclencher la Servlet qui permet de gérer
la partie Modèle de l’application
                request.setAttribute("identifiant",identifiant);
                request.setAttribute("motdepasse",motdepasse);

    getServletContext().getRequestDispatcher
(urlAuthentificationModele).forward(request, response);
            }
        }
...
        //authentification incorrecte

}
```

```
package betaboutique.servlets.client;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class ServletAuthentificationModele extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
     //écrire les informations dans la session
     HttpSession session=request.getSession();
     session.setAttribute("identifiant",
request.getAttribute("identifiant"));
     session.setAttribute("motdepasse",
request.getAttribute("motdepasse"));

     //redirection vers la page d’affichage du contenu
de la session (avec un appel par nom par exemple)
     getServletContext().getNamedDispatcher("bienvenuesession").forward
(request, response);
    }

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

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BIENVENUE SESSION</title>
</head>
<body>
<%
String identifiant=(String)session.getAttribute("identifiant");
if(identifiant==null)
{
    identifiant="inconnu";
}
```

```
String motdepasse=(String)session.getAttribute("motdepasse");
if(motdepasse==null)
{
    motdepasse="inconnu";
}
%>
<h2>Bienvenue client : <br/>
Identifiant : <%= identifiant %> <br/>
Mot de passe : <%= motdepasse %> <br/>
</h2>
</body>
</html>
```

Par la suite, les classes de gestion du modèle seront regroupées dans un même paquetage appelé *modele*.

Gestion des exceptions, erreurs et page d'accueil

1. Gestion des exceptions

La gestion des exceptions en Java est très évoluée et permet d'éviter des réponses erronées lors de calculs, de traitements particuliers... Si par exemple dans une Servlet traitant les saisies d'un formulaire, l'utilisateur saisit une chaîne de caractères à la place d'un entier, le constructeur de la classe *Integer(...)* lancera alors une exception.

```
public void doPost(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
{
    String num=request.getParameter("nombre");
    Integer i=new Integer(num);
}
```

Parfois l'utilisateur voit s'afficher en résultat la trace de l'exception dans le navigateur ou n'obtient aucune réponse. Dans tous les cas, l'application est défectueuse. Dans une application professionnelle, il est nécessaire d'effectuer des contrôles côté client et côté serveur. Parfois les contrôles côté client (en JavaScript) ne sont pas utilisés mais les contrôles côté serveur sont obligatoires.

Pour traiter les erreurs de conversions, de calculs, d'accès à des variables ou fichiers, le bloc d'instructions *try catch* est utilisé, il permet d'isoler une partie de code sensible.

Le problème est que si une instruction déclenche une exception, aucun message pour l'utilisateur n'est prévu. La solution peut être de placer des instructions d'affichage dans le bloc *catch* pour retourner une réponse spécifique en cas d'erreur.

```
public void doPost(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
{
    try
    {
        String num=request.getParameter("nombre");
        Integer i=new Integer(num);
    }
    catch(Exception e)
    {
        System.out.println("Exception");
    }
}
```

2. Gestions des pages d'erreurs

Avec Java EE il existe une façon plus robuste de définir les pages d'erreurs. Par exemple, pour le cas précédent, une page Web indiquant un problème de format (Ex : Entrez uniquement des chiffres) pourrait être créée. Il est ensuite possible de configurer l'application toujours par l'intermédiaire de son fichier de gestion *web.xml*. Pour cela, il suffit d'utiliser l'élément *<error-page>* avec l'exception associée.

Les exceptions sont placées après la gestion des sessions et des mappings de Servlets dans le fichier de configuration *web.xml*.

```
...
  <!--  fichier à afficher en cas d'erreur d'entier -->
  <error-page>
      <exception-type>java.lang.NumberFormatException</exception-type>
      <location>/erreurchiffre.html</location>
  </error-page>
...
```

Il peut être également très intéressant "d'attraper" toutes les erreurs sans en gérer à chaque fois le type. Chaque exception levée (conversion, calculs...) déclenchera alors la même page.

```
...
  <!--  fichier à afficher en cas d'erreur -->
  <error-page>
```

```
      <exception-type>java.lang.Throwable</exception-type>
      <location>/erreurglobale.html</location>
  </error-page>
...
```

Nous pouvons vérifier cette utilisation avec notre application d'authentification en modifiant un des paramètres du fichier *web.xml* pour déclencher l'exception suivante :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    ...
    <servlet-mapping>
        ...
    </servlet-mapping>
    <error-page>
          <exception-type>java.lang.Throwable</exception-type>
          <location>/erreurglobale.html</location>
    </error-page>
</web-app>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Erreur BetaBoutique</title>
</head>
<body><h1>Erreur : une erreur est survenue, recommencez votre
opération !</h1></body>
</html>
```

```
...
    //vérifier l'égalité des valeurs
    if( (identifiant!=null && identifiant.equals(ident))
&& (motdepasse!=null && motdepasse.equals(mdp)) )
    {
        if(urlBienvenue==null)
        {
            throw new ServletException("Le paramètre
[urlBienvenue] n'a pas été initialisé");
        }
        else
        {
            //déclencher la Servlet qui permet de gérer la
partie Modèle de l'application
            request.setAttribute("identifiant",identifiant);
            request.setAttribute("motdepasse",motdepasse);

    getServletContext().getRequestDispatcher
(urlAuthentificationModele).forward(request, response);
        }
    }
    //authentification incorrecte
    else
    {
            erreursParametres.add("Les coordonnées de
l'utilisateur sont incorrectes");
    }
...
```

S'il y a une authentification correcte, la page de bienvenue est demandée mais mal orthographiée par erreur pour le développement de l'application, l'exception *ServletException* est déclenchée et provoque l'affichage de la page d'erreur générale. Pour l'utilisateur c'est totalement transparent. Par contre, l'exception est affichée dans la console système.

De la même façon, il est possible de spécifier des pages à afficher pour les codes d'erreur HTTP, par exemple gérer les erreurs 404 (page non trouvée et inexistante sur le serveur).

Pour cela, nous ajoutons le code suivant dans notre fichier de configuration *web.xml*, nous créons une page HTML

d'erreur 404 (ex: *404.html*) et nous déclenchons une URL inexistante sur notre serveur (ex : http://localhost:8080/betaboutique/unepage).

```
...
<error-page>
    <error-code>404</error-code>
    <location>/404.html</location>
</error-page>
...
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Erreur BetaBoutique</title>
</head><body><h1>Erreur : cette page n'est pas disponible !</h1></body>
</html>
```

Cette technique très puissante permet de gérer tous les codes d'erreur HTTP qui sont consultables à cette adresse : http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. Dans une application en production, les erreurs 404 (page non trouvée) et 500 (erreur interne du serveur) doivent systématiquement être traitées. Le but est de toujours fournir au client une page lisible et correctement mise en forme.

3. Gestion de la page d'accueil

Lorsque nous visitons des sites au moyen d'un navigateur, nous précisons uniquement le nom de domaine pour accéder à la page d'accueil. Dans notre cas si nous précisons l'URL suivante qui correspond au contexte, aucune page n'est affichée (erreur 404) (http://localhost:8080/betaboutique/).

Il existe des balises particulières du fichier de configuration qui permettent de gérer le contexte de l'application quand il n'y a pas d'indication de document particulier. Ces documents associés sont appelés des *welcome files*. En règle générale, les serveurs Web sont positionnés avec les pages suivantes : *index.html*, *index.php*, *index.jsp*... Nous allons modifier notre fichier de déploiement pour afficher le formulaire d'authentification lors de l'arrivée sur le site.

L'élément <*welcome-file-list*> doit être placé entre la définition des sessions et les pages d'erreurs dans le fichier de configuration de l'application *web.xml*.

```
...
<!-- fichier de point de départ de l'application -->
<!-- il interdit en plus l'exploration de l'arborescence-
attention, chemin toujours relatif -->
<welcome-file-list>
    <welcome-file>authentification.html</welcome-file>
</welcome-file-list>
...
```

Dans ce cas lors de l'accès à l'application : http://localhost:8080/betaboutique/, le serveur renvoie le premier document trouvé parmi ceux de la liste indiquée dans l'élément <*welcome-file-list*>.

En résumé

Ce chapitre a présenté le mécanisme de Servlet Java EE. La première partie a présenté le fonctionnement du protocole HTTP et l'utilisation des Servlets. La partie suivante a introduit le projet *BetaBoutique* développé tout au long de ce guide et qui permet de mettre en application toutes les technologies évoquées. Le projet commence par une première Servlet simple et la mise en place d'un service d'authentification pour les clients du site. Afin de comprendre le fonctionnement des Servlets, les interfaces *ServletConfig* et *ServletContext* ont été présentées ainsi que le traitement des requêtes et des réponses HTTP avec les méthodes de service *doGet(...)* et *doPost(...)*. Le chapitre suivant fait un rappel sur la synchronisation des traitements avec les Servlets. Ensuite, l'étape suivante a permis de gérer l'état des clients sachant que le protocole HTTP fonctionne en mode déconnecté. Par la suite, les filtres ont été détaillés ainsi que l'interface *RequestDispatcher* qui est utilisée pour le routage HTTP et qui est la base du modèle MVC avec les Servlets. Enfin, l'application a été modifiée dans les deux derniers chapitres pour coller au modèle MVC et pour gérer les erreurs et exceptions afin d'avoir un service fiable et robuste proche d'une application professionnelle.

Travailler avec une base de données

1. Présentation

La plupart des applications Java EE utilisent une base de données pour la persistance des informations. Les sites de commerce comme le projet *BetaBoutique*, stockent les informations concernant les clients, articles ou commandes.

Pour rappel, une base de données est un ensemble de tables organisées pour stocker des données manipulables par une ou plusieurs applications. Une table est un ensemble d'occurrences d'un objet défini par ses propriétés (champs) et dont chaque enregistrement (record) représente une instance (valorisation des champs de cet objet).

La plupart des bases de données commercialisées sont de nature client-serveur et recourent au langage SQL (*Structured Query Language*) pour manipuler les données qu'elles contiennent. Java possède une API pour travailler avec les bases de données. Cette technologie nommée JDBC (*Java DataBase Connectivity*) est une bibliothèque d'interfaces et de classes, utilisée pour accéder aux SGBDR (Système de Gestion de Base de Données Relationnelles). JDBC fournit aux développeurs tous les outils nécessaires pour permettre à des programmes clients de se connecter à des bases de données et de leur envoyer des requêtes. Ces requêtes sont écrites en langage SQL.

Un programme écrit en JDBC envoie à un SGBDR des requêtes écrites en SQL et exploite le résultat retourné en Java.

En effet, ce système crée une abstraction des fonctions d'une base de données sous forme d'ensemble de classes et de méthodes. Le code spécifique à une base de données particulière est contenu dans une bibliothèque appelée pilote ou driver. Si nous utilisons une base de données possédant un pilote JDBC, elle peut être employée avec l'API.

Il existe plusieurs inconvénients à l'utilisation de la technologie JDBC :

- Les instructions sont écrites sous forme de chaînes de caractères et leur exactitude ne peut être vérifiée ni par Java ni par JDBC, mais uniquement par le SGBD au moment de l'exécution.
- L'API JDBC est limitée à l'utilisation de bases de données relationnelles. Il n'existe pas d'implémentation pour des bases de données objets SGBDO (Système de Gestion de Base de Données d'Objets).
- L'utilisation de l'API JDBC nécessite de connaître un minimum du langage SQL afin de réaliser les requêtes.

Les avantages d'utiliser la technologie JDBC sont :

- En cas de changement de SGBDR, il suffit de changer de driver en utilisant celui qui est adapté.
- Une application JDBC peut se connecter à plusieurs SGBDR en même temps.
- Les bibliothèques JDBC reposent sur SQL, elles bénéficient de toutes ses fonctionnalités (sélections, jointures, transactions...).
- La durée d'apprentissage de JDBC est réduite, pour une personne qui connaît le langage SQL.
- L'API JDBC est portable sur la plupart des systèmes actuels.

Par la suite, nous allons utiliser l'API JDBC et ainsi développer des méthodes de manipulation de données persistantes sans nous soucier de la base de données utilisée. Nous utiliserons par exemple, une base de données MySQL et le driver adapté à ce SGBDR. Toutes les requêtes d'interrogation, de lecture, de mise à jour et de suppression seront indépendantes de la base de données. Si nous souhaitons passer à une base de données PostgreSQL en production par exemple, il suffira de changer le driver et d'utiliser celui adapté à PostgreSQL sans retoucher notre code.

Ce système portable est beaucoup plus évolué en Java qu'avec les technologies ASP ou PHP.

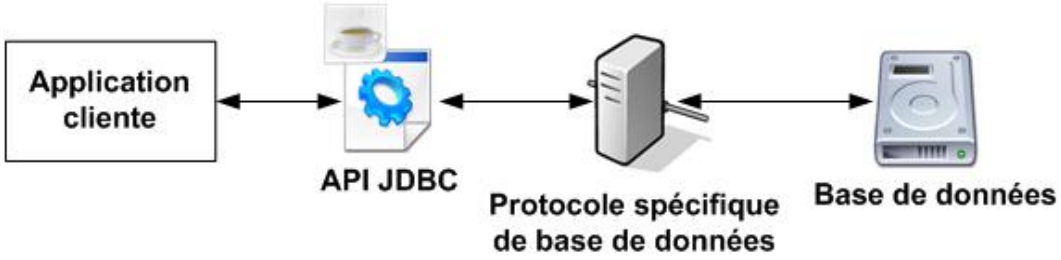
En effet, avec la technologie PHP par exemple, nous utiliserions les méthodes suivantes *mysql_connect(...)*, *mysql_close(...)*, *mysql_fetch_array(...)* qui sont propres au SGBDR MySQL. En cas de passage vers le SGBDR PostgreSQL ou Oracle, il serait nécessaire de coder de nouveau toute l'application.

2. Connexion aux bases de données

La connexion à une base de données par l'intermédiaire de l'API JDBC repose sur plusieurs étapes :

- Il est nécessaire de déterminer le pilote à utiliser pour communiquer avec la base de données.
- Il est nécessaire de réaliser le code Java afin d'établir la connexion avec la base de données.
- Il est nécessaire d'utiliser un objet spécifique pour insérer, mettre à jour ou effacer des données.
- Il est nécessaire d'utiliser un objet spécifique pour lire les résultats d'une requête.

La première étape, consiste à établir la connexion entre le programme et la base de données. Un programme travaillant avec une base de données utilise l'API JDBC pour établir une connexion avec le serveur de base de données.



Le code spécifique à la base de données utilisée est contenu dans le pilote créé par l'éditeur de la base ou une société tierce. Le principal intérêt, est qu'un programme peut communiquer avec différentes bases de données simplement en changeant son pilote. De plus, les programmes sont simples car les détails des procédures de bas niveau sont entièrement gérés par le pilote.

Les pilotes JDBC

La spécification JDBC propose quatre types de pilotes pouvant être employés pour communiquer avec les bases de données.

- **Les pilotes de type 1 appelés, pilote middleware ODBC** : ce type de pilote établit la correspondance entre l'API JDBC et une autre API. Le système ODBC a été développé pour les systèmes d'exploitation Windows. ODBC est une API permettant de communiquer avec les bases de données. Le pilote JDBC-ODBC de type 1 fournit une interface entre les programmes Java et l'API ODBC. Cette couche d'abstraction est composée de sa propre API. Les appels JDBC sont convertis en appels ODBC avant d'être passés à la base de données, ce pilote n'est pas très efficace et nécessite la configuration d'une source de données ODBC qui requiert généralement uniquement la présence d'un serveur Windows.

- **Les pilotes de type 2 appelés, pilote middleware natif** : ce type de pilote est semblable au pilote de type 1 car il communique avec la base de données par l'intermédiaire d'une API native. Un logiciel intermédiaire est conçu spécialement pour se connecter à un pilote JDBC. Ce logiciel intermédiaire appelé middleware, est écrit spécialement pour la plate-forme utilisée. Ce type de pilote entraîne parfois des baisses de performances dans la mesure où le middleware se trouve entre la base de données et JDBC. En outre, il n'existe pas toujours la possibilité de trouver des pilotes correspondant à toutes les plates-formes.

- **Les pilotes de type 3 appelés, pilote middleware Java** : ce type de pilote est similaire aux pilotes de type 2 mais le logiciel exécuté entre JDBC et la base de données est cette fois une application Java. Le pilote communique avec la base de données grâce à un composant intermédiaire. Le programme Java communique avec ce composant par le biais d'un protocole réseau indépendant. Ce pilote est écrit en pur Java, il s'exécute donc partout où Java peut être installé. Il peut donc être téléchargé et exécuté immédiatement, sans configuration utilisateur.

- **Les pilotes de type 4 appelés, pilote natif pur Java** : ce type de pilote se connecte directement à la base de données. Il bénéficie donc, comme le pilote de type 3, de la portabilité de Java. Ce type de pilote léger, communique directement avec la base de données sans qu'une conversion soit nécessaire. Il traduit les appels JDBC en requête utilisant le protocole de la base de données sans passer par ODBC ou une API native.

  Ce type de pilote offre généralement les meilleures performances. La plupart des fournisseurs de bases de données proposent désormais des pilotes de types 2 adaptés comme Oracle, MySQL, PostgreSQL, Sybase, InterBase... Si l'application doit pouvoir être installée sur différentes plates-formes il est pratiquement indispensable d'utiliser un pilote de type 4. Il sera ainsi possible de déployer l'application sur différents systèmes sans modifications.

Le plus souvent, nous utilisons des pilotes de types 3 ou 4. Les pilotes de type 1 et 2 ajoutent une couche de communication entre la couche JDBC et la base de données, ce qui nuit à l'efficacité. En terme de performances, les

pilotes de type 3 et 4 sont pratiquement équivalents.

Les bases de données

MySQL est supportée par un large éventail d'outils. Elle est soumise à la licence GNU GPL. MySQL est surtout installée pour les applications Web, elle est solide et utilisée par de grands groupes spécialisés dans l'Internet. Elle reste cependant parfois limitée en terme de fonctionnalités avancées mais elle est très évoluée en terme de performances. Plusieurs pilotes natifs de type 4 sont disponibles pour MySQL et sont conseillés pour une utilisation en Java.

PostgreSQL est disponible pour la plupart des systèmes d'exploitation modernes. PostgreSQL a été développé à l'université de Berkley en Californie sous la direction d'un groupe de développement et divers autres participants dans le monde entier.

Oracle et **DB2** sont les deux leaders sur le marché des bases de données commerciales. Oracle offre de nombreuses fonctionnalités comme l'intégration de code Java dans les procédures stockées. Ce SGBDR est robuste et très performant. Cependant, cette base de données possède deux inconvénients majeurs, le prix des licences et la complexité du système. Oracle possède un pilote JDBC de type 4 utilisable avec les applications Java. DB2 est un SGBDR qui est développé par la société IBM. Ce système très performant utilise un pilote JDBC de type 4 pour les applications développées en Java.

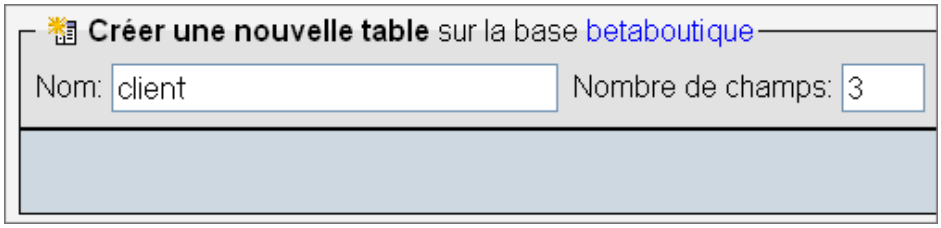
3. Utilisation de l'API JDBC

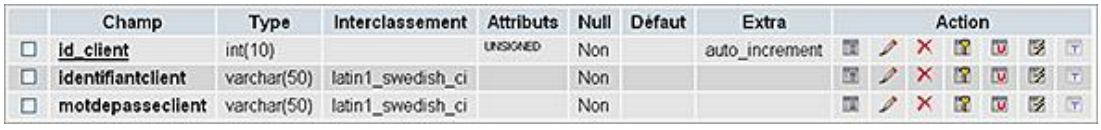
Pour la mise en application de JDBC, nous allons utiliser le SGBDR MySQL disponible à cette adresse : http://www-fr.mysql.com/.

Pour le projet, nous allons installer le paquet EasyPHP (http://www.easyphp.org/) afin de bénéficier de la base de données MySQL (version 5) mais également de l'outil PhpMyAdmin afin de créer facilement les tables, manipuler les données, gérer les encodages...

Nous allons commencer par démarrer MySQL et créer la base de données nommée : *betaboutique*.

Dans cette base de données, nous allons créer la table *client*, avec pour le moment les données suivantes :





	Champ	Type	Interclassement	Attributs	Null	Défaut	Extra	Action
☐	id_client	int(10)		UNSIGNED	Non		auto_increment	
☐	identifiantclient	varchar(50)	latin1_swedish_ci		Non			
☐	motdepasseclient	varchar(50)	latin1_swedish_ci		Non			

	id_client	identifiantclient	motdepasseclient
☐	1	jlafosse	jerome

La classe *DriverManager* est responsable de la gestion des pilotes JDBC. Cette classe permet de fournir les connexions au code Java. Pour utiliser une base de données, il suffit de passer au gestionnaire de pilotes *DriverManager* une URL afin que celui-ci retourne une connexion. Lorsqu'un programme demande une connexion, le gestionnaire de pilotes

interroge chaque pilote pour savoir s'il est capable de traiter l'URL. Dès qu'un pilote correct est trouvé, il lui demande d'établir une connexion et la retourne au programme appelant.

Installation du driver de la base de données

Quelle que soit la base de données utilisée avec JDBC, il faut installer son pilote pour qu'elle puisse fonctionner avec la programmation Java. Chaque driver est spécifique à la base de données utilisée. Pour MySQL, le connecteur utilisé dans ce guide est *mysql-connector-java-3.1.11-bin.jar*. Il peut être téléchargé à cette adresse : http://dev.mysql.com/doc/refman/5.0/fr/java-connector.html

Cette librairie au format JAR contient les classes qui seront utilisées par les applications Java, afin de fonctionner avec le SGBDR MySQL. Une fois cette bibliothèque téléchargée, nous l'installons dans le répertoire */lib* de Tomcat. Ce répertoire contient les librairies utilisées et partagées par le serveur d'applications. Cette librairie sera sans doute utilisée par plusieurs projets, il est donc conseillé de l'installer dans le répertoire partagé. Toutefois, cette opération n'est pas obligatoire, et nous pouvons installer le pilote JDBC comme une autre librairie (installation dans le répertoire */WEB-INF/lib* de l'application).

Pour faire fonctionner une application avec JDBC, il faut effectuer les tâches suivantes dans l'ordre indiqué, quelle que soit la base de données utilisée :

- Chargement du pilote/driver de la base de données.
- Obtention de la connexion à la base de données.
- Préparation de la requête d'accès à la base de données.
- Accès aux données de la base de données.
- Libération des ressources/connexions.

Pour mettre en application JDBC avec la table *client* de la base *betaboutique*, nous allons développer la Servlet : *ServletListeClientModele*.

Chargement du pilote/driver de la base de données

Le chargement du driver est effectué avec l'instruction *Class.forName(nomdudriver)*.

Pour notre projet, nous allons utiliser la configuration suivante :

```
...
<servlet>
   <servlet-name>servletlisteclientmodele</servlet-name>
   <servlet-class>betaboutique.servlets.client.ServletListe
ClientModele</servlet-class>
</servlet>
<!-- mapping des servlets -->
<servlet-mapping>
    <servlet-name>servletlisteclientmodele</servlet-name>
    <url-pattern>/listeclient</url-pattern>
</servlet-mapping>
...
```

La méthode *forName(String)* de la classe *java.lang.Class* permet d'indiquer à la JVM qu'elle doit trouver, charger et initialiser la classe désignée par le paramètre. Le programme appelant n'a donc pas besoin de créer une instance de la classe. La classe se charge de créer une instance et s'occupe elle-même de l'enregistrement. Le paramètre est le nom du driver indiqué par le fournisseur pour la base de données concernée. Ce paramètre correspond au nom de la classe qui sera chargée en mémoire par *Class.forName(...)*. Pour MySQL ce paramètre est : *com.mysql.jdbc.Driver*.

L'appel de cette méthode peut provoquer une exception de type *ClassNotFoundException* que nous devons attraper avec un bloc d'exception *try catch*.

Eclipse permet de déclarer automatiquement les blocs d'exceptions. Pour cela, il faut faire un clic sur la croix rouge qui représente l'erreur et sélectionner : *surround*.

Le code de chargement du pilote, présent dans la Servlet *ServletListeClientModele*, est donc le suivant :

```
package betaboutique.servlets.client;
```

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class ServletListeClientModele extends HttpServlet {

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

        try {
            Class.forName("com.mysql.jdbc.Driver");
            System.out.println("Pilote MySQL JDBC chargé");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.out.println("Erreur lors du chargmement du pilote");
        }

    }

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

Obtention de la connexion à la base de données

Une fois le driver chargé avec succès, nous pouvons utiliser l'API JDBC pour obtenir une connexion à la source de données. La connexion au SGBDR nécessite un nom d'utilisateur et un mot de passe. Les connexions peuvent être obtenues à partir du pilote. La méthode permettant d'obtenir une connexion est *getConnection(...)*.

Il existe plusieurs formes de cette méthode, les plus utilisées sont : *getConnection(String url)* et *getConnection(String url, String utilisateur, String motdepasse)*

Le paramètre URL est une URI HTTP avec l'adresse IP du serveur contenant la base de données (en local ou à distance) et son nom.

Pour notre projet l'instruction est de la forme suivante :

```
package betaboutique.servlets.client;
import java.io.IOException;
import java.sql.*;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class ServletListeClientModele extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        Connection connection=null;
        try {
            //chargement du driver
            Class.forName("com.mysql.jdbc.Driver");
            System.out.println("Pilote MySQL JDBC chargé");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.out.println("Erreur lors du chargmement du pilote");
        }
        try {
            //obtention de la connexion
            connection = DriverManager.getConnection
("jdbc:mysql://localhost:3306/betaboutique","root","");
```

```
            System.out.println("Connexion opérationnelle");
        } catch (SQLException e) {
            e.printStackTrace();
            System.out.println("Erreur lors de
l'établissement de la connexion");
        }
    }

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

Dans notre exemple, nous avons précisé le protocole *jdbc*, le serveur de base de données présent sur la machine locale (*localhost*), le port MySQL (*3306*) et la base de données utilisée (*betaboutique*). Le nom d'utilisateur est *root* et cet utilisateur ne possède pas de mot de passe.

Il existe également une troisième et dernière forme de la méthode *getConnection(String url, Properties proprietes)* avec une URL et un ensemble de propriétés représentées par un objet instancié à partir de la classe *Properties*. Cet objet permet de passer des paramètres supplémentaires comme le type de connexion, le cryptage utilisé, le codage de la base... La connexion ouverte est utilisable tout le temps, pour toutes les requêtes futures dans le temps, tant que celle-ci n'est pas fermée.

Préparation de la requête d'accès à la base de données

Deux cas sont envisageables pour l'accès aux données. Soit la requête est une sélection de données (sélection), soit une autre action (création, modification ou suppression). Dans le premier cas, la liste des données est renvoyée et la requête exécutée (*executeQuery*), dans le second cas, le nombre d'enregistrements affectés est retourné et la modification de l'état des données (*executeUpdate*) est exécutée.

SELECT : *ResultSet rs = statement.executeQuery(requete);*

AUTRE : *int nblignes = statement.executeUpdate(requete);*

Accès aux données de la base de données

L'accès aux données est réalisé à travers l'objet *ResultSet* obtenu lors de l'appel à *statement.executeQuery(requete)*. Nous verrons par la suite un exemple complet d'utilisation.

Libération des ressources/connexions

Lors du développement, il n'est pas rare d'arriver à la saturation de l'application à cause d'un problème de fermeture de connexion suite à une montée en charge. Il est important de vérifier avec des outils adaptés (fournis avec la base de données) si le nombre d'ouvertures et le nombre de connexions libérées sont identiques. Si ces nombres divergent, même lentement, il y a alors un problème de programmation. L'application va finir par se bloquer, dans l'attente de connexions qui n'arriveront jamais. Il est donc très important en Java EE de libérer correctement les connexions.

La simple invocation de la méthode *close()* de l'objet *connexion* permet la fermeture.

Il est également nécessaire de libérer les ressources mémoire occupées par les objets *Statement* pour la préparation des requêtes et les objets *ResultSet* pour le parcours de ces requêtes. Pour pallier à certaines erreurs de programmation, le ramasse-miettes va se charger de fermer correctement certaines ressources mais n'importe quand. Nous allons montrer un exemple de chargement du driver, de connexion au SGBDR et de fermeture avec notre Servlet.

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.sql.*;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class ServletListeClientModele extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        Connection connection=null;
        try {
```

```
//chargement du driver
            Class.forName("com.mysql.jdbc.Driver");
            System.out.println("Pilote MySQL JDBC chargé");
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            System.out.println("Erreur lors du chargement du pilote");
        }

        try {
            //obtention de la connexion
            connection = DriverManager.getConnection
("jdbc:mysql://localhost:3306/betaboutique","root","");
            System.out.println("Connexion opérationnelle");

            //obtenir des informations sur la base de données
            DatabaseMetaData dbmd = connection.getMetaData();
            System.out.println("Nom de la base de données :
"+dbmd.getDatabaseProductName());
            System.out.println("Version de la base de données :
"+dbmd.getDatabaseProductVersion());
            System.out.println("Nom du pilote de la
base de données : "+dbmd.getDriverName());
            System.out.println("Version du pilote de la
base de données : "+dbmd.getDriverVersion());
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            System.out.println("Erreur lors de
l'établissement de la connexion");
        }

        //fermer la connexion
        try {
            connection.close();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

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

```
Pilote MySQL JDBC chargé
Connexion opérationnelle
Nom de la base de données : MySQL
Version de la base de données : 5.0.27-community-log
Nom du pilote de la base de données : MySQL-AB JDBC Driver
Version du pilote de la base de données : mysql-connector-java-3.1.11
```

Ce code est fonctionnel mais il contient une énorme erreur de programmation. Les méthodes JDBC peuvent lancer des exceptions SQL. Le problème c'est que si une telle exception est lancée, avec la méthode *getConnection()*, un *Statement* ou un *ResultSet*, l'exception liée à la fermeture de la connexion ne sera pas déclenchée. Dans notre exemple, si la connexion est réalisée mais qu'une exception est déclenchée à la suite, le code suivant ne sera jamais déclenché :

```
...
//fermer la connexion
try {
connection.close();
} catch (SQLException e) {
e.printStackTrace();
}
```

```
...
```

La façon correcte d'utiliser la méthode *close()*, consiste à placer cette instruction dans le bloc *finally* d'une structure *try*, *catch*, *finally*. Cette partie du code sera toujours déclenchée quoi qu'il arrive, qu'une exception soit levée ou pas. La Servlet doit donc avoir le code suivant :

```
...

   try
   {
        //obtention de la connexion
        connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/
betaboutique","root","");
        System.out.println("Connexion opérationnelle");

        //obtenir des informations sur la base de données
        DatabaseMetaData dbmd = connection.getMetaData();
        System.out.println("Nom de la base de données :
"+dbmd.getDatabaseProductName());
        System.out.println("Version de la base de données :
"+dbmd.getDatabaseProductVersion());
        System.out.println("Nom du pilote de la base de données :
"+dbmd.getDriverName());
        System.out.println("Version du pilote de la base de données :
"+dbmd.getDriverVersion());
   }
   catch (SQLException e)
   {
        e.printStackTrace();
        System.out.println("Erreur lors de l'établissement de la connexion");
   }
   finally
   {
        //fermer la connexion
        try
        {
             connection.close();
        } catch (SQLException e)
        {
             e.printStackTrace();
        }
   }
...
```

L'interface Statement

Nous avons vu dans le code précédent comment établir une connexion avec une base de données. Toutefois, l'objet *connection* ne fournit pas de méthodes permettant d'utiliser la base de données. Pour créer, lire, mettre à jour ou effacer des données, nous utilisons la classe *Statement*.

Les objets *Statement* représentent l'interface principale avec la base de données. Un objet de cette classe est créé par l'appel de la méthode *createStatement()* de l'objet *Connection*. Une fois en possession d'un objet *Statement*, nous pouvons l'utiliser pour envoyer à la base de données des commandes SQL à l'aide d'une des trois méthodes suivantes :

- *executeQuery(requête SQL)* : exécute la requête SELECT pour interroger la base de données, et retourne un *ResultSet* pour parcourir le résultat.
- *executeUpdate(requête SQL)* : exécute la requête SQL pour provoquer une modification de l'état de la base de données (création, modification ou suppression) et retourne le nombre d'enregistrements concernés.
- *executeBatch()* : exécute un traitement par lot de commandes SQL en une seule opération.

L'interface ResultSet

Lorsque nous réalisons une requête sur une base de données, les résultats sont retournés dans un objet de type *ResultSet*. Cet objet permet de lire les lignes du résultat et de lire les données dans les différentes colonnes. Un objet de cette classe est créé suite à l'appel de l'une des deux méthodes :

- *Statement.executeQuery(requête SQL)* ;
- *PreparedStatement.executeQuery()*.

Un objet *ResultSet* contient le résultat de la requête SELECT effectuée vers la base de données. Les méthodes de l'objet permettent de parcourir les lignes obtenues par la requête et d'extraire les champs de ces lignes.

En supposant qu'il n'y ait pas eu de problème lors de l'exécution de la commande SQL, la méthode *executeQuery()* retourne toujours un *ResultSet* non nul. Au départ, le curseur est positionné avant la première ligne. La méthode *next ()* permet d'atteindre la première ligne, cette méthode retourne *true* s'il y a une valeur et *false* lors de l'accès à la dernière ligne.

Les méthodes suivantes peuvent être employées avec un *ResultSet* :

- *next()* : pour passer à la ligne suivante du *ResultSet*.
- *previous()* : pour passer à la ligne précédente.
- *first()* : pour se positionner à la première ligne.
- *last()* : pour se positionner à la dernière ligne.
- *beforeFirst()* : pour se positionner avant la première ligne.
- *afterLast()* : pour se positionner après la dernière ligne.
- *isFirst()* : pour tester si le positionnement est à la première ligne.
- *isLast()* : pour tester si le positionnement est à la dernière ligne.
- *isBeforeFirst()* : pour tester si le positionnement est avant la première ligne.
- *isAfterLast()* : pour tester si le positionnement est après la dernière ligne.
- *absolute(index)* : pour se positionner à l'index précisé.
- *close()* : pour fermer le *ResultSet*.

L'interface *ResultSet* contient de nombreuses méthodes permettant de lire les données retournées par une requête. Ces méthodes correspondent aux colonnes par leur nom ou par leur position. Il existe deux méthodes *getXxx()* pour toutes les primitives Java et pour certains objets. La première méthode prend en argument une valeur entière correspondant à l'indice du champ de la table, et l'autre une chaîne de caractères correspondant au nom du champ de la table.

Les numéros de colonnes commencent à 1 et non à 0. Le choix d'un argument de type *int* est plus rapide qu'un élément de type *String* mais également moins souple. En effet, les numéros de colonnes peuvent changer alors que c'est plus rarement le cas des noms. Nous retrouvons les méthodes suivantes de l'interface *ResultSet* : *getString(...)*, *getArray(...)*, *getBigDecimal(...)*, *getBoolean(...)*, *getByte(...)*, *getDate(...)*, *getDouble(...)*, *getFloat(...)*, *getInt(...)*, *getLong (...)*, *getShort(...)*, *getTime(...)*.

La valeur *NULL* a une signification particulière en SQL. En effet, *NULL* n'est pas la même chose que la chaîne de caractères vide ou la valeur 0 pour un numérique. *NULL* signifie qu'aucune donnée n'a été définie pour la valeur du champ. Pour toutes les méthodes retournant un objet, la méthode retourne *NULL* si la colonne contient *NULL* et 0 pour les primitives numériques. Pour les types booléens, la valeur retournée est *FALSE*.

Si par exemple nous utilisons *getFloat()* et que la valeur de retour est 0 comment savoir si la colonne contient *NULL* ou 0 ? L'interface *ResultSet* propose alors une méthode permettant d'obtenir cette information : *public boolean wasNull()*.

Cette méthode ne prend pas d'indication de colonne pour argument, elle agit sur la dernière colonne lue. Voici un exemple complet d'utilisation des interfaces *Statement* et *ResultSet* pour lire les clients de la base de données.

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.sql.*;
```

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class ServletListeClientModele extends HttpServlet {


    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        Connection connection=null;
        Statement st=null;
        ResultSet rs=null;
        try
        {
            //chargement du driver
            Class.forName("com.mysql.jdbc.Driver");
            System.out.println("Pilote MySQL JDBC chargé");
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors du chargement du pilote");
        }

        try
        {
            //obtention de la connexion
            connection = DriverManager.getConnection
("jdbc:mysql://localhost:3306/betaboutique","root","");
            System.out.println("Connexion opérationnelle");

            //lire les données de la table client
            st=connection.createStatement();
            rs=st.executeQuery("SELECT * FROM client");
            //affichage
            while(rs.next())
            {
                System.out.println("Client :
"+rs.getInt("id_client")+"-"+rs.getString("identifiantclient")
+"-"+rs.getString("motdepasseclient"));
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
        }
        finally
        {
            //fermer la connexion
            try
            {
                rs.close();
                st.close();
                connection.close();
            } catch (SQLException e)
            {
                e.printStackTrace();
            }
        }
    }

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

```
    }
}
```

Il faut noter dans le code précédent que les ressources sont toujours libérées dans l'ordre inverse de leur création (*rs*, *st*, *connection*). La connexion est ouverte en premier et fermée en dernier, par exemple. Il faut prendre pour habitude de reprendre le code et de le fermer.

Le code précédent fonctionne mais il n'est pas correct. Si le déclenchement de la fonction *rs.close()* provoque une erreur, l'exception est attrapée (*SQLException*) et les lignes *st.close()* et *connection.close()* ne seront pas déclenchées. Résultat, la connexion ne sera pas fermée. La solution consiste à lancer un bloc *try catch* pour chaque ouverture de ressource. Ainsi, une exception lancée dans la méthode n'empêchera pas l'appel de la méthode *close()* de la connexion.

Cette erreur de fermeture est très courante en Java même avec d'excellents développeurs. Rares sont les livres ou cours qui traitent cette erreur et il est très courant de trouver la fermeture des ressources dans le même bloc *try catch*.

Pour résoudre ce problème, il faut entourer chaque fermeture d'un bloc *try catch*. Le code est ainsi exécuté qu'il y ait une exception ou pas.

```
...
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
        }
        finally
        {
            if(rs!=null)
            {
                try
                {
                    rs.close();
                }
                catch(Exception e)
                {
                    e.printStackTrace();
                }
            }

            if(st!=null)
            {
                try
                {
                    st.close();
                }
                catch(Exception e)
                {
                    e.printStackTrace();
                }
            }

            if(connection!=null)
            {
                try
                {
                    connection.close();
                }
                catch(Exception e)
                {
                    e.printStackTrace();
                }
            }
...
```

Le code est désormais correct, par contre, la programmation est lourde et elle nécessite des enchaînements de blocs *try catch*. Pour avoir un code plus lisible et moins verbeux, il faut utiliser une classe statique (c'est-à-dire que les instances de cette classe manipulent les mêmes paramètres) qui va correctement fermer les ressources. Cette classe nommée *OutilsBaseDeDonnees* est placée dans le paquetage *betaboutique.boiteoutils*.

Une classe statique n'a pas besoin d'être instanciée pour pouvoir appeler ses méthodes. Cette classe n'a d'ailleurs pas besoin de constructeur. Dans ce genre de classe, plusieurs ''instances'' manipulent les mêmes paramètres partagés. Il faut utiliser alors le nom de la classe directement à la place d'une instance. À la différence, un singleton ne possède qu'une seule instance de la classe. De même, comme toute classe non statique, elle doit obligatoirement être instanciée.

Voici le code de la classe *OutilsBaseDeDonnees* avec les méthodes statiques génériques (technique de surcharge des signatures de méthodes).

```
package betaboutique.boiteoutils;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

public class OutilsBaseDeDonnees
{
/*********************************************************
 * fermer correctement un resultset
*********************************************************/
public static void fermerConnexion(ResultSet rs)
{
    if(rs!=null)
    {
        try
        {
            rs.close();
        }
        catch(Exception e)
        {
        System.out.println("Erreur lors de la fermeture
d'une connexion dans fermerConnexion(ResultSet)");
        }
    }
}

/*********************************************************
 * fermer correctement un statement
*********************************************************/
public static void fermerConnexion(Statement stmt)
{
    if(stmt!=null)
    {
        try
        {
            stmt.close();
        }
        catch(Exception e)
        {
        System.out.println("Erreur lors de la fermeture
d'une connexion dans fermerConnexion(Statement)");
        }
    }
}

/*********************************************************
 * fermer correctement une connection
*********************************************************/
public static void fermerConnexion(Connection con)
{
    if(con!=null)
    {
        try
        {
            con.close();
        }
        catch(Exception e)
        {
```

```
        System.out.println("Erreur lors de la fermeture
d'une connexion dans fermerConnexion(Connection)");
        }
    }
}

//fin de la classe
}
```

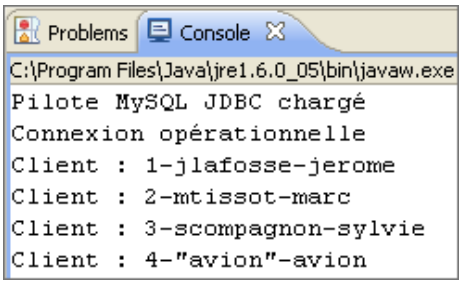
Le code précédent est modifié avec l'utilisation de la classe statique *OutilsBaseDeDonnees* afin de fermer correctement les ressources. Le code est plus léger et plus sûr.

```
...
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de
l'établissement de la connexion");
        }
        finally
        {
            OutilsBaseDeDonnees.fermerConnexion(rs);
            OutilsBaseDeDonnees.fermerConnexion(st);
            OutilsBaseDeDonnees.fermerConnexion(connection);
        }
    }
...
```

Cette Servlet permettant de lister les clients de la boutique fonctionne correctement. Maintenant, nous allons ajouter les enregistrements suivants dans la table *client* de la base de données *betaboutique*.

	id_client	identifiantclient	motdepasseclient
	1	jlafosse	jerome
	2	mtissot	marc
	3	scompagnon	sylvie
	4	"avion"	avion

Nous relançons à nouveau la Servlet afin de vérifier que le parcours du *ResultSet* est opérationnel.



Nous allons maintenant modifier légèrement notre Servlet afin de réaliser une requête pour simuler par exemple, une recherche d'un client depuis l'interface d'administration.

```
...
try
{
    //obtention de la connexion
    connection = DriverManager.getConnection
("jdbc:mysql://localhost:3306/betaboutique","root","");
    System.out.println("Connexion opérationnelle");
    //client a rechercher
    String recherche="mtissot";
    //lire les données de la table client
    st=connection.createStatement();
    rs=st.executeQuery("SELECT * FROM client WHERE
identifiantclient=\""+recherche+"\"");
```

```
      //affichage
    while(rs.next())
    {
    System.out.println("Client : "+rs.getInt("id_client")
+"-"+rs.getString("identifiantclient")+"-"+rs.getString
("motdepasseclient"));
    }
}
...
```

La Servlet est exécutée sans problème et produit le résultat souhaité. Nous allons réaliser une recherche sur l'identifiant *''avion''*. Dans un cas réel, cela pourrait correspondre à une recherche précise, par exemple une description d'un article (ex : *''le choix des armes''*).

```
...
try
{
        //obtention de la connexion
        connection = DriverManager.getConnection
("jdbc:mysql://localhost:3306/betaboutique","root","");
        System.out.println("Connexion opérationnelle");
        //client a rechercher (les slashes permettent d'échapper les guillemets)
        String recherche="\"avion\"";
        //lire les données de la table client
        st=connection.createStatement();
        rs=st.executeQuery("SELECT * FROM client WHERE
identifiantclient=\""+recherche+"\"");

        //affichage
        while(rs.next())
        {
         System.out.println("Client : "+rs.getInt("id_client")
+"-"+rs.getString("identifiantclient")+"-"+rs.getString
("motdepasseclient"));
        }
}
...
```

Cette requête provoque une erreur. Le terme de la recherche contient des guillemets qui empêchent le fonctionnement de la méthode. L'erreur affichée, de type *java.sql.SLQException* indique un problème de syntaxe près de *'avion''''*. En effet, JDBC reçoit en fait, la requête incorrecte suivante : *SELECT * FROM client WHERE identifiantclient='''';*.

Une solution serait bien sûr, de développer une méthode qui permette de réaliser des remplacements de caractères, ajouter des échappements ou remplacer les quotes comme dans d'autres langages (*addslashes()* de PHP par exemple), mais il y aura forcément des cas qui ne fonctionneront pas et cela nécessiterait un code lourd à gérer (échappement des caractères avant chaque insertion et après chaque lecture). En effet, avec ce genre de méthodes, notre identifiant *''avion''* serait alors manipulé sous la forme suivante *\"avion\"*.

Les requêtes SQL sont complexes à écrire sous forme de *String* du fait du mélange de la syntaxe SQL et des variables du programme. Pour pallier cet inconvénient, il existe une interface *PreparedStatement* dérivant de *Statement* qui permet de formater les requêtes SQL de façon beaucoup plus simple.

L'interface PreparedStatement

L'interface *PreparedStatement* hérite de *Statement*. Toutes les méthodes de cette interface, déclenchent l'exception *SQLException* de la classe *java.sql*. Les méthodes de cette interface permettent de répondre à des problèmes de portabilité, de compatibilité des bases, de temps d'exécution et de performance.

Par exemple, lors d'un développement sous le SGBD MySQL, les guillemets doubles sont acceptés, mais une base de données comme Sybase n'accepte que les guillemets simples. D'ailleurs, sur les forums du Web la question des guillements en SQL figure souvent : *''Comment puis-je insérer le titre ''l'alpagueur'' dans ma base de données sans erreur ?''*.

Une réponse courante de développeur non expérimenté serait de doubler les guillemets ou d'utiliser un antislash. Nous aurions alors, *''l\'alpagueur''* sous MySQL mais cela ne fonctionnerait pas sous Sybase et PostgreSQL.

Par exemple, sous PostgreSQL nous aurions alors *''l '' alpagueur''* (deux guillemets simples). Il est visible que cette technique est lourde en terme de développement pour le respect des compatibilités. JDBC offre une couche d'abstraction supplémentaire pour manipuler les bases de données, chaque constructeur gérant les guillemets et quotes avec son pilote.

Enfin, la plupart des bases de données évoluées gardent en mémoire cache les commandes SQL récemment

exécutées. Si une commande qui est en mémoire cache est utilisée, le résultat sera plus rapide car celle-ci est compilée et optimisée. Toutefois, il est nécessaire que la commande envoyée corresponde exactement à celle se trouvant en cache. Supposons l'exemple suivant avec deux commandes :

```
INSERT INTO client VALUES ('mtissot','marc');
INSERT INTO client VALUES ('adurand','alain');
```

Ces deux commandes sont presque identiques puisqu'elles ne diffèrent que par les valeurs littérales. Du point de vue du SGBD, les deux commandes sont quand même totalement distinctes. Si par contre, nous passons à la base de données des requêtes identiques puis qu'après, nous déclenchons des commandes comportant des variables, alors la même requête est exécutée dans les deux cas. C'est ce que permet l'interface *PreparedStatement*.

```
INSERT INTO client VALUES (?,?);
setString(1,'mtissot');
setString(2,'marc');
INSERT INTO client VALUES (?,?);
setString(1,'adurant');
setString(2,'alain');
```

La création d'un *PreparedStatement* est pratiquement identique à celle d'un objet *Statement*. La différence se situe au niveau de la précision de la commande à exécuter par le SGBD. Une fois l'objet *PreparedStatement* créé, avant que la commande SQL puisse être exécutée, les marqueurs (*?*) doivent être remplacés.

Les méthodes ci-dessous permettent de fournir des valeurs aux différents paramètres. Les paramètres sont numérotés à partir de 1, dans l'ordre d'apparition des symboles ?. Nous retrouvons alors des méthodes de la forme *setXxx()* où *Xxx* correspond au type de données Java.

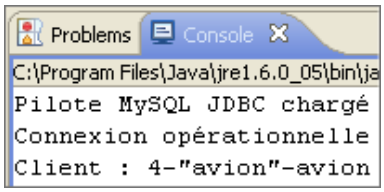
- *setString(indice,chaine)* : permet d'affecter une chaîne de caractères au paramètre d'indice.
- *setInt(indice,entier)* : permet d'affecter un entier au paramètre d'indice.
- *setBoolean(indice,booléen)* : permet d'affecter un booléen au paramètre d'indice.
- *setFloat(indice,float)* : permet d'affecter un réel au paramètre d'indice.
- ...

Comme dans le cas des *Statements*, la méthode *executeQuery()* permet d'exécuter des requêtes SQL de type SELECT (qui retournent une liste de données) et la méthode *executeUpdate()* permet d'exécuter des requêtes de modification de l'état de la base de données (qui retournent le nombre d'enregistrements concernés). Les méthodes *getConnection ()* et *close()* permettent respectivement d'accéder à la connexion et de fermer celle-ci.

Nous allons modifier notre exemple précédent, non fonctionnel, avec un *PreparedStatement* compilé.

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.sql.*;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import betaboutique.boiteoutils.OutilsBaseDeDonnees;

@SuppressWarnings("serial")
public class ServletListeClientModele extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        Connection connection=null;
        PreparedStatement requete=null;
        ResultSet rs=null;

        try
        {
            //chargement du driver
            Class.forName("com.mysql.jdbc.Driver");
```

```
            System.out.println("Pilote MySQL JDBC chargé");
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors du chargmement du pilote");
        }

        try
        {
            //obtention de la connexion
            connection = DriverManager.getConnection
("jdbc:mysql://localhost:3306/betaboutique","root","");
            System.out.println("Connexion opérationnelle");
    //preparation de la requete
    requete=connection.prepareStatement("SELECT * FROM client
WHERE identifiantclient=?");
    requete.setString(1, "\"avion\"");
            //executer la requete
            rs=requete.executeQuery();
            //affichage
            while(rs.next())
            {
                System.out.println("Client : "+rs.getInt
("id_client")+"-"+rs.getString("identifiantclient")+"-"+
rs.getString("motdepasseclient"));
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
        }
        finally
        {
            OutilsBaseDeDonnees.fermerConnexion(rs);
            OutilsBaseDeDonnees.fermerConnexion(requete);
            OutilsBaseDeDonnees.fermerConnexion(connection);
        }
    }

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

}
```



La requête fonctionne maintenant parfaitement avec une chaîne de caractères composée de quotes ou guillemets, il n'y a plus de soucis de compatibilité entre les SGBD.

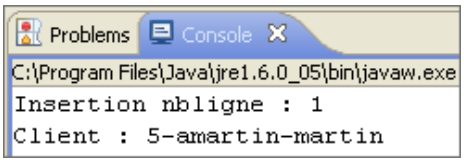
L'instruction *setString(1,"\"avion\"");* indique de remplacer le premier *'?'* dans la requête, par la chaîne de caractères *''avion''*. La requête, beaucoup plus lisible, est compilée par le SGBD, d'où un gain de temps d'exécution appréciable. De plus, la requête suivante est compilée : *SELECT * FROM client WHERE identifiantclient=?*, c'est-à-dire que le prochain appel, avec juste le changement de l'identifiant de l'utilisateur, sera exécuté très rapidement.

En effet, Java toujours soucieux de la portabilité, utilise un pilote pour chaque base mais par contre développe un code générique qui sera ensuite interprété par le pilote adapté. Ainsi, il n'y aura plus besoin d'utiliser des méthodes spécifiques à un SGBD et des échappements de chaînes plus ou moins sûrs.

Nous allons modifier notre Servlet afin d'insérer également un nouveau client lors du déclenchement de la Servlet. Cet

exemple permet de nous montrer que le principe est identique pour une insertion et une lecture, seules les fonctions *executeQuery()* et *executeUpdate()* changent.

```
...
    try
    {
        //obtention de la connexion
        connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/betaboutique",
"root","");
        //on insère un nouvel enregistrement
        requetea=connection.prepareStatement("INSERT INTO
client(identifiantclient,motdepasseclient) VALUES (?,?)");
        requetea.setString(1, "amartin");
        requetea.setString(2, "martin");
        int resinstrution=requetea.executeUpdate();
        System.out.println("Insertion nbligne : "+resinstrution);

        //preparation de la requete
        requeteb=connection.prepareStatement("SELECT * FROM client WHERE
identifiantclient=?");
        requeteb.setString(1, "amartin");

        //executer la requete
        rs=requeteb.executeQuery();
        //affichage
        while(rs.next())
        {
          System.out.println("Client : "+rs.getInt("id_client")+"-
"+rs.getString("identifiantclient")+"-"+rs.getString("motdepasseclient"));
        }
    }
    catch (SQLException e)
    {
          e.printStackTrace();
          System.out.println("Erreur lors de l'établissement de la
connexion");
    }
    finally
    {
          OutilsBaseDeDonnees.fermerConnexion(rs);
          OutilsBaseDeDonnees.fermerConnexion(requeteb);
          OutilsBaseDeDonnees.fermerConnexion(requetea);
          OutilsBaseDeDonnees.fermerConnexion(connection);
    }
...
```



	id_client	identifiantclient	motdepasseclient
	1	jlafosse	jerome
	2	mtissot	marc
	3	scompagnon	sylvie
	4	"avion"	avion
	5	amartin	martin

Le résultat de l'instruction *executeUpdate()* est très important, il permet de réaliser des tests. Dans notre cas par exemple, si la variable *resinstruction* est égale à 0 cela indique qu'aucune ligne ne sera modifiée dans la base de données (ici créée), il y aura donc une erreur.

Cette technique d'utilisation d'une connexion, d'un objet *Statement* ou *PreparedStatement* et d'un objet *ResultSet* peut

paraître complexe, mais elle est en fait très naturelle et permet d'avoir une certaine rigueur dans le code de l'application.

- Utilisation en premier d'une connexion pour dialoguer avec la base de données.
- Utilisation ensuite d'un objet *Statement* ou *PreparedStatement* (de préférence celui-ci) pour dialoguer avec la base de données et exécuter des requêtes.
- Déclenchement des requêtes avec les fonctions *executeQuery()* ou *executeUpdate()*.
- Parcourir au besoin le résultat de la requête avec un objet *ResultSet*.

Pour la suite, nous utiliserons de préférence des requêtes compilées avec l'interface *PreparedStatement*.

Insertion de valeurs nulles

Pour insérer une valeur nulle avec les *PreparedStatement*, il ne suffit pas de mettre la valeur vide pour ce marqueur, cela cause le lancement d'une *SQLException* par le pilote JDBC. Pour l'éviter, nous devons utiliser une des deux méthodes suivantes :

- *setNull(int,int)*
- *setNull(int,int,String)*

Le premier paramètre correspond à la position du marqueur. Le second paramètre est défini dans la classe *java.sql.Types*. Cette classe contient des constantes pour les types JDBC. Si par exemple, nous souhaitons affecter la valeur *NULL* à une colonne de type *String*, nous utilisons *java.sql.Types.STRING*. La version de la méthode avec un troisième paramètre de type *String* est utilisée pour des types définis par les utilisateurs.

Partage de connexions

1. Présentation

Les exemples précédents permettent de présenter l'utilisation des bases de données en Java, mais ne sont pas envisageables dans un projet de grande envergure. En effet, le code n'est pas très clair, une connexion est créée à chaque appel, les paramètres de connexion sont dans la Servlet et les codes de traitement et d'accès aux données se chevauchent.

Nous allons donc modifier notre exemple et la table *client* afin de réaliser des accès partagés, des recherches et des affichages adaptés.

2. Initialisation d'une connexion

Le code de la Servlet *ServletListeClientModele* n'est pas conforme. La connexion doit être ouverte dans la méthode *init()* et fermée dans la méthode *destroy()*. Le code conforme au standard Java EE est présenté ci-dessous :

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.sql.*;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import betaboutique.boiteoutils.OutilsBaseDeDonnees;

@SuppressWarnings("serial")
public class ServletListeClientModele extends HttpServlet {

    Connection connection=null;
    PreparedStatement requete=null;
    ResultSet rs=null;

    //initialisation de la connexion
    public void init()
    {
        try
        {
            //chargement du driver
            Class.forName("com.mysql.jdbc.Driver");
            System.out.println("Pilote MySQL JDBC chargé");
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors du chargement du pilote");
        }

        try
        {
            //obtention de la connexion
            connection = DriverManager.getConnection
("jdbc:mysql://localhost:3306/betaboutique","root","");
            System.out.println("Connexion opérationnelle");
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
        }
    }
```

```
      //traitements
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        try
        {
            //preparation de la requete
            requete=connection.prepareStatement("SELECT * FROM
client WHERE identifiantclient=?");
            requete.setString(1, "amartin");

            //executer la requete
            rs=requete.executeQuery();
            //affichage
            while(rs.next())
            {
                System.out.println("Client : "+rs.getInt
("id_client")+"-"+rs.getString("identifiantclient")+"-"+
rs.getString("motdepasseclient"));
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
        }
    }

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

    //fermeture des ressources
    public void destroy(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        try
        {
            //fermeture
            System.out.println("Connexion fermée");
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
        }
        finally
        {
            OutilsBaseDeDonnees.fermerConnexion(rs);
            OutilsBaseDeDonnees.fermerConnexion(requete);
            OutilsBaseDeDonnees.fermerConnexion(connection);
        }
    }
}
```

Cette Servlet est opérationnelle, le chargement des ressources est géré dans la méthode *init()* au démarrage de la Servlet, et la libération des ressources dans la méthode *destroy()*.

Nous allons légèrement modifier cette Servlet afin de positionner les paramètres de connexion dans le fichier de configuration de l'application (*web.xml*). Lors du changement de la base de données ou des paramètres de connexion, le code ne sera ainsi pas retouché et fonctionnera normalement. Le fichier *web.xml* est volontairement simplifié pour ne traiter que les exemples JDBC.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
```

```
<web-app>
    <!--  paramètres globaux -->
    <context-param>
        <param-name>pilotejdbc</param-name>
        <param-value>com.mysql.jdbc.Driver</param-value>
    </context-param>
    <context-param>
        <param-name>urlconnexionjdbc</param-name>
        <param-value>jdbc:mysql://localhost:3306/betaboutique</param-value>
        /context-param>
    <context-param>
        <param-name>utilisateurjdbc</param-name>
        <param-value>root</param-value>
    </context-param>
    <context-param>
        <param-name>motdepassejdbc</param-name>
        <param-value></param-value>
    </context-param>
    <!--  servlets -->
    <servlet>
        <servlet-name>servletlisteclientmodele</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
ListeClientModele</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>servletlisteclientmodele</servlet-name>
        <url-pattern>/listeclient</url-pattern>
    </servlet-mapping>
</web-app>
```

```
...

    //initialisation de la connexion
    public void init()
    {
        ServletContext servletContext=getServletContext();
        pilotejdbc=(String)servletContext.getInitParameter
("pilotejdbc");
        urlconnexionjdbc=(String)servletContext.getInit
Parameter("urlconnexionjdbc");
        utilisateurjdbc=(String)servletContext.getInit
Parameter("utilisateurjdbc");
        motdepassejdbc=(String)servletContext.getInit
Parameter("motdepassejdbc");

        try
        {
            //chargement du driver
            Class.forName(pilotejdbc);
            System.out.println("Pilote MySQL JDBC chargé");
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors du chargmement du pilote");
        }

        try
        {
            //obtention de la connexion
            connection = DriverManager.getConnection
(urlconnexionjdbc,utilisateurjdbc,motdepassejdbc);
            System.out.println("Connexion opérationnelle");
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
```

```
        }
    }
...
```

```
Pilote MySQL JDBC chargé
Connexion opérationnelle
```

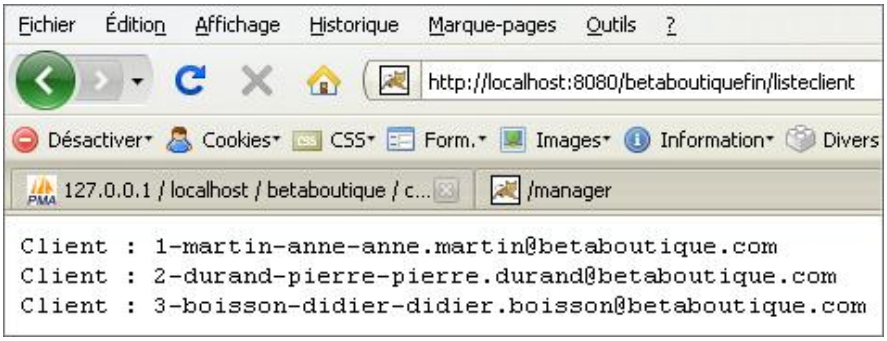
Nous allons modifier la table *client* et ajouter tous les champs nécessaires aux traitements (la structure respecte la maquette du chapitre : Les Servlets). Pour cela, nous pouvons supprimer la table actuelle et utiliser le code suivant qui va créer la nouvelle structure et ajouter quelques enregistrements.

```
--
-- Structure de la table `client`
--

CREATE TABLE `client` (
  `id_client` int(10) unsigned NOT NULL auto_increment,
  `identifiantclient` varchar(50) NOT NULL default '',
  `motdepasseclient` varchar(50) NOT NULL default '',
  `nomclient` varchar(50) NOT NULL default '',
  `prenomclient` varchar(50) NOT NULL default '',
  `emailclient` varchar(255) NOT NULL default '',
  `telephoneclient` varchar(15) NOT NULL default '',
  `datenaissanceclient` int(8) NOT NULL default '0',
  `adresseclient` varchar(255) NOT NULL default '',
  `villeclient` varchar(255) NOT NULL default '',
  `codepostalclient` varchar(10) NOT NULL default '',
  `paysclient` varchar(50) NOT NULL default '',
  PRIMARY KEY  (`id_client`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=4 ;

--
-- Contenu de la table `client`
--

INSERT INTO `client` VALUES (1, 'amartin', 'anne2de', 'martin',
'anne', 'anne.martin@betaboutique.com', '0384465958', 19821209,
'4 rue du haut', 'Poligny', '39800', 'france');
INSERT INTO `client` VALUES (2, 'pdurand', 'p8fas',
'durand', 'pierre', 'pierre.durand@betaboutique.com',
'0348759869', 19760204, '2 rue du bas', 'Lons-le-Saunier',
'39000', 'france');
INSERT INTO `client` VALUES (3, 'dboisson', 'doia9dz',
'boisson', 'didier', 'didier.boisson@betaboutique.com',
'0345968574', 19681212, '3 rue de la fontaine', 'L''étoile',
'39570', 'france');
```

Nous allons modifier notre Servlet *ServletListeClientModele* afin d'afficher la liste complète des clients et créer une nouvelle Servlet nommée *ServletListeIdentifiantClientModele*, qui va afficher tous les identifiants et mots de passe des utilisateurs. De même, les Servlets afficheront désormais les résultats dans une page Web grâce au *PrintWriter*.

```
...
    try
    {
       //flux de sortie vers le navigateur
       PrintWriter out=response.getWriter();

       //preparation de la requete
       requete=connection.prepareStatement("SELECT * FROM client ORDER BY
client.id_client");
       //executer la requete
       rs=requete.executeQuery();
       //affichage
       while(rs.next())
       {
            out.println("Client : "+rs.getInt("id_client")+"-
"+rs.getString("nomclient")+"-"+rs.getString("prenomclient")+"-"+rs.getString
("emailclient"));
       }
```

```
    }
    catch (SQLException e)
    {
       e.printStackTrace();
       System.out.println("Erreur lors de l'exécution de la requete");
    }
 }

...
```



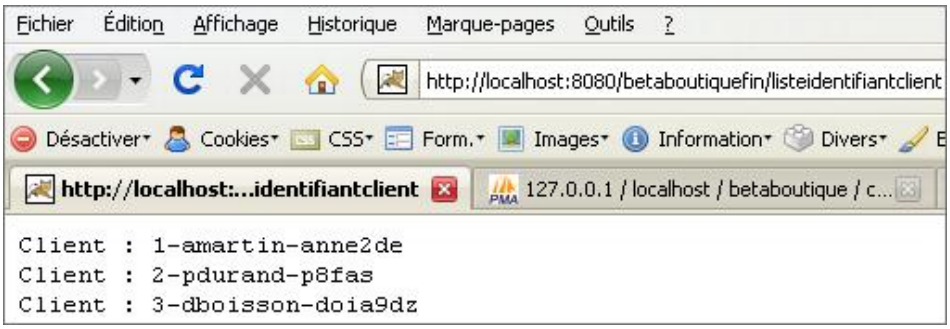
Le code de la nouvelle Servlet *ServletListeIdentifiantClientModele* est présenté ci-dessous. Le fichier *web.xml* contient la déclaration de cette Servlet.

```
...
<!--  servlets -->
<servlet>
        <servlet-name>servletlisteclientmodele</servlet-name>
    <servlet-class>betaboutique.servlets.client.ServletListe
ClientModele</servlet-class>
</servlet>
<servlet>
  <servlet-name>servletlisteidentifiantclientmodele
</servlet-name>
  <servlet-class>betaboutique.servlets.client.ServletListe
IdentifiantClientModele</servlet-class>
</servlet>
<!-- mapping des servlets -->
<servlet-mapping>
    <servlet-name>servletlisteclientmodele</servlet-name>
    <url-pattern>/listeclient</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>servletlisteidentifiantclientmodele
</servlet-name>
    <url-pattern>/listeidentifiantclient</url-pattern>
</servlet-mapping>
...
```

```
package betaboutique.servlets.client;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.*;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import betaboutique.boiteoutils.OutilsBaseDeDonnees;

@SuppressWarnings("serial")
public class ServletListeIdentifiantClientModele extends
HttpServlet {

    Connection connection=null;
    PreparedStatement requete=null;
```

```
    ResultSet rs=null;
    //parametres de connexion
    String pilotejdbc=null;
    String urlconnexionjdbc=null;
    String utilisateurjdbc=null;
    String motdepassejdbc=null;

    //initialisation de la connexion
    public void init()
    {
        ServletContext servletContext=getServletContext();
        pilotejdbc=(String)servletContext.getInitParameter
("pilotejdbc");
        urlconnexionjdbc=(String)servletContext.getInit
Parameter("urlconnexionjdbc");
        utilisateurjdbc=(String)servletContext.getInit
Parameter("utilisateurjdbc");
        motdepassejdbc=(String)servletContext.getInit
Parameter("motdepassejdbc");

        try
        {
            //chargement du driver
            Class.forName(pilotejdbc);
            System.out.println("Pilote MySQL JDBC chargé");
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors du chargmement du pilote");
        }

        try
        {
            //obtention de la connexion
            connection = DriverManager.getConnection
(urlconnexionjdbc,utilisateurjdbc,motdepassejdbc);
            System.out.println("Connexion opérationnelle");
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
        }
    }

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

        try
        {
            //flux de sortie vers le navigateur
            PrintWriter out=response.getWriter();

            //preparation de la requete
            requete=connection.prepareStatement("SELECT
id_client,identifiantclient,motdepasseclient FROM client
ORDER BY client.id_client");

            //executer la requete
            rs=requete.executeQuery();
            //affichage
            while(rs.next())
            {
                out.println("Client : "+rs.getInt
("id_client")+"-"+rs.getString("identifiantclient")+"-"+
```

```
rs.getString("motdepasseclient"));
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'exécution de la requete");
        }
    }

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

    //fermeture des ressources
    public void destroy(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        try
        {
            //fermeture
            System.out.println("Connexion fermée");
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
        }
        finally
        {
            OutilsBaseDeDonnees.fermerConnexion(rs);
            OutilsBaseDeDonnees.fermerConnexion(requete);
            OutilsBaseDeDonnees.fermerConnexion(connection);
        }
    }
}
```



Ces deux exemples sont corrects et affichent les résultats souhaités. Par contre, le code est lourd. Il faut déclarer la méthode *init()* et la méthode *destroy()* dans chaque Servlet afin de gérer une connexion à la base de données. Une solution serait de déclarer la connexion dans une seule des Servlets. Ensuite, il faudrait déclarer une variable de type objet qui serait insérée dans le descripteur de déploiement (*web.xml*) pour le partage.

Le code serait alors semblable à celui-ci :

```
...
connection = DriverManager.getConnection(urlconnexionjdbc,
utilisateurjdbc,motdepassejdbc);
ServletContext servletContext=servlet.getServletContext();
servletContext.setAttribute("connection",connection);
...
```

La variable *connection* est ainsi partagée et pourra être réutilisée par les autres Servlets avec le code suivant :

```
...
ServletContext servletContext=servlet.getServletContext();
connection = (Connection)servletContext.getAttribute("connection");
...
```

Cette technique très utile présente un problème. Quelle est la Servlet qui doit lancer la création de la connexion à la ressource ? Avec cette technique, la Servlet de connexion doit être la première appelée afin d'initialiser la connexion. Il serait alors possible de déclencher cette Servlet de connexion avec le paramètre <*welcome-file*> du fichier de configuration de l'application *web.xml*.

Cette technique bien que fonctionnelle n'est pas très pratique et surtout illogique du point de vue de la programmation dans le cas où l'utilisateur n'accède pas directement à la page d'accueil du site et ne déclenche donc pas le fichier déclaré dans la balise <*welcome-file*>.

Nous allons aborder dans le paragraphe suivant une solution à ce problème, les écouteurs.

Écouteurs/listeners et cycle de vie

1. Présentation

Les événements étudiés jusqu'ici affectent tous la Servlet, puisqu'ils appellent des méthodes sur elle. Le conteneur de Servlets gère plusieurs opérations intéressantes. Si un événement nous intéresse, nous créons une classe qui implémente l'interface d'écoute de celui-ci et nous l'enregistrons avec le moteur de Servlets. La spécification 2.3 des Servlets présente quatre interfaces d'écoute d'événements. Ces interfaces sont relativement simples et aucune ne requiert l'implémentation de plus de trois méthodes.

2. Utilisation

L'interface javax.servlet.ServletContextListener

L'interface *ServletContextListener* écoute l'un des deux événements liés au contexte : lors de sa création et lors de sa destruction. Cette interface sert à gérer l'initialisation et l'arrêt des services sous-jacents, tels que les connexions à des bases de données à partager par toutes les Servlets de l'application.

Voici un exemple d'une classe nommée *InitialisationContext* présente dans le paquetage boiteoutils qui implémente cette interface, avec la première méthode *contextInitialized(...)* qui est déclenchée au chargement de l'application, et la seconde méthode *contextDestroyed(...)*, déclenchée lors de la suppression ou l'arrêt de l'application.

```
package betaboutique.boiteoutils;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;


public class InitialisationContext implements ServletContextListener{


    //action déclenchée lors du chargement du context
    public void contextInitialized(ServletContextEvent event)
    {
        System.out.println("----------- Context initialisé -----------");
    }


    //action qui permet de détruire le filtre
    public void contextDestroyed(ServletContextEvent event)
    {
        System.out.println("----------- Context détruit -----------");
    }

//fin de la classe
}
```

Chaque méthode est appelée lorsque l'événement spécifié se produit et un objet d'événement est transmis en paramètre. La deuxième étape consiste à installer cette classe écouteur dans le descripteur de déploiement. Pour installer un écouteur au sein de l'application Web, nous déclarons cet écouteur en début de fichier *web.xml* après les variables <*context-param*>.

Chaque classe d'écoute est simplement déclarée dans un élément <*listener-class/*>. Une fois le fichier correctement installé, et lorsque le contexte est redémarré, l'écouteur propre au contexte est déclenché.

```
...
<!--  écouteurs -->
<listener>
      <listener-class>betaboutique.boiteoutils.InitialisationContext
</listener-class>
</listener>
...
```

```
---------- Context détruit ----------
14 oct. 2008 15:27:49 org.apache.catalina.c
INFO: HTMLManager: list: Listing contexts f
14 oct. 2008 15:27:55 org.apache.catalina.c
INFO: HTMLManager: start: Starting web appl
14 oct. 2008 15:27:57 org.apache.catalina.c
INFO: The listener "betaboutique.boiteutil
---------- Context initialisé ----------
```

Cette technique est donc idéale pour charger des ressources ou déclencher des traces ou fichiers journaux lors du chargement des applications.

L'interface javax.servlet.ServletContextAttributeListener

Parmi les autres écouteurs, il existe l'interface *javax.servlet.ServletContextAttributeListener* qui permet de savoir quand les attributs de contexte sont ajoutés, supprimés ou remplacés. Cette interface déclare trois méthodes qui sont *attributeAdded()*, *attributeRemoved()* et *attributeReplaced()*.

L'interface javax.servlet.http.HttpSessionListener

L'interface *javax.servlet.http.HttpSessionListener* écoute les événements liés à la création et à la destruction des sessions. Cette interface déclare deux méthodes qui sont *sessionCreated()* et *sessionDestroyed()* et qui correspondent à la création et destruction des sessions.

L'interface javax.servlet.http.HttpSession.AttributeListener

Enfin, la quatrième et dernière interface est *javax.servlet.http.HttpSession.AttributeListener* qui écoute les événements d'ajout, de suppression ou de modification d'attributs de session. Cette interface déclare alors trois méthodes : *attributeAdded()*, *attributeRemoved()* et *attributeReplaced()*, chacune prend comme paramètre une instance de l'événement *HttpSessionBindingEvent*.

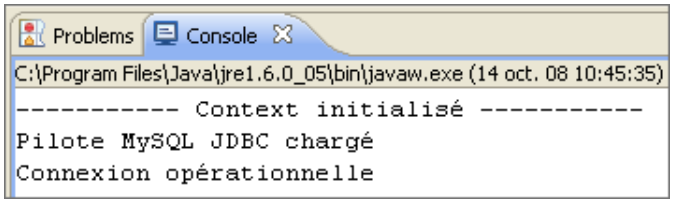
3. Mise en place d'une connexion partagée

Notre classe *betaboutique.boiteoutils.InitialisationContext* sera donc déclenchée au chargement de l'application avant le premier déclenchement de Servlet, JSP ou pages. Nous allons créer notre connexion au SGBD dans cette classe et partager cette connexion grâce au descripteur de déploiement. Dans la méthode de destruction du contexte, nous allons mettre le code de libération des ressources. Nous aurons ainsi une seule connexion proprement partagée par l'ensemble des pages de l'application.

```
package betaboutique.boiteoutils;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class InitialisationContext implements ServletContextListener{

    //parametres de connexion
    Connection connection=null;
    String pilotejdbc=null;
    String urlconnexionjdbc=null;
    String utilisateurjdbc=null;
    String motdepassejdbc=null;

    //action déclenchée lors du chargement du context
    public void contextInitialized(ServletContextEvent event)
    {
        System.out.println("----------- Contexte initialisé -----------");

        //lire le contexte
        ServletContext servletContext=event.getServletContext();
        pilotejdbc=(String)servletContext.getInitParameter
("pilotejdbc");
```

```
        urlconnexionjdbc=(String)servletContext.getInit
Parameter("urlconnexionjdbc");
        utilisateurjdbc=(String)servletContext.getInit
Parameter("utilisateurjdbc");
        motdepassejdbc=(String)servletContext.getInit
Parameter("motdepassejdbc");

        try
        {
            //chargement du driver
            Class.forName(pilotejdbc);
            System.out.println("Pilote MySQL JDBC chargé");
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors du chargmement du pilote");
        }

        try
        {
            //obtention de la connexion
            connection = DriverManager.getConnection
(urlconnexionjdbc,utilisateurjdbc,motdepassejdbc);
            //sauvegarder la connexion dans le context
            servletContext.setAttribute("connection",connection);
            System.out.println("Connexion opérationnelle");
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'établissement
de la connexion");
        }
    }

    //action qui permet de détruire le filtre
    public void contextDestroyed(ServletContextEvent event)
    {
        System.out.println("----------- Contexte détruit -----------");
        try
        {
            //fermeture
            System.out.println("Connexion fermée");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            OutilsBaseDeDonnees.fermerConnexion(connection);
        }
    }

//fin de la classe
}
```

Au chargement de notre application *betaboutique*, la connexion est alors déposée dans le contexte. Nous pouvons désormais l'utiliser avec le code très simple suivant présent dans la méthode *init()* de la Servlet *ServletListeClient*.



```
package betaboutique.servlets.client;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.*;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class ServletListeClientModele extends HttpServlet {

    Connection connection=null;
    PreparedStatement requete=null;
    ResultSet rs=null;

    //initialisation de la connexion
    public void init()
    {
        ServletContext servletContext=getServletContext();
        connection=(Connection)servletContext.getAttribute("connection");
    }

     ...
}
```

Il existe une autre solution qui permet de charger une Servlet au démarrage de l'application plutôt que lors du déclenchement de la première requête (URL). Ce paramètre est placé dans le fichier de configuration de l'application et se nomme <*load-on-startup/*>.

```
<servlet>
 <servlet-name>…</servlet-name>
 <servlet-class>…</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
```

Les Servlets doivent être chargées en commençant par l'instance dont le numéro est le plus faible. Les instances sans numéro ou avec valeurs entières positives peuvent être chargées à tout moment lors du démarrage, sur décision du conteneur.

Cette technique bien que pratique n'est pas très utilisée pour les accès aux bases de données, du fait de cette notion de priorité, et surtout de l'absence de notion de fin ou destruction. En effet, il est facile de déclencher une Servlet à la création du contexte qui va instancier une connexion au SGBD mais la fermeture correcte de la connexion n'est pas certaine avec cette technique.

Sources de données et pools de connexion

1. Présentation

La technique présentée précédemment avec plusieurs connexions, ou une connexion partagée, est fonctionnelle dans le cadre d'un petit projet, mais n'est pas adaptée à des projets de plus grande envergure. En effet, dans les premiers exemples, la connexion doit être établie avec chaque Servlet d'où facilement des problèmes en cas de grande charge. Enfin, la seconde méthode utilise une simple et unique connexion partagée, qui devient très vite non opérationnelle au-delà de cent accès simultanées.

Pour répondre à ces problèmes, la spécification JDBC 2.0 a introduit la notion de sources de données. Désormais, c'est la méthode recommandée pour établir une connexion à une base de données. L'interface *DataSource* fournit une architecture plus souple que *DriverManager* pour la création et la gestion des connexions. Un objet *DataSource* permet d'accéder à différentes bases de données en modifiant le code d'une application en un seul endroit. Cet objet permet de masquer au programmeur les détails de programmation et d'accès, afin qu'il n'ait plus qu'à se préoccuper de l'URL, de l'hôte, du port et de l'utilisateur SGBD.

Un point essentiel des pools de connexion et qu'ils permettent de créer à l'avance un certain nombre de connexions. De cette façon, l'obtention d'une connexion est beaucoup plus rapide. La différence concerne la création et le recyclage des connexions. La création et la libération d'une connexion pour chaque Servlet supposent un certain nombre d'opérations peu rapides, à savoir l'allocation des ressources, la négociation de la connexion avec le SGBD, l'authentification et enfin la libération des ressources. Une source de données est généralement obtenue en effectuant une recherche dans un contexte. Un moyen d'associer un nom à une ressource est donc défini.

2. JNDI

Java dispose d'une interface neutre de connexion. Cette interface nommée JNDI (*Java Naming and Directory Interface*) définit un ensemble de fonctions permettant d'accéder aux services de répertoires (noms). Pour utiliser un tel service, nos programmes doivent utiliser l'API JNDI. Cette API permet d'accéder à différents services de nommage de façon uniforme. Elle permet également d'organiser et rechercher des informations avec une technique de nommage. Cette interface permet de gérer les connexions à des sources de données qui peuvent être des répertoires, des bases de données mais aussi des annuaires (DNS, systèmes de fichiers, annuaire LDAP, NIS...).

3. Utilisation d'un objet DataSource

Un objet *DataSource* fournit toutes les méthodes permettant d'obtenir une connexion à une base de données par l'intermédiaire d'un service de nommage JNDI. La méthode principale de l'objet *DataSource* est *getConnection()*. Toutefois, avant qu'un client puisse obtenir une connexion, le serveur doit créer cet objet, le placer dans le contexte et l'associer à un nom. Pour obtenir une connexion à une source de données, le client JDBC n'a besoin d'aucune information au niveau de la structure de la base de données.

L'obtention d'une source de données se déroule en deux étapes :

- Il faut tout d'abord créer un objet *InitialContext* qui permet de rechercher dans le contexte de l'application.
- Il faut ensuite appeler la méthode *lookup()* qui permet de rechercher la connexion dans le contexte JNDI.

La chaîne de caractères passée à la méthode *lookup()* est le nom associé à la source de données. Une fois la connexion obtenue, le client peut l'utiliser de la même façon que si elle avait été fournie par un *DriverManager*.

Nous allons modifier notre classe *InitialisationContext* afin d'utiliser une source de données. Pour cela, nous allons créer une instance de la classe *InitialContext* pour effectuer une recherche de la ressource nommée dans le contexte. Le préfixe *java:comp/env* est utilisé pour rechercher une ressource se trouvant sur le même serveur que le composant.

```
package betaboutique.boiteoutils;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.sql.DataSource;
```

```
public class InitialisationContext implements ServletContextListener{

    //action déclenchée lors du chargement du context
    public void contextInitialized(ServletContextEvent event)
    {
        //initaliser le contexte
        Context initCtx=null;

        try
        {
            //initaliser le contexte
            initCtx=new InitialContext();

            if(initCtx==null)
            {
                throw new Exception ("Il n'y a pas de contexte !");
            }
            else
            {
                System.out.println("Contexte chargé !");
            }
            //se connecter au JNDI
            Context envCtx=(Context) initCtx.lookup
("java:comp/env");
            DataSource ds=(DataSource) envCtx.lookup
("jdbc_betaboutiquemysql");
            if(ds==null)
            {
                throw new Exception ("Il n'y a pas de DataSource !");
            }
            else
            {
                System.out.println("DataSource chargée !");
            }
            //stocker la DataSource dans un attribut nommé
''datasource'' du context
            ServletContext servletContext=event.getServletContext();
            servletContext.setAttribute("datasource",ds);
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        finally
        {
            try
            {
                //fermer le contexte
                if(initCtx!=null)
                {
                    initCtx.close();
                    System.out.println("initCtx correctement déchargé !");
                }
            }
            catch(Exception e)
            {
                System.out.println("Erreur lors de initCtx !");
            }
        }
    }
    //action qui permet de détruire le filtre
    public void contextDestroyed(ServletContextEvent event)
    {
        System.out.println("----------- Contexte détruit -----------");
        try
        {
            //fermeture
            System.out.println("DataSource fermée");
        }
```

```
catch (Exception e)
        {
            e.printStackTrace();
        }
        finally
        {

        }
    }
//fin de la classe
}
```

Pour que cet exemple fonctionne, nous devons créer le descripteur de déploiement adapté. Cette configuration permet d'associer le nom utilisé pour la recherche et le nom de la ressource JNDI.

Cette opération est réalisée en deux étapes :

- Tout d'abord, dans le descripteur de déploiement, nous devons indiquer la ressource nommée *jdbc_betaboutiquemysql* qui est une instance de *javax.sql.DataSource*. Cette déclaration est réalisée en fin de fichier *web.xml* et possède la structure suivante :

```
...
<!--  datasource a la base de donnees -->
<resource-ref>
       <description>DB Connection</description>
       <res-ref-name>jdbc_betaboutiquemysql</res-ref-name>
       <res-type>javax.sql.DataSource</res-type>
       <res-auth>Container</res-auth>
 </resource-ref>
...
```

- Ensuite, il faut paramétrer la connexion à la base de données par le pool de connexion. Pour cela, il faut préciser au serveur Java EE les informations de connexions. Cette configuration peut être alors déclarée dans le fichier */conf/server.xml* de Tomcat ou mieux, dans le fichier spécifique à l'application. Dans notre cas, nous insérons le code suivant dans le fichier */conf/Catalina/localhost/betaboutique.xml*.

```
<Context path="/betaboutique" reloadable="true"
docBase="E:\PROJETWEB\betaboutique"
workDir="E:\PROJETWEB\betaboutique\work">
<Resource name="jdbc_betaboutiquemysql" auth="Container"
type="javax.sql.DataSource" username="root" password=""
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/betaboutique"
maxActive="20" maxIdle="10"/>
</Context>
```

Une ressource JNDI est déclarée avec la partie *<Resource name.../>*. Dans cette déclaration, nous remarquons que notre projet possède une connexion nommée *jdbc_betaboutiquemysql* qui est une source de données (type), avec l'utilisateur root pour la connexion, sans mot de passe, avec un pilote JDBC MySQL et l'utilisation de la base de données *betaboutique*. Le paramètre important est *maxActive* qui permet de gérer le nombre de connexions actives/ouvertes en même temps. Cette technique permet de déclarer plusieurs pools de connexions, ce qui est extrêmement puissant. En effet, cette déclaration permet d'utiliser notre base de données MySQL pour le projet et une autre connexion pour la gestion des utilisateurs (par exemple) avec un autre SGBD.

Une astuce afin de vérifier la syntaxe d'un fichier de configuration consiste à ouvrir ce fichier dans un navigateur Web. Si l'affichage est correct, le fichier est alors opérationnel du point de vue syntaxique.

Voici ci-après un exemple de configuration pour l'utilisation de deux pôles de connexion.

```
<Context path="/betaboutique" reloadable="true"
docBase="E:\PROJETWEB\betaboutique"
workDir="E:\PROJETWEB\betaboutique\work">
<Resource name="jdbc_betaboutiquemysql" auth="Container"
type="javax.sql.DataSource" username="root" password=""
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/betaboutique"
maxActive="20" maxIdle="10"/>
```

```
<Resource name="jdbc_betaboutiqueutilisateurnmysql"
auth="Container" type="javax.sql.DataSource" username="jerome"
    password="jerome" driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/betaboutiqueutilisateur"
    maxActive="20" maxIdle="10"/>
</Context>
```

Les fichiers de l'application *betaboutique.xml* (ou *server.xml*) et *web.xml* permettent de définir les paramètres nécessaires afin d'habiliter le serveur Java EE à établir une connexion avec la base de données. Dans le fichier *betaboutique.xml* la source de données *jdbc_betaboutiquemysql* est déclarée et dans le fichier de configuration de l'application *web.xml*, le lien avec cette connexion est réalisé.

Les différents paramètres sont présentés dans cette liste :

- *driverClassName* : nom qualifié de la classe du pilote JDBC.
- *maxActive* : nombre maximal de connexions actives dans le pool.
- *maxIdle* : nombre maximal de connexions en attente dans le pool.
- *maxWait* : délai maximal, en millisecondes d'attente d'une connexion. Au-delà de ce délai, si aucune connexion n'est disponible, une exception est lancée.
- *user* : nom de l'utilisateur pour la base de données.
- *password* : mot de passe de l'utilisateur pour l'accès à la base de données.
- *url* : URL d'accès à la base de données.
- *validationQuery* : requête de test envoyée à la base de données pour s'assurer qu'une connexion est valide.

Lorsque Tomcat démarre, il analyse le fichier *server.xml* et tous les fichiers de configuration, et crée une source de données en fonction des paramètres contenus dans l'élément *ResourceParams*. Il la met ensuite à la disposition des clients par l'intermédiaire d'une interface JNDI.

4. Mise en place

Nous allons maintenant utiliser un pool de connexion pour notre projet *betaboutique*. Le fichier */conf/Catalina/host/betaboutique.xml* possède la syntaxe suivante :

```
<Context path="/betaboutique" reloadable="true"
docBase="E:\PROJETWEB\betaboutique"
workDir="E:\PROJETWEB\betaboutique\work">
<Resource name="jdbc_betaboutiquemysql" auth="Container"
type="javax.sql.DataSource" username="root" password=""
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/betaboutique"
maxActive="20" maxIdle="10" validationQuery="SELECT 1"/>
</Context>
```

Le fichier *web.xml* possède la syntaxe simplifiée suivante :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!--  écouteurs -->
    <listener>
        <listener-class>betaboutique.boiteoutils.Initialisation
Context</listener-class>
    </listener>
    <!--  servlets -->
    <servlet>
        <servlet-name>servletlisteclientmodele</servlet-name>
```

```
        <servlet-class>betaboutique.servlets.client.Servlet
ListeClientModele</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>servletlisteidentifiantclientmodele
</servlet-name>
        <servlet-class>betaboutique.servlets.client.Servlet
ListeIdentifiantClientModele</servlet-class>
    </servlet>
    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletlisteclientmodele</servlet-name>
        <url-pattern>/listeclient</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>servletlisteidentifiantclientmodele
</servlet-name>
        <url-pattern>/listeidentifiantclient</url-pattern>
    </servlet-mapping>
    <!--  datasource a la base de donnees -->
    <resource-ref>
        <description>DB Connection</description>
        <res-ref-name>jdbc_betaboutiquemysql</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</web-app>
```

Enfin, la Servlet *ServletListeClientModele* possède le code ci-dessous. Désormais une seule ligne est nécessaire pour obtenir une connexion à la source de données.

```
package betaboutique.servlets.client;
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.*;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import betaboutique.boiteoutils.OutilsBaseDeDonnees;

@SuppressWarnings("serial")
public class ServletListeClientModele extends HttpServlet {

    //variables de la classe
    DataSource ds=null;
    Connection connection=null;
    PreparedStatement requete=null;
    ResultSet rs=null;

      //traitements
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
        ds=(DataSource)getServletContext().getAttribute("datasource");
        try
        {
            //ouvrir une connexion
            connection=ds.getConnection();
            //flux de sortie vers le navigateur
            PrintWriter out=response.getWriter();
            //preparation de la requete
            requete=connection.prepareStatement("SELECT * FROM
client ORDER BY client.id_client");
            //executer la requete
            rs=requete.executeQuery();
```

```
            //affichage
            while(rs.next())
            {
                out.println("Client : "+rs.getInt
("id_client")+"-"+rs.getString("nomclient")+"-"+
rs.getString("prenomclient")+"-"+rs.getString("emailclient"));
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            System.out.println("Erreur lors de l'exécution de la requete");
        }
        finally
        {
            OutilsBaseDeDonnees.fermerConnexion(rs);
            OutilsBaseDeDonnees.fermerConnexion(requete);
            OutilsBaseDeDonnees.fermerConnexion(connection);
            ds=null;
        }
    }
    //traitements
    public void doPost(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        doGet(request, response);
    }
}
```

Le code JDBC est identique aux exemples précédents. Le client ne sait pas qu'il obtient une connexion provenant d'un pool. Il continue d'utiliser directement la même interface *Connection*. La méthode *close()* de l'objet *Connection* est toujours appelée, mais elle ne ferme pas physiquement la connexion, elle la remet à la disposition du pool. Du point de vue du client, il n'y a pas de différence entre les connexions provenant d'un pool et celles créées directement. Cette technique peut être employée dans des JSP ou des Servlets.

Cette technique est plus simple, fiable, et adaptée aux projets de grandes tailles. En effet, il faut déclarer la source de données dans le fichier de configuration de l'application (*server.xml* ou *nomapplication.xml*), préciser dans le fichier de description de l'application (*web.xml*) le lien vers cette ressource, et enfin réaliser une classe qui gère cette connexion indépendamment du SGBD utilisé.

Ainsi, il n'y a plus qu'à se soucier de la libération des ressources, toutes les bases de données compatibles JDBC peuvent être utilisées sans toucher au code source et il est possible d'optimiser les accès de façon très précise avec les attributs de la balise *<Resource.../>* (nombre de connexions actives, temps d'attente maximal, nombre de connexions en attente...). Toutes ces variables pourront ainsi être adaptées en fonction du projet, serveur et population du site sans retoucher au code.

Bases de données et MVC

1. Présentation

Le design pattern ou modèle MVC (Modèle Vue Contrôleur) est très utilisé dans le domaine de l'informatique et notamment avec les technologies Java EE. Nous avons étudié jusqu'à maintenant la partie Contrôleur avec les Servlets et les traitements associés et la partie Vue avec toutes les pages JSP utilisées pour l'affichage.

La partie Modèle concerne la persistance des données. Nous utilisons une base de données pour sauvegarder les données. Les nouveaux exemples sont basés sur les précédents (liste des clients et liste des identifiants clients) et vont être adaptés pour correspondre au modèle MVC. Nous allons développer une Servlet nommée *ServletListeArticles* qui permet d'afficher la liste des articles de la base de données.

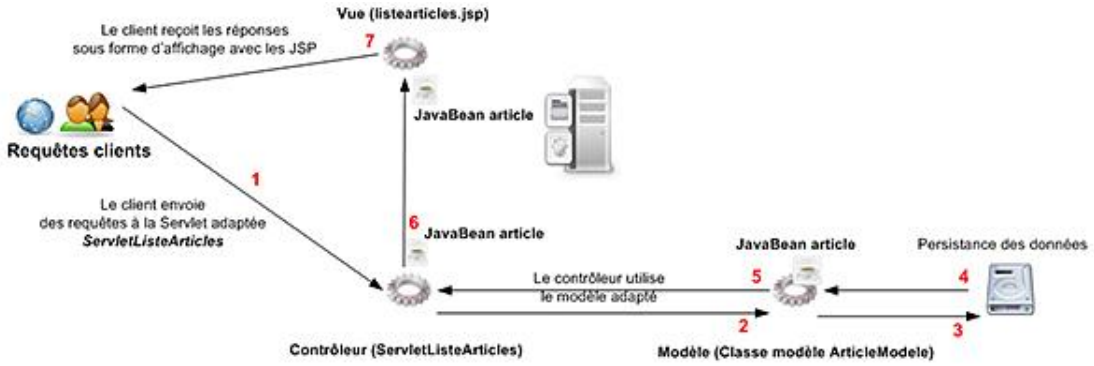
Du point de vue de l'architecture, nous aurons trois paquetages :

- *betaboutique.boiteoutils*.
- *betaboutique.modeles*.
- *betaboutique.servlets*.

Le premier paquetage correspond à la connexion au SGBD, aux classes statiques pour les ressources et bases de données, aux JavaBeans... Le second paquetage correspond aux classes modèles qui possèdent uniquement des accès aux données. Le troisième et dernier paquetage correspond aux Servlets (Contrôleur) pour le traitement des requêtes clients.

Nous allons volontairement supprimer les autres paquetages et classes du projet *betaboutique* pour commencer l'application de persistance des données avec un projet simple et clair.

Le principe de fonctionnement du modèle MVC pour le projet *betaboutique* est expliqué dans le schéma ci-dessous :



Le client déclenche la Servlet de liste des articles. Cette Servlet qui est la partie Contrôleur du modèle MVC réalise des traitements simples et déclenche le modèle *ArticleModele* qui gère la persistance des données des articles. Les articles sont lus dans la base de données et retournés sous la forme d'une collection d'objets JavaBean *Article*. Cette collection est ensuite renvoyée à la vue adaptée *listearticles.jsp* pour l'affichage des enregistrements.

2. Modèle conceptuel des données et base de données

Le modèle conceptuel des données propre à la méthodologie MERISE est présenté dans le schéma ci-dessous. Il comprend la table *article* avec sa clé primaire et ses différents champs. Ce modèle conceptuel de données est volontairement simplifié mais permet la mise en place d'un projet sans erreur.

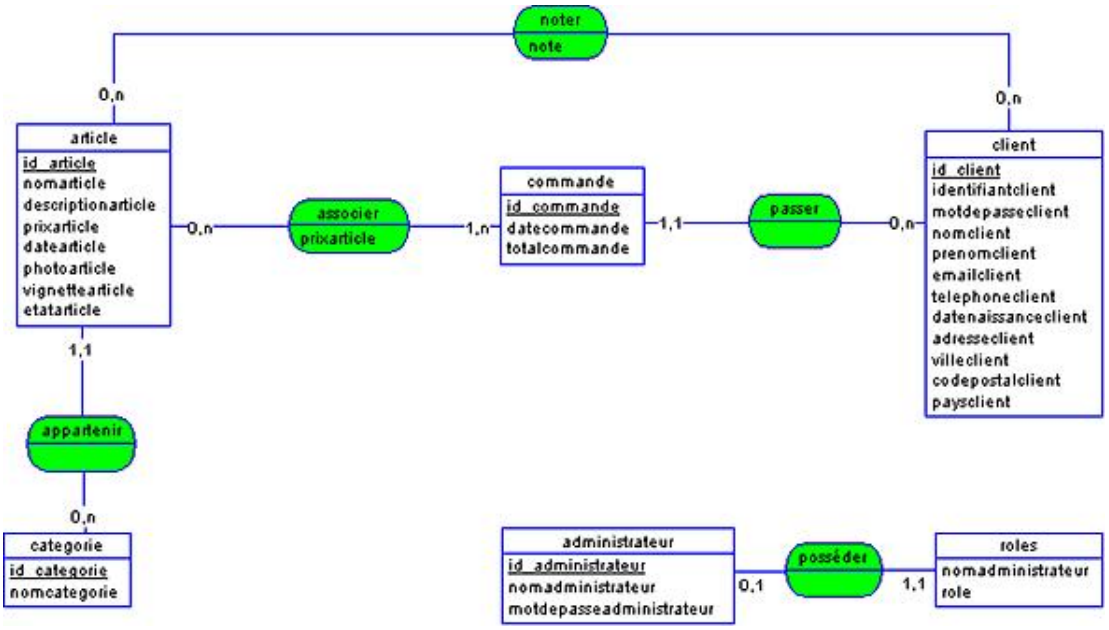
Article : le champ *datearticle* correspond à la date d'insertion de l'article dans la base de données. Le champ *photoarticle* correspond au chemin de l'image grand format de la pochette du DVD. Le champ *vignettearticle* correspond au chemin de l'image petit format (ou vignette) du DVD. Enfin, l'attribut *etatarticle* est positionné par défaut à 0 dans la base de données et permet de mettre en ligne ou pas l'article concerné.

Catégorie : un article est relié à une et une seule catégorie. Une catégorie est représentée par son nom.

Commande : une commande correspond à au moins un article ou plusieurs. La date de la commande et le total de la commande sont conservés. Le prix de l'article au moment de la commande est également conservé pour l'affichage de la facture exacte.

Client : un client peut passer plusieurs commandes, mais il peut très bien ne rien acheter sur le site. Un client peut également noter tous les articles du site ou ne jamais noter un article. Par contre, un client ne peut donner qu'une seule note par article.

Administrateur/Rôle : un administrateur de la plate-forme *betaboutique* dispose d'un identifiant et d'un mot de passe et possède un rôle unique pour l'accès. Le rôle est utilisé uniquement pour l'authentification et la gestion des droits. La table "Roles" est modélisée de la sorte pour l'utilisation avec les REALMS dans un prochain paragraphe. Cette table doit respecter la spécification des REALMS. La clé primaire de la table "roles" est composée des attributs "nomadministrateur" et "role" pour permettre à un utilisateur d'avoir plusieurs rôles.



La table *article* possède la structure suivante :

	Champ	Type	Interclassement	Attributs	Null	Défaut	Extra	Action
☐	id_article	int(10)		UNSIGNED	Non		auto_increment	
☐	nomarticle	varchar(255)	latin1_swedish_ci		Non			
☐	descriptionarticle	text	latin1_swedish_ci		Non			
☐	prixarticle	double			Non	0		
☐	datearticle	int(11)			Non	0		
☐	photoarticle	varchar(255)	latin1_swedish_ci		Non			
☐	vignettearticle	varchar(255)	latin1_swedish_ci		Non			
☐	etatarticle	tinyint(1)			Non	0		
☐	id_categorie	int(11)			Non	0		

La première étape consiste à créer le JavaBean *Article* adapté avec la même structure que la table *article*.

```
package betaboutique.boiteoutils;

@SuppressWarnings("serial")
public class Article implements java.io.Serializable {

    private String id_article=null;
    private String nomarticle=null;
    private String descriptionarticle=null;
    private String prixarticle=null;
    private String datearticle=null;
    private String photoarticle=null;
    private String vignettearticle=null;
    private String etatarticle=null;
    private String id_categoriearticle=null;

    //Constructeur par défaut (sans paramètre)
    public Article()
    {

    }
    public String getDatearticle() {
```

```
        return datearticle;
    }
    public void setDatearticle(String datearticle) {
        this.datearticle = datearticle;
    }
    public String getDescriptionarticle() {
        return descriptionarticle;
    }
    public void setDescriptionarticle(String descriptionarticle) {
        this.descriptionarticle = descriptionarticle;
    }
    public String getEtatarticle() {
        return etatarticle;
    }
    public void setEtatarticle(String etatarticle) {
        this.etatarticle = etatarticle;
    }
    public String getId_article() {
        return id_article;
    }
    public void setId_article(String id_article) {
        this.id_article = id_article;
    }
    public String getId_categoriearticle() {
        return id_categoriearticle;
    }
    public void setId_categoriearticle(String id_categoriearticle) {
        this.id_categoriearticle = id_categoriearticle;
    }
    public String getNomarticle() {
        return nomarticle;
    }
    public void setNomarticle(String nomarticle) {
        this.nomarticle = nomarticle;
    }
    public String getPhotoarticle() {
        return photoarticle;
    }
    public void setPhotoarticle(String photoarticle) {
        this.photoarticle = photoarticle;
    }
    public String getPrixarticle() {
        return prixarticle;
    }
    public void setPrixarticle(String prixarticle) {
        this.prixarticle = prixarticle;
    }
    public String getVignettearticle() {
        return vignettearticle;
    }
    public void setVignettearticle(String vignettearticle) {
        this.vignettearticle = vignettearticle;
    }
}
```

Dans cet exemple, le JavaBean *Article* n'utilise que des types *String* pour les paramètres. Cette technique de programmation permet juste d'éviter des conversions et de coder plus rapidement. Dans un cas réel, il serait préférable d'utiliser des types en correspondance avec ceux de la base de données (ex : *prixarticle* est un réel).

Le code de la Servlet contrôleur est désormais très simple, il utilise uniquement le modèle pour récupérer des données.

```
package betaboutique.servlets;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import betaboutique.modeles.ArticleModele;

@SuppressWarnings("serial")
public class ServletListeArticles extends HttpServlet {

    //variables de la classe
    DataSource ds=null;

      //traitements
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer la datasource du plug-in dans un attribut
présent dans le context de la servlet
        ds=(DataSource)getServletContext().getAttribute("datasource");
        //créer le modèle
        ArticleModele articlemodele=new ArticleModele(ds);
        //redonner la datasource
        this.ds=null;
        //retourner la liste des articles
        ArrayList listearticles=(ArrayList)articlemodele.ListeArticle();
        request.setAttribute("listearticles",listearticles);
        //retourner sur la page d'affichage des articles
    getServletContext().getRequestDispatcher
("/vues/article/listearticles.jsp").forward(request, response);
    }


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

La classe modèle *ArticleModele* possède une seule fonction qui permet de retourner la collection d'objets JavaBean présents dans la base de données.

```
package betaboutique.modeles;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.sql.DataSource;
import betaboutique.boiteoutils.Article;
import betaboutique.boiteoutils.OutilsBaseDeDonnees;
import java.util.ArrayList;

public class ArticleModele
{
    //variables de classe
    DataSource ds=null;
    Connection connection=null;
    ResultSet rs=null;
    //liste des objets
    ArrayList<Article> listearticle=new ArrayList<Article>();

/*********************************************************
 * constructeur
*********************************************************/
public ArticleModele(DataSource ds)
{
    //récupérer la DataSource de la servlet
    this.ds=ds;
}
```

```
/*********************************************************
 * liste complète des articles
*********************************************************/
public ArrayList ListeArticle()
{
    PreparedStatement requete=null;
    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        //enregistrements
        requete=connection.prepareStatement("SELECT * FROM
article ORDER BY article.nomarticle");
        rs=requete.executeQuery();
        //exécuter la requête
        if(rs!=null)
        {
            //stocker tous les articles dans une liste
            while(rs.next())
            {
                //créer un objet article
                Article article=new Article();

                //renseigner l'objet article avec ses accesseurs
                if(rs.getString("id_article")==null)
article.setId_article("0");
                else article.setId_article(rs.getString
("id_article"));
                if(rs.getString("nomarticle")==null)
article.setNomarticle("");
                else article.setNomarticle(rs.getString
("nomarticle"));
                if(rs.getString("descriptionarticle")==null)
article.setDescriptionarticle("")

                else article.setDescriptionarticle(rs.getString
("descriptionarticle"));
                if(rs.getString("prixarticle")==null)
article.setPrixarticle("0");
                else article.setPrixarticle(rs.getString
("prixarticle"));
                if(rs.getString("datearticle")==null)
article.setDatearticle("0");
                else article.setDatearticle(rs.getString
("datearticle"));
                if(rs.getString("photoarticle")==null)
article.setPhotoarticle("");
                else article.setPhotoarticle(rs.getString
("photoarticle"));
                if(rs.getString("vignettearticle")==null)
article.setVignettearticle("");
                else article.setVignettearticle(rs.getString
("vignettearticle"));
                if(rs.getString("etatarticle")==null)
article.setEtatarticle("0");
                else article.setEtatarticle(rs.getString
("etatarticle"));

                //stocker l'objet article dans la liste desarticles
                listearticle.add((Article)article);
            }
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
ArticleModele.java fonction ListeArticle");
    }
```

```
    finally
    {
        try
        {
            //fermer la connexion
            if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ArticleModele.java fonction ListeArticle");
        }
    }
    //retourner la liste des articles
    return listearticle;
}

//fin de la classe
}
```

Nous allons désormais déclarer notre Servlet contrôleur dans le fichier de configuration de l'application.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!--  url de l'application -->
    <context-param>
        <param-name>urlapplication</param-name>
        <param-value>http://localhost:8080/betaboutiquemvc/</param-value>
    </context-param>
    <!--  écouteurs -->
    <listener>
        <listener-class>betaboutique.boiteoutils.Initialisation
Context</listener-class>
    </listener>
    <!--  servlets -->
    <servlet>
        <servlet-name>servletlistearticles</servlet-name>
        <servlet-class>betaboutique.servlets.ServletListe
Articles</servlet-class>
    </servlet>
    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletlistearticles</servlet-name>
        <url-pattern>/listearticles</url-pattern>
    </servlet-mapping>
    <!--  datasource a la base de donnees -->
    <resource-ref>
            <description>DB Connection</description>
            <res-ref-name>jdbc_betaboutiquemysql</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
    </resource-ref>
</web-app>
```

Nous retrouvons la connexion à la base de données avec le listener pour déclencher la connexion au chargement du contexte, la déclaration de la Servlet contrôleur et une variable globale utilisée pour référencer l'URL de l'application (pour les images, chemins divers, feuilles de style et liens).

Il ne reste plus qu'à développer la page *listearticles.jsp*. Cette page très simple ne gère que l'affichage de la collection d'objets. Nous remarquons que le principe MVC requiert beaucoup de fichiers mais le code est bien découpé et très simple (traitement, gestion de la persistance des données et affichage).

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="java.util.ArrayList" %>
<%@page import="betaboutique.boiteoutils.Article;"%>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>LISTE DES ARTICLES</title>
</head>
<body>
<%
//context de l'application
String urlapplication=getServletContext().getInitParameter("urlapplication");
//récupérer les articles dans la requête
ArrayList listearticles=(ArrayList)request.getAttribute("listearticles");
%>
<h2>Liste des articles de la boutique</h2>
<table border="1" cellspacing="0" cellpadding="0">
<tr><td>Id article</td><td>Nom</td><td>Description</td>
<td>Prix</td><td>Date</td><td>Vignette</td></tr>
    <%
    for(int i=0;i<listearticles.size();i++)
    {
    Article article=(Article)listearticles.get(i);
    out.println("<tr>");
    out.println("<td>"+article.getId_article()+"</td>");
    out.println("<td>"+article.getNomarticle()+"</td>");
    out.println("<td>"+article.getDescriptionarticle()+"</td>");
    out.println("<td>"+article.getPrixarticle()+" Euros</td>");
    out.println("<td>"+article.getDatearticle()+"</td>");
    out.println("<td><img src=\""+urlapplication+"/
imgarticles/"+article.getVignettearticle()+"\"/></td>");
    out.println("</tr>");
    }
    %>
</table>
</body>
</html>
```

L'arborescence du projet à cette étape est la suivante :

Nous déclenchons alors l'URL : http://localhost:8080/betaboutiquemvc/listearticles

Le résultat produit est correct. Nous allons cependant apporter quelques améliorations à la classe modèle afin d'afficher correctement la date, la catégorie associée à l'article et gérer l'état de l'article (en ligne ou pas) et les recherches sur le nom ou la description de l'article.

22	Fantômas	# Acteurs : Jean Marais, Louis de Funès, Mylène Demongeot, Robert Dalban, Jacques Dynam # Réalisateurs : André Hunebelle # Format : PAL # Région: Région 2 (Ce DVD ne pourra probablement pas être visualisé en dehors de l'Europe. Plus d'informations sur les formats DVD) # Studio: G C T H V. # Date de sortie du DVD : 8 novembre 2005 # Fonctions DVD : # ASIN: B000BNEMQY	11 6 Euros	20071001	
15	Garde à vue	# Acteurs : Lino Ventura, Michel Serrault, Guy Marchand, Romy Schneider, Michel Such # Réalisateurs : Claude Miller # Format : Couleur, PAL # Langue : Français # Région: Région 2 (Ce DVD ne pourra probablement pas être visualisé en dehors de l'Europe. Plus d'informations sur les formats DVD) # Rapport de forme : 1.66:1 # Nombre de disques: 1 # Studio: TF1 Vidéo # Date de sortie du DVD : 20 septembre 2000 # Durée : 85 minutes # Moyenne des commentateurs client : basé sur 5 commentateurs. (Écrivez un commentaire) # Fonctions DVD : # ASIN: B00004YBL8	12 0 Euros	20070202	
18	L'incruste	# Acteurs : Tiroff, Frédéric Diefenthal, Zoé Félix, Zinedine Soualem, Agnès Soral # Réalisateurs : Corentin Julus, Alexandre Castagnetti # Format : PAL # Région: Région 2 (Ce DVD ne pourra probablement pas être visualisé en dehors de l'Europe. Plus d'informations sur les formats DVD) # Studio: _ # Date de sortie du DVD : 1 janvier 2006 # Fonctions DVD : # ASIN: B000KIN7DAW	5 0 Euros	20070208	

3. Optimisations

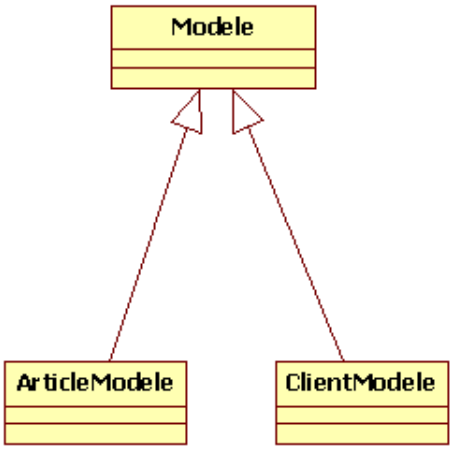
a. Informations liées et mise en forme

Notre service de liste des articles est désormais fonctionnel mais il manque quelques optimisations qui permettent de proposer un service professionnel. Dans notre cas, nous devons d'abord développer une fonction pour mettre la date au format français (aaaammjj devient jj/mm/aaaa).

Il manque également l'affichage de la catégorie de l'article. Cette information est liée à la table *categorie* par l'intermédiaire de la clé externe *id_categorie*. Ces fonctionnalités sont utilisées dans la majorité des projets que ce soient des catégories associées, des gestions de dates, le découpage de chaînes de caractères...

Ces optimisations sont liées aux données AVANT ou APRES insertion dans la base de données. De même, elles sont

déclenchées par la plupart des classes modèles. Afin d'utiliser un système souple et facilement maintenable, nous allons développer une classe nommée *Modele* qui sera placée en haut de l'arbre d'héritage des modèles. Ainsi la classe modèle *ArticleModele* hérite de la classe principale *Modele* qui contient les fonctionnalités générales et communes (principe d'héritage).



Nous allons donc ajouter pour le moment deux fonctions à la classe *Modele* à savoir : *miseEnFormeDate()* qui permet de mettre la date au format français et la fonction *getNomCategorieArticle()* qui permet de retourner le nom d'une catégorie en fonction de son identifiant. Dans cette classe, nous utilisons un autre nom de connexion (*connection1* à la place de *connection*) et également un autre nom de *ResultSet (rs1)* afin d'éviter d'éventuels conflits entre les objets *Connection* et *ResultSet* de la classe fille (fermeture ou accès simultanés) et ceux de la classe mère.

Le déclenchement du constructeur de la classe mère se fera avec la méthode *super()* dans le constructeur des classes filles, et l'utilisation des méthodes de la classe mère se fera également avec le mot clé *super*. Nous allons modifier le JavaBean *Article* afin d'ajouter un attribut pour le nom de la catégorie. Une dernière modification est réalisée dans la requête de liste des articles afin d'afficher uniquement les articles validés en administration (*etatarticle=1*).

```
package betaboutique.boiteoutils;

@SuppressWarnings("serial")
public class Article implements java.io.Serializable {
...
    private String nomcategoriearticle=null;
...
```

```
package betaboutique.modeles;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.sql.DataSource;
import betaboutique.boiteoutils.OutilsBaseDeDonnees;

public class Modele
{
    //variables de classe
    DataSource ds=null;
    Connection connection1=null;
    ResultSet rs1=null;

/*********************************************************
 * constructeur
*********************************************************/
public Modele(DataSource ds)
{
    //récupérer la DataSource de la servlet
    this.ds=ds;
}

/*********************************************************
 * mise en forme de la date (aaaammjj -> jj/mm/aaaa)
*********************************************************/
```

```
public String miseEnFormeDate(String date)
{
    if(date.length()>=4)
    {
    String jour=date.substring((date.length()-2),date.length());
    String mois=date.substring((date.length()-4),(date.length()-2));
    String annee=date.substring(0,4);
    date=jour+"/"+mois+"/"+annee;
    }
    //retourner la date
    return date;
}

/*********************************************************
 * récupérer le nom de la catégorie de l'article
*********************************************************/
public String getNomCategorieArticle(String id_categorie)
{
    String nomcategorie=null;
    //statement
    PreparedStatement requete=null;

    try
    {
        //ouvrir une connexion
        connection1=ds.getConnection();
        requete=connection1.prepareStatement("SELECT * FROM
categorie WHERE categorie.id_categorie=?");
        requete.setString(1,(String)id_categorie);
        rs1=requete.executeQuery();
        //exécuter la requête
        if(rs1!=null)
        {
        //stocker tous les types de médias dans une liste
        if(rs1.next())
        {
            if(rs1.getString("nomcategorie")==null)nomcategorie="";
            else nomcategorie=rs1.getString("nomcategorie");
            }
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe Modele.java
fonction getNomCategorieArticle");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(rs1!=null)OutilsBaseDeDonnees.fermerConnexion(rs1);
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection1!=null)OutilsBaseDeDonnees.fermerConnexion
(connection1);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
Modele.java fonction getNomCategorieArticle");
        }
    }

    //retourner le nom
    return nomcategorie;
}

//fin de la classe
}
```

```
...
public class ArticleModele extends Modele
{
    ...
    if(rs.getString("datearticle")==null)article.setDatearticle("0");
    else article.setDatearticle(super.miseEnFormeDate(rs.getString
("datearticle")));
    if(rs.getString("id_categorie")==null)article.setNomcategoriearticle("");
    else
article.setNomcategoriearticle(super.getNomCategorieArticle(rs.getString
("id_categorie")));
...
//fin de la classe
}
```

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page import="java.util.ArrayList" %>
<%@page import="betaboutique.boiteoutils.Article;"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1"><title>LISTE DES ARTICLES</title>
</head>
<body>
<%
//context de l'application
String urlapplication=getServletContext().getInitParameter
("urlapplication");
//récupérer les articles dans la requête
ArrayList listearticles=(ArrayList)request.getAttribute
("listearticles");
%>
<h2>Liste des articles de la boutique</h2>
<table border="1" cellspacing="0" cellpadding="0">
<tr><td>Id article</td><td>Nom</td><td>Description</td>
<td>Prix</td><td>Date</td><td>Vignette</td>
<td>Cat&eacute;gorie</td></tr>
    <%
    for(int i=0;i<listearticles.size();i++)
    {
    Article article=(Article)listearticles.get(i);
    out.println("<tr>");
    out.println("<td>"+article.getId_article()+"</td>");
    out.println("<td>"+article.getNomarticle()+"</td>");
    out.println("<td>"+article.getDescriptionarticle()+"</td>");
    out.println("<td>"+article.getPrixarticle()+" Euros</td>");
    out.println("<td>"+article.getDatearticle()+"</td>");
    out.println("<td><img src=\""+urlapplication+"
/imgarticles/"+article.getVignettearticle()+"\"/></td>");
    out.println("<td>"+article.getNomcategoriearticle()+"</td>");
    out.println("</tr>");
    }
    %>
</table>
</body>
</html>
```

15	Garde à vue	# Acteurs : Lino Ventura, Michel Serrault, Guy Marchand, Romy Schneider, Michel Such # Réalisateurs : Claude Miller # Format : Couleur, PAL # Langue : Français # Région: Région 2 (Ce DVD ne pourra probablement pas être visualisé en dehors de l'Europe. Plus d'informations sur les formats DVD). # Rapport de forme : 1.66:1 # Nombre de disques: 1 # Studio: TF1 Vidéo # Date de sortie du DVD : 20 septembre 2000 # Durée : 85 minutes # Moyenne des commentaires client : basé sur 5 commentaires. (Écrivez un commentaire) # Fonctions DVD : # ASIN: B00004YRL8	12.0 Euros	02/02/2007		Policier et Thriller

b. Gestion des recherches

Nous allons améliorer le service de liste des articles afin de réaliser des recherches sur le nom du DVD et sur sa description. Pour cela, nous ajoutons un petit formulaire HTML en début de fichier avec un champ de saisie et un bouton de validation. La Servlet contrôleur est légèrement modifiée pour lire et renvoyer la recherche effectuée. Enfin, la requête SQL de la classe modèle est modifiée avec l'instruction SQL *LIKE*.

```
...
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ page import="java.util.ArrayList" %>
<%@page import="betaboutique.boiteoutils.Article;"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>LISTE DES ARTICLES</title>
</head>
<body>
<%
//context de l'application
String urlapplication=getServletContext().getInitParameter
("urlapplication");
//récupérer les articles dans la requête
ArrayList listearticles=(ArrayList)request.getAttribute("listearticles");
//récupérer la recherche de l'utilisateur
String recherche=(String)request.getAttribute("recherche");
%>
<h2><a href="listearticles">Liste des articles de la boutique</a></h2>
<form action="listearticles" methode="post">
<p><input type="text" name="recherche" id="recherche" size="30"/><input
type="submit" value="Rechercher"/></p>
<%
//afficher la recherche si présente
if(recherche!=null && !recherche.equalsIgnoreCase(""))
{
out.println("<b>Votre recherche : "+recherche+"</b><br/><br/>");
}
%>
</form>
...
```

```
package betaboutique.servlets;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import betaboutique.modeles.ArticleModele;

@SuppressWarnings("serial")
public class ServletListeArticles extends HttpServlet {

    //variables de la classe
    DataSource ds=null;

      //traitements
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
        ds=(DataSource)getServletContext().getAttribute("datasource");
        //créer le modèle
```

```
        ArticleModele articlemodele=new ArticleModele(ds);
        //redonner la datasource
        this.ds=null;
        //recherche
        String recherche=(String)request.getParameter("recherche");
        //retourner la liste des articles
        ArrayList listearticles=(ArrayList)
articlemodele.ListeArticle(recherche);
        request.setAttribute("listearticles",listearticles);
        request.setAttribute("recherche",recherche);
    //retourner sur la page d'affichage des articles
    getServletContext().getRequestDispatcher
("/vues/article/listearticles.jsp").forward(request, response);
    }

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

```
...
        //ouvrir une connexion
        connection=ds.getConnection();
        //enregistrements
        String requete="SELECT * FROM article WHERE etatarticle=1";
        if(recherche!=null && !recherche.equalsIgnoreCase(""))
        {
            requete+=" AND (nomarticle LIKE ? OR descriptionarticle LIKE ?)";
        }

        //preparer la requete
        requetea=connection.prepareStatement(requete);
        //recherche
        if(recherche!=null && !recherche.equalsIgnoreCase(""))
        {
            requetea.setString(1,(String)"%"+recherche+"%");
            requetea.setString(2,(String)"%"+recherche+"%");
        }
        //exécuter la requete
        rs=requetea.executeQuery();
...
```

La recherche est désormais fonctionnelle. Nous pouvons essayer avec la chaîne *"balance"* qui dans ce cas va porter sur le nom de l'article ou avec la chaîne *"nathalie baye"* qui va porter sur la description (acteurs). Cette requête est opérationnelle mais très lente lors de l'utilisation de nombreux enregistrements et surtout sur des champs de grande taille comme la description. Nous pouvons optimiser ce système avec l'utilisation d'expressions régulières en SQL.

```
...
        //ouvrir une connexion
        connection=ds.getConnection();
        //enregistrements
        String requete="SELECT * FROM article WHERE etatarticle=1";
        if(recherche!=null && !recherche.equalsIgnoreCase(""))
        {
            requete+=" AND (nomarticle REGEXP ? OR
descriptionarticle REGEXP ?)";
        }
        //preparer la requete
        requetea=connection.prepareStatement(requete);
        //recherche
        if(recherche!=null && !recherche.equalsIgnoreCase(""))
        {
            requetea.setString(1,(String)"("+recherche+")");
```

```
            requetea.setString(2,(String)"("+recherche+")");
        }
        //exécuter la requete
        rs=requetea.executeQuery();
...
```

Il existe une dernière technique de recherche plus poussée qui permet de réaliser des affichages précis à la manière des moteurs de recherches. Cette instruction SQL nommée *MATCH* permet de faire des pondérations (ajout d'opérateurs). Par contre, il est parfois nécessaire de réaliser un index *FULLTEXT* sur les champs de recherche pour améliorer le temps d'exécution. Dans notre exemple, nous ne modifions à chaque fois que la classe modèle (d'où l'intérêt du MVC). Nous pouvons ainsi apporter des améliorations au site sans coder à nouveau l'ensemble d'une fonctionnalité.

Voici le code modifié de la requête avec la clause *MATCH*.

```
...
        //ouvrir une connexion
        connection=ds.getConnection();
        //enregistrements
        String requete="SELECT * FROM article WHERE etatarticle=1";
        if(recherche!=null && !recherche.equalsIgnoreCase(""))
        {
        requete+=" AND (MATCH (nomarticle) AGAINST (? IN BOOLEAN MODE)
OR MATCH (descriptionarticle) AGAINST (? INBOOLEAN MODE))";
        }
        //preparer la requete
        requetea=connection.prepareStatement(requete);
        //recherche
        if(recherche!=null && !recherche.equalsIgnoreCase(""))
        {
            requetea.setString(1,(String)recherche);
            requetea.setString(2,(String)recherche);
        }
        //exécuter la requete
        rs=requetea.executeQuery();
...
```

Avec cette dernière requête, nos recherches sont très puissantes. Il est possible d'utiliser l'opérateur + pour indiquer que le mot à la suite du signe doit être présent dans une ligne des enregistrements. L'opérateur - indique que le mot à la suite du signe ne doit pas être présent dans une ligne des enregistrements. L'opérateur * permet de faire des recherches sur des portions de mots. L'opérateur '' (guillemets) permet de faire des recherches exactes dans un champ (ex : citation entre guillemets).

Nous pouvons tester ces opérateurs avec les recherches suivantes :

- *nathalie baye* : tous les articles avec l'actrice Nathalie Baye dans le titre ou la description seront retrouvés.
- *baye* : tous les articles avec l'actrice Baye dans le titre ou la description seront retrouvés.
- *baye -léotard* : les articles contenant Nathalie Baye mais pas Philippe Léotard seront retrouvés.
- *pier** : les articles contenant le terme pier (Pierre Palmade, Pierre Desproges et Jean-Pierre Melville) seront retrouvés.


Ces exemples permettent de mettre en évidence l'utilisation d'une classe modèle et de requêtes SQL évoluées par rapport à un système de persistance (Hibernate ou autre). En effet, le code est parfois plus long à écrire mais les requêtes sont optimisées et utilisent TOUTES les instructions et la puissance du langage SQL.

Classe modèle

1. Présentation

L'affichage des articles et la recherche sont désormais opérationnels. Nous allons maintenant développer une classe modèle complète qui permet de gérer toutes les opérations d'administration pour un service (gestion des articles) et qui servira d'exemple pour la construction des autres services de la boutique.

La classe modèle aura donc une fonction pour lister tous les articles avec des recherches, une fonction pour récupérer toutes les informations d'un article précis, une fonction pour modifier un article, une fonction pour la création et enfin, une fonction pour la suppression.

Nous allons également gérer la pagination dans les listes et les recherches avec des contraintes sur les types (ex : recherche sur le nom ou sur la description).

2. Mise en place

Nous commençons par créer une nouvelle Servlet nommée *ServletGestionArticles*. Cette Servlet permettra de réaliser toutes les opérations sur les articles en administration. Sa définition dans le fichier *web.xml* est présentée ci-dessous.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    ...
    <!--  servlets -->
    <servlet>
        <servlet-name>servletgestionarticles</servlet-name>
        <servlet-class>betaboutique.servlets.ServletGestion
Articles</servlet-class>
    </servlet>
    <!-- mapping des servlets -->
    <servlet-mapping>
        <servlet-name>servletgestionarticles</servlet-name>
        <url-pattern>/admin/gestionarticles/*</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Nous retrouvons la déclaration de la Servlet de gestion des articles en administration (*/admin/gestionarticles/**). Ensuite, il faut réaliser le code du contrôleur (Servlet) afin de préparer la gestion des articles.

```
package betaboutique.servlets;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import betaboutique.modeles.ArticleModele;

@SuppressWarnings("serial")
public class ServletGestionArticles extends HttpServlet {

    //variables de la classe
    DataSource ds=null;

      //traitements
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
        ds=(DataSource)getServletContext().getAttribute("datasource");
```

```
        //créer le modèle
        ArticleModele articlemodele=new ArticleModele(ds);
        //redonner la datasource
        this.ds=null;
//action a réaliser (liste des articles, consultation,
modification, suppression ou création)
        String action=(String)request.getParameter("action");
        //action par défaut (liste)
        if(action==null || action.equalsIgnoreCase(""))
        {
           action="liste";
        }

        //liste des articles
        if(action.equals("liste"))
        {
           //recherche
           String typerecherche=(String)request.getParameter("typerecherche")
           String recherche=(String)request.getParameter("recherche");
           //informations pour les tris et l'affichage paginé
           String pagecourante=(String)request.getParameter("p");
           String nomtri=(String)request.getParameter("s");
           String tri=(String)request.getParameter("tri");
           //valeurs par défaut
           if(pagecourante==null)pagecourante="0";
           if(nomtri==null)nomtri="id_article";
           if(tri==null)tri="ASC";
           String maxparpage="5";

           //retourner la liste des articles
           ArrayList listearticles=(ArrayList)
articlemodele.ListeArticleAdmin(pagecourante,nomtri,
tri,maxparpage,typerecherche,recherche);
           //tri en cours
           //changer l'ordre de tri
           if(tri.equals("ASC"))tri="DESC";
           else if(tri.equals("DESC"))tri="ASC";
           request.setAttribute("nomtri",nomtri);
           request.setAttribute("tri",tri);
           //les recherches
           request.setAttribute("typerecherche",typerecherche);
           request.setAttribute("recherche",recherche);
           //retourner le maximum d'enregistrement par page
           request.setAttribute("maxparpage",(String)
articlemodele.getMaxparpage());
           //retourner la page courante de l'affichage
           request.setAttribute("pageencours",pagecourante);
           //retourner le nombre d'enregistrement trouvé
           request.setAttribute("compteurenregistrement",
articlemodele.getCompteurenregistrement());
           //retourner le nombre total d'enregistrement possibles
           request.setAttribute("totalenregistrement",
articlemodele.getTotalenregistrement());
           //retourner la liste des articles paginés
           request.setAttribute("listearticles",listearticles);
           //vider par sécurité
           listearticles=null;

           //retourner sur la page d'affichage des articles
en administration
    getServletContext().getRequestDispatcher("/admin/vues/
article/listearticles.jsp").forward(request, response);
        }

    }

    //traitements
    public void doPost(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
```

```
        doGet(request, response);
    }
}
```

Par défaut, la Servlet réalise un affichage sous forme de liste paginée des articles. Nous retrouvons plusieurs paramètres qui correspondent à la pagination, aux recherches et aux calculs d'affichage.

Ci-dessous, la fonction de liste de la classe modèle *ArticleModele*.

```
...
public class ArticleModele extends Modele
{
...
/**************************************************************************
 * liste complete des articles
**************************************************************************/
public ArrayList ListeArticleAdmin(String pagecourante,String nomtri,
String tri,
String maxparpage,String typerecherche,String recherche)
{
  //informations pour la pagination
  this.maxparpage=Integer.parseInt(maxparpage);
  this.position=this.maxparpage*Integer.parseInt(pagecourante);
  this.compteurenregistrement=0;
  PreparedStatement requetea=null,requeteb=null;
  String requete=null;

  try
  {
     //ouvrir une connexion
     connection=ds.getConnection();
     //requete de comptage pour la pagination
     requete="SELECT COUNT(DISTINCT(article.id_article)) AS
totalenregistrement FROM article WHERE 1";
     if((typerecherche!=null && !typerecherche.equalsIgnoreCase("")) &&
(recherche!=null && !recherche.equalsIgnoreCase("")))requete+=" AND
"+typerecherche+" LIKE ? ";
     //compter combien on a d'enregistrement sans les conditions de
pagination
     requetea=connection.prepareStatement(requete);
     if((typerecherche!=null && !typerecherche.equalsIgnoreCase("")) &&
(recherche!=null && !recherche.equalsIgnoreCase("")))requetea.setString(1,
(String)"%"+recherche+"%");
     rs=requetea.executeQuery();
     //exécuter la requête
     if(rs!=null)
     {
         //total d'enregistrements trouves
         if(rs.next())
         {
             if(rs.getString("totalenregistrement")!=null) this.
totalenregistrement=Integer.parseInt(rs.getString("totalenregistrement"));
         }
     }

     //enregistrements avec pagination
     requete="SELECT * FROM article WHERE 1";
     if((typerecherche!=null && !typerecherche.equalsIgnoreCase("")) &&
(recherche!=null && !recherche.equalsIgnoreCase("")))requete+=" AND
"+typerecherche+" LIKE ? ";
     requete+=" ORDER BY "+nomtri+" "+tri+" LIMIT "+this.position+",
"+this.maxparpage+"";
     requeteb=connection.prepareStatement(requete);
     if((typerecherche!=null && !typerecherche.equalsIgnoreCase("")) &&
(recherche!=null && !recherche.equalsIgnoreCase("")))requeteb.setString(1,
(String)"%"+recherche+"%");
     rs=requeteb.executeQuery();

     //executer la requete
     if(rs!=null)
```

```
    {
        //stocker tous les articles dans une liste
        while(rs.next())
        {
           //un enregistrement de plus
           this.compteurenregistrement++;
           //creer un objet article
           Article article=new Article();
           //renseigner l'objet article avec ses accesseurs
           if(rs.getString("id_article")==null)article.setId_article("0");
           else article.setId_article(rs.getString("id_article"));
           if(rs.getString("nomarticle")==null)article.setNomarticle("");
           else article.setNomarticle(rs.getString("nomarticle"));
           if(rs.getString("descriptionarticle")==null)article.
setDescriptionarticle("");
           else article.setDescriptionarticle(rs.getString
("descriptionarticle"));
           if(rs.getString("prixarticle")==null)article.setPrixarticle("0");
           else article.setPrixarticle(rs.getString("prixarticle"));
           if(rs.getString("datearticle")==null)article.setDatearticle("0");
           else article.setDatearticle(super.miseEnFormeDate(rs.getString
("datearticle")));
           if(rs.getString("photoarticle")==null)article.setPhotoarticle("");
           else article.setPhotoarticle(rs.getString("photoarticle"));
           if(rs.getString("vignettearticle")==null)article.
setVignettearticle("");
           else article.setVignettearticle(rs.getString("vignettearticle"));
           if(rs.getString("etatarticle")==null)article.setEtatarticle("0");
           else article.setEtatarticle(rs.getString("etatarticle"));
           if(rs.getString("id_categorie")==null)article.
setNomcategoriearticle("");
           else article.setNomcategoriearticle(super.getNomCategorieArticle
(rs.getString("id_categorie")));
           //stocker l'objet article dans la liste des articles
           listearticle.add((Article)article);
        }
     }
   }
   catch(Exception e)
   {
        System.out.println("Erreur dans la classe ArticleModele.java fonction
ListeArticleAdmin");
   }
   finally
   {
      try
      {
         //fermer la connexion
         if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
         if(requetea!=null)OutilsBaseDeDonnees.fermerConnexion(requetea);
         if(requeteb!=null)OutilsBaseDeDonnees.fermerConnexion(requeteb);
         if(connection!=null)OutilsBaseDeDonnees.fermerConnexion(connection);
      }
      catch(Exception ex)
      {
         System.out.println("Erreur dans la classe ArticleModele.java
fonction ListeArticleAdmin");
      }
   }
   //retourner la liste des articles
   return listearticle;
}

...
//fin de la classe
}
```

La fonction est plus complexe car elle permet de gérer la pagination complète du système avec le nombre d'enregistrements, la recherche et les tris. Il ne reste plus qu'à réaliser le codage de la vue *listearticles.jsp (/admin/vues/articles/listearticles.jsp)*. Cette vue est composée de plusieurs pages JSPF pour la mise en page et la

pagination.

```
<%@ page import="java.util.ArrayList" %>
<%@ page import="betaboutique.boiteoutils.Article" %>
<%
//url de l'application
String urlapplication=(String)getServletContext().getInit
Parameter("urlapplication");
//liste des articles
ArrayList listearticles=(ArrayList)request.getAttribute
("listearticles");
//action a appeler
String action=urlapplication+"admin/gestionarticles";
%>
<!--  inclure l'en-tete de la page -->
<%@ include file="../outils/entete.jspf" %>
<!--  inclure la pagination  -->
<%@ include file="../outils/pagination.jspf" %>

<!--  formulaire de recherche -->
<div id="rechercher"><img src="<%= urlapplication
%>img/louperecherche.png" title="Rechercher"
alt="Rechercher" align="absmiddle" border="0"/></div>
<form name="formulairerecherche" id="formulairerecherche"
action="<%= action %>" method="post">
<table cellspacing="4" cellpadding="0" id="tableau">
<tr>
<td><input type="text" class="input" id="recherche"
name="recherche" value="<%= recherche %>"/></td>
<td>
select name="typerecherche" id="typerecherche">
<option value="article.id_article"<%
if(typerecherche.equals("article.id_article"))out.print
("selected=\"selected\"");%>>Id</option>
<option value="article.nomarticle"<%
if(typerecherche.equals("article.nomarticle"))out.print
("selected=\"selected\"");%>>Nom</option>
<option value="article.prixarticle"<%
if(typerecherche.equals("article.prixarticle"))out.print
("selected=\"selected\"");%>>Prix</option>
</select>
</td>
<td>
<input type="submit" value="Rechercher" id="boutonrecherche"/>
</td>
</tr>
</table>
</form>

<br/><div class="titre"><a href="<%= action %>"><img
src="<%= urlapplication %>/img/articleadmin.png" border="0"
align="absmiddle"/> Liste des articles</a></div>

<table border="0" id="tableaubordure" cellspacing="0"cellpadding="0">
<tr align="center" class="entetetableau">
<td><a href="<%= action %>?p=<%= pageencours
%>&s=id_article&tri=<%= tri %><%= lienrecherche %>">Id<%
if(nomtri.equals("id_article") && tri.equals("ASC"))
{out.println("<img src=\"../img/asc.gif\" border=\"0\"/>");}
if(nomtri.equals("id_article") && tri.equals("DESC"))
{out.println("<img src=\"../img/desc.gif\"
border=\"0\"/>");}%></a></td>
<td><a href="<%= action %>?p=<%= pageencours
%>&s=nomarticle&tri=<%= tri %><%= lienrecherche %>">Nom<%
if(nomtri.equals("nomarticle") && tri.equals("ASC"))
{out.println("<img src=\"../img/asc.gif\" border=\"0\"/>");}
if(nomtri.equals("nomarticle") && tri.equals("DESC"))
{out.println("<img src=\"../img/desc.gif\"
border=\"0\"/>");}%></a></td>
<td><a href="<%= action %>?p=<%= pageencours %>
```

```
&s=prixarticle&tri=<%= tri %><%= lienrecherche %>">Prix<%
if(nomtri.equals("prixarticle") && tri.equals("ASC"))
{out.println("<img src=\"../img/asc.gif\" border=\"0\"/>");}
if(nomtri.equals("prixarticle") && tri.equals("DESC"))
{out.println("<img src=\"../img/desc.gif\" border=\"0\"/>");}%>
</a></td>
<td><a href="<%= action %>?p=<%= pageencours
%>&s=datearticle&tri=<%= tri %><%= lienrecherche %>">Date<%
if(nomtri.equals("datearticle") && tri.equals("ASC"))
{out.println("<img src=\"../img/asc.gif\" border=\"0\"/>");}
if(nomtri.equals("datearticle") && tri.equals("DESC"))
{out.println("<img src=\"../img/desc.gif\" border=\"0\"/>");}%>
</a></td>
<td>Vignette</td>
<td>Cat&eacute;gorie</td>
<td><a href="<%= action %>?p=<%= pageencours
%>&s=etatarticle&tri=<%= tri %><%= lienrecherche %>">Etat<%
if(nomtri.equals("etatarticle") && tri.equals("ASC"))
{out.println("<img src=\"../img/asc.gif\" border=\"0\"/>");}
if(nomtri.equals("etatarticle") && tri.equals("DESC"))
{out.println("<img src=\"../img/desc.gif\" border=\"0\"/>");}%>
</a></td>
<td colspan="3" width="130">Gestion</td>
</tr>

<%
//entete pagination
out.println(pagination);
%>
<%
for(int i=0;i<listearticles.size();i++)
{
    //récupérer l'objet dans la liste
    Article article=(Article)listearticles.get(i);
    if(i%2==0)out.println("<tr align=\"center\" class=\"ligneclaire\">");
    else out.println("<tr align=\"center\" class=\"lignefoncee\">");
    out.println("<td>"+article.getId_article()+"</td>");
    out.println("<td>"+article.getNomarticle()+"</td>");
    out.println("<td>"+article.getPrixarticle()+" Eur</td>");
    out.println("<td>"+article.getDatearticle()+"</td>");
    out.println("<td><img src=\"../imgarticles/"
+article.getVignettearticle()+"\" width=\"60\" height=\"60\"/></td>");
    out.println("<td>"+article.getNomcategoriearticle()+"</td>");
    if(article.getEtatarticle().equals("1"))
    {
        out.println("<td><img src=\"../img/actif.gif\"
alt=\"Actif\" title=\"Actif\"/></td>");
    }
    else
    {
        out.println("<td><img src=\"../img/inactif.gif\"
alt=\"Inactif\" title=\"Inactif\"/></td>");
    }
    out.println("<td><a href='"+action+"?action=
consulter&id_article="+article.getId_article()+"'>
<img src=\"../img/consulteradmin.png\" border=\"0\"
align=\"absmiddle\" title=\"Consulter\"/></a></td>");
    out.println("<td><a href='"+action+"?action=
modifier&id_article="+article.getId_article()+"'>
<img src=\"../img/modifieradmin.png\" border=\"0\"
align=\"absmiddle\" title=\"Modifier\"/></a></td>");
    out.println("<td><a href='javascript:
confirmerSuppressionArticle("+article.getId_article()+");'>
<img src=\"../img/supprimeradmin.png\" border=\"0\"
align=\"absmiddle\" title=\"Supprimer\"/></a></td>");
    out.println("</tr>");
}
%>

<%
```

```
//piedpage pagination
out.println(pagination);
%>
</table>
<%@ include file="../outils/piedpage.jspf" %>
```

Le code pour la pagination est ci-après. Ce code générique est commun à plusieurs pages, il est donc plus intéressant d'utiliser une page fragment (JSPF) (*/admin/vues/outils/pagination.jspf*).

```
<%
//pour les recherches
String typerecherche=(String)request.getAttribute
("typerecherche");
if(typerecherche==null)typerecherche="";
String recherche=(String)request.getAttribute("recherche");
if(recherche==null)recherche="";
String lienrecherche="&typerecherche="+typerecherche+"&recherche="+recherche;
//récupérer les informations de pagination
String nomtri=(String)request.getAttribute("nomtri");
String tri=(String)request.getAttribute("tri");
String maxparpage=(String)request.getAttribute("maxparpage");
String pageencours=(String)request.getAttribute("pageencours");
String compteurenregistrement=(String)request.getAttribute
("compteurenregistrement");
String totalenregistrement=(String)request.getAttribute
("totalenregistrement")
//calculs pour la pagination
int positiondebut=(int)(Integer.parseInt(maxparpage)
*Integer.parseInt(pageencours))+1;
int positionfin=(int)(positiondebut+Integer.parseInt
(compteurenregistrement)*1)-1;
int maxdepage=(int)Math.ceil((Double.parseDouble
(totalenregistrement)/Double.parseDouble(maxparpage)));
if(maxdepage==0)maxdepage=1;
int pageprecedente=Integer.parseInt(pageencours)-1;
int pagesuivante=Integer.parseInt(pageencours)+1;
int dernierepage=maxdepage-1;
int premierepage=0;
String tripagination="ASC";
if(tri.equalsIgnoreCase("ASC"))tripagination="DESC";
else tripagination="ASC";

StringBuffer pagination=new StringBuffer();
pagination.append("<tr class=\"fondblanc\"><td
colspan=\"17\">Enregistrements "+positiondebut+"-
"+positionfin+" sur "+totalenregistrement+" (Page :
"+pagesuivante+" sur "+maxdepage+")");
//afficher les boutons précédents que si nécessaire
if(!pageencours.equals("0"))
{
    pagination.append(" <a id=\"btnFirst\"
href=\""+action+"?p="+premierepage+"&tri="+tripagination+"\"
title=\"première page\"><img src=\"../img/premierepage.gif\"
border=\"0\" align=\"absmiddle\"/></a>");
    pagination.append(" <a id=\"btnPrev\"
href=\""+action+"?p="+pageprecedente+"&tri="+tripagination+"\"
title=\"Page précédente\"><img src=\"../img/pageprecedente.gif\"
border=\"0\" align=\"absmiddle\"/></a>");
}
//afficher les boutons suivants que si nécessaire
if(pagesuivante<maxdepage)
{
    pagination.append(" <a id=\"btnNext\"
href=\""+action+"?p="+pagesuivante+"&s="+nomtri+"&tri=
"+tripagination+"\" title=\"NPage suivante\"><img src=\"../img
/pagesuivante.gif\" border=\"0\" align=\"absmiddle\"/></a>");
    pagination.append(" <a id=\"btnLast\"
href=\""+action+"?p="+dernierepage+"&s="+nomtri+"&tri=
"+tripagination+"\" title=\"Dernière page\"><img src=\"../img
/dernierepage.gif\" border=\"0\" align=\"absmiddle\"/></a>");
```

```
}
pagination.append("</td></tr>");
%>
```

Deux variables sont utilisées pour les liens et la maintenabilité. La variable *action* permet de réaliser les liens vers la Servlet adaptée et la variable *urlapplication* permet de réaliser les liens, insérer les images... Enfin, dans le fichier *web.xml* la déclaration de l'URL de la Servlet est faite avec le modèle * afin de pouvoir passer des paramètres : *<url-pattern>/admin/gestionarticles/*</url-pattern>*. (ex : /admin/gestionarticles?action=modifier&id_article=4).

3. Optimisation avec JavaScript

La partie administration commence à être fournie en terme de fonctionnalités. Nous allons utiliser à cette étape du développement le langage JavaScript pour augmenter l'ergonomie de l'ensemble. Pour commencer, nous allons ajouter un fichier JavaScript qui servira de boîte à outils tout au long du projet (découpage de chaînes, messages d'alertes...).

Pour cela, nous utilisons le code suivant à enregistrer sous le nom : */javascript/boiteoutils.js*.

```
//supprimer un article en administration
function confirmerSuppressionArticle(id_article)
{
   if(confirm("Voulez-vous supprimer cet article ?"))
   {
      chemin="gestionarticles?action=supprimer&id_article="+id_article;
    document.location.href=chemin;
   }
   else
   {
   return;
   }
}
```

Le code contient pour le moment uniquement une fonction pour valider les confirmations de suppression. Il reste à inclure ce fichier dans l'en-tête JSPF avec la ligne suivante :

```
<head>
<title>Administration - BetaBoutique</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta name="description" content="BetaBoutique">
...
<!--  boite a outils -->
<script type="text/javascript" src="<%= urlapplication
%>/javascript/boiteoutils.js"></script>
</head>
```

Désormais si nous cliquons sur l'image de suppression d'un article, une boîte de confirmation est affichée.

Ce fichier *boiteoutils.js* sera incrémenté de fonctions au fur et à mesure du développement de l'application. Toutefois, il existe actuellement trois grandes librairies JavaScript qui permettent de réaliser des services intéressants. Les trois principales librairies actuelles en matière de JavaScript sont :

- *JQuery* (http://jquery.com/). C'est la librairie la plus légère et la plus simple à utiliser. Elle possède un fichier très léger pour les principales fonctionnalités et des fichiers plug-in pour des services adaptés.

- *ScriptAculous* (http://script.aculo.us/). Cette librairie est très complète mais assez lourde. Son utilisation est simple mais il y a parfois des conflits de noms entre les fonctions de la librairie et les fonctions d'autres librairies.

- *ExtJS* (http://extjs.com/). C'est actuellement la librairie la plus puissante. Son extrême lourdeur (400 Ko de librairies) et sa complexité sont ses principaux défauts. Les fonctionnalités proposées et son design Web 2.0 sont ses avantages.

Pour notre projet, nous allons utiliser la librairie JQuery de base ainsi que certains plug-ins au besoin. Pour installer la librairie, il suffit de la télécharger, de l'installer dans le répertoire */javascript* et de réaliser les inclusions nécessaires dans le fichier d'en-tête JSPF.

```
<!--  JQUERY -->
<script type="text/javascript" src="<%= urlapplication
%>/javascript/jquery/jquery.js"></script>
```

Nous allons également utiliser le plug-in *autocomplete* pour gérer les "auto-complétions". Pour cela, il est nécessaire de copier la librairie dans le répertoire */javascript/jquery/plugin/autocomplete*. Il faut aussi créer un fichier nommé recherche.js dans le répertoire */javascript/jquery/plugin/recherche*. Les inclusions sont réalisées avec les instructions suivantes :

```
<!--  boîte a outils -->
<script type="text/javascript" src="<%= urlapplication
%>/javascript/boiteoutils.js"></script>
<!-- plug-in pour les recherches -->
<script type="text/javascript" src="<%= urlapplication
%>/javascript/jquery/plugin/recherche/recherche.js"></script>
<!--  pour l'autocompletion  -->
<script type="text/javascript" src="<%= urlapplication
%>/javascript/jquery/plugin/autocomplete/jquery.bgiframe.min.js">
</script>
<script type="text/javascript" src="<%= urlapplication
%>/javascript/jquery/plugin/autocomplete/dimensions.js"></script>
<script type="text/javascript" src="<%= urlapplication
%>/javascript/jquery/plugin/autocomplete/jquery.autocomplete.js">
</script>
```

Dans le fichier *recherche.js*, nous allons pour le moment juste gérer l'affichage du formulaire de recherche. Sur l'image de la loupe, nous ajoutons le code qui permet de déclencher la fonction JavaScript *recherche()*.

```
<div id="rechercher"><a href="javascript:recherche();"><img
src="<%= urlapplication %>img/louperecherche.png"
title="Rechercher" alt="Rechercher" align="absmiddle"
border="0"/></a></div>
```

Nous allons également modifier notre feuille de style sur l'objet *formulairerecherche* (*<form name="formulairerecherche" id="formulairerecherche" action="<%= action %>" method="post">*) afin de ne pas afficher le formulaire de recherche par défaut.

```
#formulairerecherche
```

```
{
display:none;
}
```

Nous pouvons écrire le code de la fonction *recherche()* présente dans le fichier */javascript/jquery/plugin/recherche/recherche.js*.

```
//afficher ou cacher le formulaire de recherche
function recherche()
{
 //formulaire cache, l'afficher
 if($("#formulairerecherche").css("display")=="none")
 {
    $("#formulairerecherche").slideDown(500);
 }
 //formulaire affiche, le cacher
 else
 {
   $("#formulairerecherche").slideUp(500);
 }
}
```

Le code est très simple et utilise les fonctionnalités de la bibliothèque JQuery. Si le style de la feuille CSS est caché (*display:none*) alors il est montré avec la fonction *slideDown()*, sinon il est caché avec la fonction *slideUp()*. Nous avons donc une boîte de recherche qui s'ouvre de manière dynamique et ergonomique.

4. Optimisation avec Ajax

Asynchronous JavaScript and XML (XML et JavaScript asynchrones) est une solution libre pour le développement de fonctionnalités Web. Ajax est un framework (ensemble de technologies et librairies) qui regroupe HTML/XHTML, XML, CSS, JavaScript et l'objet *XMLHttpRequest* pour l'échange de données en asynchrone. Cette technologie très utilisée, n'est pas nouvelle et l'objet *XMLHttpRequest* est apparu en 2001 avec Internet Explorer 5.0. AJAX permet de déclencher de façon asynchrone en arrière plan, une page Web sans recharger la page courante. L'objet *XMLHttpRequest* est très lourd à manipuler, c'est pourquoi, beaucoup de concepteurs et sociétés ont développé des librairies plus simples pour utiliser Ajax telles que ScriptAculous, Advajax ou JQuery.

En effet, la librairie JQuery de base possède toutes les fonctionnalités pour utiliser la technologie Ajax. Nous allons ainsi mettre en œuvre la technologie Ajax pour l'auto-complétion des formulaires de recherche. Ce service d'auto-complétion sera générique à toutes nos pages, c'est-à-dire qu'il sera opérationnel sans modification du code, sans développement d'une Servlet spécifique à chaque fois...

Pour cela, le nom de la table et le nom du champ sur lequel réaliser la recherche seront utilisés en paramètre.

Voici les étapes à suivre pour installer l'auto-complétion :

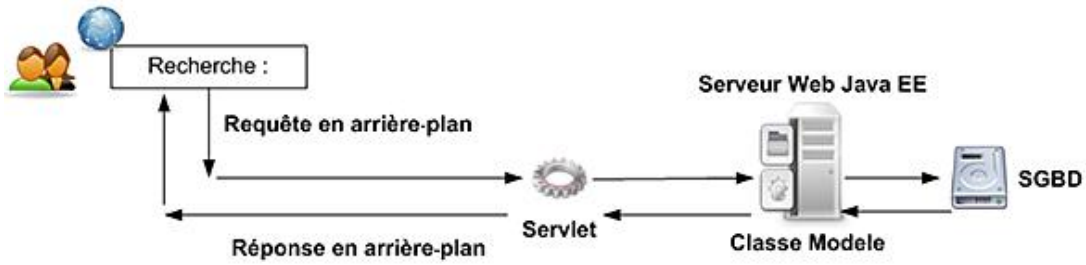
1 - Utilisation de la librairie Ajax (*JQuery*).

2 - Téléchargement de plug-in *autocomplete*.

3 - Création d'un fichier *recherche.js* avec notre code personnel.

4 - Insertion des librairies dans notre application (<script src=...).

5 - Développement d'une Servlet générique de recherche asynchrone (*ServletAutoComplete*).



Les requêtes utilisateurs seront réalisées en arrière-plan et la réponse sera effectuée sans rechargement de la page.

Nous allons commencer par télécharger et installer les librairies nécessaires à l'utilisation d'Ajax et à l'auto-complétion. Puis, nous allons coder le fichier de recherche (*recherche.js*) pour l'utilisation d'Ajax.

```
//attribut
```

```
var attribut=null;
var table=null;
var url=null;

//au chargement de la page
$(document).ready(function(){

 //ecouter la liste pour le type de recherche souhaitee
 $("#typerecherche").attr("onChange","changerTypeRecherche()");

 //declencher la fonction pour le premier type
 changerTypeRecherche();

 //afficher le formulaire de recherche si il contient
une recherche, sinon le cacher
 var recherche=$('#recherche').val();
 if(recherche!="")
 {
     //montrer le formulaire de recherche
     $("#formulairerecherche").show();
 }
 else
 {
     //cacher le formulaire de recherche
     $("#formulairerecherche").hide();
 }

});


//changer le type de recherche
function changerTypeRecherche()
{
 //recuperer le type de recherche souhaitee
 attribut=$('#typerecherche').val();
 if(attribut!=null)
 {
    //recuperer la table (premiere partie de l'attribut)
    var tab=attribut.split('.');
    table=tab[0];
    //enlever les espaces
    table=jQuery.trim(table);
    attribut=jQuery.trim(attribut);

    //declencher l'autocompletion
    if(attribut!='' && table!=''){
    //mettre en forme l'url
    url="autocomplete?attribut="+attribut+"&table="+table;
    $("#recherche").autocomplete(url, {
                    delay: 400,
                    width:400,
                    cacheLength:1,
                    matchSubset:false,
                    mustMatch : true,
                    minChars:1,
                    autoFill: true
            });
    }
  }
}

//afficher ou cacher le formulaire de recherche
function recherche()
{
 //formulaire cache, l'afficher
 if($("#formulairerecherche").css("display")=="none")
 {
    $("#formulairerecherche").slideDown(500);
 }
 //formulaire affiche, le cacher
```

```
 else
 {
   $("#formulairerecherche").slideUp(500);
 }
}
```

Dans le code précédent, il faut d'abord ajouter un écouteur sur la liste déroulante des recherches. Ainsi, lors du changement dans la liste nous aurons automatiquement le nom du champ sur lequel réaliser la recherche (*article.nomarticle*, *article.id_article*...).

Ensuite, il faut déclencher la fonction *changerTypeRecherche()* qui permet de déclarer l'auto-complétion sur le champ nommé *recherche* de la page. Le code qui réalise l'opération Ajax est le suivant :

```
//declencher l'autocompletion
if(attribut!='' && table!=''){
    //mettre en forme l'url
    url="autocomplete?attribut="+attribut+"&table="+table;
    $("#recherche").autocomplete(url, {
                    delay: 400,
                    width:400,
                    cacheLength:1,
                    matchSubset:false,
                    mustMatch : true,
                    minChars:1,
                    autoFill: true
            });
}
```

Pour résumer, si nous recevons un attribut (ex : *article.nomarticle*), alors nous mettons en forme l'URL à déclencher et nous précisons les paramètres de l'auto-complétion. Notre URL sera appelée en arrière-plan dans un délai de 400ms, la réponse sera affichée dans une boîte de 400px de large et nous déclenchons l'auto-complétion Ajax à partir de la saisie d'un caractère (*minChars:1*) et nous autorisons le choix dans la liste résultat (*autoFill:true*).

Nous avons déclaré les librairies JavaScript, développé notre fonction de gestion de l'auto-complétion, codé notre formulaire dans la page JSP (code ci-après), il ne nous reste plus que la création de la Servlet *ServletAutoComplete*.

La déclaration est réalisée comme toujours dans le fichier de gestion de l'application (*web.xml*).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    ...
    <servlet>
        <servlet-name>servletautocomplete</servlet-name>
        <servlet-class>betaboutique.servlets.ServletAutoComplete
</servlet-class>
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>servletautocomplete</servlet-name>
        <url-pattern>/admin/autocomplete/*</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Puis nous réaliserons la Servlet *ServletAutoComplete* avec un message très simple affiché dans la console en cas de recherche dans le formulaire.

```
package betaboutique.servlets;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

@SuppressWarnings("serial")
public class ServletAutoComplete extends HttpServlet {

    //variables de la classe
```

```
    DataSource ds=null;
    //traitements
    public void doGet(HttpServletRequest request,
HttpServletResponse response)throws ServletException,
IOException
    {
        System.out.println("Dans la Servlet d'auto-completion");
    }

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

Ensuite, nous rechargeons l'application et nous vérifions qu'une recherche avec au moins un caractère dans le formulaire adapté, déclenche un message dans la console en arrière-plan, sans recharger la page (voici la puissance d'Ajax).

Nous allons maintenant réaliser le code de notre contrôleur (*ServletAutoComplete*) afin de vérifier l'intégrité des données reçues et réaliser une recherche dans la base de données avec les informations adaptées. Pour cela, nous créons une classe modèle nommée *AutoCompleteModele* qui contient uniquement une fonction avec la requête SQL associée. La requête SQL reçoit en paramètre le nom du champ à rechercher et la table concernée. Nous avons bien dans ce cas un moteur d'auto-complétion générique (fonctionnel pour n'importe quel service : articles, clients et commandes).

```
package betaboutique.modeles;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.sql.DataSource;
import betaboutique.boiteoutils.OutilsBaseDeDonnees;
import java.util.ArrayList;

public class AutoCompleteModele
{
    //variables de classe
    DataSource ds=null;
    Connection connection=null;
    ResultSet rs=null;
    //liste des objets
    ArrayList<String> listeautocomplete=new ArrayList<String>();

/*********************************************************
 * constructeur
*********************************************************/
public AutoCompleteModele(DataSource ds)
{
    //récupérer la DataSource de la servlet
    this.ds=ds;
}

/*********************************************************
 * liste des données demandées en auto-complétion
*********************************************************/
public ArrayList ListeAutoCompleteAdmin(String attribut,
String table, String saisie, String limit)
{
    //statement
    PreparedStatement requete=null;

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        //enregistrements
        requete=connection.prepareStatement("SELECT
DISTINCT("+attribut+") FROM "+table+" WHERE "+attribut+"
```

```
LIKE ? ORDER BY "+attribut+" LIMIT 0,"+limit+"");
        requete.setString(1,(String)"%"+saisie+"%");
        rs=requete.executeQuery();
        //exécuter la requête
        if(rs!=null)
        {
            //stocker toutes les reponses dans une liste
            while(rs.next())
            {
                if(rs.getString(attribut)!=null)
                {
                        listeautocomplete.add((String)
rs.getString(attribut));
                }
            }
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
AutoCompleteModele.java fonction ListeAutoCompleteAdmin");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(rs!=null) OutilsBaseDeDonnees.fermerConnexion(rs);
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
AutoCompleteModele.java fonction ListeAutoCompleteAdmin");
        }
    }
    //retourner la liste
    return listeautocomplete;
}

//fin de la classe
}
```

Le code de la Servlet contrôleur est également très simple, il faut réaliser un contrôle des données, déclencher le modèle pour récupérer une liste de données et se diriger vers la vue *listeautocomplete.jsp* pour l'affichage.

```
package betaboutique.servlets;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import betaboutique.modeles.AutoCompleteModele;

@SuppressWarnings("serial")
public class ServletAutoComplete extends HttpServlet {

    //variables de la classe
    DataSource ds=null;

    //traitements
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer la datasource du plug-in dans un attribut
```

```
présent dans le contexte de la servlet
        ds=(DataSource)getServletContext().getAttribute("datasource");
        //créer le modèle
        AutoCompleteModele autocompletemodele=new AutoCompleteModele(ds);
        //fermer la datasource
        this.ds=null;
        //recuperer les attributs
        String attribut=(String)request.getParameter("attribut").trim();
        String table=(String)request.getParameter("table").trim();
        String q=(String)request.getParameter("q").trim();
        String limit=(String)request.getParameter("limit").trim();
        //verifier les parametres
        if( (attribut!=null && !attribut.equals("")
&& attribut.length()>0) && (table!=null && !table.equals("")
&& table.length()>0) && (q!=null && !q.equals("") &&
q.length()>0) && (limit!=null && !limit.equals("") &&
limit.length()>0) )
        {
            //recuperer la liste des données qui correspondent
            ArrayList listeautocomplete=autocompletemodele.Liste
AutoCompleteAdmin(attribut,table,q,limit);
            //retourner la liste
            request.setAttribute("listeautocomplete",listeautocomplete);
            //vider par sécurité
            listeautocomplete=null;
            //retourner sur la page d'affichage des enregistrements trouvés
            getServletContext().getRequestDispatcher
("/admin/vues/outils/listeautocomplete.jsp").forward(request,
response);
        }
    }

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

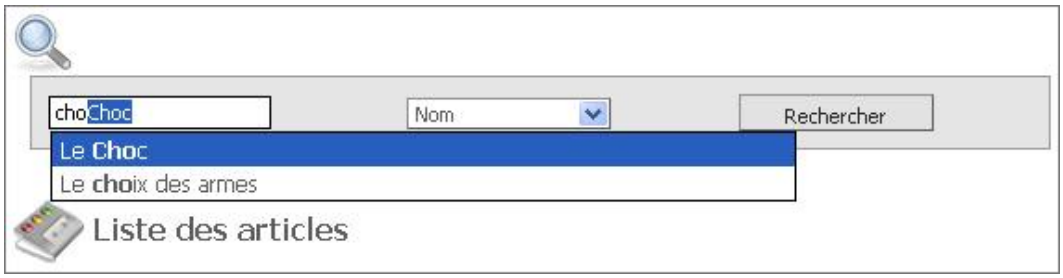
Le service d'auto-complétion est quasiment terminé, il reste le code de la vue *listeautocomplete.jsp*. Le code associé récupère la liste des données trouvées, et pour chaque attribut, affiche sa valeur avec un retour à la ligne. C'est ensuite, et de manière transparente, le code JavaScript de JQuery (autocomplete) qui va mettre en forme la réponse pour l'afficher dans la liste déroulante de recherche.

```
<%@ page import="java.util.ArrayList" %>
<%
//liste des valeurs retournés
ArrayList listeautocomplete=(ArrayList)request.getAttribute
("listeautocomplete");
%>
<%
    for(int i=0;i<listeautocomplete.size();i++)
    {
        //récupérer l'objet dans la liste
        String attribut=(String)listeautocomplete.get(i);
        out.println(attribut);
        //autre syntaxe out.println(cle+"|valeur non affichee");
    }
%>
```

Il reste enfin une dernière étape, l'intégration de la feuille de style JQuery pour la mise en page et l'affichage de l'auto-complétion. Cette opération est simple, il faut ajouter la feuille de style avec le code suivant dans la page *entete.jspf*.

```
<html>
<head>
<title>Administration - BetaBoutique</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta name="description" content="BetaBoutique">
...
```

```
<!--  feuilles de style pour autocomplete -->
<link rel="stylesheet" type="text/css" href="<%= urlapplication
%>/javascript/jquery/plugin/autocomplete/jquery.autocomplete.css" />
```



Chaque saisie de texte entraîne une requête adaptée dans le SGBD et retrouve les réponses sans rechanger la page et son formulaire de recherche.

Nous pouvons désormais réaliser des recherches et bénéficier de l'auto-complétion en fonction du type de recherche (ex : nom, id, prix...) présent dans la liste déroulante.

5. Optimisation des Servlets

Le principe de déclaration d'une Servlet pour chaque action associée est très lourd en terme de programmation, de ligne de code mais aussi d'homogénéité. Pour éviter cela, la technique consiste à utiliser un paramètre supplémentaire dans les Servlets (ex : action, mode, role...) et de déclarer une seule Servlet de gestion par service. Dans notre cas, la Servlet *ServletGestionArticles* possède un attribut supplémentaire nommé *action* qui permet de savoir si nous voulons la liste des articles, consulter un article, modifier une fiche ou supprimer un produit.

Voici les différentes URL possibles dans notre cas :

- http://localhost:8080/betaboutiquemvc/admin/gestionarticles
- http://localhost:8080/betaboutiquemvc/admin/gestionarticles?action=consulter&id_article=1
- http://localhost:8080/betaboutiquemvc/admin/gestionarticles?action=modifier&id_article=1
- http://localhost:8080/betaboutiquemvc/admin/gestionarticles?action=supprimer&id_article=1

Dans le code de la Servlet de gestion, le paramètre *action* a une valeur par défaut (liste des articles dans notre cas) et un traitement adapté en fonction de l'action associée.

```
package betaboutique.servlets;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import betaboutique.modeles.ArticleModele;

@SuppressWarnings("serial")
public class ServletGestionArticles extends HttpServlet {

    //variables de la classe
    DataSource ds=null;

    //traitements
    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException
    {
        //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
        ds=(DataSource)getServletContext().getAttribute("datasource");
```

```
        //créer le modèle
        ArticleModele articlemodele=new ArticleModele(ds);
        //redonner la datasource
        this.ds=null;

        //action a réaliser (liste des articles,
consultation, modification, suppression ou création)
        String action=(String)request.getParameter("action");
        //action par défaut (liste)
        if(action==null || action.equalsIgnoreCase(""))
        {
            action="liste";
        }

        //liste des articles
        if(action.equals("liste"))
        {
            ...
        }

        //consulter un article
        if(action.equals("consulter"))
        {
            ...
        }

        ...
    }

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

Notre Servlet est optimisée, il ne nous reste plus qu'à créer le code pour la consultation, modification et suppression d'un article.

Consultation d'un article

Nous commençons par le code de consultation d'un article. Pour réaliser la consultation, les opérations sont assez simples. Il faut récupérer l'id de l'article à consulter, déclencher la classe modèle afin de récupérer toutes les informations sur l'article sélectionné, et retourner sur la vue qui sert d'affichage.

```
...
//consulter un article
if(action.equals("consulter"))
{
        //récupérer le numéro de l'article à consulter
        String id_article=(String)request.getParameter("id_article");
        if(id_article!=null && !id_article.equals(""))
        {
            //récupérer l'article dans la base de données
avec la classe modele
            Article article=(Article)articlemodele.getArticle(id_article);
            //retourner l'objet article a la vue
            request.setAttribute("article",article);
            //retourner sur la page d'affichage des articles
en administration
getServletContext().getRequestDispatcher("/admin/vues/article/
consulterarticle.jsp").forward(request, response);
        }
}
...
```

Le code de la classe modèle utilise un objet JavaBean instance de la classe *Article* pour stocker les informations.

```
...
```

```
/*************************************************************
 * récupérer un article
*************************************************************/
public Article getArticle(String id_article)
{
    PreparedStatement requetea=null;
    //créer un objet article
    Article article=new Article();

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        requetea=connection.prepareStatement("SELECT *
FROM article WHERE article.id_article=?");
        requetea.setString(1,(String)id_article);
        rs = requetea.executeQuery();
        //exécuter la requête
        if(rs!=null)
        {
            if(rs.next())
            {
                //renseigner l'objet article avec ses accesseurs
                if(rs.getString("id_article")==null)
article.setId_article("0");
                else article.setId_article(rs.getString
("id_article"));
                if(rs.getString("nomarticle")==null)
article.setNomarticle("");
                else article.setNomarticle(rs.getString
("nomarticle"));
                if(rs.getString("descriptionarticle")==null)
article.setDescriptionarticle("");
                else article.setDescriptionarticle
(rs.getString("descriptionarticle"));
                if(rs.getString("prixarticle")==null)
article.setPrixarticle("0");
                else article.setPrixarticle(rs.getString
("prixarticle"));
                if(rs.getString("datearticle")==null)
article.setDatearticle("0");
                else article.setDatearticle
(super.miseEnFormeDate(rs.getString("datearticle")));
                if(rs.getString("photoarticle")==null)
article.setPhotoarticle("");
                else article.setPhotoarticle(rs.getString
("photoarticle"));
                if(rs.getString("vignettearticle")==null)
article.setVignettearticle("");
                else article.setVignettearticle(rs.getString
("vignettearticle"));
                if(rs.getString("etatarticle")==null)
article.setEtatarticle("0");
                else article.setEtatarticle(rs.getString
("etatarticle"));
                if(rs.getString("id_categorie")==null)
article.setNomcategorieearticle("");
                else article.setNomcategorieearticle
(super.getNomCategorieArticle(rs.getString("id_categorie")));

            }
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
ArticleModele.java fonction getArticle");
    }
    finally
    {
```

```
        try
        {
            //fermer la connexion
            if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
            if(requetea!=null)OutilsBaseDeDonnees.fermerConnexion(requetea);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ArticleModele.java fonction getArticle");
        }
    }

    //retourner l'objet article
    return article;
}
```

Enfin, la vue */admin/vues/article/consulterarticle.jsp* permet d'afficher l'article et possède un lien pour accéder au formulaire en modification.

```
<%@ page import="betaboutique.boiteoutils.Article" %>
<%
//url de l'application
String urlapplication=(String)getServletContext().getInit
Parameter("urlapplication");
//article
Article article=(Article)request.getAttribute("article");
//action a appeler
String action=urlapplication+"admin/gestionarticles";
%>
<!--  inclure l'en-tete de la page -->
<%@ include file="../outils/entete.jspf" %>
<br/><div class="titre"><a href="<%= action %>"><img
src="<%= urlapplication %>/img/articleadmin.png" border="0"
align="absmiddle"/> Liste des articles</a></div>
<form action="<%= action %>?action=modifier&id_article=<%=
article.getId_article() %>" method="POST">
<table border="0" id="tableau" cellpadding="0" cellspacing="0"
width="600">
<tr><td>Id</td><td><%= article.getId_article() %></td></tr>
<tr><td>Nom</td><td><%= article.getNomarticle() %></td></tr>
<tr><td valign="top">Description</td><td><%=
article.getDescriptionarticle() %></td></tr>
<tr><td>Prix</td><td><%= article.getPrixarticle() %>
Euros</td></tr>
<tr><td>Date cr&eacute;ation</td><td><%=
article.getDatearticle() %></td></tr>
<tr><td>Photo</td><td><img src="../imgarticles/<%=
article.getPhotoarticle() %>"/></td></tr>
<tr><td>Vignette</td><td><img src="../imgarticles/<%=
article.getVignettearticle() %>"/></td></tr>
<tr><td>Etat</td><td><%= article.getEtatarticle()
%></td></tr>
<tr><td>Cat&eacute;gorie</td><td><%=
article.getNomcategoriearticle() %></td></tr>
<tr><td align="center" colspan="2"><input type="submit"
name="modifier" value="Modifier" class="bouton"/></td></tr>
</table>
</form>

<%@ include file="../outils/piedpage.jspf" %>
```

Modification d'un article

Nous allons réaliser le formulaire de modification selon le même principe que le formulaire de consultation. Cependant, il existe une petite différence. Il faut commencer par récupérer les informations de l'article, afficher le formulaire complet et ensuite, valider les modifications.

Nous modifions la fonction *getArticle()* de la classe modèle afin de gérer l'id de la catégorie.

```
...
if(rs.next())
{
  //renseigner l'objet article avec ses accesseurs
  if(rs.getString("id_article")==null)article.setId_article("0");
  else article.setId_article(rs.getString("id_article"));
  ...
  if(rs.getString("id_categorie")==null)article.setId_categoriearticle("0");
  else article.setId_categoriearticle(rs.getString("id_categorie"));
  if(rs.getString("id_categorie")==null)article.setNomcategoriearticle("");
  else article.setNomcategoriearticle(super.getNomCategorie
Article(rs.getString("id_categorie")));
}
...
```

Il faut ensuite ajouter une nouvelle fonction nommée *getListeCategorieArticle()* dans la classe modèle mère (méthode utilisée par plusieurs sous-classes par la suite) afin de retourner uniquement la liste des catégories d'articles et une classe JavaBean pour la gestion des catégories.

```
...
public ArrayList getListeCategorieArticle()
{
    //vider la liste
    ArrayList<Categorie> listeobjetcategoriearticle=new
ArrayList<Categorie>();
    String requete=null;

    //Récupérer tous les médias mis en vente par
l'utilisateur en présence
    try
    {
        //ouvrir une connexion
        connection1=ds.getConnection();
        instruction1=connection1.createStatement();
        //liste des catégories
        requete="SELECT DISTINCT(id_categorie),nomcategorie
FROM categorie ORDER BY categorie.nomcategorie";
        rs1=instruction1.executeQuery(requete);

        //exécuter la requête
        if(rs1!=null)
        {
            //stocker tous les catégories dans une liste
            while(rs1.next())
            {
                //créer un objet categorie
                Categorie categorie=new Categorie();
                if(rs1.getString("id_categorie")==null)
categorie.setId_categorie("0");
                else categorie.setId_categorie(rs1.getString
("id_categorie"));
                if(rs1.getString("nomcategorie")==null)
categorie.setNomcategorie("");
                else categorie.setNomcategorie(rs1.getString
("nomcategorie"));
                //ajouter à la liste
                listeobjetcategoriearticle.add((Categorie)
categorie);
            }
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
Modele.java fonction getListeCategorieArticle");
    }
    finally
    {
```

```
        try
        {
            //fermer la connexion
            if(rs1!=null)OutilsBaseDeDonnees.fermerConnexion(rs1);
            if(instruction1!=null)OutilsBaseDeDonnees.fermerConnexion
(instruction1);
            if(connection1!=null)OutilsBaseDeDonnees.fermerConnexion
(connection1);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
Modele.java fonction getListeCategorieArticle");
        }
    }

    //retourner la liste des objets categorie
    return listeobjetcategoriearticle;
}
...
```

La Servlet de gestion des articles est adaptée pour traiter la modification d'un produit. La liste des catégories est récupérée (pour le choix futur dans une liste déroulante).

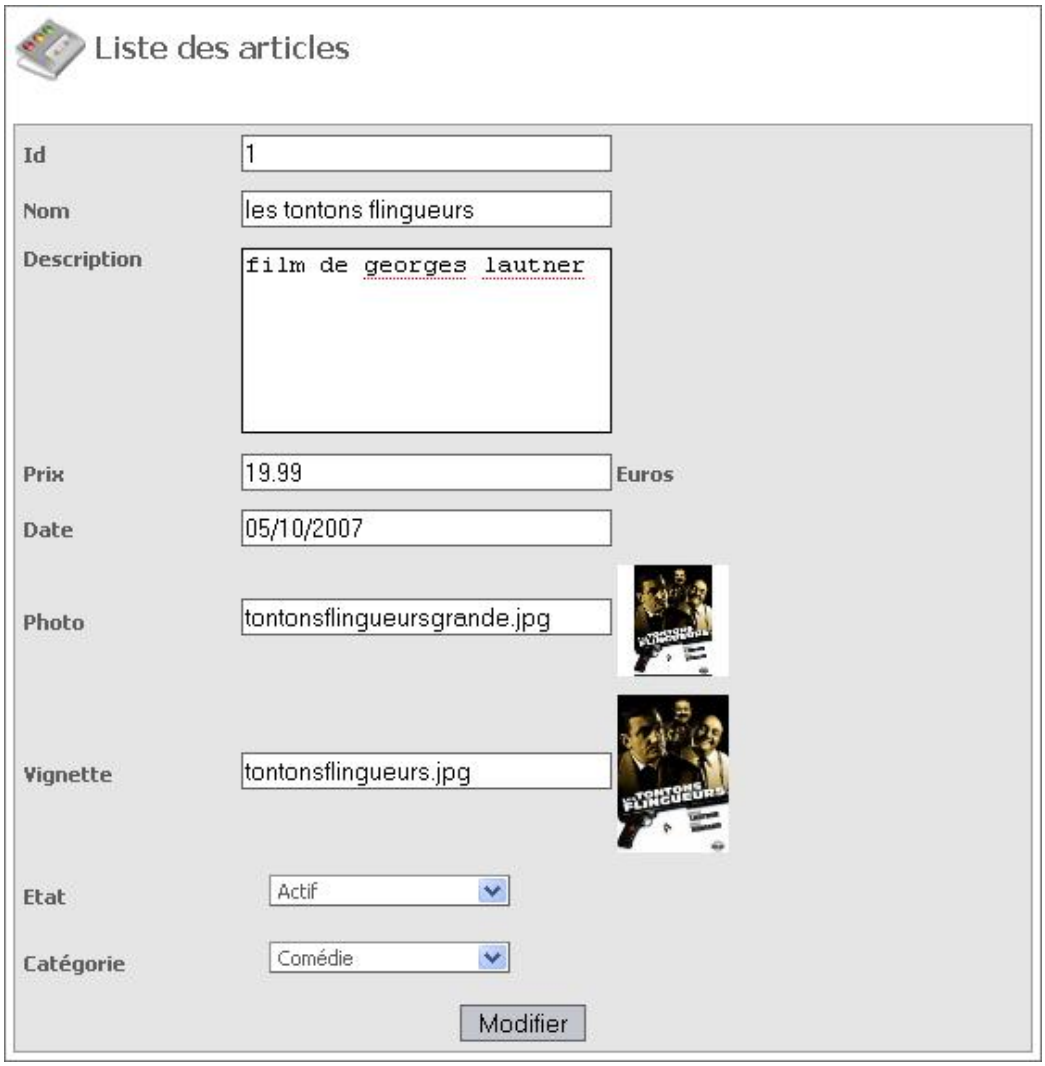
```
...
//modifier un article
else if(action.equals("modifier"))
{
    //récupérer le numéro de l'article à modifier
    String id_article=(String)request.getParameter("id_article");
    if(id_article!=null && !id_article.equals(""))
    {
        //récupérer l'article dans la base de données avec la classe modele
        Article article=(Article)articlemodele.getArticle(id_article);
        //récupérer la liste des catégories d'articles
        ArrayList listecategorieearticles=(ArrayList)
articlemodele.getListeCategorieArticle();
        //retourner l'objet article a la vue
        request.setAttribute("article",article);
        //retourner la liste des catégories d'articles à la vue
        request.setAttribute("listecategorieearticles",listecategorieearticles);
        //retourner sur la page d'affichage des articles en administration
getServletContext().getRequestDispatcher("/admin/vues/article/
modifierarticle.jsp").forward(request, response);
    }
}
...
```

Enfin, il reste le code de la vue *modifierarticle.jsp* qui permet d'afficher les informations et de déclencher l'action *validermodifier* pour insérer les nouvelles données modifiées dans la base de données.

```
<%@ page import="java.util.ArrayList" %>
<%@ page import="betaboutique.boiteoutils.Article" %>
<%@ page import="betaboutique.boiteoutils.Categorie" %>
<%
//url de l'application
String urlapplication=(String)getServletContext().getInit
Parameter("urlapplication");
//article
Article article=(Article)request.getAttribute("article");
//liste des categories
ArrayList listecategorieearticles=(ArrayList)request.get
Attribute("listecategorieearticles");
//action a appeler
String action=urlapplication+"admin/gestionarticles";
%>
<!--  inclure l'en-tete de la page -->
<%@ include file="../outils/entete.jspf" %>

<br/><div class="titre"><a href="<%= action %>"><img
```

```
src="<%= urlapplication %>/img/articleadmin.png" border="0"
align="absmiddle"/> Liste des articles</a></div>

<form action="<%= action %>?action=validermodifier"
method="POST">
<table border="0" id="tableau" cellpadding="0" cellspacing="0"
width="50%">
<tr><td>Id</td><td><input type="text" name="id_article"
class="input" value="<%= article.getId_article() %>"
readonly="readonly"/></td></tr>
<tr><td>Nom</td><td><input type="text" name="nomarticle"
class="input" value="<%= article.getNomarticle()
%>"/></td></tr>
<tr><td valign="top">Description</td><td><textarea
class="textarea" name="descriptionarticle"><%=
article.getDescriptionarticle() %></textarea></td></tr>
<tr><td>Prix</td><td><input type="text" name="prixarticle"
class="input" value="<%= article.getPrixarticle() %>"/>
Euros</td></tr>
<tr><td>Date</td><td><input type="text" name="datearticle"
class="input" value="<%= article.getDatearticle()
%>"/></td></tr>
<tr><td>Photo</td><td><input type="text" name="photoarticle"
class="input" value="<%= article.getPhotoarticle() %>"/> 
<img src="../imgarticles/<%= article.getPhotoarticle() %>"
width="60" align="absmiddle"/></td></tr>
<tr><td>Vignette</td><td><input type="text"
name="vignettearticle" class="input" value="<%=
article.getVignettearticle() %>"/> <img
src="../imgarticles/<%= article.getVignettearticle()
%>" width="60" align="absmiddle"/></td></tr>
<tr><td>Etat</td>
<td>
<%
out.println("<select name=\"etatarticle\" class=\"select\">");
out.println("<option value=\"1\"");
if(article.getEtatarticle().equals("1")) out.println("
selected=\"selected\" ");
out.println(">Actif</option>");
out.println("<option value=\"0\"");
if(article.getEtatarticle().equals("0")) out.println("
selected=\"selected\" ");
out.println(">Inactif</option>");
out.println("</select>");
%>
</td></tr>
<tr><td>Cat&eacute;gorie</td>
<td>
<%
//liste des categories
out.println("<select name=\"id_categorie\" class=\"select\">");
for(int i=0;i<listecategoriearticles.size();i++)
{
    //récupérer l'objet dans la liste
    Categorie categorie=(Categorie)listecategoriearticles.get
(i);
    out.println("<option value=\""+categorie.getId_categorie()
+"\"");
    if(categorie.getId_categorie().equals(article.get
Id_categorie())) out.println(" selected=\"selected\" ");
    out.println(">"+categorie.getNomcategorie()+"</option>");
}
out.println("</select>");
%>
</td></tr>
<tr><td align="center" colspan="2"><input type="submit"
name="modifier" value="Modifier" class="bouton"/></td></tr>
</table><br/>
</form>
<%@ include file="../outils/piedpage.jspf" %>
```

Un clic sur le bouton de modification va déclencher l'envoi des données (avec la méthode POST) à la Servlet afin d'opérer les modifications.



Dans les exemples de ce guide (notamment l'exemple précédent), il n'y a pas de vérification des données au moment de la création et de la modification. Il est évident que dans une application professionnelle, il faudrait vérifier la syntaxe de la date, la présence du prix, la présence du nom de l'article... du côté serveur au sein de la Servlet adaptée et dans l'idéal, côté serveur et côté client (avec le langage JavaScript).

Il ne reste plus qu'à développer le code de l'action *validermodifier* afin d'insérer les nouvelles données de l'article dans la base de données. Pour cela, nous allons utiliser la librairie *commons.beanutils.BeanUtils* pour gérer les JavaBeans. Par contre, cette utilisation nécessite une rigueur dans l'utilisation des noms des attributs. En effet, il est nécessaire de conserver la correspondance exacte entre les attributs de la table et du JavaBean. Dans le code de la classe *Article*, le nom de la catégorie n'est pas correct, il faut utiliser :

```
private String id_categorie=null;
```

à la place de :

```
private String id_categoriearticle=null;
```

Ce changement entraîne quelques modifications dans le code des classes modèles et des vues JSP. Il est également nécessaire de modifier la méthode *getArticle()* de la classe modèle pour gérer l'id de la catégorie.

```
...
if(rs.getString("id_categorie")==null)article.setId_categorie("0");
else article.setId_categorie(rs.getString("id_categorie"));
...
```

Le code de la Servlet permet de renseigner le JavaBean et insère les données du JavaBean dans la base de données. La requête de modification retourne une valeur qui va servir pour la gestion des messages d'erreurs ou de succès. Le code complet de la Servlet est alors le suivant :

```
...
public class ServletGestionArticles extends HttpServlet {
...
   //pour la gestion des messages d'erreurs
   ArrayList<String> erreurs=new ArrayList<String>();
   //pour la gestion des messages de succès
   ArrayList<String> succes=new ArrayList<String>();
...
    //on veut modifier un article
    else if(action.equals("modifier"))
    {
       //récupérer le numéro de l'article à modifier
       String id_article=(String)request.getParameter("id_article");
       if(id_article!=null && !id_article.equals(""))
       {
          //récupérer l'article dans la base de données avec la classe modele
          Article article=(Article)articlemodele.getArticle(id_article);
          //récupérer la liste des catégories d'articles
          ArrayList
listecategoriearticles=(ArrayList)articlemodele.getListeCategorieArticle();
          //retourner l'objet article a la vue
          request.setAttribute("article",article);
          //retourner la liste des catégories d'articles à la vue
          request.setAttribute("listecategoriearticles",
listecategoriearticles);
          //retourner sur la page d'affichage des articles en administration

  getServletContext().getRequestDispatcher("/admin/vues/article/
modifierarticle.jsp").forward(request, response);
       }
    }

    //on veut valider la modification d' un article
    else if(action.equals("validermodifier"))
    {
       //on récupère les données dans la requête et on les insère
dans le JavaBean article
       Article article=new Article();
       //création d'une collection des paramètres reçus par la requête
       HashMap map=new HashMap();
       Enumeration names=request.getParameterNames();
       while (names.hasMoreElements())
       {
           //on met chaque paramètre de la requête dans une collection
           String name=(String)names.nextElement();
           map.put(name,request.getParameterValues(name));
       }
       try
       {
           // on insère toutes nos données dans le Bean article en une seule
étape
           BeanUtils.populate(article, map);
           System.out.println("description :
"+article.getDescriptionarticle());
           //modifier l'article dans la base de données
           int resinstruction=articlemodele.modifierArticle(article);
           //en cas d'erreur avec la base de données, ajouter un message
d'erreur
           if(resinstruction!=1)
           {
               //ajouter un message d'erreur
               erreurs.add("Erreur lors de la modification de l'article");
           }
           //tout est OK
           else
           {
```

```
                //ajouter un message de succes
                succes.add("Modification de l'article réalisée avec succès");
            }
            //envoyer les liste de messages à la vue et retourner sur la vue
principale
            request.setAttribute("erreurs",erreurs);
            request.setAttribute("succes",succes);
            //retourner sur la page d'affichage des articles en
administration
   getServletContext().getRequestDispatcher("/admin/gestionarticles?action=
liste").forward(request, response);
        }
        catch(Exception e)
        {
            System.out.println("Erreur lors de la lecture des informations");
        }
...
}
```

```
/*********************************************************
 * modifier un article
*********************************************************/
public int modifierArticle(Article article)
{
    int resinstruction=0;
    PreparedStatement requete=null;
    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        requete=connection.prepareStatement("UPDATE article SET
nomarticle=?,descriptionarticle=?,prixarticle=?,datearticle=?,
photoarticle=?,vignettearticle=?,etatarticle=?,id_categorie=?
WHERE article.id_article=?");
        requete.setString(1,(String)article.getNomarticle());
        requete.setString(2,(String)article.get
Descriptionarticle());
        requete.setString(3,(String)article.getPrixarticle());
        requete.setString(4,(String)super.miseEnFormeDatevers
Base(article.getDatearticle()));
        requete.setString(5,(String)article.getPhotoarticle());
        requete.setString(6,(String)article.getVignettearticle());
        requete.setString(7,(String)article.getEtatarticle());
        requete.setString(8,(String)article.getId_categorie());
        requete.setString(9,(String)article.getId_article());
        resinstruction=requete.executeUpdate();
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
ArticleModele.java fonction modifierArticle");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ArticleModele.java fonction modifierArticle");
        }
    }

    //résultat de la modification
```

```
    return resinstruction;
}

/*********************************************************
 * mise en forme de la date (jj/mm/aaaa -> aaaammjj )
*********************************************************/
public String miseEnFormeDateversBase(String date)
{
    if(date.length()>=10)
    {
    String jour=date.substring(0,2);
    String mois=date.substring(3,5);
    String annee=date.substring(6,10);
    date=annee+mois+jour;
    }
    //retourner la date
    return date;
}
```

Il ne reste désormais plus qu'à modifier la page JSP d'en-tête pour afficher les messages de succès et d'erreurs éventuels de la requête.

```
...
<%@ page import="java.util.ArrayList" %>
<html>
<head>
<title>Administration - BetaBoutique</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta name="description" content="BetaBoutique">
...
    <!--  gestion des messages de succès -->
    <%
        ArrayList succes=(ArrayList)request.getAttribute("succes");
        if(succes!=null && succes.size()>0)
        {
            out.println("<div id=\"message_information\"><ul>");
            for(int i=0;i<succes.size();i++)
            {
                out.println("<li>"+(String)succes.get(i)+"</li>");
            }
            out.println("</ul></div>");
        }
    %>
        <!--  gestion des messages d'erreur -->
            <%
        ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
        if(succes!=null && erreurs.size()>0)
        {
            out.println("<div id=\"message_erreur\"><ul>");
            for(int i=0;i<erreurs.size();i++)
            {
                out.println("<li>"+(String)erreurs.get(i)+"</li>");
            }
            out.println("</ul></div>");
        }
    %>
...
```

Dans certains cas, il n'est pas possible d'utiliser la classe statique *BeanUtils* de gestion des JavaBeans pour insérer directement des informations d'un formulaire. En effet, si par exemple dans le formulaire il y avait des informations qui doivent être insérées dans la base de données mais qui ne sont pas dans le JavaBean *Article*, ces informations seraient perdues. Pour résoudre ce problème il faut utiliser plusieurs JavaBeans ou utiliser un mélange de JavaBeans et des informations reçues par la requête avec *request.getParameter(...)*. Rien n'empêche le développeur d'utiliser plusieurs techniques.

Certains développeurs, pour simplifier les Servlets et diminuer les lignes de code utilisent les mêmes fichiers en consultation et modification. Le client consulte le formulaire de modification sans apporter de changement.

Suppression d'un article

Il reste à réaliser l'étape de suppression d'un article. Cette étape est la plus simple, il suffit de lancer une méthode du modèle qui permet la suppression de l'article.

La portion de code à rajouter dans la Servlet est la suivante :

```
...
//supprimer un article
else if(action.equals("supprimer"))
{
    //récupérer le numéro de l'article à supprimer
    String id_article=(String)request.getParameter("id_article");
    if(id_article!=null && !id_article.equals(""))
    {
        //récupérer l'article dans la base de données
avec la classe modele
        int resinstruction=articlemodele.supprimerArticle(id_article);
        //en cas d'erreur avec la base de données, ajouter
un message d'erreur
        if(resinstruction!=1)
        {
            //ajouter un message d'erreur
            erreurs.add("Erreur lors de la suppression de l'article");
        }
        //tout est OK
        else
        {
            //ajouter un message de succes
            succes.add("Suppression de l'article réalisée avec succès");
        }
        //envoyer les listes de messages à la vue et retourner
sur la vue principale
        request.setAttribute("erreurs",erreurs);
        request.setAttribute("succes",succes);
        //retourner sur la page d'affichage des articles en administration
getServletContext().getRequestDispatcher("/admin/
```

```
gestionarticles?action=liste").forward(request, response);
    }
}
...
```

Enfin, le code de la classe modèle est très simple :

```
...
/*************************************************************
 * supprimer un article
*************************************************************/
public int supprimerArticle(String id_article)
{
    int resinstruction=0;
    PreparedStatement requete=null;

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        requete=connection.prepareStatement("DELETE FROM
article WHERE article.id_article=?");
        requete.setString(1,id_article);
        resinstruction=requete.executeUpdate();
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
ArticleModele.java fonction supprimerArticle");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ArticleModele.java fonction supprimerArticle");
        }
    }

    //retourner le résultat de la suppression
    return resinstruction;
}
...
```

Création d'un article

Comme pour l'étape de modification, l'étape de création nécessite la validation des données avant l'insertion dans la base de données.

Dans un cas réel, nous utiliserions plutôt un formulaire d'upload pour le chargement des pochettes des articles à la place d'un champ texte qui contient le chemin de l'image à afficher. Ces images sont chargées par exemple à l'aide d'un client FTP.

La Servlet est légèrement modifié pour afficher le formulaire de création (formulaire HTML).

```
...
//créer un article
else if(action.equals("creer"))
{
    //récupérer la liste des catégories d'articles
    ArrayList listecategoriearticles=(ArrayList)
```

```
articlemodele.getListeCategorieArticle();
    //retourner la liste des catégories d'articles à la vue
    request.setAttribute("listecategorieearticles",
listecategorieearticles);
    //retourner sur la page d'affichage des articles en
administration
getServletContext().getRequestDispatcher("/admin/vues/article/
creerarticle.jsp").forward(request, response);
}
...
```

Nous modifions aussi la vue *listearticles.jsp* afin d'insérer le lien de création.

```
...
<br/><div class="titre"><a href="<%= action %>"><img
src="<%= urlapplication %>/img/articleadmin.png"
border="0" align="absmiddle"/> Liste des
articles</a></div>
<a href="<%= action %>?action=creer">Cr&eacute;er un
nouvel article</a><br/>
...
```

Voici ci-après la vue *creerarticle.jsp* pour l'étape de création d'un article.

```
<%@ page import="java.util.ArrayList" %>
<%@ page import="betaboutique.boiteoutils.Article" %>
<%@ page import="betaboutique.boiteoutils.Categorie" %>
<%
//url de l'application
String urlapplication=(String)getServletContext().getInit
Parameter("urlapplication");
//liste des categories
ArrayList listecategorieearticles=(ArrayList)request.get
Attribute("listecategorieearticles");
//action a appeler
String action=urlapplication+"admin/gestionarticles";
%>
<!--  inclure l'en-tête de la page -->
<%@ include file="../outils/entete.jspf" %>

<br/><div class="titre"><a href="<%= action %>"><img src="<%=
urlapplication %>/img/articleadmin.png" border="0"
align="absmiddle"/> Liste des articles</a></div>

<form action="<%= action %>?action=validercreer" method="POST">
<table border="0" id="tableau" cellpadding="0" cellspacing="0"
width="50%">
<tr><td>Nom</td><td><input type="text" name="nomarticle"
class="input" value=""/></td></tr>
<tr><td valign="top">Description</td><td><textarea
class="textarea" name="descriptionarticle"></textarea>
</td></tr>
<tr><td>Prix</td><td><input type="text" name="prixarticle"
class="input" value=""/> Euros</td></tr>
<tr><td>Date</td><td><input type="text" name="datearticle"
class="input" value=""/></td></tr>
<tr><td>Photo</td><td><input type="text" name="photoarticle"
class="input" value=""/></td></tr>
<tr><td>Vignette</td><td><input type="text"
name="vignettearticle" class="input" value=""/></td></tr>
<tr><td>Etat</td>
<td>
<%
out.println("<select name=\"etatarticle\" class=\"select\">");
out.println("<option value=\"1\">Actif</option>");
out.println("<option value=\"0\">Inactif</option>");
out.println("</select>");
%>
</td></tr>
<tr><td>Cat&eacute;gorie</td>
```

```
<td>
<%
//liste des categories
out.println("<select name=\"id_categorie\" class=\"select\">");
for(int i=0;i<listecategoriearticles.size();i++)
{
    //récupérer l'objet dans la liste
    Categorie categorie=(Categorie)listecategoriearticles.get(i);
    out.println("<option value=\""+categorie.getId_categorie()
"\">"+categorie.getNomcategorie()+"</option>");
}
out.println("</select>");
%>
</td></tr>
<tr><td align="center" colspan="2"><input type="submit"
name="creer" value="Cr&eacute;er" class="bouton"/></td></tr>
</table><br/>
</form>
<%@ include file="../outils/piedpage.jspf" %>
```

Le code de la Servlet, déclenché par l'action *validercreer* est très proche du code de modification d'un article.

```
...
//valider la création d'un article
else if(action.equals("validercreer"))
{
        //récupérer les données dans la requête et les
insérer dans le JavaBean article
        Article article=new Article();
        //création d'une collection des paramètres reçus par la requête
        HashMap map=new HashMap();
        Enumeration names=request.getParameterNames();
        while (names.hasMoreElements())
        {
              // chaque paramètre de la requête dans une collection
              String name=(String)names.nextElement();
              map.put(name,request.getParameterValues(name));
        }
        try
        {
            // insérer toutes nos données dans le Bean
article en une seule étape
            BeanUtils.populate(article, map);
            //modifier l'article dans la base de données
                int resinstruction=articlemodele.creerArticle(article);
                //en cas d'erreur avec la base de données,
ajouter un message d'erreur
                if(resinstruction!=1)
                {
                    //ajouter un message d'erreur
                    erreurs.add("Erreur lors de la création de l'article");
                }
                //tout est OK
                else
                {
                    //ajouter un message de succes
                    succes.add("Création de l'article réalisée avec succès");
                }
                //envoyer les listes de messages à la vue et
retourner sur la vue principale
                request.setAttribute("erreurs",erreurs);
                request.setAttribute("succes",succes);
                //retourner sur la page d'affichage des
articles en administration
                getServletContext().getRequestDispatcher
("/admin/gestionarticles?action=liste").forward(request,
response);
        }
        catch(Exception e)
        {
```

```
          System.out.println("Erreur lors de la lecture
des informations");
        }
}
...
```

Enfin, le code de la méthode *creerArticle()* de la classe modèle utilise une requête pré-compilée.

```
/*********************************************************
 * créer un article
*********************************************************/
public int creerArticle(Article article)
{
    int resinstruction=0;
    PreparedStatement requete=null;
    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        //insérer l'article
        requete=connection.prepareStatement("INSERT INTO article
(nomarticle,descriptionarticle,prixarticle,datearticle,photoarticle,
vignettearticle,etatarticle,id_categorie) VALUES (?,?,?,?,?,?,?,?)");
        requete.setString(1,(String)article.getNomarticle());
        requete.setString(2,(String)article.getDescriptionarticle());
        requete.setString(3,(String)article.getPrixarticle());
        requete.setString(4,(String)super.miseEnFormeDatevers
Base(article.getDatearticle()));
        requete.setString(5,(String)article.getPhotoarticle());
        requete.setString(6,(String)article.getVignettearticle());
        requete.setString(7,(String)article.getEtatarticle());
        requete.setString(8,(String)article.getId_categorie());
        resinstruction=requete.executeUpdate();
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
ArticleModele.java fonction creerArticle");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ArticleModele.java fonction creerArticle");
        }
    }

    //résultat de la création
    return resinstruction;
}
```

Modèle et JavaBean

1. Présentation

Nous avons géré jusqu'à maintenant les accès aux données avec la méthode *getString(...)* du *ResultSet*. Comme pour la gestion des formulaires ou les fonctionnalités XML, il existe une librairie qui permet de manipuler les *ResultSet* avec des JavaBeans.

Cette librairie nommée *DbUtils* est développée par le consortium Apache/Tomcat. Cette API est relativement légère et simple d'utilisation, elle permet de simplifier le travail des développeurs lors du codage des classes modèles.

2. Utilisation

Il faut d'abord télécharger la librairie à cette adresse : http://commons.apache.org/dbutils/. Son installation est identique aux autres librairies du projet. Il faut charger la librairie dans le répertoire */WEB-INF/lib* de l'application et insérer son chemin dans le classpath (***Project - Properties - Java Build Path***). Cette API nécessite un JDK version 1.2 au moins et JDBC 2.0 ou supérieur.

Nous allons utiliser les fonctionnalités de cette librairie dans la méthode *ListeArticleAdmin* de la classe modèle *ArticleModele*. Son utilisation est simple, il suffit de préciser la liste d'objets (ou l'objet) à renseigner à partir du *ResultSet*. La classe *dbutils* va ensuite se charger de gérer les associations entre les noms de propriétés de l'objet concerné (*article*) et les attributs de la requête.

Le code présenté ci-dessous est donc extrêmement simplifié.

```
        ...
        //exécuter la requête
        if(rs!=null)
        {
            //stocker tous les articles dans une liste
            while(rs.next())
            {
                //un enregistrement de plus
                this.compteurenregistrement++;
                //créer un objet article
                Article article=new Article();
                //renseigner l'objet article avec ses accesseurs
                if(rs.getString("id_article")==null)
article.setId_article("0");
                else article.setId_article(rs.getString
("id_article"));
                if(rs.getString("nomarticle")==null)
article.setNomarticle("");
                else article.setNomarticle(rs.getString
("nomarticle"));
                if(rs.getString("descriptionarticle")==null)
article.setDescriptionarticle("");
                else article.setDescriptionarticle(rs.getString
("descriptionarticle"));
                if(rs.getString("prixarticle")==null)
article.setPrixarticle("0");
                else article.setPrixarticle(rs.getString
("prixarticle"));
                if(rs.getString("datearticle")==null)
article.setDatearticle("0");
                else article.setDatearticle
(super.miseEnFormeDate(rs.getString("datearticle")));
                if(rs.getString("photoarticle")==null)
article.setPhotoarticle("");
                else article.setPhotoarticle(rs.getString
("photoarticle"));
                if(rs.getString("vignettearticle")==null)
article.setVignettearticle("");
                else article.setVignettearticle(rs.getString
("vignettearticle"));
```

```
                if(rs.getString("etatarticle")==null)
article.setEtatarticle("0");
                else article.setEtatarticle(rs.getString
("etatarticle"));
                if(rs.getString("id_categorie")==null)
article.setNomcategoriearticle("");
                else article.setNomcategoriearticle
(super.getNomCategorieArticle(rs.getString("id_categorie")));

                //stocker l'objet article dans la liste des articles
                listearticle.add((Article)article);
            }
        }
...
```

Le nouveau code est alors ci-après.

```
...
BeanProcessor bp=new BeanProcessor();
listearticle = (ArrayList)bp.toBeanList(rs, Article.class);
//total des enregistrements
this.compteurenregistrement=listearticle.size();
...
```

Nous utilisons une instance de la classe *BeanProcessor* et sa fonction *toBeanList()* qui permet de transformer le *ResultSet (rs)* en une collection/liste (*listearticle*) d'objets de la classe *Article (Article.class)*.

L'utilisation de JavaBean permet de simplifier très clairement le code. Les tests lourds, sources d'erreurs sont ainsi évités. L'écriture du code JDBC n'est pas très difficile mais cette tâche est très répétitive. L'API *DbUtils* est simple à utiliser, elle utilise le standard JavaBean mais nécessite une rigueur de programmation. Dans notre exemple de gestion des articles, nous remarquons que le nom de catégorie n'est plus affiché. En effet, dans la requête, le nom de la catégorie de l'article n'est pas retourné mais son identifiant (*id_categorie*). À aucun moment la requête SQL ne retourne un attribut nommé *nomcategorie* qui permet de renseigner le JavaBean article par l'intermédiaire de son *setter()*.

Pour remédier à cela, il faudrait réaliser une jointure dans la requête SQL entre la table *article* et la table *categorie*. Ensuite, il faudrait que les noms des accesseurs soient strictement les mêmes que les noms des attributs des tables. Enfin, avec cette technique, il ne doit pas y avoir de conflits de noms entre attributs. En effet, cela nécessite de nommer chaque attribut de la table de manière unique (ex : *datearticle*, *datecategoriearticle*, *nomcategoriearticle*...).

Si par exemple, il existe un attribut *date* dans la table *article* et un attribut *date* dans la table *categorie*, lors de l'exécution de la requête et du renseignement du *ResultSet*, ce dernier ne saura pas comment gérer l'attribut qu'il trouvera en double.

Il existe également un autre problème à l'utilisation des JavaBeans et des *ResultSet* c'est la modification des données lors des accès. En effet, nous insérons des données brutes, c'est-à-dire telles qu'elles sont présentes dans la base de données. Dans notre gestion des articles, nous voyons que la date n'est pas dans le format correct. Précédemment, nous avons réalisé une modification de la syntaxe dans la classe modèle lors de la lecture du *ResultSet*.

```
...
if(rs.getString("datearticle")==null)article.setDatearticle("0");
else article.setDatearticle(super.miseEnFormeDate(rs.getString
("datearticle")));
...
```

Avec les JavaBeans, cette modification serait donc opérée du côté de la vue au moment de l'affichage (ce qui d'ailleurs n'est pas une mauvaise technique, il est ainsi possible de gérer différentes formes d'affichage suivant la langue par exemple : date au format anglais ou français).

3. Conclusion

L'utilisation de JavaBean et de bibliothèques adaptées du point de vue du modèle est parfois très utile en terme de performances et de ligne de code, mais pour des cas évolués, elle peut par contre complexifier le code et ne pas être adaptée.

Dans notre exemple de gestion des articles, nous aurions pu utiliser la solution 100% JavaBean en modifiant chaque nom d'attribut des tables, en utilisant une jointure sur la table *categorie* afin de retourner tous les attributs nécessaires à l'affichage et en réalisant quelques modifications de syntaxe à l'affichage.

Mais le développeur ne gagne pas forcément de temps et complexifie l'ensemble. Par contre, pour des objets simples (ex : mots-clés, notes, types, nom de catégories, familles ou formats) à manipuler, cette technique est très utile. Il s'agira donc de bien analyser le diagramme de classes du projet et le modèle conceptuel des données afin d'évaluer quelle technique est la plus adaptée et de mélanger les deux techniques suivant la pertinence (librairies ou code pur Java).

Les transactions

1. Présentation

Jusqu'à présent toutes les commandes SQL envoyées à la base de données étaient exécutées immédiatement et les modifications étaient effectuées de façon définitive (insertion, création, suppression...).

Supposons désormais que nous insérions un article et une note de 5/10 au moment de la création. Si au moment de l'insertion de la note, une panne d'alimentation survient, alors il y aura un enregistrement incorrect dans la base de données (un article sans note).

Dans cet exemple, cela n'aurait pas beaucoup de conséquences mais l'article aurait une note de 0/10 à l'affichage. En revanche, pour la gestion des commandes et des ventes, il ne serait pas envisageable d'avoir une facture vide, de l'argent encaissé sans données associées, des articles en commande sans facture associée... Dans tous ces cas, la base de données se trouverait dans un état invalide.

Ces situations sont gérées à l'aide de transactions. Le rôle d'une transaction est de passer la base de données d'un état valide à un autre. Lorsque les opérations en cours sont validées, nous sommes certains que toutes les modifications ont été effectuées. Si une seule des opérations échoue, toutes les opérations sont annulées et aucune modification n'est réalisée.

2. Utilisation

Nous allons aborder la gestion des transactions avec le service de notation des articles. En administration, lors de la création d'un article, nous allons lui associer une note par défaut de 5/10. Nous commençons par réaliser/modifier la fonction *creerArticle()* sans la gestion des transactions mais avec la notation associée par défaut.

```
/*********************************************************
 * créer un article
*********************************************************/
public int creerArticle(Article article)
{
    int resinstruction=0;
    int id_article=0;
    PreparedStatement requete=null;
    ResultSet clef=null;

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        //insérer la profession
        requete=connection.prepareStatement("INSERT INTO article
(nomarticle,descriptionarticle,prixarticle,datearticle,photoarticle,
vignettearticle,etatarticle,id_categorie) VALUES (?,?,?,?,?,?,?,?)");
        requete.setString(1,(String)article.getNomarticle());
        requete.setString(2,(String)article.getDescriptionarticle());
        requete.setString(3,(String)article.getPrixarticle());
        requete.setString(4,(String)super.miseEnForme
DateversBase(article.getDatearticle()));
        requete.setString(5,(String)article.getPhotoarticle());
        requete.setString(6,(String)article.getVignettearticle());
        requete.setString(7,(String)article.getEtatarticle());
        requete.setString(8,(String)article.getId_categorie());
        resinstruction=requete.executeUpdate();

        //récupérer le numéro de la clé générée pour cet article
        clef=requete.getGeneratedKeys();
        if(clef.next())
        {
            //id_article inséré lors de la création
            id_article=Integer.parseInt(clef.getObject(1).toString());
        }
        System.out.println("nous venons d'insérer l'article : "+id_article);
    }
```

```
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
ArticleModele.java fonction creerArticle");
    }
    finally
{
        try
        {
            //fermer la connexion
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ArticleModele.java fonction creerArticle");
        }
    }
    //résultat de la création
    return resinstruction;
}
```

Nous avons ajouté une partie importante du code qui est très souvent utilisée lors des développements et qui permet de retourner le numéro de l'enregistrement qui vient d'être créé. La méthode *getGeneratedKeys()* de l'objet *PreparedStatement* permet de réaliser cette opération. Nous insérons donc notre nouvel article, nous récupérons le numéro de l'enregistrement et nous insérons ensuite sa note par défaut.

Le code fonctionnel de la classe *ArticleModele* est présenté ci-dessous :

```
...
/**********************************************************
 * créer la note par défaut pour l'article
**********************************************************/
public int ajouterNoteDefautArticle(String id_article,String id_client)
{
    int resinstruction=0;
    PreparedStatement requetea=null;
    PreparedStatement requeteb=null;
    try
    {
        //ouvrir une connexion
        connection1=ds.getConnection();
        //vérifier si la note n'est pas déjà insérée
        requetea=connection1.prepareStatement("SELECT note FROM notearticle
WHERE notearticle.id_article=? AND notearticle.id_client=?");
        requetea.setString(1,(String)id_article);
        requetea.setString(2,(String)id_client);
        rs=requetea.executeQuery();

        //exécuter la requête
        if(rs!=null)
        {
            //on n'a pas de note, l'insérer
            if(!rs.next())
            {
                //insérer la note de l'article
                requeteb=connection1.prepareStatement("INSERT INTO
notearticle(id_article,id_client,note) VALUES (?,?,?)");
                requeteb.setString(1,(String)id_article);
                requeteb.setString(2,(String)id_client);
                requeteb.setString(3,(String)"5");
                resinstruction=requeteb.executeUpdate();
            }
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe ArticleModele.java fonction
```

```
ajouterNoteDefautArticle");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(requetea!=null)OutilsBaseDeDonnees.fermerConnexion(requetea);
            if(requeteb!=null)OutilsBaseDeDonnees.fermerConnexion(requeteb);
            if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
            if(connection1!=null)OutilsBaseDeDonnees.fermerConnexion
(connection1);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe ArticleModele.java
fonction ajouterNoteDefautArticle");
        }
    }
    //résultat de la création de la note
    return resinstruction;
}
...
```

Le code permet ainsi d'insérer l'article dans la base de données et de lui associer une note par défaut de 5/10.

3. Gestion des transactions

Dans certains systèmes SQL, il est nécessaire d'indiquer explicitement à la base de données qu'une transaction commence avant d'exécuter des commandes. Avec JDBC, cela n'est pas nécessaire, celui-ci démarre une transaction automatiquement. La transaction peut être terminée automatiquement ou manuellement par le programmeur.

Le choix entre ces deux options est déterminé par la configuration de la propriété *autocommit* de la connexion. Avec JDBC, la valeur par défaut est *true*, les transactions sont donc validées automatiquement par la connexion. Lorsque *autocommit* vaut *false*, il est de la responsabilité du programmeur de terminer explicitement les transactions. La classe *Connection* possède plusieurs méthodes permettant de manipuler les transactions :

- *setAutoCommit(boolean)* : permet la validation automatique ou non des transactions.
- *commit()* : valide la transaction en cours.
- *close()* : permet de fermer la connexion (méthode étudiée jusqu'ici).
- *rollback()* : permet d'annuler la transaction en réalisant un retour arrière à l'état initial.

Lorsque le code obtient une connexion d'un gestionnaire de pilotes, d'une source de données ou d'un pool de connexions, JDBC exige que celle-ci soit en mode *autocommit*. De ce fait, dans la plupart des applications, la première méthode appelée après l'obtention d'une connexion est *setAutoCommit(boolean)*. Après l'exécution de ce code, la gestion des transactions appartient au client. Le développeur peut créer des transactions contenant toutes les requêtes. Les commandes SQL sont envoyées à la base de données et exécutées, mais elles ne sont pas validées tant que la transaction n'est pas terminée.

La validation intervient alors lorsque la méthode *commit()* de l'objet *Connection* est appelée et que le paramètre *autocommit* est positionné à false. La transaction peut également être annulée si le programme appelle la méthode *rollback()*.

Nous allons reprendre notre exemple précédent de création des articles et de l'insertion de la note avec les transactions.

Pour le moment, nous modifions juste notre connexion JDBC afin d'indiquer que nos transactions doivent être validées manuellement par la méthode *commit()*. Dans notre exemple, lors d'une insertion, celle-ci ne doit pas être réalisée car il n'y a pas de méthode *commit()* déclenchée.

Or si nous déclenchons une création d'article, nous remarquons que l'article est bien créé dans la base de données. La gestion des transactions n'est donc pas fonctionnelle. JDBC ne déclenche pas d'erreur mais ne gère pas le mode transactionnel correctement. En fait, le problème ne vient pas de JDBC mais de la base de données MySQL. En effet, par défaut MySQL utilise des tables avec le type MyISAM. Le type MyISAM ne permet pas de gérer les transactions. Cependant, InnoDB fournit à MySQL un gestionnaire transactionnel avec un verrouillage de lignes. InnoDB a été

conçu pour maximiser les performances lors du traitement de grandes quantités de données.

Afin d'activer la gestion des transactions, il est donc nécessaire de convertir nos tables en type InnoDB. Par contre, les tables InnoDB ne supportent pas les index FULLTEXT, il est donc nécessaire de les supprimer au préalable.

Nous allons changer le type de nos tables *article* et *notearticle* afin de passer de MyISAM à InnoDB avec les commandes SQL suivantes :

```
ALTER TABLE `article` TYPE = INNODB
ALTER TABLE `notearticle` TYPE = INNODB
```

Nous pouvons vérifier les types des tables en affichant la structure :

	Table	Action	Enregistrements	Type	Interclassement
☐	administrateur		1	MyISAM	utf8_general_ci
☐	article		40	InnoDB	utf8_general_ci
☐	categorie		7	MyISAM	utf8_general_ci
☐	client		3	MyISAM	utf8_general_ci
☐	commande		0	MyISAM	utf8_general_ci
☐	commandearticle		0	MyISAM	utf8_general_ci
☐	notearticle		14	InnoDB	utf8_general_ci
☐	roles		1	MyISAM	utf8_general_ci

Nos deux tables sont maintenant de type InnoDB, nous allons vérifier la gestion des transactions en déclenchant une nouvelle création d'article. Désormais, si nous déclenchons une création avec le code précédent, celle-ci est lancée correctement (requêtes SQL exécutées) mais les requêtes SQL ne sont pas validées, l'article n'est donc pas véritablement inséré dans le SGBD.

Nous allons maintenant valider manuellement les transactions afin de pouvoir réaliser un retour arrière en cas de problème.

```
...
/*********************************************************
 * créer un article
*********************************************************/
public int creerArticle(Article article)
{
    int resinstruction=0;
    String id_article=null;
    PreparedStatement requete=null;
    ResultSet clef=null;

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        //passer en mode manuel de gestion des transactions (autocommit
désactivé)
        connection.setAutoCommit(false);

        //insérer la profession
        requete=connection.prepareStatement("INSERT INTO
article(nomarticle,descriptionarticle,prixarticle,datearticle,photoarticle,
vignettearticle,etatarticle,id_categorie) VALUES (?,?,?,?,?,?,?,?)");
        requete.setString(1,(String)article.getNomarticle());
        requete.setString(2,(String)article.getDescriptionarticle());
        requete.setString(3,(String)article.getPrixarticle());
        requete.setString(4,(String)super.miseEnFormeDateversBase
(article.getDatearticle()));
        requete.setString(5,(String)article.getPhotoarticle());
        requete.setString(6,(String)article.getVignettearticle());
        requete.setString(7,(String)article.getEtatarticle());
        requete.setString(8,(String)article.getId_categorie());
        resinstruction=requete.executeUpdate();

    //récupérer le numéro de la clé générée pour cet article
```

```
    clef=requete.getGeneratedKeys();
        if(clef.next())
        {
            //id_article inséré lors de la création
            id_article=clef.getObject(1).toString();
            //insérer la note associée (id_article et client=0 soit
administration)
            if(id_article!=null)
            {
                resinstruction=this.ajouterNoteDefautArticle(id_article,"0");
            }
        }
        //exécuter la requete de création de l'article si la tout est ok
        if(resinstruction==1)
        {
            connection.commit();
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe ArticleModele.java fonction
creerArticle");
        try
        {
            connection.rollback();
        }
        catch(Exception c)
        {
            System.out.println("Erreur de rollback de la transaction");
        }
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe ArticleModele.java
fonction creerArticle");
        }
    }

    //résultat de la création
    return resinstruction;
}
```

Si la requête est correctement exécutée, un numéro de clé d'article est alors récupéré et la note par défaut est ajoutée. Si une erreur a lieu lors de la création de la note, l'instruction *commit()* qui valide réellement la requête ne sera pas exécutée, il n'y aura donc pas d'article sans note associée dans la base de données.

L'instruction *connection.rollback()* permet de revenir en arrière en cas de déclenchement d'exception. Pour tester ce principe, nous allons volontairement modifier la requête d'ajout de la note afin d'insérer une erreur. Nous allons simuler une erreur SQL avec un accès à la table noteartícleERREUR qui n'existe pas.

```
...
/*********************************************************
 * créer la note par défaut pour l'article
*********************************************************/
public int ajouterNoteDefautArticle(String id_article,String id_client)
{
...
    //vérifier si la note n'est pas déjà insérée
    requetea=connection1.prepareStatement("SELECT note FROM noteartícleERREUR
WHERE notearticle.id_article=? AND notearticle.id_client=?");
```

```
...
}
```

Désormais, si nous insérons un nouvel article avec le formulaire de création : nous réalisons l'insertion de l'article en premier, ensuite nous ajoutons sa note par défaut. L'ajout de la note va déclencher une exception car la table *notearticleERREUR* n'existe pas. La méthode *rollback()* de l'ajout va donc être déclenchée. Dans la méthode de création de l'article le *commit() (resinstruction=0)* n'est pas validé, l'article seul sans sa note ne sera donc pas inséré.

Maintenant si nous vérifions dans la base de données, l'article n'est pas inséré malgré le fait que son insertion était correcte. Nous avons donc bien géré dans la totalité la transaction de la création d'un article.

Dans un service professionnel, les transactions seront utilisées de préférence. Par contre, il est important de tester chaque cas (création correcte, incorrecte...) afin de vérifier l'efficacité de l'ensemble et l'ordre des déclenchements. Lors du codage de la gestion des transactions, c'est essentiellement cette étape qui est compliquée.

La technique de gestion des transactions avec Java JDBC est très perfectionnée et permet de contrôler des points de sauvegarde (annulation partielle), de combiner les transactions et procédures stockées...

4. Optimisations

Nous allons apporter quelques optimisations concernant la gestion des transactions. En effet, la méthode *rollback()* nécessite la gestion d'un bloc *try catch*. Nous allons donc utiliser une méthode spécifique dans notre classe statique *OutilsBaseDeDonnees*. Nous pouvons réaliser la même opérations pour la méthode *commit()*.

```
...
/*************************************************************
 * commit Valide la transaction en cours
*************************************************************/
public static void commit(Connection con)
{
    if(con!=null)
    {
        try
        {
        con.commit();
        }
        catch(Exception e)
        {
        System.out.println("Erreur lors du commit d'une
connexion dans commit(Connection)");
        }
    }
}

/*************************************************************
 * rollback Annule la transaction en cours
*************************************************************/
public static void rollback(Connection con)
{
    if(con!=null)
    {
        try
        {
        con.rollback();
```

```
        }
        catch(Exception e)
        {
    System.out.println("Erreur lors du rollback d'une
connexion dans rollback(Connection)");
        }
    }
}
...
```

Un autre élément important et lourd à gérer lors de l'utilisation des transactions est la gestion du mode commit. Nous retrouvons après chaque récupération de la connexion la méthode suivante :

```
//passer en mode manuel de gestion des transactions (autocommit
désactivé)
connection.setAutoCommit(false);
```

La gestion du mode transactionnel peut être gérée directement au niveau du pool de connexion dans le fichier de déclaration de l'application (*/Tomcat 6.0/conf/Catalina/localhost/betaboutiquemvc.xml*). Il existe pour cela un attribut nommé *defaultAutoCommit* qui permet de préciser le niveau de gestion de façon globale. http://commons.apache.org/dbcp/configuration.html.

Par défaut, *defaultAutoCommit* est à *true* et permet de valider automatiquement les requêtes. Nous allons donc le passer à *false* afin de gérer manuellement les exécutions de requêtes.

```
<Context path="/betaboutiquemvc" reloadable="true"
docBase="E:\PROJETWEB\betaboutiquemvc"
workDir="E:\PROJETWEB\betaboutiquemvc\work" >
<Resource name="jdbc_betaboutiquemysql" auth="Container"
type="javax.sql.DataSource" username="root" password=""
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/betaboutique"
maxActive="20" maxIdle="10" validationQuery="SELECT 1"
defaultAutoCommit="false"/>
</Context>
```

Les transactions seront donc gérées manuellement (sans être obligé de préciser le mode *autocommit* à chaque fois) et les requêtes non transactionnelles seront exécutées sans problème mais ne seront pas déclenchées. Il est donc primordial de choisir le mode le plus important au niveau du pool. Si notre application utilise plus de transactions nous placerons alors *defaultAutoCommit* à *false* et nous utiliserons la méthode *setAutoCommit()* au niveau des requêtes non transactionnelles.

Pour notre projet, nous utiliserons plus de requêtes non transactionnelles. Le paramètre *defaultAutoCommit* sera donc placé à *true* (valeur par défaut) et nous utiliserons la méthode *setAutoCommit(false)* au niveau de chaque méthode transactionnelle.

Il nous reste une dernière méthode à modifier, qui concerne la suppression d'un article. Lors de la destruction d'un enregistrement, il ne faut pas oublier de supprimer les notes associées et donc de rajouter une requête de suppression.

```
/*********************************************************
 * supprimer un article
*********************************************************/
public int supprimerArticle(String id_article)
{
    int resinstruction=0;
    PreparedStatement requete=null;
    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        requete=connection.prepareStatement("DELETE FROM
article WHERE article.id_article=?");
        requete.setString(1,id_article);
        resinstruction=requete.executeUpdate();
        //supprimer les notes associées
        this.supprimerNoteArticle(id_article);
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
```

```
ArticleModele.java fonction supprimerArticle");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ArticleModele.java fonction supprimerArticle");
        }
    }
    //retourner le résultat de la suppression
    return resinstruction;
}


/*********************************************************
 * supprimer les notes de l'article concerné
*********************************************************/
public void supprimerNoteArticle(String id_article)
{
    PreparedStatement requete=null;
    //Supprimer les notes associées à un article
    try
    {
        //ouvrir une connexion
        connection1=ds.getConnection();
        requete=connection1.prepareStatement("DELETE FROM
notearticle WHERE id_article=?");
        requete.setString(1,(String)id_article);
        requete.executeUpdate();
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
ArticleModele.java fonction supprimerNoteArticle");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection1!=null)OutilsBaseDeDonnees.fermerConnexion
(connection1);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ArticleModele.java fonction supprimerNoteArticle");
        }
    }
}
```

Multilingue et JDBC

1. Présentation

Jusqu'à présent nous avons travaillé sur des données en français mais il est parfois nécessaire de gérer des données dans d'autres langues. S'il faut gérer un site en français et en anglais, le problème ne se pose pas car le jeu de caractères français/anglais est identique.

Mais que se passe-t-il quand on doit gérer par exemple la langue arabe avec des chaînes complexes :

اأنا قادر على أكل الزجاج و هذا لا يؤلمن

Par défaut, les sites Web (pages HTML, pages JSP, tables MySQL...) sont réalisés avec la norme ISO-8859-1 qui est souvent appelée Latin-1. Ce codage simple mais limité en nombre de caractères et ne permet pas de représenter les caractères d'autres langues. La vue ci-dessous fait référence au codage ISO-8859-1.

Jeu de caractères ASCII

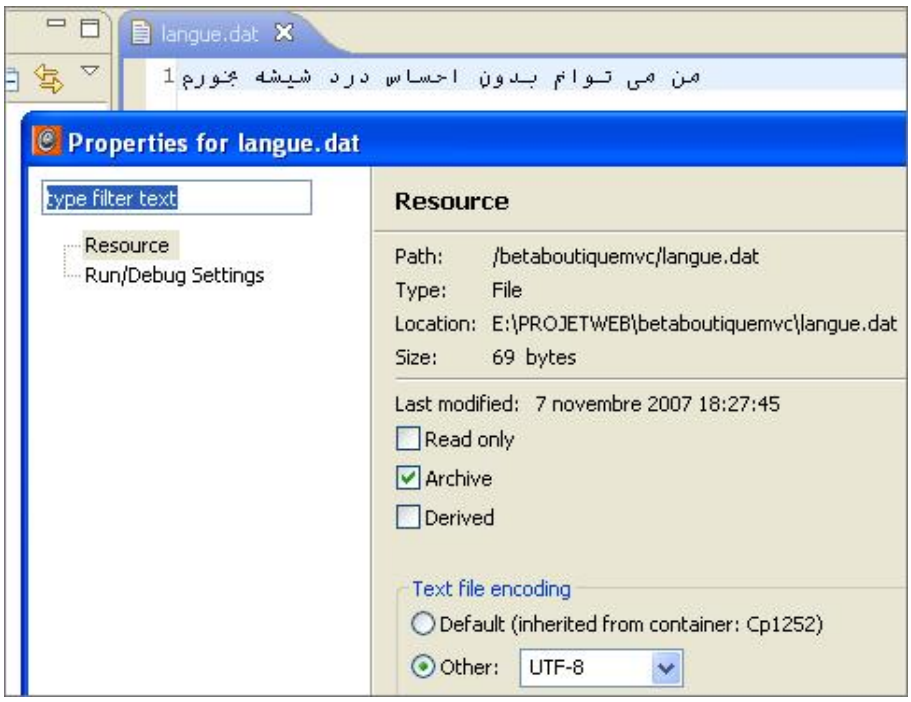
+	0	1	2	3	4	5	6	7	8	9
30				!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Le codage UTF-8 est défini pour les caractères Unicode. Chaque caractère est codé sur une suite de un à quatre octets. UTF-8 est standardisé dans la RFC 3629. Ce codage est universel et permet de représenter des milliers de caractères Unicode. Ce codage est également très rapide et nécessite moins d'octets que l'UTF-16 .

2. Mise en place

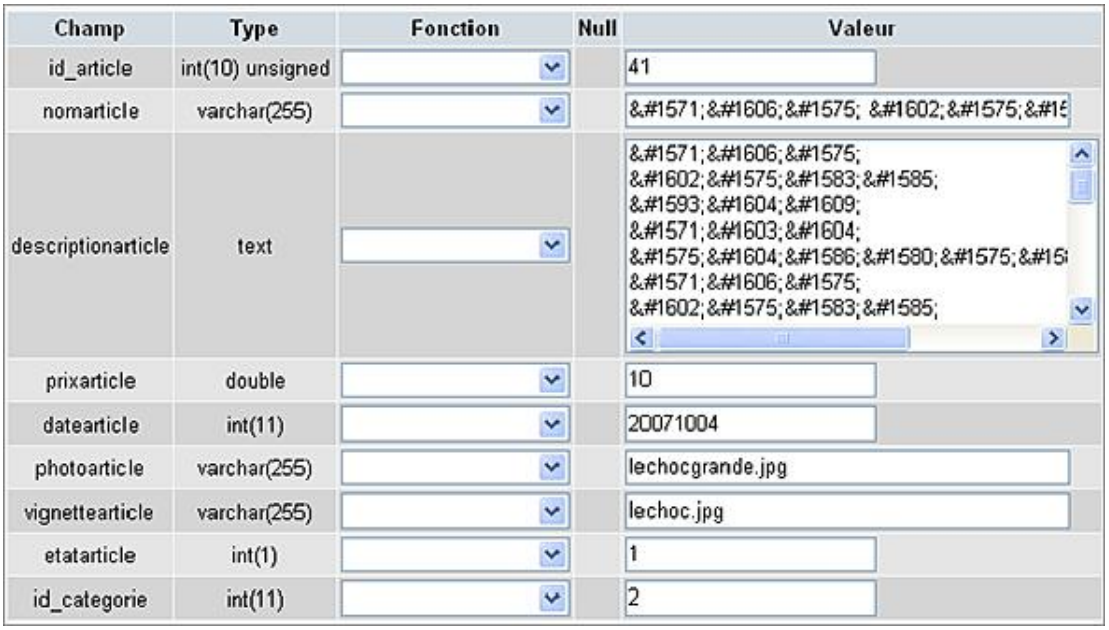
Si nous reprennons notre formulaire de création d'un nouvel article en administration, nous pouvons vérifier l'utilisation de l'encodage courant *Latin-1 (ISO-8859-1)* avec le navigateur (**Affichage - Encodage des caractères**).

Maintenant nous allons essayer d'insérer un nouvel article en Arabe (donc avec les caractères UTF-8). Pour cela, nous utilisons un fichier avec un texte d'exemple encodé en UTF-8. Pour le projet, nous utilisons un fichier nommé *langue.dat* à la racine du projet. Pour changer l'encodage d'un fichier avec Eclipse il faut utiliser le menu **File - Properties - Text file encoding**.



Nous utilisons un formulaire HTML en Latin-1 et une base de données en *Latin1_swedish_ci*. Nous faisons un copié/collé du texte arabe dans le formulaire de saisie et nous validons la création.

Si nous vérifions notre saisie dans la table MySQL, nous remarquons que les données ne sont pas interprétées correctement, les caractères ont été transformés en entités numériques. Donc, lors de l'affichage il y a des problèmes de mise en forme, les recherches ne fonctionnent plus, le stockage n'est pas au bon format...



Champ	Type	Fonction	Null	Valeur
id_article	int(10) unsigned			41
nomarticle	varchar(255)			أنا قا&#15
descriptionarticle	text			أنا قادر على أكل الزجا&#15 أنا قادر
prixarticle	double			10
datearticle	int(11)			20071004
photoarticle	varchar(255)			lechocgrande.jpg
vignettearticle	varchar(255)			lechoc.jpg
etatarticle	int(1)			1
id_categorie	int(11)			2

Pour résoudre ce problème, il faut donc utiliser le format UTF-8 de A à Z lors des développements. Cette programmation est stricte et ne tolère aucun oubli. Voici les précautions à prendre :

- Utiliser un éditeur de développement 100% UTF-8 (Eclipse est compatible UTF-8).
- Encoder les fichiers de développement en UTF-8.
- Mettre des en-têtes UTF-8 dans nos pages Web.
- Utiliser une base de données en mode UTF-8.

Pour l'instant nous avons bien respecté l'encodage de nos fichiers et nous utilisons un éditeur 100% UTF-8. Par contre, nos pages ne possèdent pas d'en-têtes UTF-8. Nous allons modifier notre page JSP *entete.jspf*. En effet, nous utilisons pour le moment l'encodage par défaut Latin-1 :

```
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
```

Nous modifions le charset en utilisant UTF-8.

```
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
```

Ensuite, nous insérons en tout début du fichier *entete.jspf* un en-tête UTF-8 afin de forcer l'encodage.

```
<%@ page contentType="text/html;charset=utf-8" %>
```

Nous venons de préciser à notre navigateur que le site est en UTF-8. Nous pouvons vérifier cette opération avec le navigateur Firefox en utilisant le menu **Affichage - Encodage des caractères**, il faut retrouver Unicode (UTF-8) coché. Le site est désormais en UTF-8 du point de vue de l'affichage et de la saisie des données (formulaire). Par contre, cet encodage nécessite l'utilisation par précaution, des entités UTF-8 dans les pages à la place des caractères spéciaux.

Il ne reste plus qu'à utiliser une base de données en mode UTF-8. Pour cela, nous allons convertir notre base de données *betaboutique* et ses tables en UTF-8. Nous utilisons le format *utf8_general_ci* afin de gérer la casse des caractères et le maximum de symboles.

Nous modifions l'encodage de la base de données avec la commande suivante :

```
ALTER DATABASE `betaboutique` DEFAULT CHARACTER SET utf8
COLLATE utf8_general_ci
```

Nous modifions ensuite l'interclassement de chaque table de la base de données afin de mettre les champs de la table en UTF-8.

```
ALTER DATABASE `betaboutique` DEFAULT CHARACTER SET utf8
COLLATE utf8_general_ci
ALTER TABLE `article` DEFAULT CHARACTER SET utf8 COLLATE
utf8_general_ci
ALTER TABLE `categorie` DEFAULT CHARACTER SET utf8 COLLATE
utf8_general_ci
ALTER TABLE `client` DEFAULT CHARACTER SET utf8 COLLATE
utf8_general_ci
ALTER TABLE `commande` DEFAULT CHARACTER SET utf8 COLLATE
utf8_general_ci
ALTER TABLE `commandearticle` DEFAULT CHARACTER SET utf8
COLLATE utf8_general_ci
ALTER TABLE `notearticle` DEFAULT CHARACTER SET utf8
COLLATE utf8_general_ci
```

Notre base de données est maintenant compatible UTF-8.

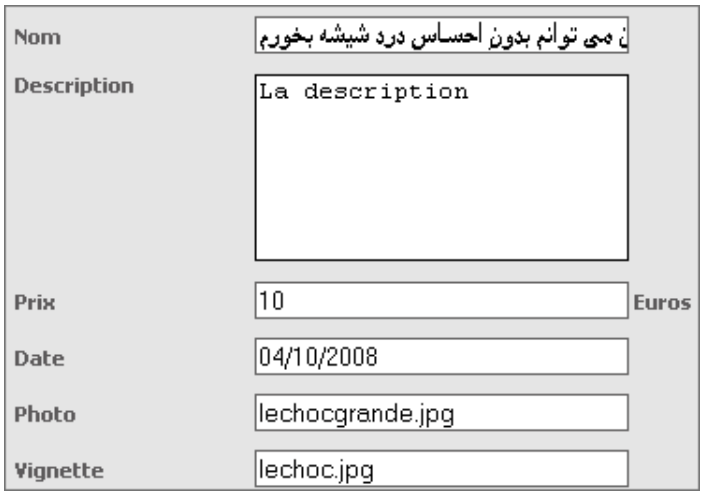
	Table	Action	Enregistrements	Type	Interclassement	Taille	Perte
☐	administrateur		1	MyISAM	utf8_general_ci	3,0 Ko	-
☐	article		41	InnoDB	utf8_general_ci	64,0 Ko	-
☐	categorie		7	MyISAM	utf8_general_ci	2,3 Ko	80 Octets
☐	client		3	MyISAM	utf8_general_ci	2,4 Ko	-
☐	commande		0	MyISAM	utf8_general_ci	1,0 Ko	-
☐	commandearticle		0	MyISAM	utf8_general_ci	1,0 Ko	-
☐	notearticle		15	InnoDB	utf8_general_ci	16,0 Ko	-
☐	roles		1	MyISAM	utf8_general_ci	3,0 Ko	20 Octets

Il reste enfin à passer les types des champs en UTF-8. Si nous prenons la table *article* par exemple, celle-ci est bien en UTF-8 mais ses champs sont encore en Latin-1, nous utilisons donc *phpmyadmin* pour changer l'interclassement des champs.

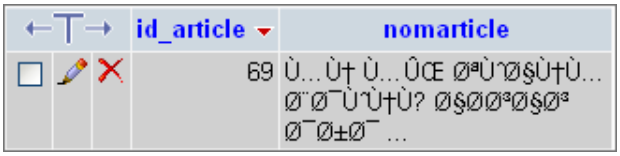
```
ALTER TABLE `categorie` CHANGE `nomcategorie` `nomcategorie`
VARCHAR( 255 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `article` CHANGE `nomarticle` `nomarticle`
VARCHAR( 255 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `article` CHANGE `descriptionarticle`
`descriptionarticle` TEXT CHARACTER SET utf8 COLLATE
utf8_general_ci NOT NULL
ALTER TABLE `article` CHANGE `photoarticle` `photoarticle`
VARCHAR( 255 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `article` CHANGE `vignettearticle`
`vignettearticle` VARCHAR( 255 ) CHARACTER SET utf8
COLLATE utf8_general_ci NOT NULL

ALTER TABLE `client` CHANGE `identifiantclient`
`identifiantclient` VARCHAR( 50 ) CHARACTER SET utf8
COLLATE utf8_general_ci NOT NULL
ALTER TABLE `client` CHANGE `motdepasseclient`
`motdepasseclient` VARCHAR( 50 ) CHARACTER SET utf8
COLLATE utf8_general_ci NOT NULL
ALTER TABLE `client` CHANGE `nomclient` `nomclient`
VARCHAR( 50 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `client` CHANGE `prenomclient` `prenomclient`
VARCHAR( 50 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `client` CHANGE `emailclient` `emailclient`
VARCHAR( 255 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `client` CHANGE `telephoneclient` `telephoneclient`
VARCHAR( 15 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `client` CHANGE `adresseclient` `adresseclient`
```

```
VARCHAR( 255 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `client` CHANGE `villeclient` `villeclient`
VARCHAR( 255 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `client` CHANGE `codepostalclient`
`codepostalclient` VARCHAR( 10 ) CHARACTER SET utf8 COLLATE
utf8_general_ci NOT NULL
ALTER TABLE `client` CHANGE `paysclient` `paysclient`
VARCHAR( 50 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
ALTER TABLE `commande` CHANGE `totalcommande` `totalcommande`
VARCHAR( 50 ) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
NULL
```

Nous allons vérifier le fonctionnement de l'ensemble en réalisant une nouvelle saisie d'un article par copié-collé.



Nous voyons que l'insertion est incorrecte. Les données ont été traitées sans précision de l'encodage reçu. Le code reçu est en UTF-8 sous la forme d'un flux d'octets ISO-8859-1.



Pour résoudre ce problème, nous devons donc travailler avec les flux au moment de l'insertion (ou modification) des données dans la base. Nous allons modifier notre requête de création afin de travailler avec un flux d'octets et non des caractères.

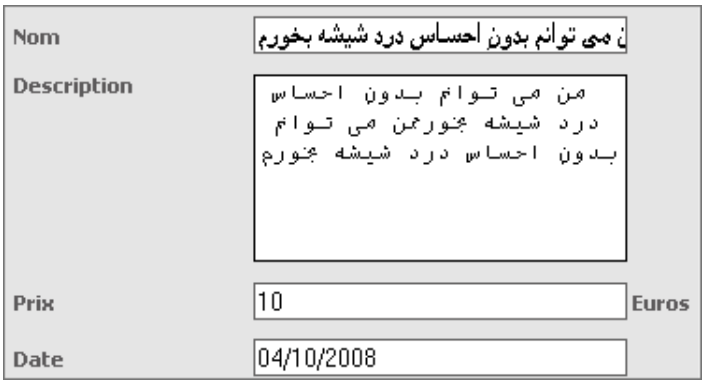
```
...
//inserer l'article
requete=connection.prepareStatement("INSERT INTO
article(nomarticle,descriptionarticle,prixarticle,datearticle,
photoarticle,vignettearticle,etatarticle,id_categorie) VALUES
(?,?,?,?,?,?,?,?)");
requete.setString(1,(String)article.getNomarticle());
requete.setString(2,(String)article.getDescriptionarticle());
```

Code avec flux d'octets :

```
...
//inserer l'article
requete=connection.prepareStatement("INSERT INTO
article(nomarticle,descriptionarticle,prixarticle,datearticle,
```

```
photoarticle,vignettearticle,etatarticle,id_categorie) VALUES
(?,?,?,?,?,?,?,?)");
requete.setBytes(1,article.getNomarticle().getBytes
("ISO-8859-1"));
requete.setBytes(2,article.getDescriptionarticle().getBytes
("ISO-8859-1"));
```

Nous gérons dans notre exemple, le nom de l'article et sa description en Unicode UTF-8. Nous recommençons la saisie et nous vérifions que les opérations sont correctes.



Les données sont correctement insérées dans la base de données (nous pourrons ainsi réaliser des recherches, tests et comparaisons sur la chaîne correcte) et affichées au format Unicode dans le navigateur. Il faut aussi modifier notre requête de modification (pas besoin de modifier les requêtes de lecture car les données sont lues au format Unicode) pour le nom et la description de l'article.

Notre système de gestion des articles (nom et description dans ce cas) est désormais multilingue. Nous pouvons vérifier l'intégrité de l'ensemble en changeant le codage de la langue du navigateur. Si nous passons de **Affichage - Encodage des caractères - Unicode (UTF-8)** à **Occidental (ISO-8859-1)**, nous retrouvons la chaîne de caractères qui correspond au problème d'insertion précédent (codage ISO-8859-1).

Parfois, avec l'utilisation d'un pool de connexion JDBC, le mécanisme de données en UTF-8 ne fonctionne pas. Pour cela, il faut ajouter les paramètres suivants au pool de connexion : *useUnicode=yes&characterEncoding=UTF-8*. Il faut préciser cela dans la partie de connexion à la base de données de l'application dans le fichier de gestion de l'application url="jdbc:mysql://localhost:3306/betaboutique?useUnicode=yes&characterEncoding=UTF-8" (server.xml ou fichier spécifique à l'application).

Nous pouvons terminer cette section en apportant quelques améliorations à l'ensemble. Il est possible par exemple d'ajouter un fichier de traduction des messages (en cas de succès et d'échec). Ainsi, suivant la langue les messages seront automatiquement traduits.

Authentification et Realms

1. Présentation

Le protocole HTTP fonctionne sous la forme de requête-réponse permettant de demander les identités des utilisateurs et de fournir l'accès aux ressources sur la base de ces identités. Lorsqu'une requête arrive à destination d'une ressource protégée, le serveur Web vérifie si le navigateur a envoyé les informations d'authentification. Si tel n'est pas le cas, le serveur envoie une page d'erreur avec le code 401 au navigateur en précisant le type d'information d'authentification qu'il attend.

Le navigateur affiche alors une boîte ou un formulaire d'authentification afin de demander à l'utilisateur les informations. Si l'utilisateur est précédemment authentifié, le navigateur utilise alors son cache local s'il détient les renseignements d'une précédente requête.

2. Les types d'authentification

a. Authentification de base

Le schéma d'authentification HTTP le plus simple se nomme autorisation de base (*Basic Authorization*). Ce mécanisme consiste à utiliser un nom d'utilisateur (identifiant) et un mot de passe en clair. Lorsqu'un utilisateur consulte pour la première fois une ressource protégée, il doit alors saisir ses informations de connexion dans une boîte de dialogue semblable à celle ci-dessous.

b. Authentification par Digest

Avec le type d'authentification basique, l'identifiant de l'utilisateur et le mot de passe sont envoyés en clair par le navigateur avec chaque requête destinée au serveur. Il s'agit là d'un moyen très simple pour gérer l'authentification utilisateur. Il existe également un autre schéma d'authentification qui autorise les transmissions codées. Avec ce type d'autorisation, l'utilisateur saisit son mot de passe en clair, mais celui-ci est envoyé sous forme crypté à partir d'une chaîne de caractères sur le réseau. Dans ce cas cette authentification est appelée authentification par Digest (*Digest Auth*) et utilise l'algorithme de hachage unilatéral SHA ou MD5.

c. Authentification par certificat

Le troisième schéma d'authentification principal se fonde sur les certificats. Avec ce type d'authentification, l'utilisateur doit posséder un certificat client pour accéder au serveur. Cette approche est la plus sûre puisque les certificats sont gérés de façon centralisée par des autorités spécialisées tel que VeriSign ou Thawte.

3. Gestion des Realms

Le protocole HTTP met en œuvre une authentification à base de Realm. Un Realm est une entité vérifiant les droits des utilisateurs pour des accès à des ressources protégées.

Sous Apache et autres serveurs Web, la configuration des Realms est gérée en ajoutant un fichier *.htaccess* au répertoire de base de la zone à sécuriser. Les informations relatives aux utilisateurs et aux rôles sont stockées dans un fichier texte ou encore dans une base de données.

Un Realm est un dispositif normalisé qui permet d'identifier les utilisateurs. Il effectue ainsi l'association entre l'identifiant et le mot de passe afin de déterminer si l'utilisateur est correctement authentifié ou non. Pour chaque utilisateur, le Realm connaît la liste des rôles associés. Les rôles sont quant à eux, les responsabilités attribuées à chaque utilisateur.

La protection des ressources est réalisée en fonction des rôles, c'est-à-dire qu'il faut préciser le rôle dont doit disposer un utilisateur pour accéder à la ressource demandée. Un Realm peut être paramétré au niveau de la balise <*Engine*>, c'est-à-dire partagé par toutes les applications, au niveau <*Host*> pour les applications d'un hôte virtuel ou au niveau <*Context*> pour une application spécifique.

Le serveur Java Tomcat dispose d'un mécanisme intégré très souple pour gérer les Realms. Il est possible d'utiliser le fichier *server.xml*, le fichier de configuration spécifique à l'application, une base de données, les données d'un serveur LDAP ou une source JNDI. Tous ces mécanismes offrent le moyen d'authentifier les utilisateurs et de les associer à des rôles.

Les règles d'utilisation des Realms sont les suivantes :

- Tout utilisateur peut être membre d'aucun ou de plusieurs rôles.
- Tous les rôles peuvent contenir aucun ou plusieurs utilisateurs.
- Les Realms permettent de définir les utilisateurs qui peuvent accéder à des ressources en fournissant la liste d'un ou de plusieurs rôle(s) disposant d'un droit d'accès.

4. Mise en place des Realms

Le fichier qui contient les informations d'authentification relatives à Tomcat est présent dans le répertoire */conf* et se nomme *tomcat-users.xml*. Voici un exemple de fichier de configuration avec les utilisateurs *tomcat* et *admin*. L'utilisateur *tomcat* possède le rôle *tomcat* et l'utilisateur *admin* possède les rôles *admin* et *manager*.

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="admin" password="admin" roles="admin,manager"/>
</tomcat-users>
```

Ce fichier texte au format XML permet de définir les rôles mais il reste encore à configurer l'application afin d'associer ces informations d'authentification à un Realm. Cette mise en place est réalisée par la balise XML <*Realm*> présente dans le fichier de configuration du serveur (*server.xml*) ou dans le fichier de configuration propre à l'application.

Voici un exemple de configuration présent dans le fichier *server.xml* :

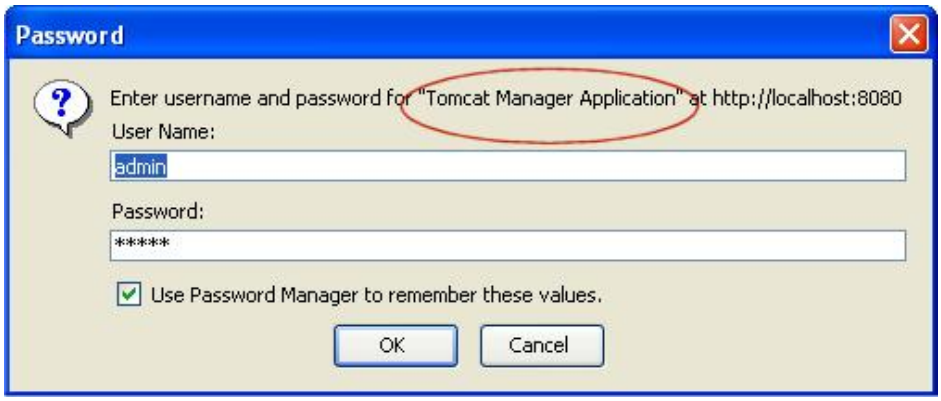
```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"/>
```

La dernière étape consiste à activer l'authentification pour une ressource donnée. Cette action est réalisée au sein du fichier de configuration de l'application *web.xml*. Le nœud <*security-constraint*> est l'élément de configuration qui doit être placé à la fin du descripteur de déploiement de l'application.

```
...
<!-- définir les accès sécurisés -->
<security-constraint>
    <display-name>Authentification</display-name>
    <web-resource-collection>
        <web-resource-name>page securisee</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Espace administration</realm-name>
</login-config>
<security-role>
    <description>Administrateur</description>
<role-name>admin</role-name>
</security-role>
...
```

La définition précédente permet de protéger tous les accès aux pages incluses dans l'application à partir de l'URL */admin* de l'application. Cette définition exige que les utilisateurs soient membre du rôle *admin*. L'authentification utilisée est de type basique.

Nous pouvons autoriser différentes contraintes pour différentes méthodes d'accès. Ainsi les méthodes d'accès HTTP peuvent être limitées. Nous pouvons ainsi utiliser une contrainte de sécurité comme pour limiter l'accès aux méthodes PUT, DELETE, GET et POST. Le nœud <*login-config*> permet de préciser le type d'authentification et le message affiché dans la boîte d'authentification (<*realm-name*>).



Pour résumer, la mise en place d'un système d'authentification basé sur un Realm, nécessite les étapes suivantes :

- Définition des utilisateurs et des rôles dans une source de données (fichier, annuaire, base de données...).
- Définition de la mise en place des accès sécurisés au sein d'un fichier de configuration.
- Définition des ressources à protéger.

5. Realm et base de données

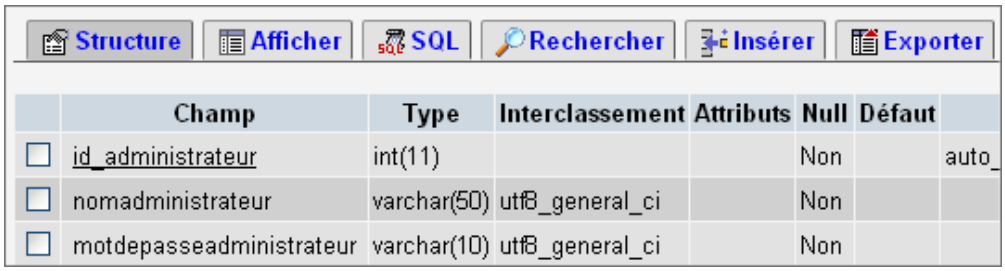
La mise en place d'un système d'authentification basé sur des fichiers est simple mais cette approche n'est pas très souple en terme de sécurité et de mise à jour. Une simple création ou modification dans le fichier de configuration (ex : *tomcat-users.xml*) peut corrompre le système d'authentification. Il est également peu prudent de laisser les mots de passe figurer en clair sur un fichier lisible par certains utilisateurs.

Il existe deux adaptateurs qui permettent de gérer les authentifications à partir d'une base de données. L'objet *JDBCRealm* stocke les informations relatives à l'utilisateur et au rôle dans une base de données et y accède via JDBC. L'objet *JNDIRealm* repose sur le même principe mais utilise un pool de connexion JNDI.

Ces objets sont très souples et permettent de définir des tables spécifiques pour gérer les utilisateurs et les rôles, mais aussi d'accéder aux informations relatives aux utilisateurs et aux rôles dans une base de données existante.

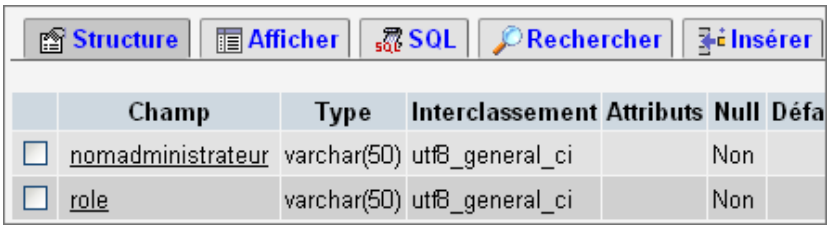
Nous allons mettre en place un système d'authentification pour le projet *betaboutique* avec une source de données JNDI. L'utilisation des Realms avec une base de données nécessite deux tables. La première contient une correspondance entre le nom d'utilisateur et le mot de passe. La seconde table contient la correspondance entre les utilisateurs et les rôles auxquels ils appartiennent. Le nom des colonnes ainsi que l'existence d'autres colonnes dans les tables ou d'autres tables dans la base de données, importent peu. Il est tout à fait possible d'adapter une application existante pour utiliser les Realms. Il suffit pour cela de bien déclarer l'application et de respecter les directives de Tomcat.

Pour notre projet, nous allons créer deux tables afin de gérer l'authentification en administration. La première table contient les informations sur les utilisateurs/administrateurs de l'application. Cette table possède la structure suivante :



	Champ	Type	Interclassement	Attributs	Null	Défaut	
☐	id_administrateur	int(11)			Non		auto_
☐	nomadministrateur	varchar(50)	utf8_general_ci		Non		
☐	motdepasseadministrateur	varchar(10)	utf8_general_ci		Non		

Le champ *nomadministrateur* correspond à l'identifiant de connexion qui sera utilisé dans la boîte d'authentification. La seconde table *roles* possède les différents rôles de la base de données avec la correspondance des utilisateurs/administrateurs.



La structure minimale imposée par Tomcat possède trois champs qui sont le nom d'utilisateur, son mot de passe et un rôle. Il est fortement conseillé de mettre le nom d'utilisateur en clé primaire afin d'avoir un index unique sur cette table. Pour le cas de l'utilisation de plusieurs rôles avec le même utilisateur, la clé primaire peut être la concaténation des champs nom et rôle.

Nous reprenons le fichier de configuration de notre application (*betaboutiquemvc.xml*) :

```
<Context path="/betaboutiquemvc" reloadable="true"
docBase="C:\PROJETWEB\applications\betaboutiquemvc"
workDir="C:\PROJETWEB\applications\betaboutiquemvc\work">
<Resource name="jdbc_betaboutiquemysql" auth="Container"
type="javax.sql.DataSource" username="root"
password="" driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/betaboutique"
maxActive="20" maxIdle="10" />
</Context>
```

Nous retrouvons la définition de la source de données JNDI ainsi que la configuration de l'application *betaboutiquemvc*. Nous allons maintenant définir notre Realm afin d'utiliser les informations d'authentification présentes dans la base de données associée au pool de connexion.

```
<Context path="/betaboutiquemvc" reloadable="true"
docBase="C:\PROJETWEB\applications\betaboutiquemvc"
workDir="C:\PROJETWEB\applications\betaboutiquemvc\work">

<Resource name="jdbc_betaboutiquemysql" auth="Container"
type="javax.sql.DataSource" username="root"
password="" driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/betaboutique"
maxActive="20" maxIdle="10" />

<Realm className="org.apache.catalina.realm.DataSourceRealm"
       dataSourceName="jdbc_betaboutiquemysql"
       localDataSource="true"

       userTable="administrateur"
       userNameCol="nomadministrateur"
       userCredCol="motdepasseadministrateur"

       userRoleTable="roles"
       roleNameCol="role"
       />
</Context>
```

La balise <*Realm*> précise la source de données utilisée (*dataSourceName*), le nom de la table qui contient les utilisateurs (*userTable*), le champ associé (*userNameCol*), le champ des mots de passe utilisateur (*userCredCol*) et enfin la table de gestion des rôles (*userRoleTable*) et la colonne correspondante (*roleNameCol*). L'attribut *localDataSource="true"* permet de préciser que la source de données JNDI est déclarée au niveau du *Context* et non dans une zone globale de la configuration du serveur.

Le Realm est maintenant opérationnel, il reste juste la dernière étape qui consiste à protéger les pages/ressources pour les accès. Comme indiqué précédemment, la configuration de la sécurité et de l'accès aux ressources est réalisée dans le fichier *web.xml* de l'application. Les pages à protéger et les rôles nécessaires pour y accéder sont déclarés dans les balises <*security-constraint*>. La balise <*display-name*> permet de définir le nom de la contrainte pour sa gestion par des outils externes (logiciel spécialisé, application JWS...). Les pages sont protégées par l'intermédiaire de la balise <*web-resource-name*> et une ou plusieurs balises <*url-pattern*> qui permettent de spécifier le motif des URL à protéger. La balise <*auth-constraint*> indique les rôles à avoir et éventuellement les méthodes HTTP qui nécessitent une authentification. La balise <*login-config*> définit la façon dont s'effectue la connexion. La balise <*real-name*> indique le nom de la zone à protéger et la balise <*auth-method*> définit le type de connexion. Nous retrouvons les quatre types d'authentification que sont :

- BASIC (authentification courante par boîte de dialogue).

- DIGEST (le mot de passe est crypté avant l'envoi par le navigateur).
- FORM (l'authentification est effectuée via un formulaire créé par le développeur).
- CLIENT-CERT (authentification SSL basée sur un certificat client).

Pour notre projet *betaboutique*, nous allons définir une authentification basée sur un Realm nommé *Authentification* qui est positionné pour les pages accessibles par l'URL */admin*. Le rôle nécessaire pour accéder à cette application est *admin* et l'authentification est de type *BASIC*.

```
...
<!-- définir les accès sécurisés -->
<security-constraint>
    <display-name>Authentification</display-name>
    <web-resource-collection>
        <web-resource-name>page securisee</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Espace administration</realm-name>
</login-config>
<security-role>
    <description>Administrateur</description>
    <role-name>admin</role-name>
</security-role>
...
```

Si nous utilisons notre application et que nous accédons à la partie administration du projet ou à une sous-page, la fenêtre d'authentification est alors affichée.

Si l'authentification n'est pas correcte, nous pouvons utiliser les *Logger* Tomcat afin de connaître la source du problème. La balise *Logger* est alors placée au niveau du Realm dans la configuration.

```
<Logger className="org.apache.catalina.logger.SystemOutLogger"
verbosity="99"/>
```

La journalisation est effectuée vers la sortie standard qui est la console Eclipse pour notre projet. Ce mécanisme d'authentification très puissant, permet de gérer les ressources de façon simple et sécurisée. Lorsqu'un utilisateur tente d'accéder à une ressource protégée pour la première fois, la méthode *authenticate()* du Realm est déclenchée. Ainsi, tous les changements effectués directement sur la base de données, tels que la configuration de nouveaux comptes, la modification de mots de passe, les rôles... sont immédiatement pris en compte.

Une fois l'authentification réalisée, l'utilisateur et ses rôles associés sont mis en cache dans Tomcat durant toute la session de l'utilisateur. Cette session dure jusqu'à ce que l'utilisateur ferme son navigateur. Toutes les modifications effectuées sur les informations de la base de données relatives à l'utilisateur déjà authentifié ne seront pas prises en compte avant la prochaine connexion de ce même utilisateur.

6. Informations relatives au Realm

Le développeur dispose de plusieurs mécanismes pour manipuler les informations suite à une authentification basée sur les Realms. L'objet *request* possède plusieurs méthodes pour tester les rôles et les accès.

```
//accès qu'avec le rôle administrateur
if(!request.isUserInRole("admin")) return mapping.findForward
("accueiladmin");
```

Le code présenté ci-dessus permet par exemple, de vérifier que l'utilisateur possède bien le rôle *admin* avant d'accéder à la page. Sinon, une redirection est réalisée sur la page d'accueil (déclarée avec une variable globale). Ce code peut par la suite être inséré dans un filtre afin de vérifier l'authentification sur les services d'un site Internet.

Nous pouvons également récupérer les informations de connexion avec l'objet *request*.

```
//récupérer l'identifiant/nom de l'utilisateur connecté
String identifiant=request.getRemoteUser();
String role="";
if(request.isUserInRole("admin"))role="Administrateur";
else if(request.isUserInRole("client"))role="Client";
```

Si la base de données ou le serveur d'annuaire ne se conforment pas aux exigences des spécifications, nous pouvons écrire notre propre classe Realm en implémentant simplement l'interface *org.apache.catalina.Realm*.

7. Realm et sécurité

L'authentification de base est facile à implémenter. Les mots de passe peu élaborés sont facilement devinés et il est alors impossible de distinguer l'utilisateur réel d'une autre personne qui utilise les mêmes nom d'utilisateur et mot de passe.

Le trafic non crypté peut facilement être mis en danger par toute personne connectée sur le même réseau. Le recours au réseau HTTPS permet de pallier cette limitation en codant chaque paquet avant de l'envoyer sur le réseau Internet. L'accès aux noms d'utilisateur et aux mots de passe transmis s'avère alors beaucoup plus difficile.

Un moyen sécurisé pour gérer l'authentification consiste à utiliser des algorithmes de cryptage. Ces algorithmes permettent de crypter le mot de passe à l'aide d'un algorithme de hachage unilatéral tel que MD5 ou SHA. Le résultat de la chaîne ne présente alors aucune ressemblance avec la chaîne d'origine. Côté client, le même algorithme est utilisé pour traiter le mot de passe avec lequel le client tente de se connecter et la chaîne codée est envoyée au serveur pour comparaison. Si les chaînes correspondent, les mots de passe sont alors considérés comme identiques.

Tomcat offre des outils permettant de générer les mots de passe cryptés manuellement à partir de lignes de commandes ou de façon dynamique. Pour générer un chiffrement dynamique MD5, nous pouvons utiliser la méthode statique *Digest()* de la classe *RealmBase*.

Le code suivant permet de générer des mots de passe cryptés selon l'algorithme précisé en paramètre.

```
import org.apache.catalina.realm.RealmBase;
...
String motDePasseEnClair="monmotdepasse";
String motDePasseCrypte= RealmBase.Digest(motDePasseEnClair,"MD5");
...
```

Pour utiliser des mots de passe cryptés dans l'implémentation du Realm, nous devons stocker ces données chiffrées dans une base de données ou un fichier texte. Ensuite, lors de la configuration du Realm, nous devons ajouter l'attribut *digest* définissant l'algorithme utilisé pour le codage des mots de passe.

```
<Realm className="org.apache.catalina.realm.DataSourceRealm"
       digest="MD5"
       dataSourceName="jdbc_betaboutiquemysql"
       localDataSource="true"
       userTable="administrateur"
       userNameCol="nomadministrateur"
       userCredCol="motdepasseadministrateur"
       userRoleTable="roles"
       roleNameCol="role"
       />
```

D'autres options existent, outre les méthodes d'authentification standards, une technique très sécurisée consiste à utiliser des systèmes de connexion basés sur des mots de passe à usage unique et des mots de passe limités dans le temps tels que SecurID. La combinaison d'un mot de passe que l'utilisateur connaît, avec un nombre généré par la

machine que l'utilisateur possède permet de créer un schéma d'identification très fort.

8. Realm et formulaire personnalisé

L'authentification de type *BASIC* présentée dans la section précédente est très simple à mettre en œuvre mais présente l'inconvénient de ne pas être gérée avec un formulaire développé par le programmeur de l'application (moins de souplesse, moins ergonomique, pas le design du site...).

Avec les Realms, nous pouvons mettre en place le même type d'authentification avec un formulaire JSP (HTML). Ce formulaire doit être conforme aux spécifications de Tomcat et posséder les champs suivants :

- Le champ pour le nom d'utilisateur (identifiant) qui doit s'appeler *j_username*.
- Le champ pour le mot de passe qui doit s'appeler *j_password*.
- L'action du formulaire qui doit être *j_security_check*.

Nous allons donc créer notre propre formulaire d'authentification nommé *authentification.jsp* présent dans le répertoire */vues*.

```
<form action="j_security_check" method="POST">
<input type="text" name="j_username"/>
<input type="password" name="j_password"/>
<input type="submit" name="Connexion"/>
</form>
```

Nous devons ensuite créer une page d'erreur qui sera référencée dans le fichier de configuration. Ces deux pages (*authentification.jsp* et *erreurauthentification.jsp*) peuvent se trouver dans un répertoire protégé.

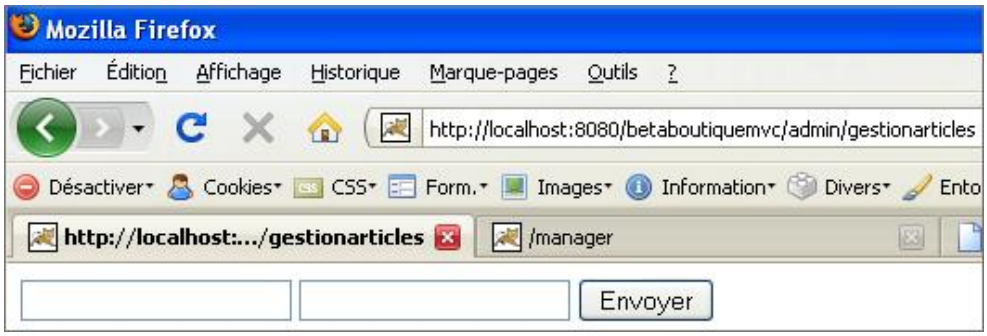
Nous devons maintenant configurer le Realm au sein du fichier *web.xml* afin d'utiliser notre formulaire d'authentification. La balise *<login-config>* permet de paramétrer les authentifications basées sur un formulaire.

```
...
<!-- définir les accès sécurisés -->
<security-constraint>
    <display-name>Authentification</display-name>
    <web-resource-collection>
        <web-resource-name>page securisee</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>default</realm-name>
    <form-login-config>
        <form-login-page>/vues/authentification.jsp</form-login-page>
        <form-error-page>/vues/erreurauthentification.jsp</form-error-page>
    </form-login-config>
</login-config>
<security-role>
    <description>Administrateur</description>
    <role-name>admin</role-name>
</security-role>
...
```

Les noms des pages d'authentification et d'erreur sont libres et sont paramétrés avec les balises *<form-login-page>* et *<form-error-page>*.

Cette mise en place nécessite un redémarrage de Tomcat. En effet, les configurations de la connexion sont conservées dans le cache local et sont donc présentes durant toute la durée de vie de l'application. L'utilisation des Realms à partir d'un formulaire personnalisé permet de gérer la déconnexion des utilisateurs. En effet, les Realms utilisent le mécanisme de session pour stocker les informations. La destruction d'une session permet ainsi de déconnecter l'utilisateur courant.

Si nous essayons d'accéder désormais à une page protégée de l'application, le formulaire d'authentification est alors affiché en lieu et place de la fenêtre HTTP.



En cas de mauvaise saisie, c'est notre page d'erreur qui est déclenchée.

Enfin, si un utilisateur correct est utilisé pour l'authentification, l'accès aux ressources protégées est alors autorisé.

	id_administrateur	nomadministrateur	motdepasseadministrateur
	5	lafosse	jerome

9. En résumé

Ce chapitre a présenté Java et l'utilisation des bases de données. La première partie a concerné la présentation des différents types de pilotes JDBC et l'utilisation d'une connexion associée. Les interfaces utilitaires ont été expliquées pour la préparation des requêtes, l'exécution de requêtes pré-compilées et le parcours des résultats.

Dans un deuxième temps, le guide a introduit le principe du partage de connexions par l'intermédiaire d'exemples concrets.

La troisième partie a introduit la notion d'écouteur afin de mettre en place des connexions partagées. La partie suivante a développé la notion de pool de connexions qui sont utilisés lors de développements de projets complexes.

La cinquième partie a insisté sur le modèle MVC et l'utilisation des bases de données pour ce modèle de conception. Le développement doit alors suivre un protocole de développement adapté, afin de respecter le standard. Le modèle a été mis en application par l'intermédiaire de la gestion des articles de la boutique betaboutique. Ce chapitre a permis de manipuler les Servlets, les classes modèles mais aussi JavaScript et la technologie Ajax.

Nous avons ensuite abordé de façon détaillée les classes modèles du projet avec les JavaBeans, les transactions et les développements multilingues. Les portions de code sont toujours détaillées à partir du projet de la boutique et le service d'administration des articles a été réalisé dans sa totalité.

Enfin la dernière partie a été axée sur la mise en place de la sécurité à base de Realm et de comptes utilisateur. Ce dernier paragraphe a ainsi permis d'utiliser l'authentification à partir de fichiers mais aussi de bases de données.

Framework

1. Présentation

Il existe en programmation deux types d'individus :

- les programmeurs système ;
- les programmeurs d'applications.

Les programmeurs système écrivent le code qui sera utilisé par les programmeurs d'applications. Les programmeurs système développent les langages Java, PHP, C ou encore C++ et les programmeurs d'applications utilisent ces langages et outils pour créer de la valeur ajoutée à des fins commerciales. Les programmeurs d'applications se concentrent sur leurs projets sans se soucier des techniques et mécaniques de bas niveaux. Les programmeurs d'applications utilisent des outils ou bibliothèques appelées framework.

Un framework est un ensemble de bibliothèques, d'outils et de règles à suivre qui aident au développement d'applications. Les frameworks sont développés par des programmeurs systèmes. Un framework est composé de plusieurs briques/composants qui sont en interaction les uns avec les autres. Les applications peuvent être écrites de manière plus efficace si le framework utilisé est adapté au projet au lieu d'être obligé de réinventer à chaque fois la roue. Un framework fournit un ensemble de fonctionnalités à partir d'une implémentation objet. Lors de développements à grande échelle et de conception par équipe, les frameworks sont alors très utiles, voire indispensables. Actuellement, il existe différents types de frameworks :

- les frameworks d'infrastructure système, qui permettent de développer des systèmes d'exploitation, des outils graphiques et des plates-formes Web (Struts, Spring...) ;
- les frameworks communicants (appelés intergiciels) ;
- les frameworks d'entreprise (développements spécifiques) ;
- les frameworks de gestion de contenu (type CMS).

Les frameworks permettent la réutilisation de code, la standardisation du développement et l'utilisation du cycle de développement de type itératif-incrémental (spécification, codage, maintenance et évolution). Un framework avec son cycle de vie est aussi désigné comme un progiciel évolué. Actuellement, il existe beaucoup de frameworks dans tous les domaines d'application et avec pratiquement tous les langages. Voici une liste non exhaustive des frameworks utilisés ainsi que les langages associés :

- Apache Struts - Java EE
- JSF (Java Server Faces) - Java EE
- Spring - Java EE
- Zend Framework - PHP
- Jelix - PHP

2. Pourquoi utiliser un framework

Les Servlets ont été définies en 1998 et deux ans après, de grandes entreprises avaient déjà misé sur Java pour leurs applications Web. Pendant plusieurs années, ces entreprises ont développé leurs projets de façon autonome sans standards. Aujourd'hui, toutes ces sociétés mesurent l'importance des frameworks. Le choix du framework de développement est stratégique pour une entreprise, il sera déterminant pour la qualité, la productivité et la pérennité des projets.

Normes et standards

Le développement informatique avec l'utilisation de normes permet de généraliser les bonnes pratiques et

d'harmoniser les développements au sein de l'entreprise, ce qui facilite la maintenance. L'utilisation de normes au sein des entreprises est en place depuis plusieurs années, toutefois le développement Java EE permet d'utiliser des normes, mais également des outils complexes eux-mêmes normalisés.

Framework et développement Web

La définition initiale de l'API Servlet est trop faible pour envisager un développement complexe d'application totalement basé sur des Servlets. Au départ, les applications Java étaient basées sur le principe de l'API CGI et progressivement les frameworks Java sont apparus pour combler les manques de l'API Servlet et JSP. Le choix de l'API aura un impact non négligeable sur les performances, la réalisation, la qualité et la maintenance de l'application. De même, puisque le framework sera le socle de base sur lequel le logiciel sera construit, sa pérennité sera elle-même fondamentale.

3. Les différents frameworks

Il existe plusieurs types d'outils pour le développement d'application. Un framework de type ''maison'', c'est-à-dire développé par l'entreprise, n'est pas la meilleure solution. Dès les premières années de Java, les équipes d'informaticiens ont développé leurs propres outils pour le développement et de grandes entreprises ont parfois construit leur propre framework. Ces développements sont à éviter car aucune entreprise ne pourra consacrer suffisamment d'efforts nécessaires pour la maintenance et l'évolution du framework. De plus, les frameworks OpenSource deviennent des standards et sont testés, validés à une échelle mondiale par l'intermédiaire des projets réalisés.

Les frameworks d'éditeur présentent un risque pour les entreprises d'un point de vue développement. En effet, ils possèdent toujours un objectif caché qui est la fidélisation de l'entreprise sur les outils de l'éditeur.

Les frameworks OpenSource sont actuellement les plus nombreux et les plus aboutis. Ils intègrent la qualité du travail et la même dynamique que le projet Apache. Une bonne part des projets de frameworks est d'ailleurs issue du consortium Apache. Les frameworks sont des outils complexes quelle que soit la qualité de développement et l'origine des projets. Il n'est pas nécessaire de maîtriser tous les frameworks existants mais ils doivent être utilisés correctement. Une fois le framework choisi, il est alors nécessaire de se former et de constituer une cellule d'assistance aux équipes de développement.

4. Quel framework choisir ?

Le développement Internet basé sur la technologie Java a été submergé par des API et outils de toutes sortes. Le choix d'un framework est basé sur différents critères :

- Est-ce que l'on doit tout concevoir de A à Z ?
- Est-ce que le développement permet l'utilisation d'une application précédemment développée ou une partie ?
- Est-ce qu'il est possible d'utiliser un environnement comme fondement de l'application ?

La conception de A à Z permet de parfaitement maîtriser une technologie mais nécessite beaucoup de temps et d'argent. Le développement à partir d'applications existantes est intéressant uniquement si les développeurs des projets antérieurs sont présents. La troisième approche (utiliser un environnement comme fondement de l'application) est sans aucun doute la meilleure dans la plupart des cas.

Le choix de l'environnement de développement ou framework entraîne plusieurs questions :

- Est-ce que l'environnement prend en charge toutes les technologies utilisées ?
- Est-ce que l'environnement s'appuie sur une communauté importante et fiable d'utilisateurs ?
- Quel est le temps de formation et de parfaite maîtrise de l'environnement ?

Le temps de formation est un élément important. Les développeurs ne doivent pas changer de framework pour chaque projet et le fait de rester fidèle à un outil et de parfaitement le maîtriser permet de suivre son évolution et pourquoi pas d'apporter sa propre personnalisation à l'outil. Le choix final d'un framework doit être étudié par un architecte de développement. La solution la plus adaptée sera celle qui utilisera les architectes les plus compétents et le meilleur framework.

Apache-Struts

1. Présentation

Le projet OpenSource Jakarta-Struts permet la modification du code afin de correspondre à nos exigences. Struts est quasiment devenu le standard de fait pour les projets Java EE. Struts est un environnement agréable et puissant qui gère l'application ainsi que les tâches courantes (routage, actions, validations...). Un autre avantage de son utilisation est le nombre croissant d'utilisateurs qui tendent à pérenniser le projet. Actuellement beaucoup d'environnements de développement comme Eclipse proposent des outils pour la programmation Struts.

Struts est un framework qui offre des outils de validation des entrées utilisateurs (saisies et formulaires), des bibliothèques de balises JSP pour la création rapide de pages, une technique de routage pour les pages et accès Web et un processus de création de formulaires sans utiliser de code Java. Struts offre également d'autres avantages :

- Struts fonctionne avec tous les serveurs Java EE (Tomcat, WebSphere, Weblogic...).
- Struts est une architecture solide et stable (projet Apache).
- Struts est adapté aux applications Web de grande taille.
- Struts permet de décomposer une application complexe en composants plus simples.
- Struts garantit un développement similaire par les équipes de programmeurs.
- Struts possède une documentation abondante.
- Struts permet un développement rapide et peu onéreux.

Struts propose une méthode de développement qui s'appuie sur un fichier de configuration au format XML qui est très simple à comprendre et à utiliser.

2. Struts et MVC

Le modèle MVC (Modèle Vue Contrôleur) est un design pattern ou patron de conception. Les designs patterns sont issus du langage UML et permettent de représenter sous forme de diagrammes de classes une solution à un problème. Le M correspond au Modèle et représente les informations que l'application manipule pour les affichages, enregistrements et lectures. Le modèle est représenté par des objets simples instanciés à partir d'une classe JavaBean. Les objets JavaBean sont alimentés à l'aide d'un code de persistance. Le V correspond à la Vue et représente la manière dont les informations sont affichées. En programmation Java EE les vues correspondent aux pages JSP. Enfin, le C correspond au Contrôleur et permet de gérer le travail des vues et du modèle. Le contrôleur est développé avec une Servlet qui est appelée par une page HTML ou JSP.

Le modèle MVC est donc composé des trois éléments suivants :

- le contrôleur : Servlets ;
- le modèle : JavaBean et outils de persistance des données ;
- la vue : pages JSP.

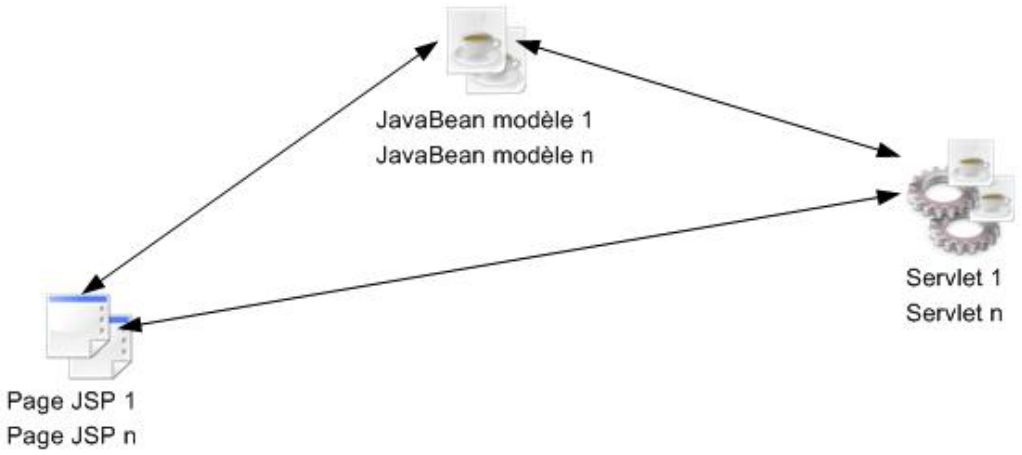
Pour chaque couche du modèle MVC il existe des frameworks. Struts est un framework qui fait partie de la couche présentation. Il permet de gérer les actions, le routage et l'affichage des données mais pas la persistance. Pour la couche de persistance des données, il est possible d'utiliser JDBC ou un framework de persistance comme JDO ou Hibernate qui sont actuellement les plus populaires.

Lors du développement MVC, pour chaque action il existe une Servlet, un JavaBean et une page JSP associée, ce qui rend le développement relativement long et complexe. Pourquoi alors, ne pas utiliser une seule Servlet qui contrôlerait toute la chaîne de production à partir d'actions ?

Ce modèle (design pattern) existe, c'est le modèle MVC II .

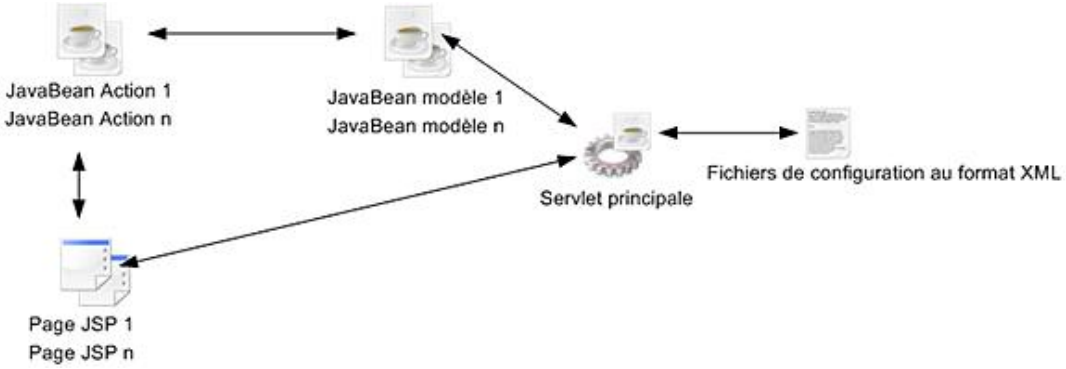
Dans le schéma ci-dessous MVC I, plusieurs Servlets jouent le rôle de contrôleur (ex: créer le client, supprimer le

client...), chacune contrôlant un JavaBean (ou plusieurs) et une page JSP (ou plusieurs). Le seul fichier de configuration est le descripteur de déploiement de l'application *web.xml*. La Servlet associée à une action/traitement s'occupe des traitements et du routage de l'utilisateur vers les vues adaptées.



Dans le second schéma, de type MVC II, il n'y a qu'une seule Servlet qui joue le rôle de contrôleur. Celle-ci s'appuie sur un fichier de configuration au format XML qui fait le lien entre les JavaBean actions de présentation et les pages JSP. Le code à ajouter habituellement dans les Servlets MVC I doit être complété dans des classes additionnelles simples qui sont les actions.

L'avantage du MVC II par rapport au MVC I est l'utilisation du fichier de configuration qui permet d'éviter de toucher au code. D'autre part, la rapidité de développement est accrue avec ce type de conception. Le modèle MVC II laisse alors libre choix d'utiliser toutes les technologies Java disponibles : Services Web, JavaBean, EJB...



Struts respecte le modèle MVC II avec les éléments suivants :

- La **vue** : elle est représentée par des pages JSP.
- Le **contrôleur** : il est constitué d'une seule Servlet *ActionServlet* qui est déjà codée par les développeurs Struts. Une classe qui hérite de la classe Struts *ActionServlet* doit alors être créée.
- Le **modèle** : il n'est pas implémenté avec Struts. Les JavaBean et des techniques associées (EJB, XML, outils de persistance ou JDBC) sont généralement utilisés.
- Le **fichier de configuration** : le fichier *struts-config.xml* proposé par Struts permet d'assembler et de configurer tous les composants du projet.

Tous les éléments du modèle MVC II sont reliés et configurés par l'intermédiaire du fichier de configuration *struts-config.xml*.

3. Installation du framework

Avant de commencer les développements, il faut installer le framework à partir des sources disponibles sur le site d'Apache (http://struts.apache.org/).

Actuellement, il existe deux versions de struts :

- Struts V1.X qui est le standard et qui permet le développement MVC II.
- Struts V2.X qui est la dernière version, basée sur l'association des frameworks WebWork et Struts V1.X.

Pour ce guide, nous utiliserons la version V1.2 (1.2.8) de Struts qui est actuellement le standard en entreprise. La dernière version très récente, ne bénéficie pas encore d'assez de recul. Il n'y a pas encore assez de projets complexes développés par des entreprises pérennes pour tirer une grande expérience de cette nouvelle version. Les guides de développements et tutoriaux des entreprises sont tous basés sur une version précédente à V1.3. En effet, beaucoup de changements sont apparus depuis cette version et sont surtout utilisés pour réaliser la transition vers Struts V2.X.

L'installation de l'API est simple, il suffit de copier les fichiers téléchargés dans les répertoires d'une application Web Java EE traditionnelle. Il sera possible de créer un nouveau projet et de copier les fichiers nécessaires (librairies *.jar* et fichiers *.xml*) ou alors d'installer le framework en utilisant une application Struts vierge livrée avec le framework (http://struts.apache.org/download.cgi http://archive.apache.org/dist/struts/binaries/struts-1.2.8-bin.zip).

Le framework Struts V1.2.8 est composé de plusieurs fichiers.

Les librairies Java (*.jar*) qui contiennent toutes les classes utilisées par le framework :

- *antlr.jar* (librairie de développement à partir d'une description grammaticale).
- *common-beanutils.jar* (librairie de manipulation de JavaBean).
- *commons-digester.jar* (librairie de gestion du mapping XML vers des objets Java).
- *commons-fileupload.jar* (librairie de gestion de l'upload en Java).
- *commons-logging.jar* (librairie de loggin/traces).
- *commons-validator.jar* (librairie de validation et de règles à partir d'un fichier XML).
- *jakarta-oro.jar* (librairie de gestion des expressions régulières Perl).
- *struts.jar* (librairie complète de Struts).

Les bibliothèques de tags ou taglibs (.tld) qui permettent de gérer les balises Struts dans les JSP :

- *struts-bean.tld*
- *struts-html.tld*
- *struts-logic.tld*
- *struts-nested.tld*
- *struts-tiles.tld*

Depuis la version V1.3 de Struts, les grammaires *.tld* ne sont plus utilisées directement mais référencées à partir d'URL liées au site d'Apache. Ex : *<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>*.

Les fichiers de configuration de l'application (*.xml* et *.properties*) :

- *struts-config.xml* (configuration de l'application MVC) .
- *validator-rules.xml* (configuration des règles de validation) .
- *web.xml* (fichier de configuration de l'application) .

- *MessageResources.properties* (fichier de gestion des propriétés et des traductions pour les sites multilingues).

Il existe deux façons d'installer le framework Struts :

- La première installation de Struts consiste à copier les librairies *.jar* dans le répertoire */WEB-INF/lib* de l'application.

  Les fichiers *.xml* doivent être copiés dans le répertoire */WEB-INF* de l'application.

  Les fichiers *.properties* doivent être copiés dans le répertoire */WEB-INF/src* du projet.

- Il existe également une application Struts vide, livrée en standard qui permet d'installer le framework. Cette application vide porte le nom de *struts-blank.war* pour indiquer qu'elle est vierge.

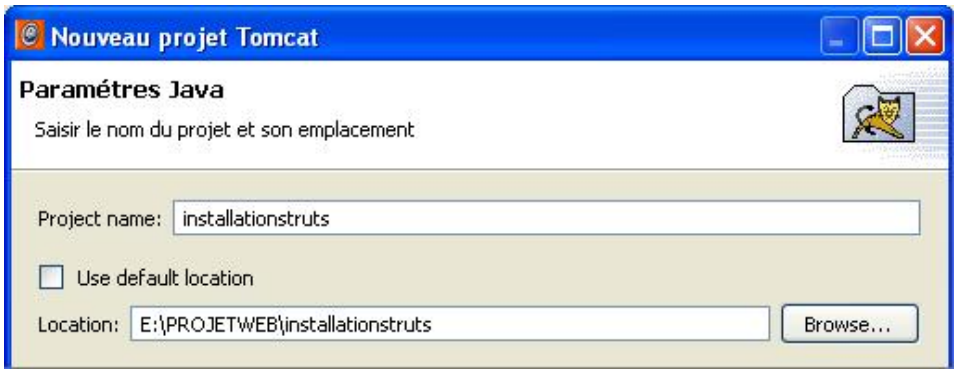
Pour installer cette application, il est nécessaire de :

- Copier l'archive *struts-blank.war* dans un répertoire (ex : *installationstruts*).

- Décompresser son contenu.

- Ouvrir **Eclipse** et cliquer sur **Fichier - Nouveau - Projet - Java - Projet Tomcat**.

- Nommer le projet (ex : *installationstruts*) et sélectionner le répertoire précédent.



- Démarrer Tomcat et un navigateur.

- Saisir l'URL suivante dans le navigateur http://localhost:8080/installationstruts/

Au lancement, une erreur est affichée à l'écran. En effet, la page d'accueil utilise un fichier pour les messages (propriétés et traductions). Ce fichier de traduction est présent dans le paquetage nommé *java*. Il faut donc correctement référencer le projet pour indiquer le nom du paquetage.

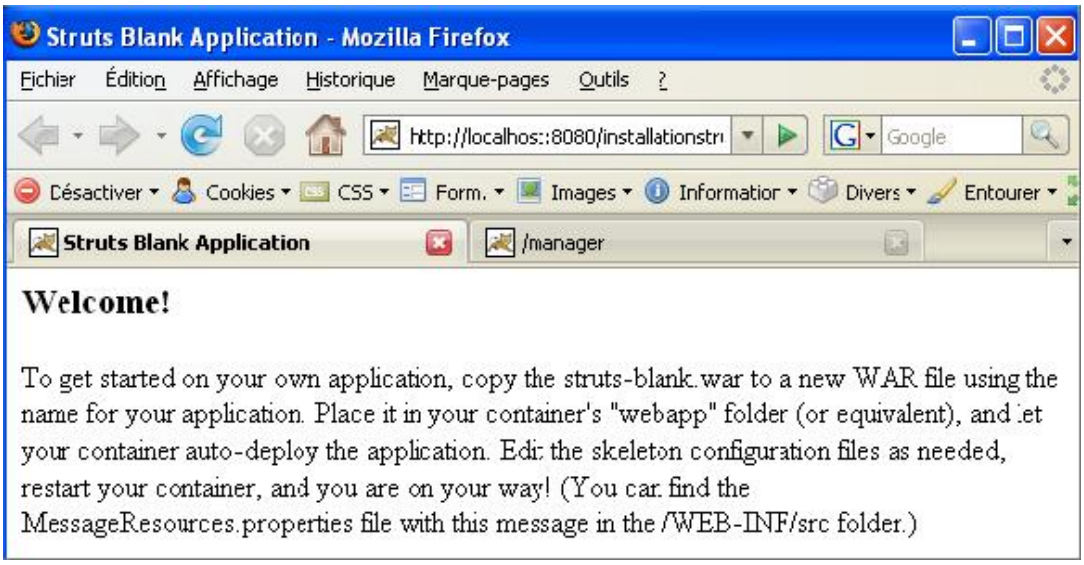
- Éditer le fichier *struts-config.xml* et modifier la balise *<message-resources .../>*.

- Ajouter le nom du paquetage à la balise *<message-resources parameter="java.MessageResources"/>*.

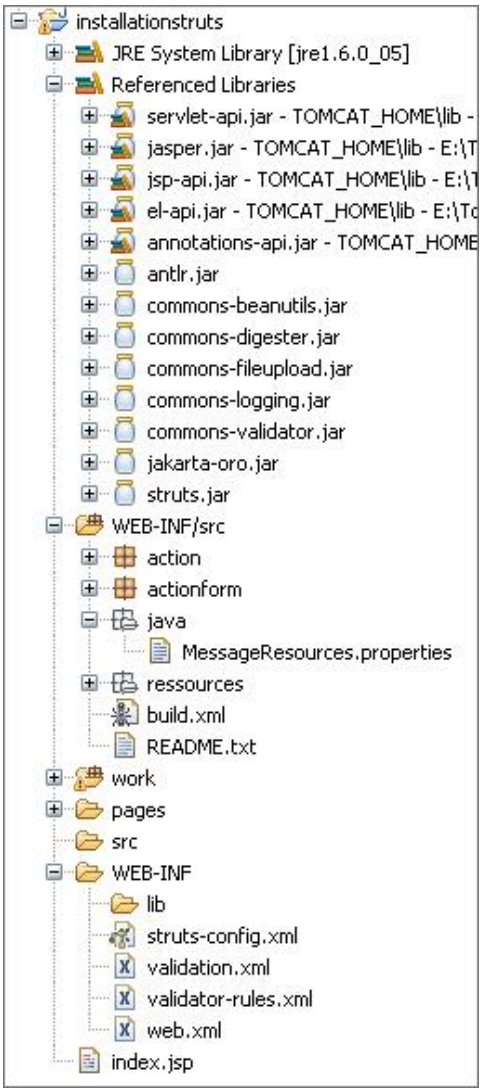
```
<!-- ================================ Message ressources Definitions -->
    <message-resources parameter="java.MessageResources" />
<!-- ========================================= Plug Ins Configuration -->
```

- Relancer l'application avec le manager Tomcat et actualiser la page.

Le résultat affiché est le suivant :



Le projet Web Struts possède la structure suivante :



Nous retrouvons le nom du projet (*installationstruts*), les librairies chargées et utilisées par le framework (*antlr.jar*, *commons-beanutils.jar*, *commons-digester.jar*...), le répertoire des sources du projet (*/WEB-INF/src*), le répertoire de configuration du projet (*/WEB-INF*) et le répertoire */META-INF*, qui n'est pas utile pour le moment et qui permet uniquement de déployer l'application.

Le répertoire */WEB-INF* contient les trois fichiers de configuration de l'application et le fichier des règles de validation. Le répertoire */pages* contient les vues JSP. Enfin, le répertoire */WEB-INF/src* contient un paquetage Java nommé *java* avec le fichier des propriétés/traductions *MessageResources.properties*, un fichier explicatif *README.txt* et un fichier ANT

(*Another Neat Tool*) *build.xml* pour le déploiement de l'application.

L'application Web est désormais correctement configurée pour utiliser le framework Struts V1.X. Nous pouvons maintenant commencer à étudier le développement avec Struts.

Projet Web

1. Présentation

Pour cette partie du guide, nous allons développer une application Struts de gestion et d'administration d'un chat utilisateur (application de communication instantanée). Ce chat lié au projet *betaboutique* est développé suite à une demande des utilisateurs/acheteurs qui souhaitent dialoguer en direct avec le service après-vente de la boutique. Cette application nommée *chatbetaboutique*, permet ainsi aux utilisateurs de poser en temps réel des questions sur les nouveautés, arrivages ou remboursements de la boutique.

Seule la partie administration et gestion du chat sera traitée dans cet ouvrage sans se soucier des outils de communication du chat (d'où l'intérêt d'un développement à partir de couches).

L'application Web de gestion du chat *betaboutique* sera développée en trois étapes ou briques :

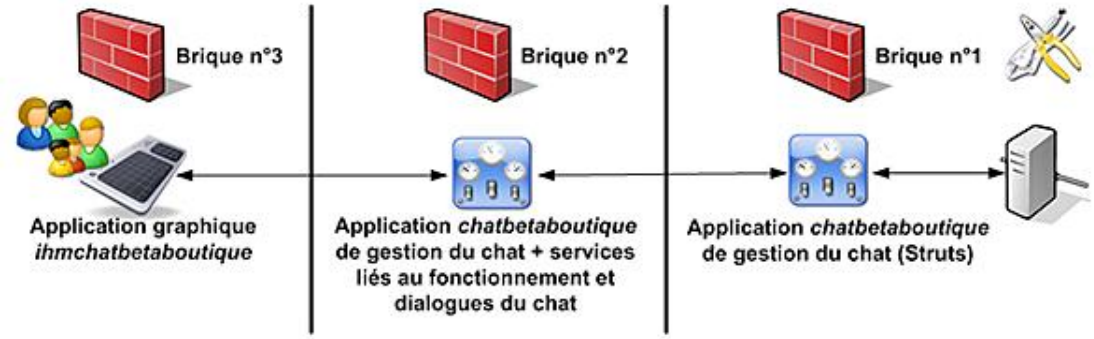
1 - La première brique qui sera développée, consiste à programmer une application Web Struts nommée *chatbetaboutique*. Ce projet s'occupe de la gestion des comptes utilisateur (création, modification, suppression), des salons (création, modification et suppression) et des inscriptions des utilisateurs aux salons. Cette application reprend le style graphique de la boutique (partie administration) mais utilise le framework Struts en lieu et place des Servlets et pages JSP traditionnelles.

2 - L'application Web Struts *chatbetaboutique* développée dans la partie 1 pourrait être agrémentée d'un ensemble de services liés aux dialogues et aux synchronisations des messages. En effet, lors de cette seconde brique, nous pourrions développer des services et un ensemble de pages JSP pour poster des messages dynamiques qui seraient stockés dans une base de données. De même, cette étape nécessiterait le développement d'un ensemble de pages JSP pour lire les messages, les utilisateurs inscrits et les salons, au format XML.

Il existe une multitude de techniques pour gérer un chat. Nous allons pour notre projet utiliser un serveur Java et la technologie Servlet pour la gestion des actions de communication du chat (insertion d'un nouveau message, lecture des messages, liste des utilisateurs inscrits...). Le plus complexe, lors du développement de services de ce type est la gestion des accès concurrents et des temps de réponse. Les sockets sont souvent utilisés en programmation. L'utilisation des technologies Java EE permet de gérer les accès concurrents, les temps de réponse et la fiabilité de l'ensemble.

---

3 - La troisième et dernière brique, concerne l'interface graphique qui est une couche supplémentaire et qui pourrait être développée avec la version Java SE. Les affichages seraient par exemple réalisés avec une application SWING et un système de parsage des données reçues au format XML.



C'est là tout l'intérêt d'utiliser des technologies adaptées et portables. En effet, rien n'empêche de développer le chat avec le framework Struts afin d'optimiser l'application et de bénéficier des outils actuels. En utilisant un affichage des données au format XML, une interface graphique pourra par la suite être développée en J2SE (Java Standard), en Flash, en JavaScript ou avec n'importe quel autre langage capable de parser du code XML.

Nous allons donc nous intéresser aux deux premières briques du projet (n°1 et n°2) indépendamment de la troisième et dernière brique liée à l'interface graphique. Cette façon de programmer par brique ou couche permet de mieux développer chaque partie, d'utiliser les meilleurs développeurs pour chaque service et de tester très précisément chaque module (ex : tests pour la création d'un nouveau message, tests pour l'affectation d'un utilisateur à un salon...). Enfin, lors du débuggage ou de la maintenance, il sera beaucoup plus aisé de trouver la brique concernée par le problème ou l'évolution du système.

2. Spécification de l'application de chat

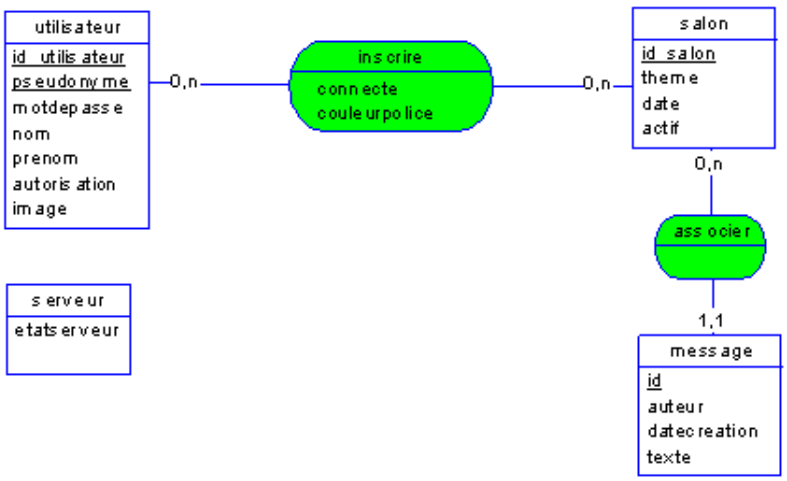
Il existe évidemment plusieurs implémentations envisageables lors du développement d'un projet. Pour le projet

chatbetaboutique, nous allons présenter le Modèle Conceptuel des Données MERISE afin d'introduire les informations à gérer par la suite.

Les utilisateurs de la base de données *chatbetaboutique* correspondent en fait aux utilisateurs de la base *betaboutique*. Nous utilisons une nouvelle base de données afin de bien séparer les deux projets et d'éviter de mélanger les services.

La nouvelle base de données créée est nommée *chatbetaboutique*. Cette base de données possède cinq tables qui sont *utilisateur*, *salon*, *utilisateursalon*, *message* et *serveur*.

La structure de la base de données est représentée avec le MCD suivant :



La table *utilisateur* est simple et permet de représenter une personne à partir d'un identifiant unique (*id_utilisateur*) et d'un pseudonyme (*pseudonyme*). La clé est composée du pseudonyme de l'utilisateur et de son id (id_utilisateur). Le pseudonyme fait partie de la clé afin d'avoir un identifiant unique dans la base et l'id est uniquement utilisée pour améliorer le développement (partie gestion des comptes). Le mot de passe de l'utilisateur (*motdepasse*) est stocké afin de permettre à l'utilisateur de s'authentifier par la suite lors de l'utilisation du chat. Enfin, il y a des informations simples comme le nom et le prénom (*nom* et *prenom*). L'image (*image*) permet d'affecter un avatar à l'utilisateur afin d'améliorer l'ergonomie de l'ensemble.

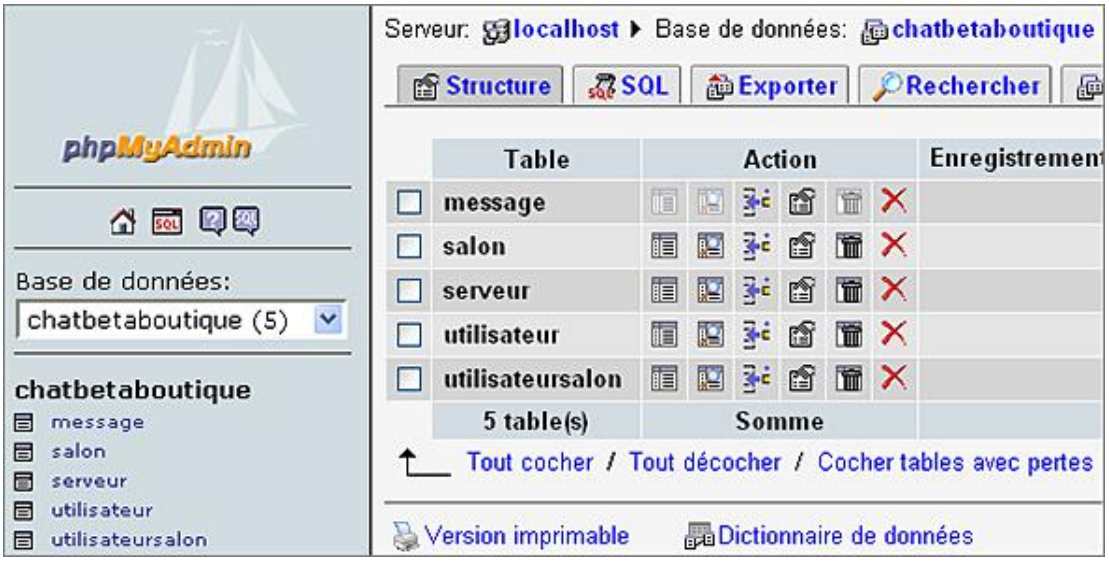
La table *salon* est composée d'une clé automatique (*id_salon*) et d'un champ texte (*theme*) pour enregistrer le thème ou nom du salon. Le champ *date* permet de spécifier à quelle date aura lieu le chat pour le salon indiqué et enfin, le champ *actif* permettra d'activer ou pas un salon avec l'interface graphique. Un chat pourra être ainsi arrêter dynamiquement (pas d'authentification, de lecture ou d'insertion de message si le chat est désactivé).

La table *utilisateursalon*, qui correspond à la relation insérée, est composée d'une clé primaire qui est la concaténation du pseudonyme utilisateur et de l'identifiant du salon (*pseudonyme* et *id_salon*). Le champ *connecte* permet d'indiquer si un utilisateur inscrit est connecté ou pas au salon. Enfin, le champ *couleurpolice* sera utilisé par la suite afin d'affecter une couleur de police pour le texte de l'utilisateur.

La table *message* permet de stocker les messages du chat. Ses champs sont l'identifiant du message (*id*) qui est un entier incrémenté automatiquement, l'auteur du salon (*auteur*) qui est en fait son pseudonyme (afin d'accélérer les lectures, pas besoin de jointure), la date de création du message au format timestamp (*datecreation*) avec les millisecondes (afin de gérer l'ensemble avec précision), le texte du message (*texte*) et le salon concerné par le message (*id_salon*).

La table *serveur* permet de désactiver en un seul clic la totalité du chat avec son champ *etatserveur*.

La base de données *chatbetaboutique* sera au format UTF-8 afin de gérer les conversations multilingues. La structure MySQL et le script de création de l'ensemble sont présentés ci-dessous.




```
-- Base de données: `chatbetaboutique`
-- --------------------------------------------------------
-- Structure de la table `message`
CREATE TABLE `message` (
  `id` int(11) NOT NULL auto_increment,
  `auteur` varchar(30) default NULL,
  `datecreation` varchar(30) default NULL,
  `texte` varchar(250) default NULL,
  `id_salon` int(11) NOT NULL default '0',
  PRIMARY KEY  (`id`),
  KEY `datecreation` (`datecreation`),
  KEY `id_salon` (`id_salon`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 AUTO_INCREMENT=229 ;
-- --------------------------------------------------------
-- Structure de la table `salon`
CREATE TABLE `salon` (
  `id_salon` int(11) NOT NULL auto_increment,
  `theme` varchar(255) NOT NULL default '',
  `date` timestamp NOT NULL default CURRENT_TIMESTAMP on
update CURRENT_TIMESTAMP,
  `actif` char(1) NOT NULL default '',
  PRIMARY KEY  (`id_salon`),
  KEY `date` (`date`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 AUTO_INCREMENT=4 ;
-- --------------------------------------------------------
-- Structure de la table `serveur`
CREATE TABLE `serveur` (
  `etatserveur` char(1) collate utf8_bin NOT NULL default '0'
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
-- --------------------------------------------------------
-- Structure de la table `utilisateur`
CREATE TABLE `utilisateur` (
  `id_utilisateur` int(11) unsigned NOT NULL auto_increment,
  `pseudonyme` varchar(30) NOT NULL default '',
  `motdepasse` varchar(15) default NULL,
  `nom` varchar(255) NOT NULL default '',
  `prenom` varchar(255) NOT NULL default '',
  `autorisation` char(1) NOT NULL default '0',
  `image` varchar(255) default NULL,
  PRIMARY KEY  (`id_utilisateur`,`pseudonyme`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 AUTO_INCREMENT=375 ;
-- --------------------------------------------------------
-- Structure de la table `utilisateursalon`
CREATE TABLE `utilisateursalon` (
  `pseudonyme` varchar(30) NOT NULL default '',
  `id_salon` int(11) NOT NULL default '0',
  `connecte` char(1) NOT NULL default '0',
  `couleurpolice` varchar(50) NOT NULL default '0',
```

```
  PRIMARY KEY  (`pseudonyme`,`id_salon`),
  KEY `pseudonyme` (`pseudonyme`),
  KEY `id_salon` (`id_salon`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Si nous considérons un exemple d'utilisation, nous aurons l'utilisateur *jlafoss* inscrit aux salons *sav* et *nouveautés*. Il sera actuellement connecté au salon *sav* avec une police de couleur *-13023407* qui correspond au format numérique de la couleur RVB et il aura posté deux messages pour ce salon avec les dates précises au format timestamp.

Nous allons donc aborder l'utilisation du framework Struts avec les exemples liés au développement de la partie administration du chat utilisateur. Nous allons concevoir la première brique du projet qui consiste à gérer les utilisateurs, les salons et les inscriptions des utilisateurs aux salons. Les techniques, validations, actions, bibliothèques de tags et formulaires Struts seront présentés à travers le développement de ces services.

Formulaires Struts

1. Présentation

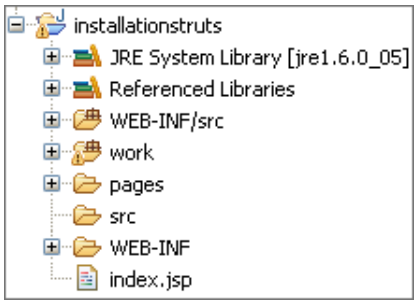
Avec le modèle MVC II, la vue est représentée par deux composants et non par un seul comme dans le pattern MVC I. Ces deux composants sont une page JSP et une classe qui est un objet JavaBean. Les pages JSP Struts sont composées de taglibs développées par Struts. Ces balises permettent de ne pas insérer de code Java dans les pages JSP en conformité avec le pattern MVC. Ces bibliothèques de tags sont en fait des balises HTML supplémentaires qui aident les développeurs de pages Web. Pour utiliser ces bibliothèques de tags dans les pages JSP, il faut les déclarer en début de page. La déclaration simple est réalisée avec la directive *<%@ taglib uri="..." prefix="..." %>*. Le paramètre *uri* permet de définir le lien vers le fichier *.tld* qui est la grammaire de la bibliothèque de balises. Le paramètre *prefix* définit le préfixe des balises dans la page JSP en cours.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
```

Dans les définitions précédentes, nous retrouvons le paramètre *uri* qui fait référence à une grammaire présente sur le site de Struts. Le préfixe utilisé est bean, donc plusieurs balises *<bean: .../>* pourront être utilisées comme la balise ci-dessous qui permet d'afficher un message présent dans le fichier de propriétés :

```
<bean:message key="welcome.title"/>
```

La déclaration des taglibs Struts est relativement simple et chaque page JSP développée avec Struts devra inclure au moins les trois librairies précédentes (*bean*, *html* et *logic*). Afin de mettre en application et de détailler l'utilisation des formulaires avec Struts, nous allons utiliser le projet *installationstruts* précédemment créé. Le projet, à cette étape du guide doit être semblable à la vue ci-après.



Par convention, les pages JSP qui utilisent des formulaires avec Struts sont suffixées par le terme *Form*.

Nous allons créer un nouveau formulaire nommé *authentificationForm.jsp* dans le répertoire */pages*.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head>
<title>Authentification</title>
<html:base/>
</head>
<body>
<html:form action="/Authentification">
Identifiant : <html:text property="identifiant"/><br/>
Mot de passe <html:text property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

Les balises Struts *<html:html>*, *<html:form>*, *<html:text>*... sont utilisées. Ces balises sont interprétées par le serveur et transformées en code HTML compatible XHTML.

Si nous essayons d'exécuter cette page avec l'URL suivante

(http://localhost:8080/installationstruts/pages/authentificationForm.jsp), nous remarquons qu'il y a plusieurs erreurs. En effet, le formulaire Struts n'est pas complet, nous avons précisé qu'avec le modèle MVC II, nous devons associer à chaque page JSP un JavaBean ActionForm pour faire le lien avec les propriétés du formulaire de la page (identifiant et motdepasse dans l'exemple).

2. JavaBean ActionForm

Un objet *ActionForm* est un JavaBean de présentation inclus dans une page JSP. Ce JavaBean est une classe qui hérite de la classe *ActionForm* de Struts. Lors de la gestion du formulaire, le JavaBean associé est créé à partir de cette classe. Dans notre exemple, la page *authentificationForm.jsp* doit donc être accompagnée d'une classe JavaBean *AuthentificationForm*.

Nous allons créer cette nouvelle classe dans un paquetage nommé *actionform*.

```
package actionform;
import org.apache.struts.action.ActionForm;

@SuppressWarnings("serial")
public class AuthentificationForm extends ActionForm {

    private String identifiant=null;
    private String motdepasse=null;

    public String getIdentifiant() {
        return identifiant;
    }
    public void setIdentifiant(String identifiant) {
        this.identifiant = identifiant;
    }
    public String getMotdepasse() {
        return motdepasse;
    }
    public void setMotdepasse(String motdepasse) {
        this.motdepasse = motdepasse;
    }
}
```

Cette classe permet d'écrire et lire les informations entrées par l'utilisateur et de les envoyer par la suite vers l'action qui va s'occuper du traitement.

3. Le contrôleur Action

La vue est créée en conformité avec le modèle MVC II. Nous pouvons maintenant développer une action associée au contrôleur. Pour rappel, dans MVC II, il existe un seul contrôleur qui est toujours une Servlet et qui se nomme avec Struts *ActionServlet*. Les développeurs Struts ont déjà développé cette Servlet, elle est d'ailleurs livrée en standard dans le paquetage *org.apache.struts.action* de la librairie *struts.jar*.

Nous allons maintenant programmer nos actions avec une nouvelle classe qui hérite de la classe *Action* de Struts. La classe *Action* définit plusieurs méthodes, la plus utilisée étant *execute()*. Nous allons créer la classe d'action nommée *AuthentificationAction* présente dans un paquetage nommé *action*.

```
package action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import actionform.AuthentificationForm;

@SuppressWarnings("serial")
public class AuthentificationAction extends Action {

    public ActionForward execute(ActionMapping mapping,
ActionForm form, HttpServletRequest request,
HttpServletResponse response) throws Exception
```

```
    {
        //lire le JavaBean de la vue
        AuthentificationForm authentificationform=(AuthentificationForm)
form;
        //vérifier les saisies (identifiant=jlafoss et motdepasse=jerome)
        if(authentificationform.getIdentifiant().equals("jlafoss") &&
authentificationform.getMotdepasse().equals("jerome"))
        {
            //redirection vers la page de succes
            return mapping.findForward("succes");
        }
        //dans tous les autres cas, retour vers la page
d'erreurs
        return mapping.findForward("erreurs");
    }
}
```

Le code ci-dessus permet de vérifier les saisies de l'utilisateur à partir du JavaBean *AuthentificationForm* qui est automatiquement renseigné. En cas de succès de l'authentification l'utilisateur est redirigé vers la page adaptée */pages/succes.jsp* et en cas d'erreur les informations sont affichées par la page */pages/erreurs.jsp*.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head>
<title>Erreurs</title>
<html:base/>
</head>
<body>
<h3>Erreur lors de l'authentification</h3>
</body>
</html:html>
```

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head>
<title>Succès</title>
<html:base/>
</head>
<body>
<h3>Authentification réalisée avec succès</h3>
</body>
</html:html>
```

La méthode *execute()* de la classe *AuthentificationAction* possède quatre paramètres :

- *ActionMapping mapping* : cet élément permet de gérer le routage de l'utilisateur par l'intermédiaire du fichier *struts-config.xml*. Les redirections seront d'ailleurs réalisées avec la fonction *findForward()* de l'objet *mapping*.
- *ActionForm form* : cet objet représente les valeurs du formulaire par l'intermédiaire du JavaBean associé.
- *HttpServletRequest request* : cet objet représente la requête HTTP dans laquelle nous pouvons récupérer les valeurs (identique à l'utilisation des Servlets).
- *HttpServletResponse response* : cet objet représente le flux de sortie de l'application dans lequel nous pouvons envoyer des informations (identique à l'utilisation des Servlets).

Notre premier exemple simple est quasiment terminé. Nous avons défini la vue avec la page */pages/authentificationForm.jsp* et son JavaBean associé *actionform.AuthentificationForm.java*. Ensuite, nous avons développé l'action qui sera réalisée *action.AuthentificationAction.java*. Maintenant, il ne reste plus qu'à assembler tous ces éléments par l'intermédiaire du fichier *struts-config.xml*.

4. Le fichier de configuration struts-config.xml

Le descripteur de déploiement *web.xml* utilisé avec les Servlets et pages JSP est toujours présent avec Struts mais il n'y a par contre plus qu'une seule Servlet déclarée dans ce fichier, la Servlet contrôleur *ActionServlet*.

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Struts Blank Application</display-name>
  <!-- Standard Action Servlet Configuration -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
 </servlet>
  <!-- Standard Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  <!-- The Usual Welcome File List -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Toutes les actions d'extension *.do* seront traitées par la Servlet *ActionServlet* de Struts qui est chargée au démarrage (*<load-on-start-up>*) de l'application et qui possède un fichier de configuration */WEB-INF/struts-config.xml*. Le fichier *struts-config.xml* est un fichier de configuration d'application Struts au format XML. C'est également un fichier d'assemblage qui permet de faire le lien entre les *ActionForm*, les classes Action et les vues.

Nous allons réaliser la configuration de notre application par l'intermédiaire de ce fichier.

- La balise *<form-bean>* permet de déclarer un objet JavaBean de type *ActionForm* présent dans la vue. Les attributs *name* et *type* permettent de donner un nom au JavaBean et son type associé.

- Les balises *<action-mappings>* et *<action>* permettent de déclarer les objets contrôleurs de type *Action*. L'attribut *name* permet de faire le lien avec l'*ActionForm* de la vue. L'attribut *scope* permet d'indiquer où récupérer l'objet *ActionForm*. L'attribut *path* est utilisé pour faire le lien entre le formulaire et la classe *Action* à déclencher.

Si par exemple le formulaire contient la valeur suivante : *<form method="post" action="Authentification.do">*, l'attribut *path* aura la valeur suivante *path="/Authentification"*. L'attribut *type* permet de faire le lien avec la classe associée à l'action. Enfin, la balise *<action>* peut englober des balises *<forward>* qui correspondent aux redirections utilisées par la classe *Action*. L'attribut *name* permet de donner un nom à la redirection.

La balise *<form-bean>* permet de définir les formulaires présents dans les vues JSP. L'attribut *type* de cette balise permet de donner le type lié à l'attribut *name*. Dans notre exemple le type est une classe Java, mais cela peut très bien être un type simple comme un *String*, *String[]*, *int*, ou encore *float*.

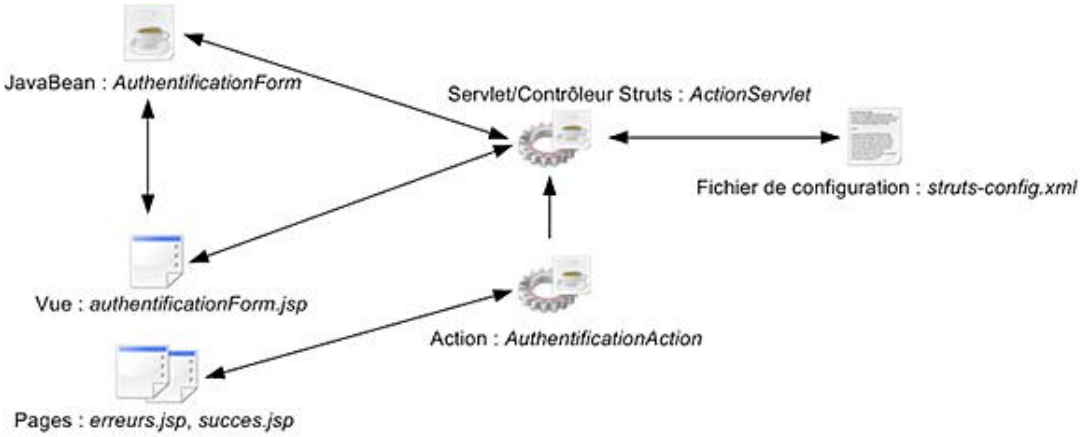
Voici le fichier de configuration *struts-config.xml* utilisé pour notre exemple simple d'authentification.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
<!-- ============================================ Form
Bean Definitions -->
    <form-beans>
        <!--  bean d'authentification -->
```

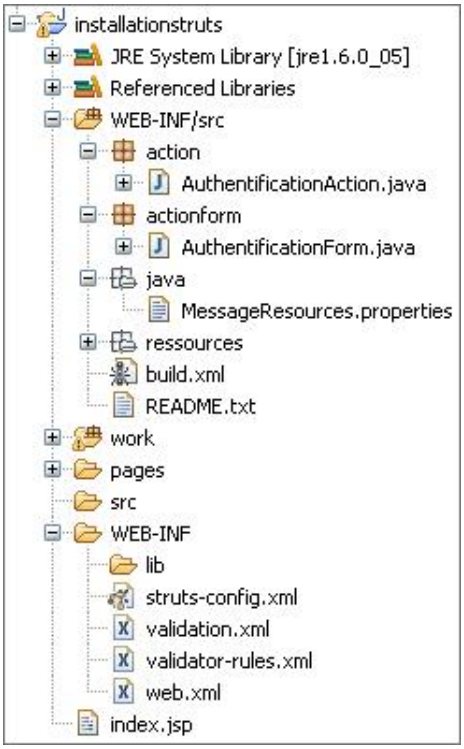
```
        <form-bean
        name="authentificationForm"
        type="actionform.AuthentificationForm"/>
    </form-beans>

<!-- ========================================== Global
Exception Definitions -->
    <global-exceptions>
        <!-- sample exception handler
        <exception
            key="expired.password"
            type="app.ExpiredPasswordException"
            path="/changePassword.jsp"/>
        end sample -->
    </global-exceptions>
<!-- ========================================== Global
Forward Definitions -->
    <global-forwards>
        <!-- Default forward to "Welcome" action -->
        <!-- Demonstrates using index.jsp to forward -->
        <forward
            name="welcome"
            path="/Welcome.do"/>
    </global-forwards>
<!-- ========================================== Action Mapping
Definitions -->
    <action-mappings>

        <action
            path="/Welcome"
            forward="/pages/Welcome.jsp"/>

    <action
         path="/Authentification"
         type="action.AuthentificationAction"
         name="authentificationForm"
         scope="request"
         input="/pages/authentification.jsp">
     <forward name="erreurs" path="/pages/erreurs.jsp"/>
     <forward name="succes" path="/pages/succes.jsp"/>
    </action>

    </action-mappings>
<!-- ======================================== Message
Resources Definitions -->
  <message-resources parameter="java.MessageResources" />
<!-- ================================================ Validator plugin -->
  <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames"
    value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
  </plug-in>

</struts-config>
```

La mise en place de notre premier exemple est terminée. Nous pouvons recharger le contexte de l'application avec le manager Tomcat et tester le service. Pour résumer, lors de la mise en place d'un nouveau service, il est nécessaire de créer les différents éléments d'une application Struts, à savoir une classe de type *ActionForm* pour les pages JSP avec formulaire, une classe type *Action* pour gérer l'action du contrôleur, une classe modèle supplémentaire pour la gestion éventuelle de la persistance des données et enfin des pages JSP de résultat. Tout ceci est lié et assemblé par un seul et unique fichier de configuration nommé *struts-config.xml*.

Notre schéma précédent concernant le modèle MVC II adapté à notre exemple est le suivant :



L'arborescence du projet est la suivante :



Vues et Struts

1. Présentation

Les applications Web actuelles bénéficient d'interfaces graphiques très évoluées. L'interface graphique encore appelée IHM ou GUI, permet d'utiliser l'application par le biais de zones de textes, formulaires, boutons, calendriers ou tout autre objet graphique qui permet de réaliser des actions. Tous ces objets font partie de la Vue utilisateur.

Pour les développements Web, la vue est représentée par des pages HTML/XHTML. Ces pages sont également composées de code JavaScript pour enrichir l'ensemble. Avec le framework Struts, les vues sont également représentées avec du code HTML/XHTML, le langage JavaScript, mais également un ensemble de balises/taglibs dédiées. Ces balises sont fournies sous forme de taglibs et répondent aux besoins des programmeurs d'applications Internet. Les taglibs sont interprétées par le serveur et traduites du côté serveur avant affichage. L'intérêt de cette technique est de réaliser des pages/vues simples avec un minimum de code Java.

La vue en Struts est composée d'une page JSP et d'un JavaBean instance de la classe *ActionForm*. Cette classe permet de représenter les informations saisies par un utilisateur dans un formulaire HTML contenu dans la page JSP. Il est également possible de ne pas créer un JavaBean instance de la classe *ActionForm* mais d'utiliser un outil dynamique grâce aux *DynaForms* qui permettent de laisser le serveur Java créer le JavaBean à partir d'informations renseignées dans le fichier de configuration *struts-config.xml*.

Il existe enfin un fichier de propriétés qui permet de gérer des informations additionnelles et l'internationalisation. Il sera possible d'utiliser les taglibs associés pour gérer des propriétés, labels de champs, textes informatifs, traductions de menus...

2. Les taglibs Struts

Nous avons étudié dans l'exemple précédent les trois taglibs Struts les plus utilisées. Les bibliothèques de tags Struts sont au nombre de cinq :

- *struts-bean.tld*
- *struts-html.tld*
- *struts-logic.tld*
- *struts-nested.tld*
- *struts-tiles.tld*

Les déclarations des taglibs sont réalisées avec les lignes suivantes :

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<%@ taglib uri="http://struts.apache.org/tags-nested" prefix="nested" %>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
```

La documentation des bibliothèques de tags Struts est présente à cette adresse : http://struts.apache.org/1.x/struts-taglib/index.html

---

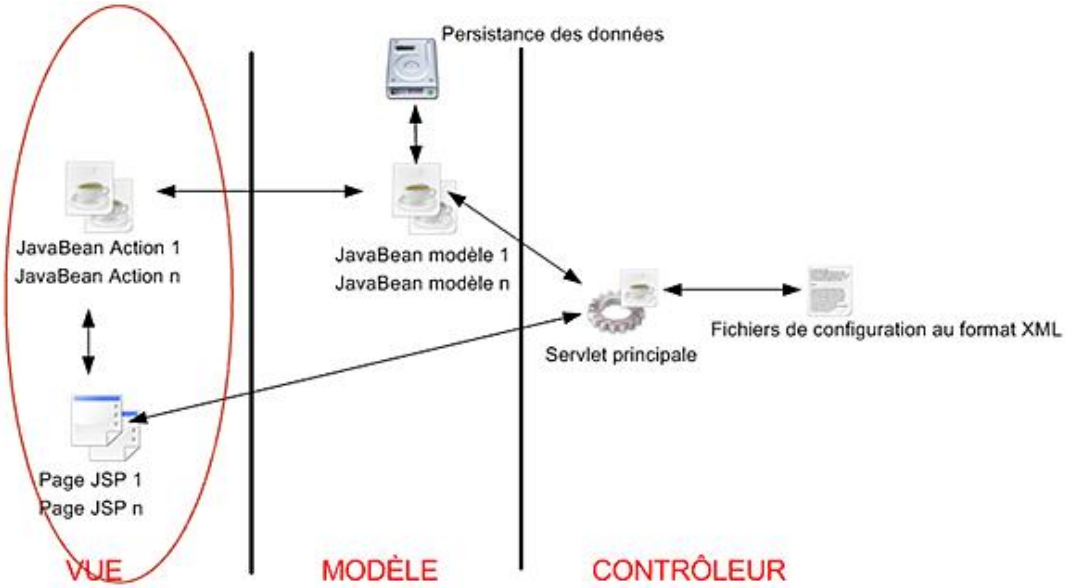
Il est tout à fait possible de changer les préfixes des balises Struts. Mais par convention, il est préférable de conserver les noms proposés. Ex : *<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="programmation" %>*. *<programmation:notEmpty name="monbean" scope="request"></programmation:notEmpty>*. Il n'est également pas nécessaire de déclarer les taglibs qui ne sont pas utilisées dans une page JSP.

Si nous reprenons le formulaire d'authentification précédent, nous remarquons l'utilisation de la bibliothèque de tags *html*. Seuls les tags *<html:...>* sont utilisés dans cet exemple, le code aurait donc pu être simplifié en supprimant les lignes *<%@ taglib...>* avec *bean* et *logic*.

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<html:html>
```

```
<head><title>Authentification</title><html:base/></head>
<body>
<html:form action="/Authentification">
Identifiant : <html:text property="identifiant"/><br/>
Mot de passe <html:text property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

Nous allons à cette étape de ce chapitre, étudier la partie VUE du modèle MVC II.



3. La classe de gestion des formulaires (ActionForm)

Avec le modèle MVC II, la vue est composée de deux éléments :

- La page JSP qui contient du code HTML, des balises Struts et des formulaires de données.
- Des classes qui accompagnent chaque formulaire dans les pages JSP. Ces classes JavaBean héritent de la classe Struts : *org.apache.struts.action.ActionForm*.

La classe abstraite *ActionForm* déclare neuf méthodes qu'il est possible de surcharger :

- *getServlet()* : retourne l'instance de la servlet attachée.
- *getServletWrapper()* : retourne l'instance du contrôleur attaché.
- *getMultipartRequestHandler()* : retourne la requête du formulaire.
- *setServlet(ActionServlet servlet)* : retourne l'instance de la servlet attachée.
- *setMultipartRequestHandler(MultipartRequestHandler multipartRequestHandler)* : utilisé pour les requêtes *Multipoint* comme l'upload de fichiers.
- *reset(ActionMapping mapping, ServletRequest request)* : vide tous les JavaBeans.
- *reset(ActionMapping mapping, HttpServletRequest request)* : vide les propriétés du JavaBean de la requête.
- *validate(ActionMapping mapping, ServletRequest request)* : valide les propriétés du formulaire.

- *validate(ActionMapping mapping, HttpServletRequest request)* : valide les propriétés du formulaire de la requête.

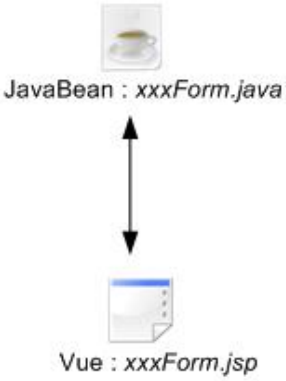
Mise en place d'un formulaire

Toute classe qui hérite d'ActionForm est un JavaBean qui correspond aux champs d'un formulaire HTML Struts. Dans notre exemple précédent d'authentification, le formulaire *authentificationForm.jsp* illustre ce propos.

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<html:html>
<head>
<title>Authentification</title>
<html:base/>
</head>
<body>
<html:form action="/Authentification">
Identifiant : <html:text property="identifiant"/><br/>
Mot de passe <html:text property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

Cette page JSP déclare bien les taglibs Struts afin d'utiliser les balises adaptées. Par convention, pour ce formulaire, il faut développer une classe JavaBean nommée *authentificationForm.java*, dans laquelle un attribut est déclaré pour chaque champ présent dans le formulaire JSP. Ces attributs sont déclarés comme privés ou protégés (*private* ou *protected*) afin de respecter l'encapsulation des JavaBeans. Ce terme usuel sera alors Bean ou JavaBean de formulaire. La classe JavaBean formulaire pour la page JSP précédente est présentée ci-après.

```
package actionform;
import org.apache.struts.action.ActionForm;

@SuppressWarnings("serial")
public class AuthentificationForm extends ActionForm {

    private String identifiant=null;
    private String motdepasse=null;

    public String getIdentifiant() {
        return identifiant;
    }
    public void setIdentifiant(String identifiant) {
        this.identifiant = identifiant;
    }
    public String getMotdepasse() {
        return motdepasse;
    }
    public void setMotdepasse(String motdepasse) {
        this.motdepasse = motdepasse;
    }
}
```



Lorsque la page JSP *authentificationForm.jsp* est renseignée et soumise au serveur, celui-ci crée une instance de la classe *AuthentificationForm.java* à partir des valeurs saisies dans le formulaire de la page JSP. L'instance est référencée par l'intermédiaire du fichier de configuration *struts-config.xml*. Cette référence réalisée avec les balises

<form-beans> et *<form-bean>* est ensuite utilisée par différents éléments comme *ActionServlet* pour les traitements à réaliser.

Le fichier *struts-config.xml* comporte plusieurs balises réservées à la déclaration d'une instance d'*ActionForm*. La balise *<form-beans>* permet de déclarer des JavaBeans formulaires et la balise *<form-bean>* permet de déclarer un JavaBean spécifique d'un formulaire. L'élément *<form-bean>* comporte l'attribut *name* qui permet d'identifier le JavaBean et l'attribut *type* qui définit le type d'instanciation du JavaBean. L'élément *<form-bean>* peut également contenir des éléments *<form-property>* pour les *DynaForms*.

```
...
<form-beans>
    <!--  bean d'authentification -->
    <form-bean
    name="authentificationForm"
    type="actionform.AuthentificationForm"/>
</form-beans>
...
```

Dans l'exemple précédent, le Bean formulaire est identifié par le nom *authentificationForm*. Il est instancié à partir de la classe *AuthentificationForm* présente dans le paquetage *actionform*. Une fois le formulaire JSP déclaré et le JavaBean formulaire mis en place, nous pouvons utiliser l'instance du JavaBean formulaire dans la Servlet d'action *ActionServlet*.

Pour utiliser le JavaBean formulaire, nous devons créer une classe d'action. Il existe plusieurs classes d'action avec Struts (classe d'action simple : *org.apache.struts.actions.Action*, classe de redirection : *org.apache.struts.actions.ForwardAction*...). De même, cette classe d'action est déclarée dans le fichier de configuration *struts-config.xml*.

```
package action;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import actionform.AuthentificationForm;

@SuppressWarnings("serial")
public class AuthentificationAction extends Action {

    public ActionForward execute(ActionMapping mapping,
ActionForm form, HttpServletRequest request,
HttpServletResponse response) throws Exception
    {
        //lire le JavaBean de la vue
        AuthentificationForm authentificationform=(AuthentificationForm)
form;
        //vérifier les saisies (identifiant=jlafoss et motdepasse=jerome)
        if(authentificationform.getIdentifiant().equals("jlafoss") &&
authentificationform.getMotdepasse().equals("jerome"))
        {
            //redirection vers la page de succes
            return mapping.findForward("succes");
        }
        //dans tous les autres cas, retour vers la page d'erreurs
        return mapping.findForward("erreurs");
    }

}
```

```
...
<action-mappings>
    <action
        path="/Authentification"
        type="action.AuthentificationAction"
        name="authentificationForm"
        scope="request"
        input="/pages/authentification.jsp">
     <forward name="erreurs" path="/pages/erreurs.jsp"/>
     <forward name="succes" path="/pages/succes.jsp"/>
    </action>
</action-mappings>
```

```
...
```

Pour résumer, l'utilisation d'un formulaire simple avec Struts nécessite les étapes suivantes :

- Déclaration des librairies de tags Struts.
- Création du formulaire dans la page JSP nommée *xxxForm.jsp*.
- Création de la classe JavaBean associée au formulaire *xxxForm.java*.
- Déclaration du JavaBean de formulaire dans le fichier de configuration *struts-config.xml*.
- Création (au besoin) d'une classe d'action pour les traitements *xxxAction.java*.
- Déclaration de l'action dans le fichier de configuration *struts-config.xml*.
- Tests de la mise en place totale.

4. Les formulaires dynamiques (DynaForms)

La méthode précédente de création de formulaire est fiable mais assez lourde à mettre en place. La création d'un JavaBean associé à chaque formulaire est assez contraignante. Il existe une autre méthode très rapide pour créer un JavaBean formulaire sans taper une seule ligne de code Java mais en déclarant celui-ci dans le fichier *struts-config.xml*.

La balise *<form-bean>* du fichier de configuration *struts-config.xml* est constituée d'un élément *<form-property>* qui permet la création de propriétés aidant à la création de JavaBean formulaire dynamique. Cette balise *<form-property>* possède les attributs *name* (identifiant de la propriété), *type* (type de la propriété) et *initial* (valeur par défaut de la propriété). Les DynaForms sont créés à partir de la classe *DynaActionForm* qui est contenue dans le paquetage *org.apache.struts.action*. Lorsque le serveur, au moment de l'analyse du fichier *struts-config.xml* rencontre la balise *<form-bean>*, il crée une classe JavaBean dynamiquement avec les propriétés déclarées et une instance directement utilisable.

Nous allons reprendre notre exemple d'authentification avec l'utilisation d'un *DynaForms*. Pour cela nous commençons par créer une page JSP nommée *authentificationDynaforms.jsp* avec le code suivant :

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head>
<title>Authentification DynaForms</title>
<html:base/>
</head>
<body>
<html:form action="/AuthentificationDynaForms">
Identifiant : <html:text property="identifiant"/><br/>
Mot de passe <html:text property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

Si nous essayons de déclencher cette page JSP, le navigateur nous indique l'absence de mapping pour le formulaire à l'adresse */AuthentificationDynaForms*. Nous devons donc définir cette action dans le fichier de configuration *struts-config.xml*.

```
...
<action-mappings>
        <action
            path="/AuthentificationDynaForms"
            name="authentificationDynaForms"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/pages/bienvenueDynaForms.jsp">
        </action>
</action-mappings>
```

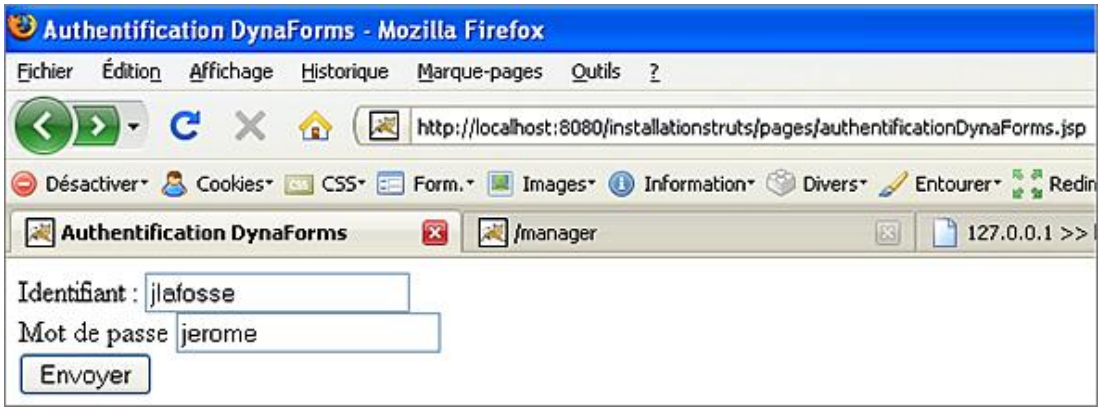
```
...
```

Nous indiquons que le nom du formulaire dynamique sera *authentificationDynaForms* et que le chemin qui permet de déclencher cette action est */AuthentificationDynaForms*. L'action liée à cette URL sera la classe *org.apache.struts.actions.ForwardAction* de Struts. La redirection est réalisée vers la page */pages/bienvenueDynaForms.jsp*. Si nous déclenchons à nouveau l'URL http://localhost:8080/installationstruts/pages/authentificationDynaForms.jsp, après rafraîchissement de l'application par l'intermédiaire du manager Tomcat, nous remarquons une erreur liée au JavaBean formulaire *authentificationDynaForms* qui n'existe pas. Nous allons donc créer ce formulaire dynamique dans le fichier de configuration *struts-config.xml*.

```
...
<form-beans>
      <!--  bean d'authentification dynaforms -->
      <form-bean name="authentificationDynaForms"
type="org.apache.struts.action.DynaActionForm">
            <form-property name="identifiant"
type="java.lang.String" initial=""/>
            <form-property name="motdepasse"
type="java.lang.String" initial=""/>
      </form-bean>
</form-beans>
...
```

Le JavaBean formulaire nommé *authentificationDynaForms* est créé et possède les propriétés *identifiant* et *motdepasse*. Une fois la déclaration réalisée, ce JavaBean s'utilise comme un Bean de formulaire classique. Nous pouvons maintenant développer la page de redirection *bienvenueDynaForms.jsp* qui permet uniquement de lire le *DynaForms* dans la requête HTTP.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head><title>Succès</title><html:base/></head>
<body>
<h3>Authentification réalisée avec succès</h3>
<bean:write name="authentificationDynaForms" property="identifiant"/><br/>
<bean:write name="authentificationDynaForms" property="motdepasse"/><br/>
</body>
</html:html>
```



Nous avons manipulé des objets de type chaînes de caractères avec le *DynaForms* précédent. Comme nous pouvons manipuler tous les objets avec cette classe, nous allons créer une classe nommée *Utilisateur* avec les propriétés *identifiant* et *motdepasse*. Pour cela nous allons ajouter un nouveau paquetage nommé *boiteoutils* qui contiendra les classes JavaBean simples. Chaque classe manipulée par un *DynaForms* doit être un JavaBean standard, c'est-à-dire sérialisable.

```
package boiteoutils;
import java.io.Serializable;

public class Utilisateur implements Serializable
{
    //variables de classe
    private String identifiant=null;
    private String motdepasse=null;
    //java 5.0 pour avoir un identifiant unique de la sérialisation
    private static final long serialVersionUID = 1L;
    public Utilisateur()
    {
    }
    public String getIdentifiant() {
    return identifiant;
    }
    public void setIdentifiant(String identifiant) {
    this.identifiant = identifiant;
    }
    public String getMotdepasse() {
    return motdepasse;
    }
    public void setMotdepasse(String motdepasse) {
    this.motdepasse = motdepasse;
    }
//fin de la classe
}
```

Nous allons maintenant créer une page JSP nommée *authentificationDynaFormsClasse.jsp* pour gérer un *DynaForms* avec notre classe *Utilisateur*. Maintenant, si nous démarrons l'application en appelant la page JSP, nous remarquons une erreur liée à la déclaration du *DynaForms* dans le fichier de configuration *struts-config.xml* et au mapping */AuthentificationDynaFormsClasse*.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head>
<title>Authentification Dynaforms</title>
<html:base/>
</head>
<body>
<html:form action="/AuthentificationDynaFormsClasse">
Identifiant : <html:text property="utilisateur.identifiant"/><br/>
Mot de passe <html:text property="utilisateur.motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form></body>
</html:html>
```

Nous allons donc réaliser la déclaration du formulaire et du mapping dans le fichier de configuration Struts.

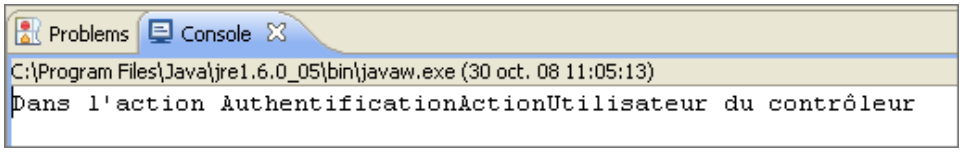
```
...
<form-beans>
    ...
    <!-- ========== Formulaire de gestion des utilisateurs ========== -->
    <form-bean name="utilisateurDynaForms"
type="org.apache.struts.validator.DynaValidatorForm">
        <form-property name="utilisateur" type="boiteoutils.Utilisateur" />
    </form-bean>
    ...
</form-beans>
<action-mappings>
    ...
    <!-- ===action qui permet de gérer les utilisateurs=== -->
        <action
            path="/AuthentificationDynaFormsClasse"
            name="utilisateurDynaForms"
            scope="request"
            input="/pages/authentificationDynaFormsClasse.jsp"
            type="action.AuthentificationActionUtilisateur">
            <forward name="erreurs" path="/pages/erreurs.jsp"/>
            <forward name="succes" path="/pages/succes.jsp"/>
    </action>
    ...
</action-mappings>
...
```

Nous remarquons la définition du formulaire nommé *utilisateurDynaForms* qui est une instance de la classe *org.apache.struts.validator.DynaValidatorForm* et qui possède une propriété nommée *utilisateur* qui est un objet, instance de la classe *boiteoutils.Utilisateur*. Ensuite, nous déclarons le mapping avec l'accès à l'URL */AuthentificationDynaFormsClasse*. Cette URL déclenche l'objet formulaire *utilisateurDynaForms*. Les données possèdent une portée *request* et en cas d'erreur, on déclenche la page */pages/authentificationDynaFormsClasse.jsp*.

Nous allons désormais coder cette classe afin de faire un test sur les saisies de l'utilisateur.

```
package action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import actionform.AuthentificationForm;

@SuppressWarnings("serial")
public class AuthentificationActionUtilisateur extends Action {

    public ActionForward execute(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response) throws Exception
    {
        System.out.println("Dans l'action AuthentificationActionUtilisateur
du contrôleur");
        return null;
    }
}
```

L'action du formulaire est codée simplement dans un premier temps en rapport avec la valeur *path (path="/AuthentificationDynaFormsClasse")* du fichier de configuration *struts-config.xml*.



Nous pouvons maintenant passer à l'étape de vérification de l'authentification utilisateur. La technique est la même qu'avec les *ActionForms* mais il faut remarquer la simplicité d'utilisation de l'ensemble avec les *DynaForms*. Il faut d'abord récupérer le *DynaForms* et réaliser la création de l'objet *utilisateur* directement à partir du *DynaForms*. L'objet *utilisateur* pourra être utilisé normalement. Il faut noter également l'utilisation des accès aux propriétés de la classe *Utilisateur* avec la notation pointée (*utilisateur.identifiant*).

```
package action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
mport org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.DynaActionForm;
import boiteoutils.Utilisateur;

@SuppressWarnings("serial")
public class AuthentificationActionUtilisateur extends Action {

    public ActionForward execute(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response) throws Exception
    {
        //le bean formulaire
        DynaActionForm authentificationdynaform=(DynaActionForm)form;
        //récupérer l'objet du formulaire
        Utilisateur utilisateur=(Utilisateur)
authentificationdynaform.get("utilisateur");

        //vérifier les saisies (identifiant=jlafoss et motdepasse=jerome)
        if(utilisateur.getIdentifiant().equals("jlafoss") &&
utilisateur.getMotdepasse().equals("jerome"))
        {
            redirection vers la page de succes
            return mapping.findForward("succes");
        }
        //dans tous les autres cas, retour vers la page d'erreurs
        return mapping.findForward("erreurs");
    }
}
```

5. Le fichier de propriétés et de langues

Une page JSP contient un ensemble d'éléments en plus du code HTML, comme du texte, des images, des chemins d'accès pour les liens... Dans la plupart des cas, le texte est statique et inséré en dur dans les pages. L'utilisation de

texte statique dans une page JSP ne permet pas l'évolution rapide d'une application. De même, si l'application est accessible dans plusieurs langues, il est intéressant d'utiliser un outil adapté à ces besoins.

Les technologies Java utilisent des fichiers texte pour gérer les propriétés et langues des applications. Un fichier de propriétés est un simple fichier texte portant l'extension *.properties*. Ce fichier permet de déclarer des constantes utilisables dans toute l'application. Il est possible d'utiliser plusieurs fichiers de configuration pour gérer les propriétés et la localisation (langues).

Ce ou ces fichiers de propriétés doivent être placés dans un paquetage spécifique (ex : *ressources*). Ensuite, on utilise la balise *<message-resources>* du fichier de configuration Struts pour gérer ces fichiers.

```
<message-resources parameter="java.MessageResources" />
```

Dans cet exemple, le fichier de propriétés se nomme *MessageResources* et se trouve dans le paquetage *java*. Une utilisation plus courante serait alors :

```
<message-resources parameter="ressources.MessageResources" />
```

On remarque que l'extension *.properties* n'est pas déclarée dans l'attribut *parameter*.

Une fois le fichier de propriétés créé, nous pouvons insérer des informations sous forme de constantes avec le format :

```
nomconstante=valeur constante
```

La première partie est l'identifiant de la constante, elle ne doit pas contenir d'espace. La partie droite de l'expression correspond à la valeur. Dans cette partie, il est possible d'utiliser tous les types de caractères, même les espaces et caractères spéciaux. Par convention des termes explicites séparés par des points sont utilisés.

```
utilisateur.labelpseudonyme=Pseudonyme de l'utilisateur
```

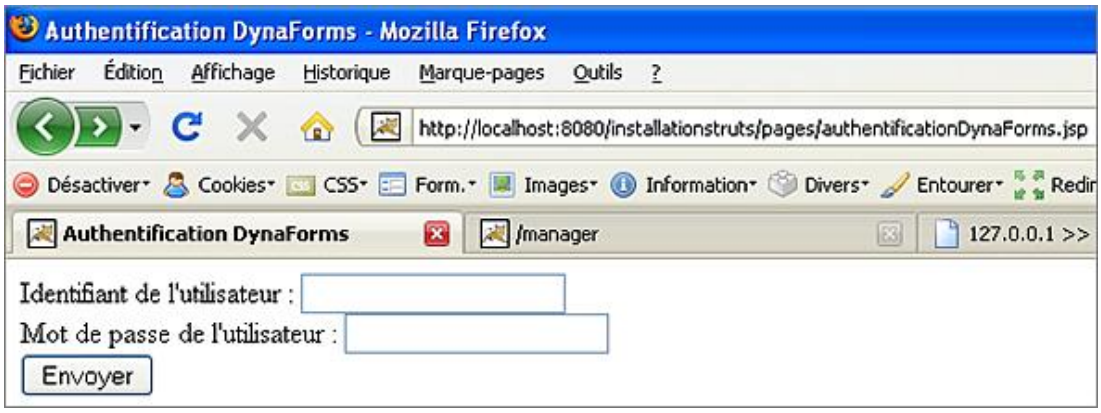
Il est vivement conseillé d'organiser ce fichier de propriétés par groupe de messages communs afin de retrouver rapidement les informations. Une fois renseignées, les propriétés sont accessibles par toutes les pages JSP (grâce aux tags Struts et JSTL) mais aussi par les Servlets.

Pour la localisation (les langues), le tag *<bean:message>* permet de récupérer la valeur d'une propriété et de l'afficher dans une page JSP. Ce tag comporte un attribut nommé *key* qui permet de faire référence à un identifiant du fichier de propriétés *MessageResources.properties*.

Si nous reprenons l'exemple de la page JSP d'authentification avec le DynaForms *authentificationDynaForms.jsp*, nous allons insérer les informations textuelles à partir du fichier de propriétés. Le code source ci-dessous est alors modifié avec les balises Struts.

```
...
<html:form action="/AuthentificationDynaForms">
Identifiant : <html:text property="identifiant"/><br/>
Mot de passe <html:text property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
...

...
<html:form action="/AuthentificationDynaForms">
<bean:message key="utilisateur.identifiant"/><html:text
property="identifiant"/><br/>
<bean:message key="utilisateur.motdepasse"/><html:text
property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
...
```

L'affichage textuel est associé aux informations présentes dans le fichier *MessageResources.properties*.




```
# -- gestion des utilisateurs --
utilisateur.identifiant=Identifiant de l'utilisateur :
utilisateur.motdepasse=Mot de passe de l'utilisateur :
```

Ce principe est applicable pour tout le texte (les images, les sources flash...) présent dans la page JSP, cette dernière sera alors totalement paramétrable à partir du fichier de propriétés. Pour la gestion de l'internationalisation, il ne reste plus qu'à créer autant de fichiers de propriétés que de langues à supporter en respectant toujours les identifiants définis dans les fichiers.

Le fichier d'une langue spécifique doit porter l'extension *_xx.properties* où *xx* identifie les lettres de la langue souhaitée, exemples :

- français : *ressources_fr.properties*
- anglais : *ressources_en.properties*

Le fichier de propriétés peut être suffixé de la locale sous la forme code langue et code pays afin d'identifier un pays, une langue et un dialecte.

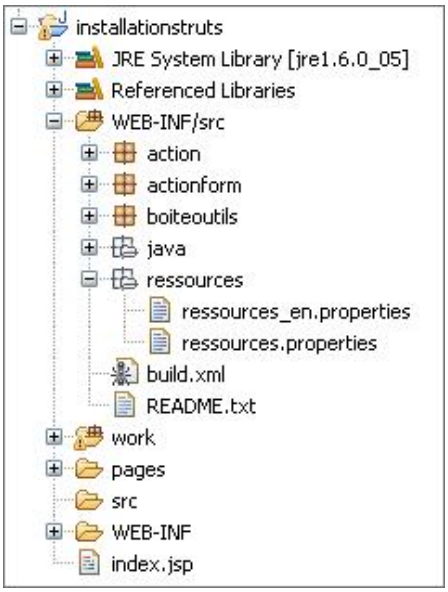
Nous allons définir deux fichiers de propriétés, un en français et un en anglais. Nous allons renommer le fichier *MessageResources.properties* en *ressources.properties*, cela sera le fichier par défaut (langue française). Nous allons également mettre à jour la configuration dans le fichier *struts-config.xml*.

```
...
<!-- ================================== Message Resources Definitions -->
<message-resources parameter="ressources.ressources" />
...
```

Ensuite, nous allons créer un fichier nommé *ressources_en.properties*. Ce fichier contient les mêmes informations mais en anglais.

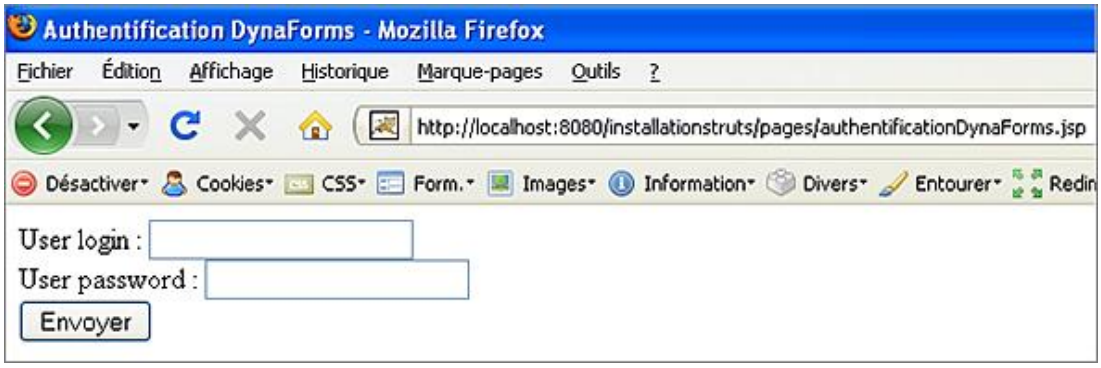
```
# -- gestion des utilisateurs --
utilisateur.identifiant=User login :
utilisateur.motdepasse=User password :
```

Notre arborescence du projet est alors la suivante :



Pour tester l'application, nous pouvons modifier la langue du navigateur (**Outils - Options - Avancé - Général - Langues - Choisir... - Anglais/Etats-Unis - Ajouter**) et sélectionner l'anglais.

Nous pourrons évidemment développer une page spécifique pour passer d'une langue à l'autre par l'intermédiaire de liens en modifiant la locale en programmation.



Les validations et vérifications de données

1. Présentation

Nous avons étudié dans la section précédente les *ActionsForms* et *DynaForms* ainsi que plusieurs balises Struts et le fichier de propriétés. Ces éléments permettent de créer des services d'applications Internet mais rien n'interdit jusqu'ici à un utilisateur de rentrer, par exemple, un email dans un champ date.

Afin d'éviter ce type d'erreur, il est recommandé de prévoir des traitements spécifiques et de prévenir l'utilisateur que le type d'information n'est pas compatible avec la saisie qu'il vient d'effectuer.

Dans les applications Web traditionnelles, il est nécessaire de coder ses propres validations en JavaScript et/ou du côté serveur avec des traitements spécifiques. Avec le modèle MVC I, une page JSP est associée à une Servlet, la validation peut alors représenter un volume de travail considérable si l'application est de taille conséquente.

Struts propose les deux types de validations, client et serveur, en utilisant différentes techniques selon que les composants *ActionForm* ou *DynaActionForm* sont utilisés. La méthode conventionnelle consiste à implémenter la méthode *validate()* dans un Bean de formulaire. La seconde alternative consiste à utiliser le plug-in *Validators* qui met à disposition des programmeurs des règles de validation.

2. Validation par méthode (reset() et validate())

Dans la partie précédente, nous avons utilisé le formulaire d'authentification *authentificationForm.jsp* de type *ActionForm*. Le Bean de formulaire hérite de la classe abstraite *ActionForm*, il peut donc implémenter les méthodes déclarées dans *ActionForm*. Les deux méthodes intéressantes sont *reset()*, pour initialiser des données, et *validate()* pour valider les saisies de l'utilisateur.

a. La méthode reset()

Cette méthode permet d'initialiser les propriétés du Bean formulaire. La signature de la méthode *reset()* est la suivante :

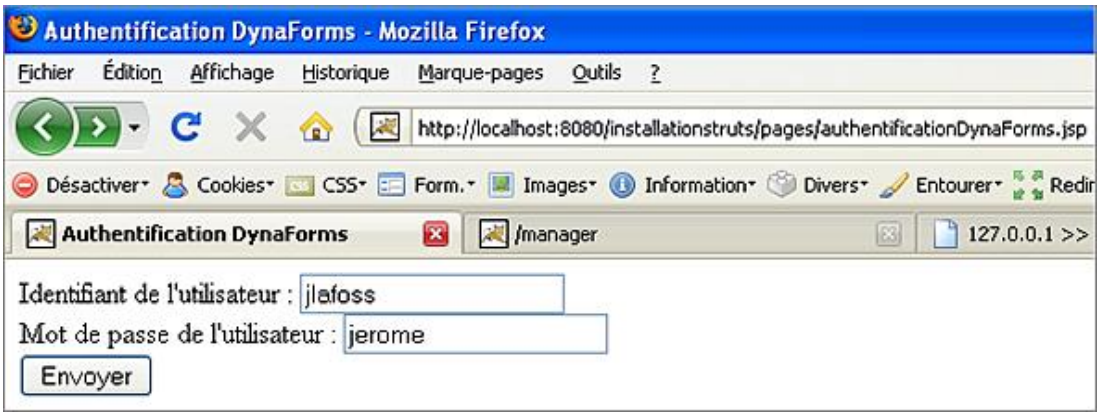
public void reset(ActionMapping mapping, HttpServletRequest request)

- *ActionMapping* : représente l'objet qui permet de récupérer le JavaBean de formulaire.
- *HttpServletRequest* : représente la requête HTTP.

Reprenons notre exemple d'authentification avec un formulaire *ActionForm*. La page JSP nommée *authentificationForm.jsp* est liée avec la classe *actionform.AuthentificationForm*. Nous pouvons donc implémenter la méthode *reset()* dans cette classe pour initialiser des valeurs.

```
package actionform;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

@SuppressWarnings("serial")
public class AuthentificationForm extends ActionForm {
    private String identifiant=null;
    private String motdepasse=null;
    public String getIdentifiant() {
        return identifiant;
    }
...
    public void reset(ActionMapping mapping, HttpServletRequest request)
    {
        this.identifiant="jlafoss";
        this.motdepasse="jerome";
    }
}
```

Maintenant, si nous relançons notre navigateur avec la page d'authentification, nous retrouvons bien les valeurs initialisées.



b. La méthode validate()

La méthode *validate()* est une des méthodes de la classe abstraite *ActionForm* qu'il est possible d'implémenter dans une classe concrète. Cette méthode permet de vérifier la saisie d'un utilisateur dans un champ (texte, liste déroulante...). Si une propriété n'est pas valide, alors le formulaire renvoie une erreur sur une page JSP adaptée. Il est alors possible d'afficher les erreurs avec le tag adapté *<html:errors>*.

public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)

- *ActionMapping* : représente l'objet qui a permis de récupérer le JavaBean de formulaire.
- *HttpServletRequest* : représente la requête HTTP.

La méthode *validate()* renvoie un objet de type *ActionErrors*, c'est une collection de type *java.util.HashMap* qui contient la liste des erreurs.

Pour remplir cette collection d'erreurs, il faut utiliser la méthode *add()*.

public void add(String property, ActionMessage message)

- *String property* : représente une chaîne de caractères qui est la clé d'un message d'erreur associé à la propriété du Bean de formulaire posant problème.
- *ActionMessage message* : représente le message qui est lu dans le fichier de propriétés et qui est renvoyé.

Nous allons modifier notre classe *AuthentificationForm* pour gérer la validation des données.

```
package actionform;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;

@SuppressWarnings("serial")
public class AuthentificationForm extends ActionForm {

    private String identifiant=null;
    private String motdepasse=null;
    public String getIdentifiant() {
        return identifiant;
    }
    ...
    public ActionErrors validate(ActionMapping mapping,
HttpServletRequest request)
    {
        ActionErrors erreurs=new ActionErrors();
        if(this.identifiant==null || this.identifiant.equals("".trim()))
        {
        erreurs.add("identifiant", new ActionMessage
```

```
("erreur.utilisateur.identifiant"));
        }
        if(this.motdepasse==null || this.motdepasse.equals("".trim()))
        {
        erreurs.add("motdepasse", new ActionMessage
("erreur.utilisateur.motdepasse"));
        }
        return erreurs;
    }
}
```

Dans cet exemple, si les attributs *identifiant* et *motdepasse* ne sont pas renseignés, une erreur est renvoyée sur la page JSP. Les attributs sont testés, les valeurs ne doivent pas être *null* ou égales à une chaîne vide. Dans le cas contraire, une erreur est ajoutée dans la collection avec comme clé le nom de l'attribut du JavaBean de formulaire et pour valeur, le message d'erreur renseigné dans le fichier de propriétés.

Si une erreur survient sur le champ *identifiant* par exemple, l'entrée *erreur.utilisateur.identifiant* du fichier de propriétés sera utilisée.

```
# -- gestion des erreurs
erreur.utilisateur.identifiant=Le champ identifiant doit être renseigné
erreur.utilisateur.motdepasse=Le champ mot de passe doit être renseigné
```

Les saisies seront ainsi validées. Il reste désormais à afficher tous les messages d'erreurs dans un même bloc au sein d'une page JSP avec le tag *<html:errors>*. Nous allons également paramétrer le routage dans le fichier de configuration de Struts (*struts-config.xml*).

```
<action-mappings>
    ...
    <action
        path="/Authentification"
        type="action.AuthentificationAction"
        name="authentificationForm"
        scope="request"
        input="/pages/authentificationForm.jsp">
        <forward name="erreurs" path="/pages/erreurs.jsp"/>
        <forward name="succes" path="/pages/succes.jsp"/>
    </action>
    ...
</action-mappings>
```

Dans ce routage, la validation du formulaire par l'utilisateur, déclenche l'URL */Authentification* avec la classe associée *action.AuthentificationAction*. En cas d'erreur dans la classe d'action, c'est la page */pages/authentificationForm.jsp* (paramètre *input*) qui est appelée. Donc si la méthode *validate()* déclenche une erreur, c'est la page d'authentification qui est affichée. Nous pouvons donc ajouter dans cette page la balise d'affichage de toutes les erreurs *<html:errors/>*.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head>
<title>Authentification</title>
<html:base/>
</head>
<body>
<html:errors/>
<html:form action="/Authentification">
Identifiant : <html:text property="identifiant"/><br/>
Mot de passe <html:text property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

Maintenant, nous pouvons utiliser la propriété *property* pour indiquer d'afficher un message spécifique.

```
...
Identifiant : <html:text property="identifiant"/><html:errors
property="identifiant"/><br/>
...
```

Nous remarquons que les erreurs sont affichées sous la forme d'une liste à puces. Cela provient des entrées suivantes dans le fichier de propriétés qui englobent chaque message d'erreur.

```
...
# -- standard errors --
errors.header=<UL>
errors.prefix=<LI>
errors.suffix=</LI>
errors.footer=</UL>
...
```

En effet, la balise <*html:error*> possède quatre attributs intéressants :

- *header* : permet d'identifier la clé à utiliser pour formater l'en-tête du message d'erreur.
- *footer* : permet d'identifier la clé du fichier de propriétés à utiliser pour formater le pied du message d'erreur.
- *prefix* : permet d'identifier la clé du fichier de propriétés à utiliser pour formater le préfixe de chaque message d'erreur.
- *suffix* : permet d'identifier la clé du fichier de propriétés à utiliser pour formater le suffixe de chaque message d'erreur.

La syntaxe est la suivante : *<html:errors property="nompropriété" header="erreurs.entete" footer="erreurs.pied" prefix="erreurs.nomprefixe" suffix="erreurs.nomsuffixe"/>*.

3. Les Validators

La méthode *validate()* présentée précédemment permet de gérer les validations d'entrées dans une classe *ActionForm*. Il existe une seconde méthode pour valider les entrées à partir de fichiers XML. Pour mettre en place les validations ou Validators, il faut utiliser le plug-in Struts qui permet d'utiliser deux fichiers XML qui sont *validation.xml* et *validator-rules.xml*.

Tout d'abord, la déclaration du plug-in s'effectue à la fin du fichier *struts-config.xml* :

```
...
<!-- ========================================================= Validator plugin -->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
   <set-property property="pathnames" value="/WEB-
INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
...
```

Dans cette déclaration, nous indiquons que les deux fichiers de gestion des Validators sont présents dans le répertoire */WEB-INF* sous les noms suivants : *validator-rules.xml* et *validation.xml*. Il faut ensuite déclarer nos validations dans le fichier *validation.xml* et utiliser les validations Struts déclarées dans le fichier *validator-rules.xml*.

Le fichier de déclaration des validations (*validation.xml*) possède la structure suivante :

```
<!DOCTYPE form-validation PUBLIC"-//Apache Software
Foundation//DTD Commons Validator Rules Configuration 1.1.3//EN"
"http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">
<form-validation>
....
</form-validation>
```

La balise <*form-validation*> contient les éléments qui sont utilisés pour réaliser les validations dans un formulaire. Les éléments que l'on peut implémenter dans ce fichier sont les suivants :

- <*global*> : permet de déclarer des constantes globales au fichier *validation.xml*.
- <*constant*> : permet de déclarer des constantes avec les sous-éléments <*constant-name*> et <*constant-value*>.

Nous pouvons alors déclarer des constantes globales à base d'expressions régulières pour gérer la syntaxe des numéros de téléphone, des mots de passe, des sites Web...

```
...
<global>
    <constant>
        <constant-name>entierpositif</constant-name>
        <constant-value>^\d+$</constant-value>
    </constant>
    <constant>
        <constant-name>codepostal</constant-name>
        <constant-value>^[0-9a-zA-Z]{5}$</constant-value>
    </constant>
</global>
...
```

Pour valider un JavaBean de formulaire, il faut déclarer le formulaire et les propriétés à valider dans le fichier *validation.xml* à partir de la balise <*formset*> qui contient des sous-éléments. La balise <*formset*> possède les sous-éléments suivants :

- <*form name=''nomduformulaire''*> : permet d'identifier le JavaBean de formulaire.
- <*field property=''nomduchamp''/*> : permet d'identifier le nom du champ à valider.
- <*arg key=''errors.nomdelacle''/*> : permet d'identifier une clé provenant du fichier de propriétés afin de définir le texte à afficher dans le message d'erreur.
- <*var name=''nomdelavariable''/*> : permet d'identifier les variables pour les masques de validation.

Nous allons mettre en œuvre l'application des validations à partir de notre exemple d'authentification. Pour rappel, le fichier *struts-config.xml* contient la déclaration du formulaire sous la forme suivante :

```
...
<!--  bean d'authentification actionform -->
<form-bean  name="authentificationForm"
type="actionform.AuthentificationForm"/>
...
```

Le formulaire d'authentification est donc nommé *authentificationForm*. Nous pouvons alors déclarer ses règles de validations dans le fichier *validation.xml* en réalisant une référence à ce nom.

```
...
<formset>

    <!-- ========== Formulaire authentification utilisateur ========== -->
    <form name="authentificationForm">
        field property="identifiant" depends="required,minlength,
maxlength,mask">
            <arg0 key="identifiantutilisateur"/>
            <arg1 name="minlength" key="${var:minlength}" resource="false"/>
            <arg1 name="maxlength" key="${var:maxlength}" resource="false"/>
            <arg3 key="identifiantutilisateur"/>
            <var>
                <var-name>minlength</var-name>
                <var-value>4</var-value>
            </var>
            <var>
                <var-name>maxlength</var-name>
                <var-value>7</var-value>
            </var>
            <var>
                <var-name>mask</var-name>
                <var-value>^[a-zA-Z]{1}[a-zA-Z]*$</var-value>
            </var>
        </field>
    </form>

</formset>
```

```
...
```

Nous déclarons notre formulaire par l'intermédiaire de la balise *<form/>*. Ensuite, nous précisons les règles pour le champ *identifiant*. L'attribut *depends* de l'élément *field* permet de déterminer quelles sont les validations à réaliser. L'attribut *depends* peut avoir plusieurs valeurs qui sont des références à des méthodes déclarées dans le fichier *validator-rules.xml*.

Struts propose déjà plusieurs règles de validations que nous pouvons utiliser directement sans ajouter de déclaration personnalisée.

- *required* : permet de vérifier si le champ n'est pas *null* ou vide.
- *validwhen* : permet de vérifier si un champ a la même valeur qu'un autre.
- *maxlength* : permet de vérifier la taille maximum d'un champ.
- *minlength* : permet de vérifier la taille minimum d'un champ.
- *mask* : permet de vérifier si la valeur respecte une expression régulière personnelle.
- *byte* : permet de vérifier si la valeur peut être convertie en type *Byte*.
- *short* : permet de vérifier si la valeur peut être convertie en type *Short*.
- *integer* : permet de vérifier si la valeur peut être convertie en type *Integer*.
- *long* : permet de vérifier si la valeur peut être convertie en type *Long*.
- *float* : permet de vérifier si la valeur peut être convertie en type *Float*.
- *double* : permet de vérifier si la valeur peut être convertie en type *Double*.
- *date* : permet de vérifier si la valeur peut être convertie en type *Date*.
- *intRange* : permet de vérifier si la valeur est contenue dans la fourchette de type *Integer*.
- *floatRange* : permet de vérifier si la valeur est contenue dans la fourchette de type *Float*.
- *doubleRange* : permet de vérifier si la valeur est contenue dans la fourchette de type *Double*.
- *creditCard* : permet de vérifier si la valeur respecte le format carte de crédit.
- *email* : permet de vérifier si la valeur respecte le format email.
- *url* : permet de vérifier si la valeur respecte le format des liens Internet ou URL.

Dans notre exemple, la validation précise que la saisie est obligatoire (*required*), la taille mini est de 4 caractères (*minlength*), la taille maxi est de 7 caractères (*maxlength*) et que la saisie doit respecter le masque indiqué par l'expression régulière (*mask*). Chacune des méthodes de validation est associée à un message dans le fichier de propriétés. Cette technique permet de gérer de façon complexe les paramètres et langues.

Reprenons l'exemple de la méthode *required*.

```
errors.required=Attention, le champ [{0}] doit être renseigné !
```

Nous remarquons l'utilisation des accolades avec la valeur zéro *{0}*. En fait, cette déclaration est liée à un attribut *<arg key="..."/>* présent dans le ficher *validation.xml*. Ainsi, le *{0}* est remplacé par la valeur de la clé *identifiantutilisateur* contenue dans le fichier de propriétés.

```
# -- standard errors --
errors.header=<UL>
```

```
errors.prefix=<LI>
errors.suffix=</LI>
errors.footer=</UL>
errors.required=Attention, le champ [{0}] doit être renseigné !
errors.minlength=Attention, le champ [{0}] doit avoir au moins
{1} caractères !
errors.maxlength=Attention, le champ [{0}] ne peut avoir plus de
{1} caractères !
errors.invalid=Attention, le champ [{0}] est incorrect!
errors.date=Attention, le champ [{0}] n'est pas une date valide !
errors.byte=Attention, le champ [{0}] doit être un octet !
errors.date=Attention, le champ [{0}] doit être une date !
errors.double=Attention, le champ [{0}] doit être un double !
errors.float=Attention, le champ [{0}] doit être un réel !
errors.integer=Attention, le champ [{0}] doit être un entier !
errors.long=Attention, le champ [{0}] doit être un entier long !
errors.short=Attention, le champ [{0}] doit être un entier court !
errors.range=Attention, le champ [{0}] doit être dans l'intervalle
{1} et {2} !
errors.creditcard=Attention, le champ [{0}] n'est pas un numéro de
carte valide !
errors.email=Attention, le champ [{0}] n'est pas une adresse
électronique valide !
# -- gestion des utilisateurs --
utilisateur.identifiant=Identifiant :
utilisateur.motdepasse=Mot de passe :
identifiantutilisateur=identifiant
motdepasseutilisateur=mot de passe
# -- gestion des erreurs
erreur.utilisateur.identifiant=Le champ identifiant doit être renseigné
erreur.utilisateur.motdepasse=Le champ mot de passe doit être renseigné
```

Nous pouvons enfin mettre en place les validations à partir des accesseurs pour chaque champ du formulaire en utilisant les *ActionForm*. Cette technique très lourde, nécessite beaucoup de code. Les validations les plus utilisées étant les *Validators* sur les *DynaForms*.

4. Les DynaForms et Validators

Il est très simple de créer des validations complexes avec les formulaires dynamiques Struts. Il suffit de déclarer le formulaire dans le fichier *struts-config.xml* et de créer les *Validators* associés.

Si nous reprenons notre fichier *authentificationDynaForms.jsp*, nous trouvons la déclaration suivante :

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head><title>Authentification Dynaforms</title><html:base/></head>
<body>
<html:errors/>
<html:form action="/AuthentificationDynaForms">
<bean:message key="utilisateur.identifiant"/><html:text
property="identifiant"/><br/>
<bean:message key="utilisateur.motdepasse"/><html:text
property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

Ensuite, nous retrouvons la déclaration associée dans le fichier *struts-config.xml*. Nous remarquons que la classe utilisée n'est plus *org.apache.struts.action.DynaActionForm* mais *org.apache.struts.validator.DynaValidatorForm*.

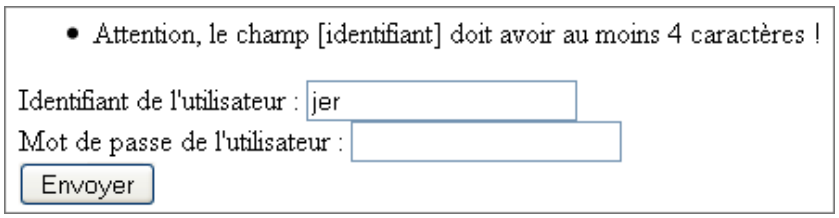
```
...
<form-beans>
<!--  bean d'authentification dynaforms -->
<form-bean name="authentificationDynaForms"
type="org.apache.struts.validator.DynaValidatorForm">
```

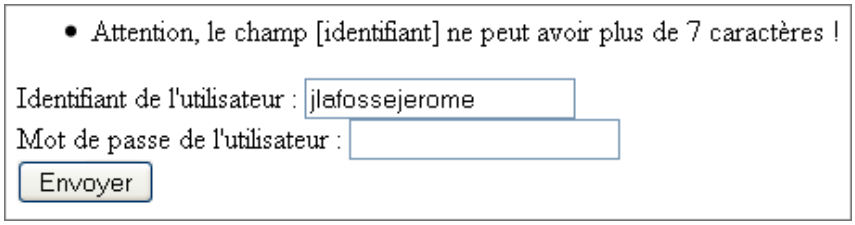
```
    <form-property name="identifiant" type="java.lang.String" initial=""/>
    <form-property name="motdepasse" type="java.lang.String" initial=""/>
</form-bean>
</form-beans>
...
<action
    path="/AuthentificationDynaForms"
    name="authentificationDynaForms"
    validate="true"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/pages/bienvenueDynaForms.jsp"
    input="/pages/authentificationDynaForms.jsp"
    scope="request">
</action>
...
```

Le paramètre *validate* permet de préciser s'il faut vérifier les saisies ou non. Le paramètre *input* est très important, il permet d'indiquer sur quelle page retourner en cas d'erreur. Le nom de notre formulaire est *authentificationDynaForms*, il ne reste plus qu'à créer les validations dans le fichier *validation.xml*.

```
...
<!-- ========== Formulaire authentification utilisateur ========== -->
<form name="authentificationDynaForms">
    <field property="identifiant" depends="required,minlength,
maxlength,mask">
        <arg0 key="identifiantutilisateur"/>
        <arg1 name="minlength" key="${var:minlength}" resource="false"/>
        <arg1 name="maxlength" key="${var:maxlength}" resource="false"/>
        <var>
            <var-name>minlength</var-name>
            <var-value>4</var-value>
        </var>
        <var>
            <var-name>maxlength</var-name>
            <var-value>7</var-value>
        </var>
        <var>
            <var-name>mask</var-name>
           <var-value>^[a-zA-Z]*$</var-value>
        </var>
</field>
</form>
...
```

Dans ce cas, nous indiquons que le champ est obligatoire, qu'il doit contenir une taille mini et maxi et qu'il doit respecter l'expression régulière précisée dans le masque. Le paramètre *<arg0 key="...">* permet de réaliser les remplacements des noms de propriétés. Dans notre cas, la valeur de la clé identifiantutilisateur sera remplacée dans le message d'erreur.





Le formulaire dynamique est opérationnel. Nous venons de mettre en place une vérification très pointue avec quatre syntaxes différentes sur un même champ de formulaire. Ce service permet de mettre en évidence l'utilisation d'un framework pour faciliter la tâche du programmeur. Nous imaginons tout de suite la quantité de travail à réaliser au sein d'une action de la Servlet pour gérer ces vérifications. De même, les règles sont déclarées dans un fichier statique

au format XML. Nous pourrons ainsi modifier une règle sans compiler à nouveau l'application ni tout redéployer.

Nous pouvons enfin terminer par la mise en place de règles sur le mot de passe utilisateur.

```
<!DOCTYPE form-validation PUBLIC
         "-//Apache Software Foundation//DTD Commons Validator Rules
Configuration 1.1.3//EN"
          "http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">

<form-validation>
    <global>
        <constant>
            <constant-name>password</constant-name>
            <constant-value>^[0-9a-zA-Z]*$</constant-value>
        </constant>
    </global>

    <formset>

        <!-- ========== Formulaire authentification utilisateur ========= -->
        <form name="authentificationDynaForms">
            <field property="identifiant" depends="required,minlength,
maxlength,mask">
                <arg0 key="identifiantutilisateur"/>
                <arg1 name="minlength" key="${var:minlength}"
resource="false"/>
                <arg1 name="maxlength" key="${var:maxlength}"
resource="false"/>
                <var>
                    <var-name>minlength</var-name>
                    <var-value>4</var-value>
                </var>
                <var>
                    <var-name>maxlength</var-name>
                    <var-value>7</var-value>
                </var>
                <var>
                    <var-name>mask</var-name>
                    <var-value>^[a-zA-Z]*$</var-value>
                </var>
            </field>
            <field property="motdepasse" depends="mask">
                <arg0 key="motdepasseutilisateur"/>
                <var>
                    <var-name>mask</var-name>
                    <var-value>${password}</var-value>
                </var>
            </field>
        </form>

    </formset>
</form-validation>
```

Nous remarquons au cours de la présentation des exemples, un élément très important de l'utilisation du framework Struts, la conservation des données. En effet, nous n'avons pas besoin de gérer le retour des données lors des saisies. Chaque saisie et choix de l'utilisateur sont conservés en cas d'erreur : les informations dans les formulaires n'ont pas besoin d'être de nouveau saisies. Cette technique est extrêmement souple et pratique par rapport aux langages Internet courants.

Nous allons présenter le principe de validation avec un formulaire dynamique de type classe. Pour mettre en application cette technique, nous pouvons reprendre le *DynaForms* suivant :

```
<!-- ========== Formulaire de gestion des utilisateur ========== -->
<form-bean name="utilisateurDynaForms"
type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="utilisateur" type="boiteoutils.Utilisateur" />
</form-bean>
```

Dans cette déclaration, le formulaire est associé directement à la classe *Utilisateur*. Cependant, Struts permet de gérer entièrement des formulaires associés à des classes Java.

Le formulaire d'authentification avec un *DynaForms* lié à la classe est présenté ci-dessous :

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head>
<title>Authentification Dynaforms</title>
<html:base/>
</head>
<body>
<html:errors/>
<html:form action="/AuthentificationDynaFormsClasse">
Identifiant : <html:text property="utilisateur.identifiant"/><br/>
Mot de passe <html:text property="utilisateur.motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

Ce formulaire déclenche l'action */AuthentificationDynaFormsClasse* déclarée dans le fichier de configuration *struts-config.xml*.

```
<!-- ===action qui permet de gérer les utilisateurs=== -->
<action
    path="/AuthentificationDynaFormsClasse"
    name="utilisateurDynaForms"
    scope="request"
    validate="true"
    input="/pages/authentificationDynaFormsClasse.jsp"
    type="action.AuthentificationActionUtilisateur"
    >
    <forward name="erreurs" path="/pages/erreurs.jsp"/>
    <forward name="succes" path="/pages/succes.jsp"/>
</action>
```

Le déclenchement du formulaire permet d'appeler l'action *execute()* de la classe *AuthentificationActionUtilisateur*. Par contre, le paramètre *validate* de la déclaration de l'action est désormais placé à *true* afin de valider la vérification des saisies utilisateur. En cas d'erreur, c'est la page */pages/authentificationDynaFormsClasse.jsp* qui sera appelée. Nous allons réaliser une vérification sur la présence obligatoire des informations (*required*).

```
<!-- ======= Formulaire authentification utilisateur avec classe ======== -->
<form name="utilisateurDynaForms">
    <field property="utilisateur.identifiant" depends="required">
        <arg0 key="identifiantutilisateur"/>
    </field>
    <field property="utilisateur.motdepasse" depends="required">
        <arg0 key="motdepasseutilisateur"/>
    </field>
</form>
```

La syntaxe du fichier *validation.xml* n'est pas tout à fait la même qu'avec l'utilisation de types simples. En effet, il faut préfixer chaque champ de la classe par le nom de l'instance créée. Dans le fichier *struts-config.xml*, nous avons déclaré le formulaire de cette manière :

```
<form-property name="utilisateur" type="boiteoutils.Utilisateur" />
```

Le nom de l'instance est donc *utilisateur*. Ainsi, lors des validations ou accès, nous utiliserons le préfixe *utilisateur* pour faire référence aux attributs de la classe *boiteoutils.Utilisateur*. Maintenant, du point de vue de l'utilisation de cet objet, nous récupérons le *DynaForms* dans la classe d'action après avoir réalisé un transtypage.

```
package action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.DynaActionForm;
```

```
import boiteoutils.Utilisateur;

@SuppressWarnings("serial")
public class AuthentificationActionUtilisateur extends Action {

    public ActionForward execute(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response) throws Exception
    {
        //le bean formulaire
        DynaActionForm authentificationdynaform=(DynaActionForm)form;
        //récupérer l'objet du formulaire
        Utilisateur utilisateur=(Utilisateur)authentificationdynaform.get
("utilisateur");
        //vérifier les saisies (identifiant=jlafoss et motdepasse=jerome)
        if(utilisateur.getIdentifiant().equals("jlafoss") &&
utilisateur.getMotdepasse().equals("jerome"))
        {
            //redirection vers la page de succes
            return mapping.findForward("succes");
        }
        //dans tous les autres cas, retour vers la page d'erreurs
        return mapping.findForward("erreurs");
    }

}
```

Cette technique d'utilisation de formulaires dynamiques à partir de classe est très pratique et souple à utiliser. Dans la plupart des cas, c'est ce type de programmation qui est utilisé avec Struts.

5. Mise en forme

Nous avons vu tout l'intérêt d'utiliser les validations dynamiques avec les Validators et le fichier de propriétés. Comme les messages affichés en cas d'erreur sont gérés à partir du fichier de propriétés, nous pouvons améliorer ces affichages pour respecter les standards HTML.

En effet, en HTML, la balise *<label for="idchamp"/>* permet de donner un nom/label et de réaliser un lien automatique sur le champ indiqué par l'attribut *for*. Nous pouvons reprendre notre page JSP *authentificationDynaForms.jsp* pour gérer l'attribut *id* d'une balise HTML. Pour le moment, la traduction du tag *<html:text property="identifiant"/>* produit le code source suivant : *<input type="text" name="identifiant" value="">*. Il manque dans la balise *<input/>* l'attribut *id* avec le nom du champ. Pour cela, la librairie de tag Struts propose l'attribut *errorStyleId*.

Le nouveau fichier est donc le suivant :

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head><title>Authentification Dynaforms</title><html:base/></head>
<body>
<html:errors/>
<html:form action="/AuthentificationDynaForms">
<bean:message key="utilisateur.identifiant"/>
<html:text property="identifiant" errorStyleId="identifiant"/><br/>
<bean:message key="utilisateur.motdepasse"/>
<html:text property="motdepasse" errorStyleId="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

Ensuite, nous allons modifier le fichier de propriétés pour retourner la balise *<label/>* dans le message d'erreur.

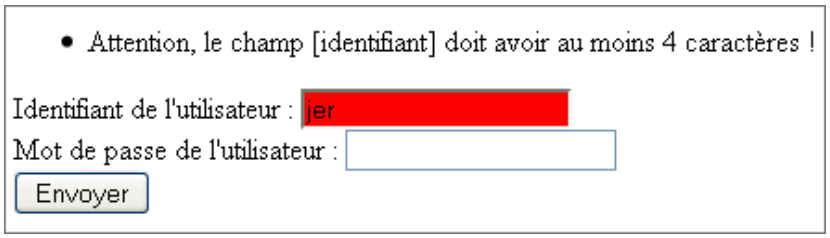
```
errors.required=<label for="{0}">Attention, le champ [{0}]
doit être renseigné !</label>
errors.minlength=<label for="{0}">Attention, le champ [{0}]
doit avoir au moins {1} caractères !</label>
errors.maxlength=<label for="{0}">Attention, le champ [{0}]
ne peut avoir plus de {1} caractères !</label>
```

Nous réalisons ensuite une authentification incorrecte pour tester notre exemple. Le message d'erreur en retour est alors dynamique, nous pouvons cliquer sur celui-ci pour se positionner automatiquement dans le champ de saisie concerné et afficher les dernières propositions.

Nous pouvons encore améliorer les affichages avec l'utilisation de styles CSS. Nous allons définir une classe d'erreur en style avec un fond rouge. La page d'authentification est alors la suivante :

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head><title>Authentification Dynaforms</title>
<style type="text/css">
.erreur
{
background-color:#FF0000;
}
</style>
<html:base/>
</head>
<body>
<html:errors/>
<html:form action="/AuthentificationDynaForms">
<bean:message key="utilisateur.identifiant"/>
<html:text property="identifiant" errorStyleId="identifiant"
errorStyleClass="erreur"/><br/>
<bean:message key="utilisateur.motdepasse"/>
<html:text property="motdepasse" errorStyleId="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

En cas d'erreur de saisie, la classe CSS *erreur* est appliquée grâce à l'attribut *errorStyleClass* de la bibliothèque de tag Struts (http://struts.apache.org/1.x/struts-taglib/tagreference.html).



Les techniques de mise en forme avec Struts sont très puissantes, les seules limites sont notre imagination.

6. Validations en JavaScript

Jusqu'à présent nous avons réalisé les validations côté serveur avec les technologies Struts ou Java. Les informations sont envoyées, analysées, validées puis retournées vers l'utilisateur pour correction si besoin. Pour éviter les allers-retours entre le client et le serveur et ajouter de la souplesse à l'application, nous pouvons insérer des validations côté client. En développement Web traditionnel, il est nécessaire de développer ses propres scripts JavaScript en

rapport avec les tests côté serveur. Avec Struts, il est possible d'utiliser les Validators et d'ajouter une seule balise de tag dans la page concernée.

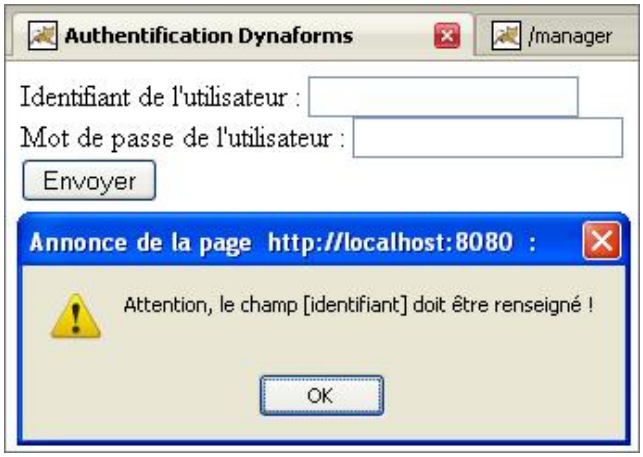
Si nous voulons ajouter la validation JavaScript sur notre page d'authentification précédente, nous devons ajouter la balise *<html:javascript forName="nomduformulaire"/>* dans l'en-tête de la page JSP. La seconde instruction doit être ajoutée dans la balise *<html:form/>* pour indiquer la validation en JavaScript (*onsubmit="return validatenomduformulaire(this)"*) sur le formulaire. La première instruction utilise l'attribut *formName*, il doit avoir pour valeur le nom du JavaBean de formulaire. La seconde instruction utilise la méthode JavaScript commençant par *validate* et se terminant par le nom du JavaBean de formulaire.

Si nous prenons notre page d'authentification nommée *authentificationDynaForms.jsp*, voici le nouveau code :

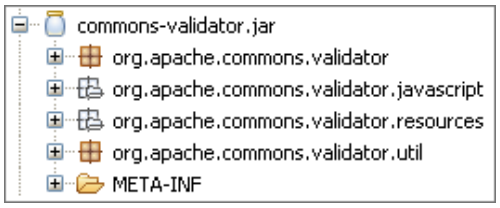
```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head>
<title>Authentification Dynaforms</title>
<style type="text/css">
.erreur
{
background-color:#FF0000;
}
</style>
<html:javascript formName="authentificationDynaForms"/>
<html:base/>
</head>
<body>
<html:errors/>
<html:form action="/AuthentificationDynaForms" onsubmit="return
validateAuthentificationDynaForms(this)">
<bean:message key="utilisateur.identifiant"/><html:text property=
"identifiant"
errorStyleId="identifiant" errorStyleClass="erreur"/><br/>
<bean:message key="utilisateur.motdepasse"/><html:text property="motdepasse"
errorStyleId="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
</body>
</html:html>
```

Le paramètre *onsubmit="..."* contient toujours une méthode JavaScript avec la première lettre du préfixe en majuscule. Dans notre cas, le formulaire se nomme *authentificationDynaForms* mais la méthode est appelée avec le nom *validateAuthentificationDynaForms*.

Nous pouvons désormais actualiser la page et Struts va automatiquement générer les validations JavaScript côté client en accord avec les validations Java côté Serveur.

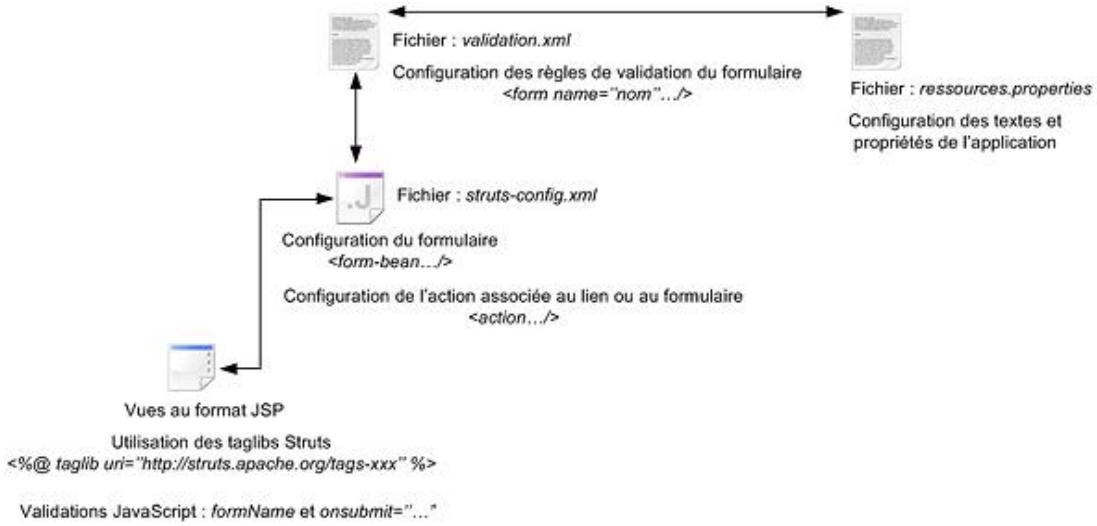


Par défaut, avec Struts, c'est une fenêtre d'alerte qui est affichée pour les validations JavaScript. Il est également possible de changer cette présentation en modifiant les scripts présents dans la librairie *commons-validator.jar* et le paquetage *org.apache.commons.validator.javascript*.



Pour résumer, le principe d'utilisation de Struts est assez simple une fois que tous les services et fichiers de configurations sont bien assimilés. Nous utiliserons essentiellement les *DynaForms* avec des validations par fichier de configuration (*validation.xml*) et JavaScript.

Voici ci-après un schéma récapitulatif du fonctionnement global et des fichiers à connaître pour mettre en place les services Struts.



Le contrôleur Struts

1. Présentation

Avec Java EE le modèle ou design pattern MVC I pour Modèle-Vue-Contrôleur est souvent utilisé. Chacun des éléments de ce modèle est associé à un rôle particulier :

- Un JavaBean qui joue le rôle de modèle et qui est rendu persistant dans une base de données.
- Une page JSP qui joue le rôle de vue et qui permet d'afficher à l'écran les attributs du JavaBean.
- Une Servlet qui joue le rôle de contrôleur, elle gère les traitements et actions entre le modèle, le JavaBean et le système de persistance.

Avec ce modèle MVC I, il existe quasiment autant de Servlets contrôleur que de pages JSP et de JavaBeans. Chaque Servlet déclarée dans le fichier *web.xml* réalise une action (ou un ensemble). Pour alléger le code et éviter ces redondances, le modèle a évolué en modèle MVC II. Il n'existe alors plus qu'une seule Servlet pour toutes les pages JSP et les JavaBeans. Seule cette Servlet est déclarée dans le fichier *web.xml* et gère la totalité des demandes clients. Struts s'appuie sur ce modèle MVC II et possède donc une seule Servlet de gestion.

La Servlet principale utilisée par Struts est *ActionServlet*, elle fonctionne en accord avec le fichier de configuration *struts-config.xml*. Dans ce fichier, toutes les actions associées au contrôleur sont déclarées. Ces actions sont des classes de type *Action* qui en fait, remplacent les Servlets du modèle MVC I.

2. Utilisation et déclaration de la classe ActionServlet

La classe *org.apache.struts.action.ActionServlet* est la Servlet principale de toute application Struts. Cette Servlet possède donc la méthode *init()* qui est lancée au démarrage de l'application, *destroy()* qui est lancée lors de l'arrêt de l'application, *doGet()* qui est la méthode de service associée à la méthode GET HTTP et *doPost()* qui est la méthode de service associée à la méthode POST HTTP.

Cette classe *ActionServlet* est la Servlet contrôleur de l'application Struts, elle s'appuie sur le fichier de configuration *struts-config.xml* pour retrouver les différents éléments de configuration dont elle a besoin afin que l'application Web fonctionne correctement.

La déclaration de cette Servlet principale est réalisée dans le fichier *web.xml* de l'application. Il faut remarquer la déclaration du paramètre de configuration qui est le fichier de gestion Struts et la priorité du chargement avec la balise *<load-on-startup/>*.

```
<!-- Standard Action Servlet Configuration -->
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
```

La Servlet *ActionServlet* est le contrôleur principal Struts mais nous pouvons très bien déclarer d'autres Servlet dans le fichier *web.xml* avec le principe du modèle MVC I et mixer les deux technologies.

Pour rappel, la balise *<servlet-name>* permet de donner un nom à la Servlet et *<servlet-class>* identifie la classe de la Servlet Struts, toujours *org.apache.struts.action.ActionServlet*. La Servlet possède également, comme pour toute application Web, un alias déclaré avec la balise *<servlet-mapping>*.

Toutes les URL se terminant par le suffixe *.do* seront associées et déclencheront la Servlet *ActionServlet*.

```
<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
```

```
</servlet-mapping>
```

Par exemple, si une page JSP possède un lien ou un formulaire avec l'URL */authentification.do*, l'*ActionServlet* Struts cherchera dans son fichier *struts-config.xml* une action dont le paramètre *path* aura pour valeur *path="/authentification"*. Par convention avec Struts le suffixe *.do* est utilisé mais rien n'empêche d'utiliser un autre suffixe comme *.jss*, *.jst*...

3. Les classes Actions

La classe *ActionServlet* est développée par la fondation Struts et ne doit pas être modifiée. Tout le code Java doit être implémenté dans une des classes d'action. Une classe d'action est une classe Java qui hérite de la classe *Action*. Cette classe *Action*, implémente des méthodes qui peuvent être redéfinies dans la classe d'action développée par le programmeur, la plus importante étant la méthode *execute()* qui est le traitement de l'action. Cette méthode est le point d'entrée de la classe d'action, c'est l'équivalent de la méthode *doGet()* ou *doPost()* d'une Servlet.

Toute classe d'action doit être déclarée dans le fichier de configuration de Struts, *struts-config.xml*. La déclaration d'une action s'effectue par l'intermédiaire des éléments <*action*> présents dans la balise <*action-mappings*>. L'élément <*action*> possède plusieurs attributs qui permettent de configurer une action :

- *path* : ce paramètre permet d'identifier le nom de l'action sans le suffixe. Ce nom est lié à l'attribut *action* d'un formulaire ou à l'URL d'un lien.
- *type* : ce paramètre permet d'identifier le nom complet de la classe d'action que l'*ActionServlet* doit utiliser comme action. Le nom est précisé avec le paquetage de la classe.
- *name* : ce paramètre est le nom du JavaBean de formulaire qui va recevoir les informations de l'action.
- *scope* : ce paramètre définit la portée du JavaBean de formulaire (requête ou session). Si la valeur est *session*, les informations saisies par l'utilisateur sont conservées tout le temps de la session utilisateur.
- *attribute* : ce paramètre permet d'identifier le JavaBean de formulaire par un nom. Si cet attribut est absent de la déclaration, le nom du JavaBean est le même que celui défini dans l'attribut *name*.
- *input* : ce paramètre permet d'identifier la vue à l'origine de l'action mais également la vue ou action qui sera appelée en cas d'erreur de saisie.
- *forward* : ce paramètre permet de réaliser des liens vers une autre vue ou action.
- *include* : ce paramètre permet d'inclure une autre vue dans la vue à l'origine de l'action.

La balise <*action*> possède également des sous-éléments :

- *exceptions* : permet d'identifier un type d'exception et de configurer les opérations à effectuer en cas d'erreur.
- *set-property* : permet de transmettre des informations à l'action.
- *forward* : permet de réaliser une redirection vers une vue ou action spécifique. Une déclaration d'action peut être associée à plusieurs balises <*forward/*>.

```
<action
    path="/Authentification"
    type="action.AuthentificationAction"
    name="authentificationForm"
    scope="request"
    input="/pages/authentificationForm.jsp">
    <forward name="erreurs" path="/pages/erreurs.jsp"/>
    <forward name="succes" path="/pages/succes.jsp"/>
</action>
```

Dans l'exemple précédent, l'attribut *path* identifie l'action appelée par un lien ou un formulaire. Nous retrouverons alors un lien comme ceci <*a href="/Authentification.do"*>*lien*</*a*> ou un formulaire <*form action="/Authentification.do"*> ou un formulaire Struts <*html:form action="/Authentification.do"*>. L'attribut *type* permet d'identifier la classe de l'application qui réalisera l'action associée (*action.AuthentificationAction*). L'attribut *name* définit le JavaBean de formulaire déclaré

dans le fichier de configuration *struts-config.xml*.

```
<!--  bean d'authentification actionform -->
<form-bean  name="authentificationForm"
type="actionform.AuthentificationForm"/>
```

Les attributs *name* de la déclaration du JavaBean de formulaire et de la déclaration de l'action sont strictement identiques. L'attribut *scope* indique que le JavaBean est renseigné seulement pour la requête de l'utilisateur. L'attribut *attribute* n'est pas précisé car le nom du JavaBean de formulaire est identique à l'attribut *name* de l'action. Enfin, l'attribut *input* indique que la page de soumission est la JSP */pages/authentificationForm.jsp*. Si des erreurs de validation sont déclenchées, la page déclarée à *input* sera alors affichée en retour sans perte d'informations (éventuelles saisies de l'utilisateur).

Il existe également deux sous-éléments <*forward*>, ces éléments permettent de rediriger l'utilisateur vers d'autres vues ou actions en conservant ou pas les informations présentes dans la requête. Son fonctionnement est identique à la classe *RequestDispatcher* avec la méthode *forward()*.

L'élément <*forward*> comporte les attributs suivants :

- *name* : ce paramètre permet de donner un nom à la redirection qui sera utilisée dans l'action.
- *path* : ce paramètre permet d'identifier la vue ou l'action vers laquelle réaliser le transfert.
- *redirect* : ce paramètre optionnel attend une valeur booléenne pour savoir si les informations de la requête sont conservées ou non dans la redirection (classe *RequestDispatcher* avec *forward()* ou *response.sendRedirect ()*).

Il est également possible de définir des redirections globales à toutes les actions de l'application. Ces redirections doivent être décrites en dehors de l'élément <*action-mapping*> dans la balise <*global-forwards*>.

```
<global-forwards>
        <forward name="welcome" path="/Welcome.do"/>
</global-forwards>
```

Cette technique permet de définir des redirections globales comme la page d'accueil de l'application, la page d'erreurs... Le développement et la maintenance sont alors facilités.

L'élément <*exception*> comporte également plusieurs attributs. Cette balise permet de gérer les erreurs qui peuvent survenir dans l'application. Comme pour la balise <*forward*>, il est possible de déclarer des exceptions dans une action ou de façon globale.

- *key* : ce paramètre permet d'identifier le message d'erreur à utiliser dans le fichier de propriétés.
- *path* : ce paramètre identifie la vue ou l'action utilisée pour afficher l'erreur.
- *type* : ce paramètre permet de déclarer le type d'exception à traiter.
- *scope* : ce paramètre indique si les erreurs doivent être enregistrées dans la session ou la requête de l'utilisateur.

4. La méthode execute()

Une action est composée de sa déclaration dans le fichier *struts-config.xml* et d'une classe d'action qui implémente la méthode *execute()*.

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response) throws Exception
```

- *ActionMapping mapping* : ce paramètre représente l'instance de l'action en cours.
- *ActionForm form* : ce paramètre représente le JavaBean de formulaire soumis à l'action.
- *ServletRequest request* : ce paramètre représente la requête HTTP de l'utilisateur.

- *ServletResponse response* : ce paramètre représente la réponse HTTP de l'utilisateur.

Cette méthode *execute()* est lancée lorsque l'action est instanciée par l'*ActionServlet*. Il est alors possible de récupérer les informations du JavaBean de formulaire grâce au paramètre, objet *form*. On peut ainsi récupérer par simple transtypage le JavaBean de formulaire. Le principe est le même, que cela soit avec un *ActionForm* ou un *DynaActionForm*.

```
//lire le JavaBean
AuthentificationForm authentificationform=(AuthentificationForm)form;

//lire le JavaBean
DynaActionForm authentificationdynaform=(DynaActionForm)form;
```

La classe *ActionMapping* possède une méthode *findForward()* qui permet de retrouver une redirection définie par le ou les élément(s) <*forward*> de l'action ou une redirection globale définie dans <*global-forwards*>. En résumé, la méthode *execute()* est le traitement à réaliser par l'action. Elle permet de récupérer les informations transmises par l'utilisateur dans le JavaBean de formulaire, de réaliser une action spécifique et de rediriger l'utilisateur.

La classe *Action* possède des classes filles ou classes dérivées. Ces classes permettent de simplifier le travail du programmeur au sein d'une action.

ForwardAction

La classe *ForwardAction* est l'action la plus simple et une des plus utilisées. Cette classe permet de réaliser des redirections sans implémentation de code Java. Elle fonctionne de la même façon que la méthode *forward()* de la classe *RequestDispatcher*. Cette classe est surtout utilisée pour faire du routage, lorsqu'une page JSP souhaite transférer ses données vers une autre page JSP, action ou Servlet.

```
<action
    path="/AuthentificationDynaForms"
    name="authentificationDynaForms"
    validate="true"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/pages/bienvenueDynaForms.jsp"
    input="/pages/authentificationDynaForms.jsp"
    scope="request">
</action>
```

Dans l'exemple précédent, lorsque l'utilisateur soumet le formulaire dynamique, les saisies sont validées. En cas de succès, l'utilisateur est redirigé vers la page */pages/bienvenueDynaForms.jsp* par l'intermédiaire de la configuration *type="org.apache.struts.actions.ForwardAction"* et *parameter="/pages/bienvenueDynaForms.jsp"*.

IncludeAction

La classe *IncludeAction* est une action simple. Elle ne nécessite pas d'implémentation de code. Le développeur n'a donc pas besoin de créer sa propre classe. Elle est utilisée lorsqu'un développeur souhaite transférer ses données vers une autre page JSP et qu'une balise <*jsp:include...*> existe dans la page appelée. Elle fonctionne de la même façon que la méthode *RequestDispatcher.include(request,response)*.

DispatchAction

La classe *DispatchAction* permet de déclarer plusieurs méthodes *execute()* dans une même classe d'action. Ces méthodes sont ensuite déclenchées dans les pages JSP par l'intermédiaire de l'attribut *action* du tag <*html:form*> avec le nom de la méthode en paramètre. Si nous reprenons notre exemple avec la page d'authentification */pages/authentificationForm.jsp*, nous pouvons modifier l'action du formulaire avec un paramètre supplémentaire.

```
<html:form action="/Authentification.do?methode=verifierAuthentification">
```

Nous remarquons qu'avec l'utilisation d'un paramètre pour le *DispatchAction*, nous précisons alors l'extension du chemin déclenché. Ex : *Authentification.do à la place de Authentification*.

Dans ce cas, le formulaire est soumis à l'action *Authentification* et utilise la méthode *verifierAuthentification()*. Les méthodes doivent avoir un nom différent et la méthode *execute()* n'est pas implémentée dans ce cas. Toutes les méthodes doivent avoir la même signature que la méthode *execute()*.

Voici un exemple d'authentification qui pourrait être composée de plusieurs étapes. Cette technique est idéale pour gérer des formulaires par étape sur plusieurs pages.

```
<action
    path="/Authentification"
    type="action.AuthentificationAction"
    name="authentificationForm"
    scope="request"
    input="/pages/authentificationForm.jsp"
    validate="false"
    parameter="methode">
    <forward name="erreurs" path="/pages/erreurs.jsp"/>
    <forward name="succes" path="/pages/succes.jsp"/>
</action>
```

La déclaration d'une action de type *DispatchAction* ne diffère pas d'une action courante. Nous pouvons remarquer cependant l'utilisation du paramètre nommé *parameter* qui correspond au nom utilisé comme référence dans les actions ou liens.

```
package action;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.DispatchAction;
import actionform.AuthentificationForm;

@SuppressWarnings("serial")
public class AuthentificationAction extends DispatchAction {

    public ActionForward verifierAuthentification(ActionMapping
mapping, ActionForm form, HttpServletRequest request, HttpServletResponse
response) throws Exception
    {
            //lire le JavaBean de la vue
        AuthentificationForm authentificationform=(AuthentificationForm)
form;
        //vérifier les saisies (identifiant=jlafoss et motdepasse=jerome)
        if(authentificationform.getIdentifiant().equals("jlafoss") &&
authentificationform.getMotdepasse().equals("jerome"))
        {
            //redirection vers la page de succes
            return mapping.findForward("succes");
        }
        //dans tous les autres cas, retour vers la page d'erreurs
        return mapping.findForward("erreurs");
    }
}
```

LookUpDispatchAction

La classe *LookUpDispatchAction* est semblable à la classe *DispatchAction*, elle implémente des méthodes qui sont identiques à la signature de la méthode *execute()*. La différence entre *DispatchAction* et *LookUpDispatchAction* réside dans la manière d'appeler les méthodes. Avec *LookUpDispatchAction*, il n'est pas nécessaire d'ajouter des paramètres aux URL mais l'appel d'une méthode s'effectue grâce aux boutons de soumission inclus dans une page JSP.

En effet, avec une classe de type *LookUpDispatchAction*, les boutons <*html:submit*> doivent avoir un attribut *property* supplémentaire.

```
<html:submit property="action">
  <bean:message key="action.verifierauthentification"/>
</html:submit>
```

L'élément <*bean*> présent dans l'élément <*html:submit*> permet de donner un nom au bouton de type submit et à l'action. L'attribut *property* de chaque bouton de type submit fait référence à une valeur nommée *action* qui doit être identique à l'attribut *parameter* de l'action déclarée dans le fichier de configuration de Struts.

```
<action
    path="/Authentification"
    type="action.AuthentificationAction"
    name="authentificationForm"
```

```
    scope="request"
    input="/pages/authentificationForm.jsp"
    validate="false"
    parameter="action">
    <forward name="erreurs" path="/pages/erreurs.jsp"/>
    <forward name="succes" path="/pages/succes.jsp"/>
</action>
```

La classe d'action doit hériter de la classe *LookUpDispatchAction* et doit implémenter la méthode *getKeyMethodMap()* qui renvoie une collection avec les différentes actions à réaliser. Du point de vue du programmeur, le *LookUpDispatchAction* ne présente pas un véritable intérêt par rapport à l'utilisation du *DispatchAction* qui est beaucoup plus souple.

MappingDispatchAction

La classe *MappingDispatchAction* est sans doute la plus utilisée et la plus souple pour le programmeur. La déclaration des méthodes associées a la même signature que la méthode *execute()* et la déclaration dans le fichier *struts-config.xml* permet de définir plusieurs actions avec l'attribut *parameter*. La valeur de *parameter* correspond alors à la méthode d'action à utiliser.

Nous allons montrer ce principe en reprenant notre page JSP d'origine */pages/authentificationForm.jsp* et en réalisant deux actions distinctes. La première action permet de réaliser une authentification, la seconde permet une redirection sur la page de bienvenue.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head><title>Authentification</title><html:base/></head>
<body>
<html:errors/>
<html:form action="/Authentification">
Identifiant : <html:text property="identifiant"/><br/>
Mot de passe <html:text property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
</html:form>
<html:form action="/PasAuthentification">
<html:submit value="Sans authentification"/>
</html:form>
</body>
</html:html>
```

Le fichier de mapping est très intéressant dans ce cas. C'est la même classe qui gère ces deux actions. L'attribut *parameter* permet de déclencher le bon traitement en fonction du choix de l'utilisateur.

```
<action
    path="/Authentification"
    type="action.AuthentificationAction"
    name="authentificationForm"
    scope="request"
    input="/pages/authentificationForm.jsp"
    validate="false"
    parameter="verifierAuthentification">
    <forward name="erreurs" path="/pages/erreurs.jsp"/>
    <forward name="succes" path="/pages/succes.jsp"/>
</action>
<action
    path="/PasAuthentification"
    type="action.AuthentificationAction"
    name="authentificationForm"
    scope="request"
    input="/pages/authentificationForm.jsp"
    validate="false"
    parameter="pasAuthentification">
    <forward name="succes" path="/pages/succes.jsp"/>
</action>
```

```
package action;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.ActionForm;
```

```
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.MappingDispatchAction;
import actionform.AuthentificationForm;

@SuppressWarnings("serial")
public class AuthentificationAction extends MappingDispatchAction {

    public ActionForward verifierAuthentification(ActionMapping
mapping, ActionForm form, HttpServletRequest request,
HttpServletResponse response) throws Exception
    {
            //lire le JavaBean de la vue
        AuthentificationForm authentificationform=(AuthentificationForm)
form;
        //vérifier les saisies (identifiant=jlafoss et motdepasse=jerome)
        if(authentificationform.getIdentifiant().equals("jlafoss") &&
authentificationform.getMotdepasse().equals("jerome"))
        {
            //redirection vers la page de succes
            return mapping.findForward("succes");
        }
        //dans tous les autres cas, retour vers la page d'erreurs
        return mapping.findForward("erreurs");
    }

    public ActionForward pasAuthentification(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response) throws Exception
    {
        //redirection vers la page de succes
        return mapping.findForward("succes");
    }
}
```

Nous remarquons qu'un clic sur le bouton **Envoyer** nécessite une authentification correcte avec le déclenchement de la méthode *verifierAuthentification()* alors qu'un clic sur le bouton **Sans authentification** permet de déclencher la fonction *pasAuthentification()* de la même classe.

La classe d'action hérite de la classe *MappingDispatchAction* et implémente, comme la classe *DispatchAction*, des méthodes respectant la signature de la méthode *execute()*. Lors des développements il faut utiliser dans la plupart des cas la classe *MappingDispatchAction* avec un paramètre pour le traitement de l'action à déclencher. Il y aura par exemple une fonction pour la liste des articles, une pour la consultation, une pour la modification, une pour la validation de la modification, une pour la création, une pour la validation de la création et une dernière pour sa suppression. La totalité du code sera alors regroupée par thème dans la même classe d'action.

Ensuite, la classe simple *Action* sera utilisée pour le développement de services spécifiques comme par exemple, une action de téléchargement, une action de copie de fichier, une action de chargement ou upload de données.

ActionDispatcher

La classe *ActionDispatcher* est identique à la classe *MappingDispatchAction* mais hérite de la classe *Action* et possède une propriété de type *ActionDispatcher*. Cette classe est peu utilisée car plus lourde à mettre en place que la classe *MappingDispatchAction* et nécessite en plus la déclaration de la méthode *execute()* pour la gestion du dispatcher.

SwitchAction

La classe *SwitchAction* permet d'utiliser une action présente dans un autre fichier de configuration *struts-config.xml* que celui du module principal. En effet, nous pouvons très bien créer un second module, en utilisant un autre fichier *struts-config.xml* qui sera appelé par exemple *struts-configadmin.xml* pour l'interface d'administration.

La déclaration d'un nouveau fichier de configuration est très simple, elle est réalisée dans le fichier de configuration de l'application *web.xml*.

```
<servlet>
         <servlet-name>action</servlet-name>
         <servlet-class>org.apache.struts.action.ActionServlet
</servlet-class>
         <init-param>
           <param-name>config</param-name>
           <param-value>/WEB-INF/struts-config.xml,/WEB-INF/struts-
```

```
configadmin.xml</param-value>
         </init-param>
         <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
         <servlet-name>action</servlet-name>
         <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Cette déclaration fonctionnelle autorise le développement d'actions et formulaires dans plusieurs fichiers de configuration mais ne permet pas de distinguer les formulaires de manière unique. Pour cela, nous devons donner un nom au paramètre de configuration comme avec la déclaration suivante :

```
<servlet>
   <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>

    <init-param>
      <param-name>config/admin</param-name>
      <param-value>/WEB-INF/struts-configadmin.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
```

On utilisera le fichier *struts-config.xml* pour gérer les déclarations des formulaires, actions et redirections pour la partie en accès public du site (*frontoffice*) et le fichier *struts-configadmin.xml* pour gérer la partie privée (*backoffice*). Il faut cependant prévoir le cas où certains utilisateurs, avec des droits d'accès plus importants, voudraient accéder à des services du module d'administration. C'est le rôle de la classe *SwitchAction*.

Pour utiliser ce service, il faut déclarer une action de type *SwitchAction* dans le module principal *struts-config.xml*.

```
<action path="/switch" type="org.apache.struts.actions.SwitchAction"/>
```

Une fois cette déclaration réalisée, il est nécessaire de créer les actions du module principal. Nous allons reprendre le formulaire d'authentification *authentificationForm.jsp* et en cas de succès, une action du module administrateur sera appelée.

La déclaration dans le fichier de configuration principale *struts-config.xml* est la suivante :

```
<action
    path="/Authentification"
    type="action.AuthentificationAction"
    name="authentificationForm"
    scope="request"
    input="/pages/authentificationForm.jsp"
    validate="false"
    parameter="verifierAuthentification">
        <forward name="erreurs" path="/pages/erreurs.jsp"/>
        <forward name="succes" path="/switch.do?page=/
succesAdmin.do&amp;prefix=/admin"/>
</action>
```

Nous remarquons qu'en cas de succès suite à une authentification correcte, c'est le module *switch.do* qui va réaliser le routage vers l'action */succesAdmin.do* présente dans l'autre module. Le paramètre *prefix* indique le nom du module déclaré dans le fichier *web.xml* avec dans notre exemple, le terme *admin* pour la déclaration :

```
<param-name>config/admin</param-name>
<param-value>/WEB-INF/struts-configadmin.xml</param-value>
```

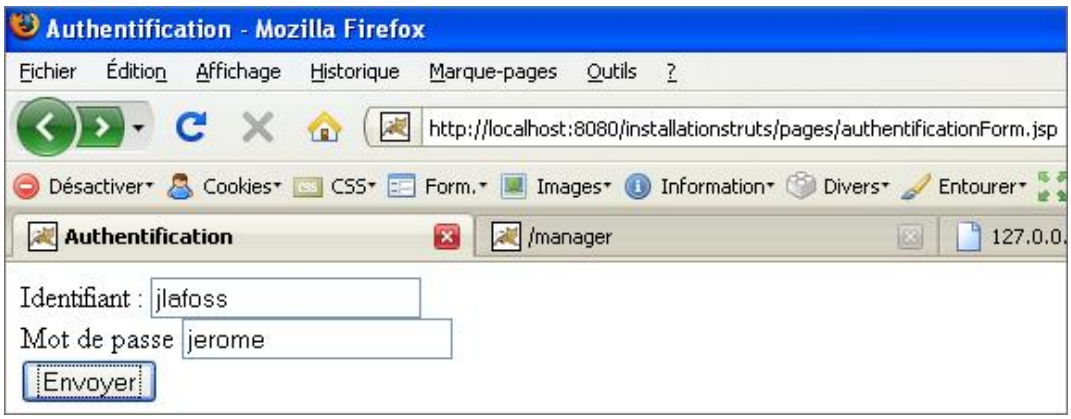
Il est important de bien utiliser le symbole *&* dans le fichier de configuration au format XML. Le caractère & est réservé lors de l'utilisation du langage XML.

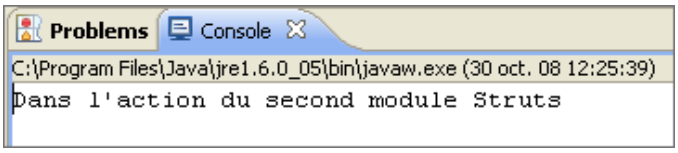
Enfin, il nous reste à déclarer l'action et la classe associée dans le second module *struts-configadmin.xml*.

```
<action-mappings>
    <action
        path="/succesAdmin"
        type="action.SuccesAdminAction">
    </action>
</action-mappings>
```

Pour terminer, la classe action *.SuccesAdminAction* permet uniquement d'afficher un message dans la console Eclipse afin de valider le fonctionnement de l'ensemble.

```
package action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

@SuppressWarnings("serial")
public class SuccesAdminAction extends Action {

    public ActionForward execute(ActionMapping mapping,
ActionForm form, HttpServletRequest request,
HttpServletResponse response) throws Exception
    {
        System.out.println("Dans l'action du second module Struts");
        return null;
    }
}
```





5. Les méthodes Helper

La classe *Action* possède un ensemble de méthodes appelées *Helper* qui permettent de fournir des informations sur l'action ou de tester certaines opérations déclenchées par l'utilisateur comme un clic sur un bouton annuler, l'ajout d'un message d'erreur, la modification de la locale pour la langue...

- *addErrors()* : cette méthode permet d'ajouter une collection de messages d'erreurs à partir de clés. Ces messages sont ensuite affichés par la balise *<html:errors/>*.

- *addMessages()* : cette méthode permet d'ajouter une collection de messages (de succès en général) à partir de clés. Ces messages sont ensuite affichés par la balise *<html:messages/>*.

- *generateToken()* : cette méthode permet de générer un jeton de transaction pour forcer les requêtes HTTP.
- *isTokenValid()* : cette méthode renvoie *true* si un jeton de transaction existe dans la session de l'utilisateur et que la valeur soumise avec la requête est identique.
- *isCancelled()* : cette méthode renvoie *true* si le bouton Cancel du formulaire est cliqué.
- *getLocale()* : cette méthode renvoie un objet de type *Locale* qui représente la langue utilisée par l'utilisateur.
- *setLocale()* : cette méthode définit la localisation de l'utilisateur. Elle permet de pouvoir changer de langue.
- *saveToken()* : cette méthode enregistre un nouveau jeton de transaction dans la session de l'utilisateur.
- *getDataSource()* : cette méthode renvoie un pool de connexion de l'application.

Nous allons mettre en application une de ces méthodes *Helper* avec la fonction *isCancelled()*. Cette méthode permet de vérifier si un bouton de type *Cancel* a été cliqué dans un formulaire pour l'initialiser par exemple. Si le bouton a été cliqué, la méthode *isCancelled()* renvoie *true* sinon *false*.

Pour mettre en application cet exemple, nous allons reprendre le formulaire */pages/authentificationForm.jsp* et initialiser les champs du formulaire avec un utilisateur public qui a pour coordonnées, l'identifiant *public* et le mot de passe *public* avec la déclaration suivante de l'action :

```
<action
    path="/Authentification"
    type="action.AuthentificationAction"
    name="authentificationForm"
    scope="request"
    input="/pages/authentificationForm.jsp"
    validate="false"
    parameter="verifierAuthentification">
    <forward name="erreurs" path="/pages/erreurs.jsp"/>
    <forward name="succes" path="/pages/succes.jsp"/>
    <forward name="initialisation" path="/pages/authentificationForm.jsp"/>
</action>
```

Nous avons déclaré une nouvelle redirection possible nommée *initialisation* qui permet de retourner à la page d'authentification. Maintenant dans la page JSP nous allons ajouter un bouton de type *Cancel*.

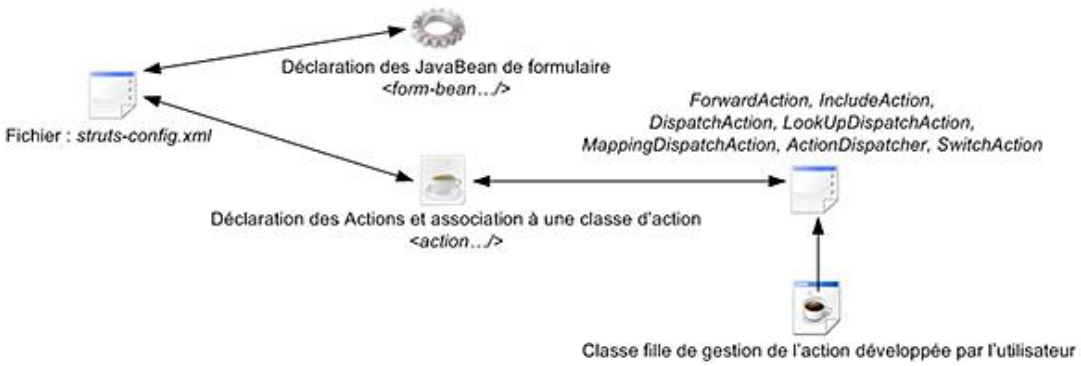
```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head><title>Authentification</title><html:base/></head>
<body>
<html:errors/>
<html:form action="/Authentification">
Identifiant : <html:text property="identifiant"/><br/>
Mot de passe <html:text property="motdepasse"/><br/>
<html:submit value="Envoyer"/>
<html:cancel></html:cancel>
</html:form>
</body>
</html:html>
```

Enfin, il nous reste à traiter cette méthode *Helper* dans l'action *ActionAuthentificationAction* de l'application. Nous vérifions si l'utilisateur a cliqué sur le bouton **Cancel** et dans ce cas, nous insérons les valeurs par défaut pour une visite en mode public.

```
package action;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.MappingDispatchAction;
import actionform.AuthentificationForm;
```

```
@SuppressWarnings("serial")
public class AuthentificationAction extends MappingDispatchAction {

    public ActionForward verifierAuthentification(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response) throws Exception
    {
          //lire le JavaBean de la vue
        AuthentificationForm authentificationform=(AuthentificationForm)
form;
        //clique sur le bouton Cancel
        if(isCancelled(request))
        {
           //vider le formulaire par sécurité
           authentificationform.reset(mapping, request);
           //renvoyer le formulaire dans la requete avec
les paramètres par défaut
           authentificationform.setIdentifiant("public");
           authentificationform.setMotdepasse("public");
           //renvoyer le formulaire dans la requete
           request.setAttribute("authentificationForm",authentificationform);
           return mapping.findForward("initialisation");
        }
        //vérifier les saisies (identifiant=jlafoss et motdepasse=jerome)
        if(authentificationform.getIdentifiant().equals("jlafoss") &&
authentificationform.getMotdepasse().equals("jerome"))
        {
           //redirection vers la page de succes
           return mapping.findForward("succes");
        }
        //dans tous les autres cas, retour vers la page d'erreurs
        return mapping.findForward("erreurs");
    }
}
```

Pour résumer, voici un schéma explicatif des différentes actions envisageables dans une application développée avec le framework Struts.



Il est également important de préciser qu'il existe plusieurs méthodes très précieuses lors de l'utilisation du framework Struts. La première est une méthode statique de la classe *BeanUtils* qui permet de copier la totalité d'un JavaBean de formulaire reçu (de type *ActionForm* ou *DynaForms*) dans un objet Java. Nous pourrons ainsi par la suite directement manipuler notre objet pour le sérialiser, le rendre persistant dans une base de données...

Si nous modifions notre méthode précédente pour tester cette fonction :

```
public ActionForward verifierAuthentification(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response) throws Exception
{
    //lire le JavaBean de la vue
    AuthentificationForm authentificationform=(AuthentificationForm) form;
    //instancier un objet utilisateur
    Utilisateur utilisateur=new Utilisateur();
    //copier en une seule ligne l'ensemble des saisies
du formulaire dans notre objet
    BeanUtils.copyProperties(utilisateur,authentificationform);
```

```
    //Afficher les valeurs saisies
    System.out.println("Identifiant : "+utilisateur.getIdentifiant());
    System.out.println("Mot de passe : "+utilisateur.getMotdepasse());
    return null;
}
```

La seconde méthode intéressante est *request.setAttribute()* qui permet de renvoyer l'objet formulaire directement dans la requête. Il sera ainsi possible de modifier les valeurs saisies et renvoyer l'ensemble sans perte d'informations.

```
public ActionForward verifierAuthentification(ActionMapping mapping,
ActionForm form,
HttpServletRequest request, HttpServletResponse response) throws Exception
{
   //lire le JavaBean de la vue
   AuthentificationForm authentificationform=(AuthentificationForm) form;
   //modification de la saisie, on vide tout
   authentificationform.setIdentifiant("");
   authentificationform.setMotdepasse("");
   //retour sur la page de saisie
   return mapping.findForward("initialisation");
}
```

Enfin, la méthode *request.setAttribute()* permet également de renvoyer tous les paramètres reçus en une seule ligne :

```
public ActionForward verifierAuthentification(ActionMapping
mapping, ActionForm form, HttpServletRequest request,
HttpServletResponse response) throws Exception
{
    //lire le JavaBean de la vue
    AuthentificationForm authentificationform=(AuthentificationForm) form;
    //renvoyer tous les paramètres reçus dans la requête
    request.setAttribute(mapping.getAttribute(), authentificationform);
    //retour sur la page de saisie
    return mapping.findForward("initialisation");
}
```

Développement du module d'administration

1. Présentation

Après avoir présenté le framework Java EE Struts, nous allons développer la partie administration et gestion du chat *BetaBoutique*. Cette application représente la première brique de notre précédent schéma explicatif.

Cette nouvelle application nommée *chatbetaboutique* sera développée en respectant la charte graphique du site mais sans introduire la partie service du chat avec les actions liées à la communication et aux dialogues.

Ce projet permet de gérer le serveur, les salons, utilisateurs et messages.

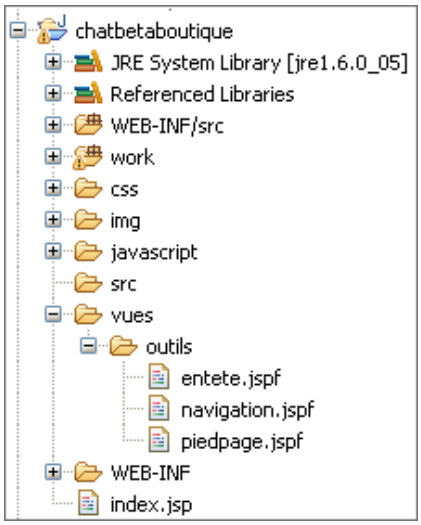
2. Mise en place

Pour le développement de cette partie du projet nous allons utiliser les technologies Struts les plus adaptées à la réalisation rapide et efficace d'une application Java EE. Bien entendu, il existe plusieurs manières de développer une application suivant les algorithmes, solutions techniques, choix et compétences de chacun. La solution proposée n'est pas LA solution parfaite mais une solution envisageable, d'une application de gestion d'un chat.

Nous allons commencer la mise en place de cette application en créant un nouveau projet nommé *chatbetaboutique* avec Eclipse à partir du projet *betaboutique*. Pour cela, le plus simple est de réaliser une copie du répertoire *betaboutique* présent dans nos projets Eclipse et de le nommer *chatbetaboutique*. Ensuite, avec Eclipse, nous réalisons une nouvelle création de projet à partir des sources.

Le projet est désormais opérationnel, nous allons supprimer plusieurs fichiers qui ne sont pas nécessaires pour le développement du service de gestion du chat. La page *cgv.jsp*, le répertoire */admin/vues/articles*, et tous les paquetages du projet (*betaboutique.servlets.clients*...).

L'application sera directement accessible par une URL protégée et sera installée à la racine de l'application. Le projet proposé possède donc la structure suivante accessible à cette adresse : http://localhost:8080/chatbetaboutique/



Les répertoires */css*, */img* et */javascript* sont identiques à ceux de l'application *betaboutique*. Le fichier */WEB-INF/web.xml* est très simple et possède la structure suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- fichier de point de départ de l'application -->
    <!-- il interdit en plus l'exploration de l'arborescence -->
    <!-- attention, chemin toujours relatif -->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Enfin les fichiers */index.jsp*, */entete.jspf*, */navigation.jspf* et */piedpage.jspf* sont présentés ci-dessous:

```
<%@ include file="vues/outils/entete.jspf" %>
<%@ include file="vues/outils/piedpage.jspf" %>

<html>
<head>
<title>Administration - BetaBoutique</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta name="description" content="BetaBoutique">
<!-- feuilles de style -->
<link rel="stylesheet" type="text/css" href="css/styles.css"
title="defaut" />
<link rel="stylesheet" type="text/css" href="css/styles_admin.css"
title="defaut" />
<!--  pour les bulles d'aide -->
<link type="text/css" rel="stylesheet" media="all"
href="javascript/jtip/jtip.css"/>
<script type="text/javascript" src="javascript/jquery/jquery.js"></script>
<script type="text/javascript" src="javascript/jtip/jtip.js"></script>
</head>
<body>
<div id="global">
     <div id="entete">
         <table border="0" cellspacing="0" cellpadding="0"
width="100%" height="100%">
         <tr>
             <td align="left" valign="top"><a href="/"><img
src="img/bandeau_haut_gauche.jpg" border="0"/></a></td>
             <td align="left" valign="top"><img
src="img/bandeau_haut_centre.jpg"/></td>
             <td align="left" valign="top" background="img/
bandeau_haut_droite_admin.jpg" width="222"> </td>
         </tr>
         </table>
     </div>
     <div id="contenu">
     <table border="0" cellspacing="0" cellpadding="0" width="100%">
     <tr>
         <td valign="top" width="200">
          <!-- COLONNE GAUCHE -->
          <%@ include file="navigation.jspf" %>
         </td>
         <!-- COLONNE CENTRE -->
         <td valign="top" width="100%" style="padding-left:5px">
```

```
<table border="0" cellspacing="0" cellpadding="0"
name="menugauche" id="menugauche"
style="background: url('img/fondmenu.gif') repeat-x;"  height="500">
<tr>
  <td width="100%" valign="top"><br/><span style="margin-left:
30px;"><b>ADMINISTRATION</b></span>
   <ul class="menucategorie">
       <li><a href=""><img src="img/fleche.gif" border="0"
align="absmiddle"/> Gestion des clients</a></li>
       <li><a href=""><img src="img/fleche.gif" border="0"
align="absmiddle"/> Gestion des articles</a></li>
       <li><a href=""><img src="img/fleche.gif" border="0"
align="absmiddle"/> Gestion des commandes</a></li>
   </ul>
  </td>
</tr>
</table>
```

```
</td>
</tr>
</table>
<div id="betaboutique">
 BetaBoutique est une boutique de et par la soci&eacute;t&eacute;
BetaBoutique SARL au Capital 10 000Euros  n° siret 111 222 333 444 555
```

```
      </div>
</div>
</body>
</html>
```

À cette étape du projet, le lancement de l'application permet d'afficher l'interface graphique du projet dans le navigateur ainsi que le menu de navigation qui sera modifié par la suite.

3. Installation de Struts

L'application *chatbetaboutique* est maintenant correctement déployée. Nous allons procéder à l'installation de Struts en copiant les archives Java *.jar* et les fichiers de configuration associés. Pour cela, nous allons commencer par copier les archives suivantes dans le répertoire */WEB-INF/lib* de l'application :

- *antlr.jar*
- *commons-beanutils.jar*
- *commons-digester.jar*
- *commons-fileupload.jar*
- *commons-logging.jar*
- *commons-validator.jar*
- *jakarta-oro.jar*
- *struts.jar*

Ensuite, nous allons inclure ces archives au *classpath* Java avec la commande suivante dans Eclipse **Project - Properties - Java Build Path - Add External JARs** et sélection de toutes les archives.

L'installation est presque terminée, il nous reste à installer par simple copier/coller les fichiers de configuration *struts-config.xml*, *validation.xml* et *validator-rules.xml* dans le répertoire */WEB-INF* de l'application et à créer un paquetage nommé *ressources* avec un fichier de propriétés nommé dans notre cas *ressources.properties*.

La gestion de ce fichier est réalisée dans le fichier de configuration *struts-config.xml*.

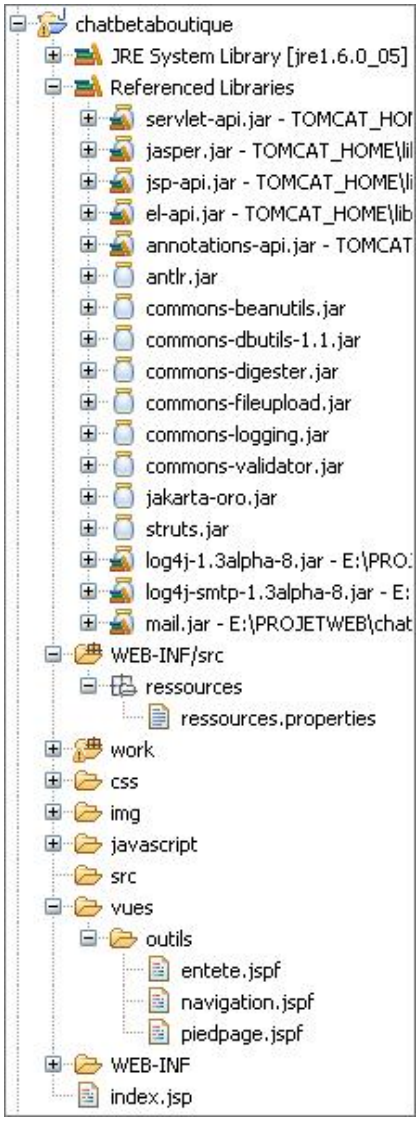
```
<!--  Message Resources Definitions -->
<message-resources parameter="ressources.ressources" />
```

Nous terminons l'installation du projet par la configuration de l'*ActionServlet* principale de l'application au sein du fichier */WEB-INF/web.xml*. Cette partie de code permet de gérer la totalité des actions avec Struts ainsi que la déclaration des fichiers de configuration.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
  <display-name>Application chatbetaboutique</display-name>
  <!-- Standard Action Servlet Configuration -->
  <servlet>
```

```
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
 </servlet>
  <!-- Standard Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <!-- The Usual Welcome File List -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

À cette étape du projet, l'arborescence de l'application est la suivante :



4. Installation du pool de connexion à la base de données

Nous allons terminer la partie installation de l'application par la mise en place de la source de connexion à la base de données. Cette étape est strictement identique au projet étudié dans le chapitre précédent consacré aux bases de données. Le fichier de configuration de l'application */TOMCAT/conf/Catalina/localhost/chatbetaboutique.xml* est présenté

ci-après :

```
<Context path="/chatbetaboutique" reloadable="true"
docBase="E:\PROJETWEB\chatbetaboutique"
workDir="E:\PROJETWEB\chatbetaboutique\work">
<Resource name="jdbc_chatbetaboutiquemysql" auth="Container"
type="javax.sql.DataSource"
username="root" password="" driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/chatbetaboutique"
maxActive="20" maxIdle="10" validationQuery="SELECT 1"/>
</Context>
```

Nous passons ensuite à la définition de la *DataSource* au sein du fichier */WEB-INF/web.xml* de l'application.

```
<!-- datasource a la base de donnees -->
<resource-ref>
     <description>DB Connection</description>
     <res-ref-name>jdbc_chatbetaboutiquemysql</res-ref-name>
     <res-type>javax.sql.DataSource</res-type>
     <res-auth>Container</res-auth>
</resource-ref>
```

Pour ce qui est de la création de la connexion au pool, nous pourrions définir une classe chargée au lancement de l'application comme dans le chapitre précédent. Avec le framework Struts, il existe la possibilité de créer des plug-ins qui seront également chargés au démarrage de l'application. La déclaration d'un plug-in est réalisée à la fin du fichier de configuration de l'application.

```
<!-- connexion a la source de données -->
<plug-in className="chatbetaboutique.PluginDataSource" />
```

Avec cette déclaration, nous précisons qu'un plug-in nommé *PluginDataSource*, présent dans le paquetage *chatbetaboutique* sera chargé au démarrage de l'application.

Par souplesse lors du développement, nous allons créer deux constantes au sein du fichier de configuration */WEB-INF/web.xml*.

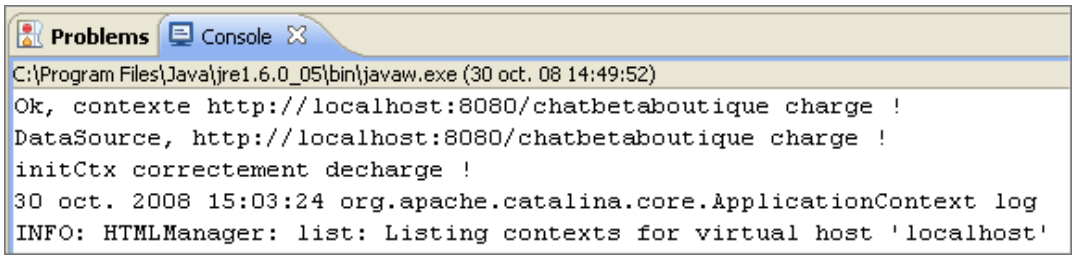
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
  <display-name>Application chatbetaboutique</display-name>
  <!-- paramètres globaux accessibles dans les JSP et Servlets -->
  <!-- nom du connecteur jdbc -->
  <context-param>
    <param-name>connecteurjdbc</param-name>
    <param-value>jdbc_chatbetaboutiquemysql</param-value>
  </context-param>
  <!--  url pour accéder à l'application -->
  <context-param>
        <param-name>urlapplication</param-name>
        <param-value>http://localhost:8080/chatbetaboutique</param-value>
  </context-param>
  ...
</web-app>
```

La classe *chatbetaboutique.PluginDataSource* possède une syntaxe déjà abordée dans le chapitre consacré aux bases de données. Ce plug-in permet uniquement de charger le pool de connexion nommé *datasource* au démarrage de l'application Struts.

```
package chatbetaboutique;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.sql.DataSource;
import org.apache.struts.action.ActionServlet;
import org.apache.struts.action.PlugIn;
import org.apache.struts.config.ModuleConfig;

public class PluginDataSource implements PlugIn
{
```

```
    //fonction appelée lors de la création du plugin
    public void init(ActionServlet servlet,ModuleConfig
moduleConfig) throws ServletException
    {
        //récupérer les paramètres présents dans le
fichier de configuration web.xml
        String nomprojet=(String)servlet.getServlet
Context().getInitParameter("urlapplication");
        String connecteurjdbc=(String)servlet.getServlet
Context().getInitParameter("connecteurjdbc");
        //initialiser le context

        try
        {
            //initialiser le context
            initCtx=new InitialContext();
                        if(initCtx==null)throw new Exception
("Il n'y a pas de contexte !");
            else
            {
                System.out.println("Ok, contexte "+nomprojet+" charge !");
            }

            //se connecter au JNDI mysql
            Context envCtx=(Context) initCtx.lookup("java:comp/env");
            DataSource ds=(DataSource) envCtx.lookup(connecteurjdbc);
            if(ds==null)throw new Exception ("Il n'y a pas de DataSource !");
            else
            {
                System.out.println("DataSource, "+nomprojet+" charge !");
            }
                        //stocker la DataSource dans un attribut
du context de la servlet application
            ServletContext servletContext=servlet.getServletContext();
            servletContext.setAttribute("datasource",ds);
        }
        catch(Exception e)
        {
            throw new ServletException(e.getMessage());
        }
        finally
        {
            try
            {
                //fermer le context
                if(initCtx!=null)initCtx.close();
                System.out.println("initCtx correctement decharge !");
            }
            catch(Exception e)
            {
                System.out.println("Erreur lors de initCtx !");
            }
        }
    }

    //fonction appelée lors de la destruction du plugin
    public void destroy(){}

//fin de la classe
}
```

Nous pouvons tester le fonctionnement du plug-in en démarrant notre application de gestion du chat.



5. Liste des utilisateurs

Nous allons commencer la première partie de cette application de gestion du chat par le service qui permet de lister tous les utilisateurs de l'application. Pour rappel, les utilisateurs sont définis dans la table *chatbetaboutique.utilisateur* qui possède la structure suivante :

Pour la réalisation des services avec Struts nous allons procéder méthodiquement par étape :

- Définition du JavaBean associé à la classe à gérer.
- Définition du formulaire de JavaBean dans le fichier de configuration *struts-config.xml*.
- Définition du routage du service en cours dans le fichier de configuration *struts-config.xml*.
- Définition des validations si nécessaire.
- Codage de la classe de gestion des actions.
- Codage du modèle si nécessaire.
- Codage de la vue JSP.

Pour des raisons de confort de programmation et de maintenance future, nous allons développer la totalité de l'application avec des *DynaForms* Struts et le fichier de validation des entrées *validation.xml*.

a. Création du JavaBean

Nous commençons par créer un nouveau paquetage nommé *boiteoutils*. Ce paquetage Java va contenir par la suite les classes JavaBean ainsi que si besoin certaines classes statiques. Au sein de ce paquetage, une nouvelle classe JavaBean nommée *Utilisateur* est créée en correspondance avec les champs du diagramme de classes UML et de la table.

```
package boiteoutils;
import java.io.Serializable;

public class Utilisateur implements Serializable
{
   //java 5.0 pour avoir un identifiant unique de la sérialisation
   private static final long serialVersionUID = 1L;

   //variables de classe
   private String id_utilisateur=null;
   private String pseudonyme=null;
   private String motdepasse=null;
```

```
   private String nom=null;
   private String prenom=null;
   private int autorisation=0;
   private String image=null;

public Utilisateur() {
}
public int getAutorisation() {
   return autorisation;
}
...
//fin de la classe
}
```

b. Déclaration du JavaBean de formulaire

Le JavaBean *Utilisateur* est correctement codé, l'étape suivante consiste à déclarer ce JavaBean de formulaire au sein du fichier de configuration de l'application Struts *struts-config.xml*.

```
<form-beans>
   <!-- ========== Formulaire de gestion des utilisateur ========== -->
      <form-bean name="FormUtilisateur"
type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="utilisateur" type="boiteoutils.Utilisateur" />
   </form-bean>
</form-beans>
```

Le JavaBean de formulaire est déclaré sous forme d'un *DynaForms* avec pour nom *FormUtilisateur* et pour propriété un objet *utilisateur* instance de la classe *boiteoutils.Utilisateur*.

c. Définition du routage

Le JavaBean est correctement configuré pour Struts, nous pouvons désormais passer à la déclaration du routage de l'application. Cette étape est également réalisée dans le fichier de configuration *struts-config.xml* par l'intermédiaire de la balise <*action/*>.

```
<action-mappings>
    <!-- ================= UTILISATEUR ==================== -->
    <!-- ===action qui permet de lister les utilisateurs=== -->
        <action
            path="/listeutilisateur"
            name="FormUtilisateur"
            scope="request"
            validate="false"
            input="accueil"
            type="chatbetaboutique.GestionUtilisateurAction"
            parameter="listeutilisateur"
    >
<forward name="listeutilisateur"
path="/vues/utilisateur/listeutilisateur.jsp" redirect="false"/>
        </action>
</action-mappings>
```

Cette déclaration d'action sera associée à une classe de type *MappingDispatchAction*. Le paramètre *path* permet d'indiquer que cette action sera déclenchée avec un lien nommé *listeutilisateur*. Le formulaire JavaBean associé à cette action est *FormUtilisateur* déclaré plus haut par l'intermédiaire de la balise <*form-bean/*>. Le paramètre *scope* indique que les paramètres auront une portée valable durant le temps de la requête utilisateur. L'attribut *validate="false"* indique qu'il n'y aura pas de validation des saisies du formulaire avec ce lien. Le paramètre *input="accueil"* sera déclenché en cas d'erreur dans l'action ou d'erreur de saisie. Cette action nommée *accueil* est une action globale déclarée comme ceci au sein du fichier *struts-config.xml* :

```
<!-- ========================================== Global Forward
Definitions -->
<global-forwards>
        <forward name="accueil" path="/index.jsp" redirect="false"/>
</global-forwards>
```

Cette action pourra ainsi être référencée depuis n'importe quelle action Struts. Le paramètre *type* permet d'indiquer la classe d'action qui sera chargée de traiter et déclarer l'action associée au chemin */listeutilisateur*. Enfin, le paramètre *parameter* permet avec la classe *MappingDispatchAction* de préciser la méthode de la classe d'action qui sera déclenchée. Dans notre cas, un lien sur */listeutilisateur* va déclencher la méthode *listeutilisateur* de la classe *chatbetaboutique.GestionUtilisateurAction*. Enfin, la redirection déclarée avec la balise *<forward/>* permet d'indiquer la vue qui sera affichée en retour du traitement.

d. Codage de la classe de gestion des actions

La déclaration de l'action a été effectuée dans l'étape précédente. Nous pouvons réaliser le codage de la classe d'action nommée *GestionUtilisateurAction* présente dans le paquetage *chatbetaboutique*.

```
package chatbetaboutique;

import org.apache.struts.action.*;
import org.apache.struts.actions.MappingDispatchAction;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.*;

public class GestionUtilisateurAction extends MappingDispatchAction{

    //afficher la liste des utilisateurs
    public ActionForward listeutilisateur(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
    {
        System.out.println("Liste des utilisateurs");
        return null;
    }

//fin de la classe
}
```

Pour le moment cette action permet uniquement d'afficher un message dans la console Java. Nous allons recharger le contexte de l'application et réaliser un lien vers cette action Struts pour tester dans un premier temps cette première déclaration. Pour cela, nous devons réaliser le lien suivant et le déclencher dans la page JSP */vues/outils/navigations.jspf*.

```
<table border="0" cellspacing="0" cellpadding="0"
name="menugauche" id="menugauche" style="background:
url('img/fondmenu.gif') repeat-x;"  height="500">
<tr>
  <td width="100%" valign="top"><br/><span style="margin-left:30px;">
<b>ADMINISTRATION</b></span>
   <ul class="menucategorie">
       <li><a href="listeutilisateur.do"><img src="img/fleche.gif"
border="0" align="absmiddle"/> Gestion des utilisateurs</a></li>
   </ul>
  </td>
</tr>
</table>
```

Le fonctionnement de l'action est opérationnel, nous pouvons passer à son codage qui doit permettre de lire et de retourner la liste des utilisateurs de l'application du chat de *BetaBoutique*.

Pour l'affichage des informations, nous ne réaliserons pas de tri, recherche et pagination afin de simplifier le code et les explications associées.

```
package chatbetaboutique;

import modele.UtilisateurModele;
import org.apache.struts.action.*;
import org.apache.struts.actions.MappingDispatchAction;
import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
```

```
import javax.servlet.http.*;
import javax.sql.DataSource;

public class GestionUtilisateurAction extends MappingDispatchAction{

    //variables de la classe
    DataSource ds=null;

    //afficher la liste des utilisateurs
    public ActionForward listeutilisateur(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
    {
        //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
        ds=(DataSource)servlet.getServletContext().getAttribute
("datasource");
        //créer le modèle
        UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
        //fermer la datasource
        this.ds=null;

        //retourner la liste des utilisateur
        ArrayList listeutilisateur=(ArrayList)
utilisateurmodele.getListeUtilisateur();
        //retourner la liste des utilisateurs
        request.setAttribute("listeutilisateur",listeutilisateur);
        //vider par sécurité
        listeutilisateur=null;

        //retourner sur la page d'affichage des utilisateurs
        return mapping.findForward("listeutilisateur");
    }

//fin de la classe
}
```

Le code est très simple et nous remarquons alors tout l'intérêt d'utiliser le modèle MVC II. Nous commençons par déclarer la connexion à la base de données (*ds*), ensuite nous instancions un objet qui représente le modèle (*utilisateurmodele*). Nous déclenchons la fonction du modèle qui va retourner tous les objets de type *Utilisateur* sans se soucier du traitement (*getListeUtilisateur()*), nous postons cette collection dans la requête (*request.setAttribute()*) et nous nous dirigeons vers la vue qui va afficher les données (*mapping.findForward()*).

e. Codage du modèle

Le traitement de l'action est désormais réalisé. Nous pouvons passer au codage du modèle. Pour cela, nous allons créer un nouveau paquetage nommé *modele* et la classe *UtilisateurModele*. Cette classe possède pour le moment une seule méthode qui permet de lire la liste des utilisateurs de la base de données et de les insérer dans une collection de type *ArrayList*.

```
package modele;

import boiteoutils.*;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.sql.DataSource;
import org.apache.commons.dbutils.BeanProcessor;
import java.util.ArrayList;

public class UtilisateurModele
{
    //variables de classe
    DataSource ds=null;
    Connection connection=null;
    ResultSet rs=null;
    //liste des objets
    ArrayList<Utilisateur> listeutilisateur=new ArrayList<Utilisateur>();
```

```
/*************************************************************
 * constructeur
*************************************************************/
public UtilisateurModele(DataSource ds)
{
    //récupérer le DataSource de la servlet
    this.ds=ds;
}

/*************************************************************
 * liste des utilisateurs
*************************************************************/
public ArrayList getListeUtilisateur()
{
    //statement
    PreparedStatement requete=null;

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        //enregistrements
        requete=connection.prepareStatement("SELECT * FROM
utilisateur ORDER BY pseudonyme,id_utilisateur");
        rs=requete.executeQuery();
        //exécuter la requête
        if(rs!=null)
        {
            //utilisation de la librairie DbUtils qui permet
de transformer un ResultSet en Objet Java
            BeanProcessor bp=new BeanProcessor();
                        listeutilisateur =
(ArrayList<Utilisateur>)bp.toBeanList(rs,Utilisateur.class);
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe UtilisateurModele.java
fonction getListeUtilisateurModele");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
UtilisateurModele.java fonction getListeUtilisateurModele");
        }
    }

    //retourner la liste des utilisateurs
    return listeutilisateur;
}
//fin de la classe
}
```

La fonction *getListeUtilisateur()* utilise la bibliothèque *DbUtils* afin de transformer le *ResultSet* réponse en une collection d'objets de la classe *Utilisateur*. Cette collection est ensuite retournée à l'action qui va elle-même réaliser le routage vers la page JSP d'affichage des données.

Nous utilisons également la classe statique *boiteoutils.OutilsBaseDeDonnees* pour la gestion du SGBD.

f. Codage de la vue JSP

La dernière étape consiste à coder la vue JSP. Pour cela, nous allons utiliser les bibliothèques de tags livrées en standard avec Struts. L'inclusion de ces bibliothèques est réalisée dans le fichier */vues/outils/entete.jspf*.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html>
<head>
<title>Administration - BetaBoutique</title>
...
```

La vue JSP */vues/utilisateur/listeutilisateur.jsp* est codée en HTML avec une partie de code Java qui permet d'afficher les données. Nous pourrions tout à fait utiliser des balises 100% HTML en utilisant les taglibs Struts.

```
<%@ page import="java.util.ArrayList" %>
<%@ page import="boiteoutils.Utilisateur" %>
<%
//liste des utilisateurs
ArrayList listeutilisateur=(ArrayList)request.getAttribute
("listeutilisateur");
%>
<!--  inclure l'entete de la page -->
<%@ include file="../outils/entete.jspf" %>

<br/><div class="titre"><a href="listeutilisateur.do"><img
src="img/utilisateur.jpg" border="0" align="absmiddle"/> Liste
des utilisateurs</a></div>
<div class="titreaction"><a href="creerutilisateur.do">Cr&eacute;er
un utilisateur</a></div>
<table border="0" id="tableaubordure" cellspacing="0" cellpadding="0">
<tr align="center" class="entetetableau">
<td>ID</td>
<td>Pseudonyme</td>
<td>Mot de passe</td>
<td>Nom</td>
<td>Pr&eacute;nom</td>
<td>Autorisation</td>
<td>Image</td>
<td colspan="3" width="130">Gestion</td>
</tr>
<%
for(int i=0;i<listeutilisateur.size();i++)
{
    //récupérer l'objet dans la liste
    Utilisateur utilisateur=(Utilisateur)listeutilisateur.get(i);
    if(i%2==0)out.println("<tr align=\"center\" class=\"ligneclaire\">");
    else out.println("<tr align=\"center\" class=\"lignefoncee\">");
    out.println("<td>"+utilisateur.getId_utilisateur()+"</td>");
    out.println("<td>"+utilisateur.getPseudonyme()+"</td>");
    out.println("<td>"+utilisateur.getMotdepasse()+"</td>");
    out.println("<td>"+utilisateur.getNom()+"</td>");
    out.println("<td>"+utilisateur.getPrenom()+"</td>");
    if(utilisateur.getAutorisation()==1)
    {
        out.println("<td><img src=\"img/actif.gif\" alt=\"Actif\"
title=\"Actif\"/></td>");
    }
    else
    {
        out.println("<td><img src=\"img/inactif.gif\" alt=\"Inactif\"
title=\"Inactif\"/></td>");
    }
    out.println("<td>"+utilisateur.getImage()+"</td>");
    out.println("<td><a href='consulterutilisateur.do?
id_utilisateur="+utilisateur.getId_utilisateur()+"'><img
src=\"img/consulter.png\" border=\"0\" align=\"absmiddle\"
title=\"Consulter\"/></a></td>");
    out.println("<td><a href='modifierutilisateur.do?
id_utilisateur="+utilisateur.getId_utilisateur()+"'><img
```

```
src=\"img/modifier.png\" border=\"0\" align=\"absmiddle\"
title=\"Modifier\"/></a></td>");
    out.println("<td><a href='javascript:confirmerSuppressionUtilisateur
("+utilisateur.getId_utilisateur()+",\""+utilisateur.getPseudonyme()+"\");'>
<img src=\"img/supprimer.png\" border=\"0\" align=\"absmiddle\"
title=\"Supprimer\"/></a></td>");
    out.println("</tr>");
}
%>
</table>
<%@ include file="../outils/piedpage.jspf" %>
```

Notre premier service développé entièrement en Struts est terminé. Chaque partie est correctement découpée en respectant le pattern MVC II. Nous allons dans la suite de ce chapitre présenter la technique pour consulter la fiche détaillée d'un utilisateur, le formulaire de création et de modification d'un utilisateur avec les validations XML et l'action de suppression. Ces techniques seront alors utilisées à l'identique pour réaliser la partie gestion des salons du chat.

6. Consultation de la fiche utilisateur

L'action de consultation est la plus simple à développer. Le JavaBean *Utilisateur* est déjà est place ainsi que sa définition. Nous pouvons passer directement à l'étape de déclaration de l'action dans le fichier *struts-config.xml*.

```
<!-- ===action qui permet de consulter la fiche d'un utilisateur=== -->
<action
    path="/consulterutilisateur"
    name="FormUtilisateur"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="chatbetaboutique.GestionUtilisateurAction"
    parameter="consulterutilisateur"
    >
<forward name="consulterutilisateur" path="/vues/utilisateur/
consulterutilisateur.jsp" redirect="false"/>
</action>
```

La déclaration est pratiquement identique à celle de la liste des utilisateurs. Nous retrouvons le paramètre *path* qui permet de définir le nom de l'action qui va être déclenché : */consulterutilisateur.do*. Le JavaBean de formulaire est toujours le même, il n'y a pas de validation de saisie sur une consultation et c'est la méthode *consulterutilisateur()* de la classe *chatbetaboutique.GestionUtilisateurAction* qui sera déclenchée. Enfin, une seule définition de redirection est définie, elle correspond à la vue JSP pour l'affichage.

Le code de l'action de consultation est assez simple. Le paramètre passé dans le lien est récupéré et testé. Cet attribut correspond au numéro de l'utilisateur à consulter. Ensuite, l'action déclenche le modèle afin de récupérer la totalité de l'objet associé à cet identifiant. Le code est ajouté à la suite de la classe *GestionUtilisateurAction*.

```
//consulter un utilisateur
public ActionForward consulterutilisateur(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
{
    ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
    //créer le modèle
    UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
```

```
    //fermer la datasource
    this.ds=null;

    //récupérer l'id de l'utilisateur envoyé
    String id_utilisateur=(String)request.getParameter("id_utilisateur");
    //récupérer l'objet dans la base de données
    if(id_utilisateur!=null && !id_utilisateur.equals(""))
    {
    Utilisateur utilisateur=(Utilisateur)utilisateurmodele.getUtilisateur
(id_utilisateur);
    request.setAttribute("utilisateur",utilisateur);
    //redirection vers la page de consultation
    return mapping.findForward("consulterutilisateur");
    }

    //en cas d'erreur retourner sur la page d'accueil
    return mapping.findForward("accueil");
}
```

La classe modèle *UtilisateurModele* possède désormais une nouvelle méthode qui permet de récupérer une image de l'objet *utilisateur* sauvegardé en base.

```
/*********************************************************
 * récupérer l'utilisateur indiqué
*********************************************************/
public Utilisateur getUtilisateur(String id_utilisateur)
{
    //créer un objet utilisateur
    Utilisateur utilisateur=new Utilisateur();
    //statement
    PreparedStatement requete=null;

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        //enregistrements
        requete=connection.prepareStatement("SELECT * FROM
utilisateur WHERE id_utilisateur=?");
        requete.setString(1,(String)id_utilisateur);
        rs=requete.executeQuery();
        //exécuter la requête
        if(rs!=null)
        {
            if(rs.next())
        {
    //utilisation de la librairie DbUtils qui permet de
transformer un ResultSet en Objet Java
            BeanProcessor bp=new BeanProcessor();
            utilisateur = (Utilisateur)bp.toBean(rs, Utilisateur.class);
            }
        }
    }
    catch(Exception e)
    {
    System.out.println("Erreur dans la classe
UtilisateurModele.java fonction getUtilisateur");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
```

```
UtilisateurModele.java fonction getUtilisateur");
        }
    }

    //retourner l'objet utilisateur
    return utilisateur;
}
```

L'action de consultation est quasiment terminée, il ne reste plus que le code de la vue */vues/utilisateur/consulterutilisateur.jsp*.

```
<%@ page import="boiteoutils.Utilisateur" %>
<%
//objet utilisateur
Utilisateur utilisateur=(Utilisateur)request.getAttribute("utilisateur");
%>
<!--  inclure l'en-tete de la page -->
<%@ include file="../outils/entete.jspf" %>

<br/><div class="titre"><a href="listeutilisateur.do"><img
src="img/utilisateur.jpg" border="0" align="absmiddle"/> Liste
des utilisateurs</a></div>

<table border="0" id="tableau" cellpadding="0" cellspacing="0" width="50%">
<tr><td>Id</td><td><%= utilisateur.getId_utilisateur() %></td></tr>
<tr><td>Pseudonyme</td><td><%= utilisateur.getPseudonyme() %></td></tr>
<tr><td>Mot de passe</td><td><%= utilisateur.getMotdepasse() %></td></tr>
<tr><td>Nom</td><td><%= utilisateur.getNom() %></td></tr>
<tr><td>Pr&eacute;nom</td><td><%= utilisateur.getPrenom() %></td></tr>
<tr><td>Autorisation</td><td><%= utilisateur.getAutorisation() %></td></tr>
<tr><td>Image</td><td><%= utilisateur.getImage() %></td></tr>
<tr><td align="center" colspan="2"><html:submit value="Modifier"
styleClass="bouton"/></td></tr>
</table>

<!--  inclure le pied de page -->
<%@ include file="../outils/piedpage.jspf" %>
```

Le service de consultation est totalement opérationnel. Nous allons cependant ajouter un formulaire dans la page JSP de consultation afin de réaliser un lien vers le formulaire de modification par l'intermédiaire du bouton *Modifier*. La balise Struts *<html:form/>* permet d'ajouter un formulaire à la page. Cependant l'ajout de cette balise provoque une erreur. En effet, l'action */modifierutilisateur* n'est pas encore déclarée dans le fichier de configuration *struts-config.xml*.

```
<%@ page import="boiteoutils.Utilisateur" %>
<%
//objet utilisateur
```

```
Utilisateur utilisateur=(Utilisateur)request.getAttribute("utilisateur");
%>
<!--  inclure l'entete de la page -->
<%@ include file="../outils/entete.jspf" %>
<br/><div class="titre"><a href="listeutilisateur.do"><img src="img/
utilisateur.jpg" border="0"
align="absmiddle"/> Liste des utilisateurs</a></div>
<html:form action="/modifierutilisateur" method="POST">
<table border="0" id="tableau" cellpadding="0" cellspacing="0" width="50%">
...
</table>
<html:hidden property="id_utilisateur" value="<%= utilisateur.
getId_utilisateur() %>"/>
</html:form>
<!--  inclure le pied de page -->
<%@ include file="../outils/piedpage.jspf" %>
```

7. Création d'un nouvel utilisateur

Le formulaire de création est affiché par l'intermédiaire d'une page JSP simple. Les parties création et modification sont réalisées en deux étapes. La première étape consiste à afficher le formulaire avant validation. Cette action déclarée dans le fichier de configuration *struts-config.xml* permet de réaliser une redirection simple vers le formulaire de création.

```
<!-- action qui permet d'afficher le formulaire de
création d'un utilisateur -->
<action
    path="/creerutilisateur"
    name="FormUtilisateur"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/vues/utilisateur/creerutilisateur.jsp">
</action>
```

Le formulaire de création */vues/utilisateur/creerutilisateur.jsp* reprend les différents champs de la table *Utilisateur*.

```
<!--  inclure l'en-tete de la page -->
<%@ include file="../outils/entete.jspf" %>

<br/><div class="titre"><a href="listeutilisateur.do"><img
src="img/utilisateur.jpg" border="0" align="absmiddle"/> Liste
des utilisateurs</a></div>

<html:form action="/validercreerutilisateur" method="POST">
<table border="0" id="tableau" cellpadding="0" cellspacing="0" width="50%">
<tr><td>Pseudonyme</td><td><html:text property="utilisateur.pseudonyme"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Mot de passe</td><td><html:text property="utilisateur.motdepasse"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Nom</td><td><html:text property="utilisateur.nom"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Pr&eacute;nom</td><td><html:text property="utilisateur.prenom"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Autorisation</td><td><html:checkbox
property="utilisateur.autorisation" styleClass="input"
errorStyleClass="inputerreur"/></td></tr>
<tr><td>Image</td><td><html:text property="utilisateur.image"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td align="center" colspan="2"><html:submit value="Envoyer"
styleClass="bouton"/></td></tr>
</table>
</html:form>

<!--  inclure le pied de page -->
<jsp:include page="../outils/piedpage.jspf" />
```

Avec Struts, toutes les actions et JavaBean de formulaire sont liés. Dans la page ci-dessus, nous utilisons le taglib *<html:form/>* avec le paramètre *action* qui déclenche l'action */validercreerutilisateur*. Cette action doit être déclarée dans le fichier *struts-config.xml* afin que la page JSP */vues/utilisateur/creerutilisateur.jsp* puisse être compilée.

```
<!-- action qui va valider la création d'un utilisateur -->
<action
    path="/validercreerutilisateur"
    name="FormUtilisateur"
    scope="request"
    validate="true"
    input="/creerutilisateur.do"
    type="chatbetaboutique.GestionUtilisateurAction"
    parameter="validercreerutilisateur">
<forward name="listeutilisateur" path="/listeutilisateur.do"
redirect="false"/>
</action>
```

Afin d'afficher les messages d'erreur et de succès, nous allons ajouter quelques lignes à la fin du fichier */vues/outils/entete.jspf*.

```
...
<!-- COLONNE CENTRE -->
<td valign="top" width="100%" style="padding-left:5px">

      <!-- AFFICHER LES MESSAGES DE SUCCES ET ERREUR -->
      <div id="message_erreur"><html:errors/></div>
      <div id="message_information">
        <html:messages id="message" message="true">
            <ul><li><bean:write name="message"/></li></ul>
        </html:messages>
      </div>
```

Le formulaire de création est opérationnel, nous pouvons passer à la validation des saisies. Dans la déclaration de l'action */validercreerutilisateur* qui est déclenchée suite à la validation du formulaire de saisie, nous avons déclaré le paramètre *validate* à *true*. Les contrôles de saisies seront donc réalisés en correspondance avec les déclarations du fichier *validation.xml*. En cas d'erreur, c'est l'action déclarée avec le paramètre *input* qui sera appelée.

Dans notre cas, le formulaire de création est rappelé sans perdre les saisies. Nous déclarons en début de fichier */WEB-INF/validation.xml* deux constantes globales qui correspondent à la syntaxe respectivement du mot de passe et du pseudonyme utilisateur. Ensuite, nous déclarons une référence au formulaire utilisateur *<form name="FormUtilisateur"/>* en correspondance avec le nom dans le fichier *struts-config.xml <form-bean name="FormUtilisateur"/>*. Les validations de saisies sont très pointues, les syntaxes sont contrôlées par rapport à des expressions régulières, des présences et des tailles.

```
<!DOCTYPE form-validation PUBLIC
      "-//Apache Software Foundation//DTD Commons Validator Rules
Configuration 1.1.3//EN"
      "http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">
<form-validation>
   <global>
      <constant>
         <constant-name>motdepasse</constant-name>
         <constant-value>^[0-9a-zA-Z]*$</constant-value>
      </constant>
      <constant>
         <constant-name>pseudonyme</constant-name>
         <constant-value>^[0-9a-zA-Z\-]{4,20}$</constant-value>
      </constant>
      <constant>
         <constant-name>autorisation</constant-name>
         <constant-value>^[0|1]{1}$</constant-value>
      </constant>
   </global>
   <formset>
      <!-- ========== Formulaire authentification utilisateur ========== -->
      <form name="FormUtilisateur">
         <field property="utilisateur.pseudonyme"
depends="required,minlength,maxlength,mask">
            <arg0 key="pseudonymeutilisateur"/>
            <arg1 name="minlength" key="${var:minlength}" resource="false"/>
```

```
            <arg1 name="maxlength" key="${var:maxlength}" resource="false"/>
            <var>
               <var-name>minlength</var-name>
               <var-value>4</var-value>
            </var>
            <var>
               <var-name>maxlength</var-name>
               <var-value>20</var-value>
            </var>
            <var>
               <var-name>mask</var-name>
               <var-value>${pseudonyme}</var-value>
            </var>
         </field>
         ...
      </form>
   </formset>
</form-validation>
```

Les champs *peusdonyme*, *motdepasse* et *autorisation* sont analysés de façon précise alors que les champs *nom* et *prenom* doivent uniquement être renseignés. La balise *<arg0/>* permet de faire le lien avec les noms dans le fichier *ressources.properties*. Comme indiqué précédemment, les validations et messages d'erreurs sont associés au fichier de propriétés qui est situé pour notre projet dans le paquetage *ressources* et qui a pour nom *ressources.properties*. Voici son contenu :

```
# -- standard errors --
errors.header=<UL>
errors.prefix=<LI>
errors.suffix=</LI>
errors.footer=</UL>
# -- messages --
errors.required=Attention, le champ [{0}] doit être renseigné !
errors.minlength=Attention, le champ [{0}] doit avoir au moins
{1} caractères !
errors.maxlength=Attention, le champ [{0}] ne peut avoir plus de
{1} caractères !
errors.invalid=Attention, le champ [{0}] est incorrect!
errors.date=Attention, le champ [{0}] n'est pas une date valide !
errors.byte=Attention, le champ [{0}] doit être un octet !
errors.date=Attention, le champ [{0}] doit être une date !
errors.double=Attention, le champ [{0}] doit être un double !
errors.float=Attention, le champ [{0}] doit être un réel !
errors.integer=Attention, le champ [{0}] doit être un entier !
errors.long=Attention, le champ [{0}] doit être un entier long !
errors.short=Attention, le champ [{0}] doit être un entier court !
errors.range=Attention, le champ [{0}] doit être dans l'intervalle
{1} et {2} !
errors.creditcard=Attention, le champ [{0}] n'est pas un numéro
de carte valide !
errors.email=Attention, le champ [{0}] n'est pas une adresse
électronique valide !
# -- gestion des utilisateurs --
pseudonymeutilisateur=Pseudonyme
motdepasseutilisateur=Mot de passe
nomutilisateur=Nom
prenomutilisateur=Prénom
autorisationutilisateur=Autorisation
imageutilisateur=Image
```

Nous pouvons vérifier les saisies en réalisant différents tests précis sur le formulaire de création.

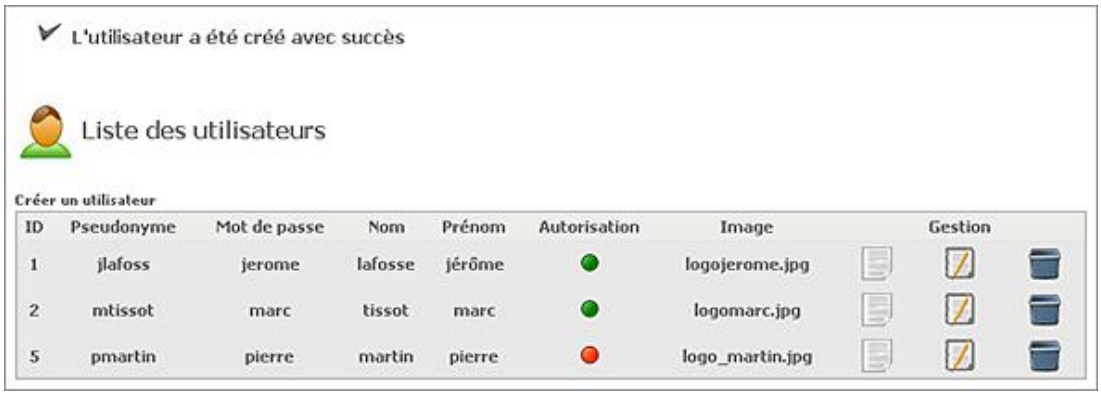
Le service de création d'un nouvel utilisateur est presque terminé. Il nous reste à développer la méthode *validercreerutilisateur()* de la classe *GestionUtilisateurAction* ainsi que la méthode associée au modèle.

```
//valider la création d'un utilisateur
public ActionForward validercreerutilisateur(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
{
        //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
        ds=(DataSource)servlet.getServletContext().getAttribute
("datasource");
        //créer le modèle
        UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
        //fermer la datasource
        this.ds=null;
        //récupérer le bean formulaire
        DynaActionForm FormUtilisateur=(DynaActionForm)form;
        //créer un objet utilisateur directement à partir des saisies
        Utilisateur utilisateur=(Utilisateur)FormUtilisateur.get
("utilisateur");

        //rendre l'objet persistant dans la base de données
        int resinstruction=utilisateurmodele.creerUtilisateur(utilisateur);

        //en cas d'erreur avec la base de données
        if (resinstruction!=1)
        {
            //ajouter un message d'erreur
            ActionMessages erreurs=new ActionErrors();
            erreurs.add("message", new ActionMessage
("erreurs.creationutilisateur"));
            saveErrors(request,erreurs);
        }
        //tout est OK, enregistrer l'utilisateur est créé
        else
        {
            //toutes les vérifications sont correctes
            ActionMessages messages=new ActionMessages();
            messages.add("message", new ActionMessage
("succes.creationutilisateur"));
```

```
            saveMessages(request,messages);
            //retourner sur la page de listing des utilisateurs
            return mapping.findForward("listeutilisateur");
        }

        //en cas d'erreur retour sur la page d'accueil
        return mapping.findForward("accueil");
}
```

L'action de validation de la création permet d'instancier un objet *utilisateur* à l'image des saisies de l'utilisateur en une seule ligne. Ensuite, cet objet est rendu persistant par le modèle. Un message dynamique de succès (ou d'erreur) est alors inséré dans la requête après lecture des valeurs dans le fichier de propriétés.

```
/*********************************************************
 * créer un utilisateur
*********************************************************/
public int creerUtilisateur(Utilisateur utilisateur)
{
   int resinstruction=0;
   PreparedStatement requete=null;

   try
   {
      //ouvrir une connexion
      connection=ds.getConnection();
      requete=connection.prepareStatement("INSERT INTO
utilisateur(pseudonyme,motdepasse,nom,prenom,autorisation,image) VALUES
(?,?,?,?,?,?)");
      requete.setString(1,(String)utilisateur.getPseudonyme());
      requete.setString(2,(String)utilisateur.getMotdepasse());
      requete.setString(3,(String)utilisateur.getNom());
      requete.setString(4,(String)utilisateur.getPrenom());
      requete.setInt(5,(Integer)utilisateur.getAutorisation());
      requete.setString(6,(String)utilisateur.getImage());
      resinstruction=requete.executeUpdate();
   }
   ...
}
```



8. Modification de la fiche utilisateur

Le formulaire de modification est toujours l'élément le plus complexe à développer en programmation Web. Certaines informations doivent être testées, des messages d'erreurs sont affichés en conséquence, des redirections sont réalisées en fonction des traitements et surtout les saisies doivent être conservées en cas d'erreur.

Comme dans le cas de la création, la validation est réalisée avec deux étapes. La première consiste à lire les données de la fiche à modifier et la seconde permet la modification de celle-ci. Nous commençons par la réalisation du routage avec la définition des deux actions nécessaires.

```
<!-- ===action qui permet de modifier un utilisateur
en récupérant ses valeurs=== -->
<action
    path="/modifierutilisateur"
```

```
    name="FormUtilisateur"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="chatbetaboutique.GestionUtilisateurAction"
    parameter="modifierutilisateur"
    >
<forward name="modifierutilisateur"
path="/vues/utilisateur/modifierutilisateur.jsp" redirect="false"/>
</action>
<!-- ===action qui permet de valider la modification d'un utilisateur=== -->
<action
    path="/validermodifierutilisateur"
    name="FormUtilisateur"
    scope="request"
    validate="true"
    input="/modifierutilisateur.do"
    type="chatbetaboutique.GestionUtilisateurAction"
    parameter="validermodifierutilisateur"
>
<forward name="modifierutilisateur"
path="/modifierutilisateur.do" redirect="false"/>
<forward name="listeutilisateur"
path="/listeutilisateur.do" redirect="false"/>
</action>
```

L'action Struts utilise la fonction *getUtilisateur()* du modèle précédemment développé et le JavaBean de formulaire *FormUtilisateur*.

```
//afficher le formulaire en modification pour l'utilisateur indiqué
public ActionForward modifierutilisateur(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
{
    //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
    ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
    //créer le modèle
    UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
    //fermer la datasource
    this.ds=null;
    //récupérer l'id de l'utilisateur
    String id_utilisateur=(String)request.getParameter("id_utilisateur");
    //le bean formulaire
    DynaActionForm FormUtilisateur=(DynaActionForm)form;
    //si validation du formulaire mais qu'il y a
une erreur, récupérer les informations
    if(request.getParameter("utilisateur.id_utilisateur")!=null)
    {
        id_utilisateur=request.getParameter("utilisateur.id_utilisateur");
    }

    if(id_utilisateur!=null && !id_utilisateur.equals(""))
    {
    Utilisateur utilisateur=(Utilisateur)utilisateurmodele.getUtilisateur
(id_utilisateur);
    //copier l'intégralité de l'objet dans le JavaBean de formulaire
    FormUtilisateur.set("utilisateur", utilisateur);
    //redirection vers la page de modification
    return mapping.findForward("modifierutilisateur");
    }
    //en cas d'erreur retourner sur la page d'accueil
    return mapping.findForward("accueil");
}
```

Enfin, la vue JSP */vues/utilisateur/modifierutilisateur.jsp* permet d'afficher les données du JavaBean de formulaire en correspondance avec l'attribut *id* de l'utilisateur (*<html:hidden/>*).

```
<!--  inclure l'en-tete de la page -->
<%@ include file="../outils/entete.jspf" %>
```

```
<br/><div class="titre"><a href="listeutilisateur.do"><img
src="img/utilisateur.jpg" border="0" align="absmiddle"/> Liste
des utilisateurs</a></div>

<html:form action="/validermodifierutilisateur" method="POST">
<table border="0" id="tableau" cellpadding="0" cellspacing="0" width="50%">
<tr><td>Pseudonyme</td><td><html:text property="utilisateur.pseudonyme"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Mot de passe</td><td><html:text property="utilisateur.motdepasse"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Nom</td><td><html:text property="utilisateur.nom"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Pr&eacute;nom</td><td><html:text property="utilisateur.prenom"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Autorisation</td><td><html:text property="utilisateur.autorisation"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Image</td><td><html:text property="utilisateur.image"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td align="center" colspan="2"><html:submit value="Envoyer"
styleClass="bouton"/></td></tr>
</table>
<html:hidden property="utilisateur.id_utilisateur"/>
</html:form>

<!--  inclure le pied de page -->
<jsp:include page="../outils/piedpage.jspf" />
```

Il ne reste plus qu'à développer l'action *validermodifierutilisateur()* de la classe *GestionUtilisateurAction* ainsi que la fonction du modèle, afin de réaliser la modification en base.

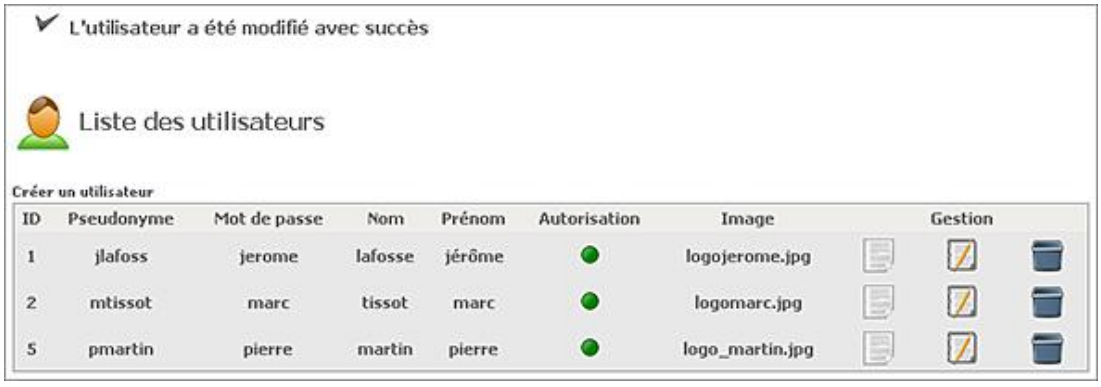
```
//valider la modification de l'utilisateur
public ActionForward validermodifierutilisateur(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
{
    //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
    ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
    //créer le modèle
    UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
    //fermer la datasource
    this.ds=null;
    //récupérer le bean formulaire
    DynaActionForm FormUtilisateur=(DynaActionForm)form;
    //créer un objet utilisateur directement à partir des saisies
    Utilisateur utilisateur=(Utilisateur)FormUtilisateur.get("utilisateur");

    //modifier l'utilisateur
    int resinstruction=utilisateurmodele.modifierUtilisateur(utilisateur);

    //en cas d'erreur avec la base de données
    if (resinstruction!=1)
    {
        //ajouter un message d'erreur
        ActionMessages erreurs=new ActionErrors();
        erreurs.add("message", new ActionMessage
("erreurs.modificationutilisateur"));
        saveErrors(request,erreurs);
    }
    //tout est OK
    else
    {
        //toutes les vérifications syntaxiques sont correctes si arrivée ici
        ActionMessages messages=new ActionMessages();
        messages.add("message", new ActionMessage
("succes.modificationutilisateur"));
        saveMessages(request,messages);
        //retourner sur la page de listing des utilisateurs
        return mapping.findForward("listeutilisateur");
```

```
    }
        //en cas d'erreur, retourner sur la page d'accueil
    return mapping.findForward("accueil");
}
```

```
/**********************************************************
 * modifier l'utilisateur
**********************************************************/
public int modifierUtilisateur(Utilisateur utilisateur)
{
   int resinstruction=0;
   PreparedStatement requete=null;

   try
   {
      //ouvrir une connexion
      connection=ds.getConnection();
      requete=connection.prepareStatement("UPDATE utilisateur SET
pseudonyme=?,motdepasse=?,nom=?,prenom=?,autorisation=?,image=? WHERE
id_utilisateur=?");
      requete.setString(1,(String)utilisateur.getPseudonyme());
      requete.setString(2,(String)utilisateur.getMotdepasse());
      requete.setString(3,(String)utilisateur.getNom());
      requete.setString(4,(String)utilisateur.getPrenom());
      requete.setInt(5,(Integer)utilisateur.getAutorisation());
      requete.setString(6,(String)utilisateur.getImage());
      requete.setString(7,(String)utilisateur.getId_utilisateur());
      resinstruction=requete.executeUpdate();
   }
   ...
}
```

Nous terminons ensuite par la définition des deux nouveaux messages dans le fichier de propriétés.

```
erreurs.modificationutilisateur=Erreur lors de la modification
de l'utilisateur
succes.modificationutilisateur=L'utilisateur a été modifié avec succès
```



9. Activation des vérifications JavaScript

Nous pouvons activer les validations JavaScript pour le formulaire de création et de modification. Pour cela, il suffit de modifier la vue JSP avec deux lignes qui permettent de générer le code en rapport avec les règles de validation du fichier *validation.xml*. Si nous prenons par exemple la page */vues/utilisateur/creerutilisateur.jsp*, nous devons simplement ajouter les lignes suivantes :

```
...
<html:javascript formName="FormUtilisateur"/>
<html:form action="/validercreerutilisateur" method="POST" onsubmit="return
validateFormUtilisateur(this)">
...
```

10. Suppression d'un utilisateur

La suppression est le service le plus simple à développer avec la consultation. Nous allons cependant ajouter un fichier JavaScript */javascript/boiteoutils.js* qui permet d'afficher des messages de confirmation. Ce fichier contient le code suivant :

```
//supprimer un utilisateur
function confirmerSuppressionUtilisateur(id_utilisateur,pseudonyme)
{
   if(confirm("Voulez-vous supprimer cet utilisateur ?"))
   {
       chemin="supprimerutilisateur.do?id_utilisateur="+id_utilisateur
+"&pseudonyme="+pseudonyme;
    document.location.href=chemin;
   }
   else
   {
   return;
   }
}
```

Ce script est inséré dans la page */vues/outils/entete.jspf* avec la ligne ci-après :

```
<script type="text/javascript" src="javascript/boiteoutils.js"></script>
```

Nous pouvons désormais réaliser la configuration de l'action par l'intermédiaire du fichier *struts-config.xml*.

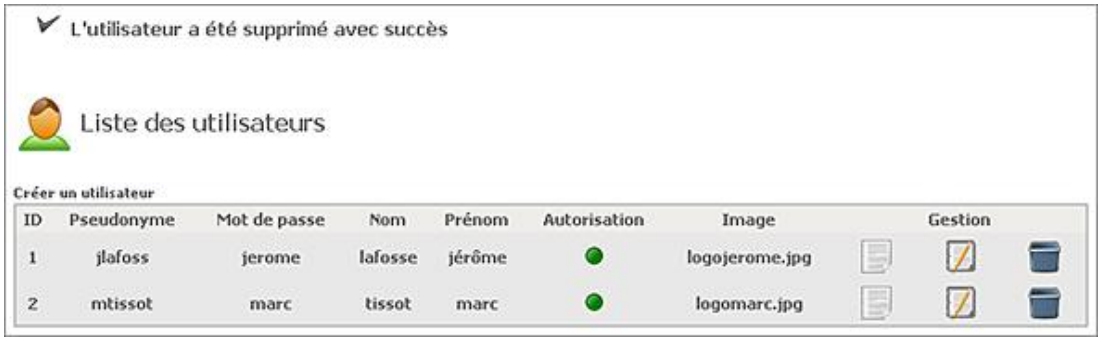
```
<!-- ===action qui permet de supprimer un utilisateur=== -->
<action
    path="/supprimerutilisateur"
    scope="request"
    input="/listeutilisateur.do"
    type="chatbetaboutique.GestionUtilisateurAction"
    parameter="supprimerutilisateur"
    >
<forward name="listeutilisateur" path="/listeutilisateur.do"
redirect="false"/>
</action>
```

Enfin, il reste à coder l'action *supprimerutilisateur()* et du modèle avec les messages associés.

```
//supprimer l'utilisateur indiqué
public ActionForward supprimerutilisateur(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
{
    //récupérer la datasource du plugin dans un attribut
présent dans le contexte de la servlet
    ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
    //créer le modèle
    UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
    //fermer la datasource
    this.ds=null;
    //récupérer l'id de l'utilisateur
    String id_utilisateur=(String)request.getParameter("id_utilisateur");
    //récupérer le pseudonyme de l'utilisateur
    String pseudonyme=(String)request.getParameter("pseudonyme");
    //supprimer l'administrateur
    if(id_utilisateur!=null && !id_utilisateur.equals("") &&
pseudonyme!=null && !pseudonyme.equals(""))
    {
        int resinstruction=utilisateurmodele.supprimerUtilisateur
(id_utilisateur,pseudonyme);
        //en cas d'erreur avec la base de données
        if (resinstruction!=1)
        {
        //ajouter un message d'erreur
        ActionMessages erreurs=new ActionErrors();
```

```
        erreurs.add("message", new ActionMessage
("erreurs.suppressionutilisateur"));
        saveErrors(request,erreurs);
        }
        //tout est OK
        else
        {
        ActionMessages messages=new ActionMessages();
            messages.add("message", new ActionMessage
("succes.suppressionutilisateur"));
                saveMessages(request,messages);
        }
    }
    //retourner sur la page de listing des utilisateurs
    return mapping.findForward("listeutilisateur");
}
```

```
/*********************************************************
 * supprimer l'utilisateur
*********************************************************/
public int supprimerUtilisateur(String id_utilisateur,String pseudonyme)
{
   int resinstruction=0;
   PreparedStatement requetea=null,requeteb=null;
   try
   {
      //ouvrir une connexion
      connection=ds.getConnection();
      //supprimer l'utilisateur
      requetea=connection.prepareStatement("DELETE FROM utilisateur WHERE
id_utilisateur=?");
      requetea.setString(1,(String)id_utilisateur);
      resinstruction=requetea.executeUpdate();
      //supprimer ses inscriptions aux salons
      requeteb=connection.prepareStatement("DELETE FROM utilisateursalon
WHERE pseudonyme=?");
      requeteb.setString(1,(String)pseudonyme);
      requeteb.executeUpdate();
   }
...
}
```



11. Gestion de l'état du serveur

La gestion de l'état du serveur est réalisée avec la table *serveur*. Cette table possède un seul attribut nommé *etatserveur* qui permet d'activer ou désactiver le serveur.

Pour commencer, nous allons développer une action Struts avec un paramètre envoyé par un lien HTML qui permet de changer l'état du serveur. Pour cela il est nécessaire d'ajouter un nouveau lien dans le menu de la page */vues/outils/navigation.jspf*. Ce lien va déclencher l'action *etatserveur* avec le paramètre en cours, représentant le nouvel état. Pour ce service, nous pouvons utiliser la classe *Action* simple proposée par Struts. De même, la table serveur possède un seul attribut, il est donc plus simple de ne pas créer une classe JavaBean de formulaire mais d'utiliser à la place un paramètre de type chaîne de caractères qui représente l'état du serveur.

Nous commençons par ajouter un lien HTML pour déclencher le service de gestion de l'état du serveur.

```
<table border="0" cellspacing="0" cellpadding="0"
name="menugauche" id="menugauche" style="background:
url('img/fondmenu.gif') repeat-x;"  height="500">
<tr>
  <td width="100%" valign="top"><br/><span style="margin-left:
30px;"><b>ADMINISTRATION</b></span>
   <ul class="menucategorie">
       <li><a href="etatserveur.do"><img src="img/fleche.gif"
border="0" align="absmiddle"/> Gestion du serveur</a></li>
       <li><a href="listeutilisateur.do"><img src="img/fleche.gif"
border="0" align="absmiddle"/> Gestion des utilisateurs</a></li>
   </ul>
  </td>
</tr>
</table>
```

Nous passons ensuite à la définition de l'action *etatserveur.do* dans le fichier de configuration *struts-config.xml*.

```
<!-- ================= SERVEUR ==================== -->
<!-- ===action qui permet de gérer l'état du serveur=== -->
<action
    path="/etatserveur"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="chatbetaboutique.GestionServeurAction"
    >
<forward name="etatserveur" path="/vues/serveur/etatserveur.jsp"
redirect="false"/>
</action>
```

Le mapping de l'action est réalisé, il reste à créer le code de la classe *chatbetaboutique.GestionServeurAction* ainsi que le modèle associé. Nous pouvons également adapter notre action afin de vérifier si l'état du serveur est passé en paramètre et si nous devons dans ce cas changer son état avec une méthode du modèle.

```
package chatbetaboutique;

import modele.ServeurModele;
import org.apache.struts.action.*;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import javax.sql.DataSource;

public class GestionServeurAction extends Action{

    //variables de la classe
    DataSource ds=null;

    //gérer l'état du serveur
    public ActionForward execute(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
    {
```

```
        //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
        ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
        //créer le modèle
        ServeurModele serveurmodele=new ServeurModele(ds);
        //fermer la datasource
        this.ds=null;

        //récupérer le paramètre et changer l'état du serveur si nécessaire
        String etat=(String)request.getParameter("etat");
        if(etat!=null && !etat.equals(""))
        {
            //modifier l'état du serveur
            int resinstruction=serveurmodele.setEtatServeur(etat);
            //en cas d'erreur avec la base de données
            if (resinstruction!=1)
            {
                //ajouter un message d'erreur
                ActionMessages erreurs=new ActionErrors();
                                erreurs.add("message", new
ActionMessage("erreurs.etatserveur"));
                saveErrors(request,erreurs);
            }
            //tout est OK
            else
            {
                //toutes les vérifications sont correctes
                ActionMessages messages=new ActionMessages();
                messages.add("message", new
ActionMessage("succes.etatserveur"));
                saveMessages(request,messages);
            }
        }

        //retourner l'état actuel du serveur
        String etatserveur=(String)serveurmodele.getEtatServeur();
        //retourner l'état du serveur
        request.setAttribute("etatserveur",etatserveur);
        //retourner sur la page d'affichage de l'état du serveur
        return mapping.findForward("etatserveur");
    }


//fin de la classe
}
```

Le modèle permet de lire l'état actuel du serveur dans la base de données.

```
package modele;

import boiteoutils.*;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.sql.DataSource;

public class ServeurModele
{
    //variables de classe
    DataSource ds=null;
    Connection connection=null;
    ResultSet rs=null;

/*********************************************************
 * constructeur
*********************************************************/
public ServeurModele(DataSource ds)
{
    //récupérer le DataSource de la servlet
    this.ds=ds;
```

```
}

/*********************************************************
 * etat du serveur
*********************************************************/
public String getEtatServeur()
{
    String etatserveur="0";
    //statement
    PreparedStatement requete=null;

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        //enregistrements
        requete=connection.prepareStatement("SELECT
etatserveur FROM serveur");
        rs=requete.executeQuery();
        //exécuter la requête
        if(rs!=null)
        {
            if(rs.next())
            {
                etatserveur=rs.getString("etatserveur");
            }
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
ServeurModele.java fonction getEtatServeur");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ServeurModele.java fonction getEtatServeur");
        }
    }

    //retourner l'état du serveur
    return etatserveur;
}

//fin de la classe
}
```

La page JSP permet d'afficher l'état du serveur et de passer un paramètre dans un lien pour changer sa valeur.

```
<%
//récupérer l'état du serveur
String etatserveur=(String)request.getAttribute("etatserveur");
%>
<!--  inclure l'en-tete de la page -->
<%@ include file="../outils/entete.jspf" %>

<br/><div class="titre"><a href="etatserveur.do"><img
src="img/serveur.jpg" border="0" align="absmiddle"/> Etat
du serveur</a></div>

<table border="0" id="tableau" cellpadding="0" cellspacing="0" width="50%">
```

```
<tr><td>Etat</td><td>
<% if(etatserveur.equals("1"))
{
 out.println("<a href=\"etatserveur.do?etat=0\">Serveur Actif
<img src=\"img/actif.gif\" border=\"0\" align=\"absmiddle\"/></a>");
}
else
{
 out.println("<a href=\"etatserveur.do?etat=1\">Serveur Inactif
<img src=\"img/inactif.gif\" border=\"0\" align=\"absmiddle\"/></a>");
}
%></td></tr>
</table>

<!--  inclure le pied de page -->
<%@ include file="../outils/piedpage.jspf" %>
```

La fonction du modèle qui permet de changer l'état du serveur est la suivante :

```
/*********************************************************
 * modifier l'etat du serveur
*********************************************************/
public int setEtatServeur(String etat)
{
    int resinstrution=0;
    PreparedStatement requete=null;

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        requete=connection.prepareStatement("UPDATE
serveur SET etatserveur=?");
        requete.setString(1,(String)etat);
        resinstrution=requete.executeUpdate();
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
ServeurModele.java fonction setEtatServeur");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(requete!=null)OutilsBaseDeDonnees.fermerConnexion(requete);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
ServeurModele.java fonction setEtatServeur");
        }
    }

    //résultat de la modification
    return resinstrution;
}
```

Enfin, le fichier de propriétés possède les deux nouveaux messages suivants :

```
#serveur
erreurs.etatserveur=Erreur lors de la modification de l'état du serveur
succes.etatserveur=L'état du serveur a été modifié avec succès
```

12. Gestion des salons

Nous allons développer un service de gestion des salons avec le même principe que celui utilisé pour les utilisateurs du chat. Le fonctionnement est identique, il consiste à développer le JavaBean de formulaire *Salon* en correspondance avec le modèle de la table *salon* de la base de données.

```
package boiteoutils;
import java.io.Serializable;

public class Salon implements Serializable
{
   //java 5.0 pour avoir un identifiant unique de la sérialisation
   private static final long serialVersionUID = 1L;
   //variables de classe
   private String id_salon=null;
   private String theme=null;
   private String date=null;
   private int actif=0;

public Salon(){
}
...
//fin de la classe
}
```

Nous réalisons ensuite, la définition de l'élément *<form-bean/>* et du mapping pour les actions liées à la gestion des salons du chat.

```
<!-- ========== Formulaire de gestion des salons ========== -->
<form-bean name="FormSalon"
type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="salon" type="boiteoutils.Salon" />
</form-bean>

<!-- ================= SALON ====================== -->
<!-- ===action qui permet de lister les salons=== -->
<action
    path="/listesalon"
    name="FormSalon"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="chatbetaboutique.GestionSalonAction"
    parameter="listesalon">
<forward name="listesalon" path="/vues/salon/listesalon.jsp"
redirect="false"/>
</action>
<!-- ===action qui permet de consulter la fiche d'un salon=== -->
<action
    path="/consultersalon"
    name="FormSalon"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="chatbetaboutique.GestionSalonAction"
    parameter="consultersalon">
<forward name="consultersalon" path="/vues/salon/consultersalon.jsp"
redirect="false"/>
</action>
<!-- action qui permet d'afficher le formulaire de création d'un salon -->
<action
    path="/creersalon"
    name="FormSalon"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/vues/salon/creersalon.jsp">
</action>
```

```
<!-- action qui va valider la création d'un salon -->
<action
    path="/validercreersalon"
    name="FormSalon"
    scope="request"
    validate="true"
    input="/creersalon.do"
    type="chatbetaboutique.GestionSalonAction"
    parameter="validercreersalon">
<forward name="listesalon" path="/listesalon.do" redirect="false"/>
</action>
<!-- ===action qui permet de modifier un salon en
récupérant ses valeurs=== -->
<action
    path="/modifiersalon"
    name="FormSalon"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="chatbetaboutique.GestionSalonAction"
    parameter="modifiersalon">
<forward name="modifiersalon" path="/vues/salon/modifiersalon.jsp"
redirect="false"/>
</action>
<!-- ===action qui permet de valider la modification d'un salon=== -->
<action
    path="/validermodifiersalon"
    name="FormSalon"
    scope="request"
    validate="true"
    input="/modifiersalon.do"
    type="chatbetaboutique.GestionSalonAction"
    parameter="validermodifiersalon">
<forward name="modifiersalon" path="/modifiersalon.do" redirect="false"/>
<forward name="listesalon" path="/listesalon.do" redirect="false"/>
</action>
<!-- ===action qui permet de supprimer un salon=== -->
<action
    path="/supprimersalon"
    scope="request"
    input="/listesalon.do"
    type="chatbetaboutique.GestionSalonAction"
    parameter="supprimersalon">
<forward name="listesalon" path="/listesalon.do" redirect="false"/>
</action>
```

Nous pouvons désormais passer au codage des actions associées, avec la classe de gestion nommée *chatbetaboutique.GestionSalonAction*.

```
package chatbetaboutique;

import modele.SalonModele;
import org.apache.struts.action.*;
import org.apache.struts.actions.MappingDispatchAction;
import boiteoutils.Salon;
import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import javax.sql.DataSource;

public class GestionSalonAction extends MappingDispatchAction{

   //variables de la classe
   DataSource ds=null;

   //afficher la liste des salons
   public ActionForward listesalon(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response)throws IOException,
ServletException
```

```
   {
      ...
   }


   //consulter un salon
   public ActionForward consultersalon(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response)throws IOException,
ServletException
   {
      ...
   }



   //valider la création d'un salon
   public ActionForward validercreersalon(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
   {
      ...
   }


   //afficher le formulaire en modification pour le salon indiqué
   public ActionForward modifiersalon(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response)throws IOException,
ServletException
   {
      ...
   }


   //valider la modification du salon
   public ActionForward validermodifiersalon(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
   {
      ...
   }


   //supprimer le salon indiqué
   public ActionForward supprimersalon(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response)throws IOException,
ServletException
   {
      ...
   }

//fin de la classe
}
```

La classe de gestion du modèle modele.SalonModele est également très proche du modèle de gestion des utilisateurs.

```
package modele;

import boiteoutils.*;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.sql.DataSource;
import org.apache.commons.dbutils.BeanProcessor;
import java.util.ArrayList;

public class SalonModele
{
   //variables de classe
   DataSource ds=null;
   Connection connection=null;
```

```
   ResultSet rs=null;
   //liste des objets
   ArrayList<Salon> listesalon=new ArrayList<Salon>();

/*********************************************************
 * constructeur
*********************************************************/
public SalonModele(DataSource ds)
{
   //récupérer le DataSource de la servlet
   this.ds=ds;
}

/*********************************************************
 * liste des salons
*********************************************************/
public ArrayList getListeSalon()
{
   //statement
   PreparedStatement requete=null;

   try
   {
      //ouvrir une connexion
      connection=ds.getConnection();
      //enregistrements
      requete=connection.prepareStatement("SELECT * FROM salon ORDER BY
theme,date");
      rs=requete.executeQuery();
      //exécuter la requête
      if(rs!=null)
      {
         //utilisation de la librairie DbUtils qui permet de transformer
un ResultSet en Objet Java
         BeanProcessor bp=new BeanProcessor();
         listesalon = (ArrayList<Salon>)bp.toBeanList(rs,Salon.class);
       }
   }
...
}
/*********************************************************
 * récupérer le salon indiqué
*********************************************************/
public Salon getSalon(String id_salon)
{
   //créer un objet salon
   Salon salon=new Salon();
   //statement
   PreparedStatement requete=null;

   try
   {
      //ouvrir une connexion
      connection=ds.getConnection();
      //enregistrements
      requete=connection.prepareStatement("SELECT * FROM salon WHERE
id_salon=?");
      requete.setString(1,(String)id_salon);
      rs=requete.executeQuery();
      //exécuter la requête
      if(rs!=null)
      {
         if(rs.next())
         {
      //utilisation de la librairie DbUtils qui permet de transformer
un ResultSet en Objet Java
         BeanProcessor bp=new BeanProcessor();
         salon = (Salon)bp.toBean(rs, Salon.class);
         }
       }
```

```
   }
...
}

/*********************************************************
 * créer un salon
*********************************************************/
public int creerSalon(Salon salon)
{
...
}

/*********************************************************
 * modifier le salon
*********************************************************/
public int modifierSalon(Salon salon)
{
...
}

/*********************************************************
 * supprimer le salon
*********************************************************/
public int supprimerSalon(String id_salon)
{
   ...
}

//fin de la classe
}
```

La totalité du service de gestion des salons est pratiquement terminé, nous allons ajouter les fonctionnalités d'usage pour l'ensemble. Nous insérons d'abord un lien dans le menu de navigation */vues/outils/navigation.jspf*.

```
<table border="0" cellspacing="0" cellpadding="0" name="menugauche"
id="menugauche" style="background: url('img/fondmenu.gif') repeat-x;"
height="500">
<tr>
  <td width="100%" valign="top"><br/><span style="margin-left:
30px;"><b>ADMINISTRATION</b></span>
   <ul class="menucategorie">
       <li><a href="etatserveur.do" class="info"><img
src="img/fleche.gif" border="0" align="absmiddle"/><span>Cliquez ici
pour afficher le service de gestion du serveur</span> Gestion
du serveur</a></li>
       <li><a href="listeutilisateur.do" class="info"><img
src="img/fleche.gif" border="0" align="absmiddle"/><span>Cliquez ici
pour afficher le service de gestion des utilisateurs</span> Gestion
des utilisateurs</a></li>
       <li><a href="listesalon.do" class="info"><img
src="img/fleche.gif" border="0" align="absmiddle"/><span>Cliquez ici
pour afficher le service de gestion des salons</span> Gestion
des salons</a></li>
   </ul>
  </td>
</tr>
</table>
```

Ensuite, nous passons au codage des règles de validation pour le formulaire. Dans notre cas, nous indiquons une règle sur les champs theme, date et actif dans le fichier */WEB-INF/validation.xml*.

```
...
<global>
...
<constant>
<constant-name>datesalon</constant-name>
<constant-value>^[0-9]{4}-[0-9]{2}-[0-9]{2}\s[0-9]{2}:[0-9]{2}:[0-9]{2}.{0,1}
[0-9]{0,1}$</constant-value>
</constant>
```

```
</global>
...
<!-- ========== Formulaire salon ========== -->
<form name="FormSalon">
    <field property="salon.theme" depends="required">
        <arg0 key="themesalon"/>
    </field>
    <field property="salon.date" depends="required,mask">
        <arg0 key="datesalon"/>
        <var>
            <var-name>mask</var-name>
            <var-value>${datesalon}</var-value>
        </var>
    </field>
    <field property="salon.actif" depends="required,mask">
        <arg0 key="actifsalon"/>
        <var>
            <var-name>mask</var-name>
            <var-value>${autorisation}</var-value>
        </var>
    </field>
</form>
...
```

Enfin nous terminons par le fichier de propriétés pour les messages et les noms des champs.

```
#salon
erreurs.creationsalon=Erreur lors de la création du salon
succes.creationsalon=Le salon a été créé avec succès
erreurs.modificationsalon=Erreur lors de la modification du salon
succes.modificationsalon=Le salon a été modifié avec succès
erreurs.suppressionsalon=Erreur lors de la suppression du salon
succes.suppressionsalon=Le salon a été supprimé avec succès
# -- gestion des salons --
themesalon=Thème
datesalon=Date
nomutilisateur=Nom
actifsalon=Actif
```

Lors de la suppression d'un salon, un message de confirmation est affiché. La fonction *confirmerSuppressionSalon* doit donc être codée dans le fichier */javascript/boiteoutils.js*.

```
//supprimer un salon
function confirmerSuppressionSalon(id_salon)
{
   if(confirm("Voulez-vous supprimer ce salon ?"))
   {
       chemin="supprimersalon.do?id_salon="+id_salon;
    document.location.href=chemin;
   }
   else
   {
   return;
   }
}
```

Le service est opérationnel, nous pouvons tester les saisies, modifications et messages.

13. Gestion des inscriptions et interactivité

Pour le développement du service de gestion des inscriptions utilisateurs aux salons, il existe plusieurs approches possibles :

- Développement d'un service spécifique qui permet d'associer des utilisateurs au salon indiqué à partir de cases à cocher.
- Développement d'un service interactif à base d'Ajax et de sérialisation des données.

C'est la seconde solution que nous allons utiliser, avec un formulaire dynamique associé à chaque salon. Pour la réalisation de cette partie plus complexe, il est nécessaire de détailler un scénario explicatif :

- La première étape consiste à créer une balise * * HTML associée à chaque ligne de la liste des salons. Par exemple, le salon avec *id=6* aura une balise * * cachée avec un identifiant unique et pourra recevoir un formulaire Ajax suite à un clic administrateur.
- Suite à ce clic, une fonction JavaScript Ajax va récupérer toute la liste des utilisateurs avec l'état de l'inscription associée (inscrit ou pas au salon) et afficher la réponse dans la balise * * cachée.
- Tous les utilisateurs seront affichés dans ce formulaire et pourront donc être inscrits au salon sélectionné en une seule étape. Il serait possible, par facilité, de réaliser une inscription en Ajax sur chaque clic de la case à cocher mais le service ne serait pas très ergonomique. Nous allons donc utiliser un mécanisme de sérialisation pour envoyer tous les choix de l'administrateur avec un seul clic de souris.

Le développement de ce service nécessite l'utilisation de la librairie JavaScript JQuery (http://jquery.com/). Cette librairie est référencée dans la page *entete.jspf*.

Tout d'abord nous ajoutons un bouton de gestion dans la vue qui permet d'afficher la liste des salons.

```
<%
...
out.println("<td><a href='javascript:listeUtilisateurSalon
(\""+salon.getId_salon()+"\");'><img src=\"img/utilisateurchat.png\
" border=\"0\" align=\"absmiddle\" title=\"Utilisateur\"/></a><div
id=\"listeutilisateursalon_"+salon.getId_salon()+"\"\"
class=\"listeutilisateursalon\"> </div></td>");
...
%>
```

Ensuite, nous codons la fonction JavaScript *listeUtilisateurSalon()* présente dans le fichier *boiteoutils.js* qui permet de récupérer tous les utilisateurs de la base de données ainsi que les états associés, pour ce salon.

```
//fonction qui permet d'afficher la liste des utilisateurs
inscrits ou pas au salon
function listeUtilisateurSalon(id_salon)
{
    //fermer la boîte en cours
    if($("#listeutilisateursalon_"+id_salon).css("display")=="block")
```

```
    {
        $("#listeutilisateursalon_"+id_salon).slideUp("slow");
        return;
    }
    //fermer toutes les boîtes qui ont la classe css
    $(".listeutilisateursalon").hide();
    //détruire leur formulaire
    $(".listeutilisateursalon").html("");

    //récupérer la liste des utilisateurs inscrits ou pas au salon indique
    if(id_salon!=null)
    {
        //envoyer les donnees en POST
        $.ajax(
        {
            type: "POST",
            url: "listeutilisateursalon.do",
            dataType: "html",
            data: "id_salon="+id_salon,
            timeout : 4000,
            error: function(){
            },
            beforeSend : function()
            {
            },
            success: function(html)
            {
                //mettre le résultat dans la balise <div/> concernee
                $("#listeutilisateursalon_"+id_salon).html(html);
                $("#listeutilisateursalon_"+id_salon).slideDown("slow");
            }
        });
    }
}
```

L'instruction suivante *$(".listeutilisateursalon").html("");* présente dans la fonction JavaScript *listeUtilisateurSalon()* est importante. En effet, elle permet de détruire tous les formulaires qui ont pour classe CSS *.listeutilisateur*. En effet, la fermeture des blocs * * ne permet pas la suppression des cases à cocher qui sont alors présentes dans la page HTML mais non visibles. Cette instruction permet de détruire tous les formulaires pour qu'il n'y ait pas de conflits de noms.

Ensuite, nous codons l'action dans le fichier de configuration *struts-config.xml*.

```
<!-- ===action qui permet de retourner la liste
des utilisateurs pour un salon indiqué=== -->
<action
    path="/listeutilisateursalon"
    name="FormUtilisateur"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="chatbetaboutique.GestionUtilisateurAction"
    parameter="listeutilisateursalon"
    >
<forward name="listeutilisateursalon"
path="/vues/utilisateur/listeutilisateursalon.jsp"
redirect="false"/>
</action>
```

Le code de l'action *listeutilisateursalon* est ajouté dans la classe *GestionUtilisateur*. Ce développement est assez simple, il permet de déclencher une nouvelle fonction du modèle qui retourne tous les utilisateurs et leur état respectif, pour le salon indiqué en paramètre.

```
//afficher la liste des utilisateurs pour le salon
public ActionForward listeutilisateursalon(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
{
    //récupérer la datasource du plug-in dans un attribut
```

```
présent dans le contexte de la servlet
    ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
    //créer le modèle
    UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
    //fermer la datasource
    this.ds=null;
    //récupérer le salon en paramètre
    String id_salon=(String)request.getParameter("id_salon");

    //récupérer la liste de tous les utilisateurs et l'état pour ce salon
    if(id_salon!=null && !id_salon.equals(""))
    {
        //retourner la liste des utilisateurs
        ArrayList listeutilisateursalon=
(ArrayList)utilisateurmodele.getListeUtilisateurSalon(id_salon);
        //retourner la liste des utilisateurs
        request.setAttribute("listeutilisateursalon",listeutilisateursalon);
        //retourner le numéro du salon
        request.setAttribute("id_salon",id_salon);
        //vider par sécurité
        listeutilisateursalon=null;
        //retourner sur la page d'affichage des utilisateurs
        return mapping.findForward("listeutilisateursalon");
    }

    //en cas d'erreur
    return mapping.findForward("accueil");
}
```

```
/**********************************************************
 * liste des utilisateurs et informations pour le salon
**********************************************************/
public ArrayList getListeUtilisateurSalon(String id_salon)
{
    // récupérer la liste de tous les utilisateurs
    this.listeutilisateur=this.getListeUtilisateur();

    //nouvelle liste des utilisateurs avec l'état de l'inscription
    ArrayList<Utilisateur> listeutilisateursalon=new
ArrayList<Utilisateur>();

    // parcours de chaque utilisateur
    for(int i=0;i<listeutilisateur.size();i++)
    {
        Utilisateur utilisateur=(Utilisateur)listeutilisateur.get(i);
        //récupérer son inscription pour le salon indiqué
        int inscrit=this.getInscriptionUtilisateurSalon
(utilisateur.getPseudonyme(),id_salon);
        //positionner cette valeur pour l'utilisateur
        utilisateur.setInscrit(inscrit);
        //mettre l'utilisateur dans la nouvelle liste
        listeutilisateursalon.add(utilisateur);
    }

    return listeutilisateursalon;
}
/**********************************************************
 * tester si l'utilisateur concerné est inscrit au salon
**********************************************************/
public int getInscriptionUtilisateurSalon(String pseudonyme,String id_salon)
{
    //inscription au salon
    int inscrit=0;

    //statement
    PreparedStatement requete=null;
    try
    {
        //ouvrir une connexion
```

```
        connection=ds.getConnection();
        //enregistrements
        requete=connection.prepareStatement("SELECT * FROM
utilisateursalon WHERE pseudonyme=? AND id_salon=?");
        requete.setString(1,(String)pseudonyme);
        requete.setString(2,(String)id_salon);
        rs=requete.executeQuery();
        //exécuter la requête
        if(rs!=null)
        {
            //l'utilisateur est inscrit à ce salon
            if(rs.next())
            {
                inscrit=1;
            }
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
UtilisateurModele.java fonction getInscriptionUtilisateurSalon");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
UtilisateurModele.java fonction getInscriptionUtilisateurSalon");
        }
    }

    //retourner l'état de la connexion
    return inscrit;
}
```

Les affichages sont réalisés avec la feuille de style par défaut et la nouvelle classe CSS *.listeutilisateursalon*.

```
.listeutilisateursalon
{
display:none;
position:absolute;
height:5cm;
width:15cm;
overflow:auto;
z-index:1000;
margin-left:-500px;
border-style:solid;
border-top-width:1px;
border-right-width:2px;
border-bottom-width:2px;
border-left-width:1px;
border-color:#A8A6A7;
background-color:#fff;
}
```

Le mécanisme permet ainsi de fermer toutes les boîtes déjà ouvertes (et de les détruire) et de récupérer en arrière-plan, avec la technologie Ajax, la liste des utilisateurs et leur état respectif. Le codage de la vue */vues/utilisateur/listeutilisateursalon.jsp* est assez proche de la vue */vues/utilisateur/listeutilisateur.jsp*.

```
<%@ page import="java.util.ArrayList" %>
<%@ page import="boiteoutils.Utilisateur" %>
<%
//liste des utilisateurs
```

```
ArrayList listeutilisateursalon=
(ArrayList)request.getAttribute("listeutilisateursalon");
//numéro du salon
String id_salon=(String)request.getAttribute("id_salon");
%>
<form name="formulaire" id="formulaire">
<table border="0" id="tableaubordure" cellspacing="0" cellpadding="0">
<tr align="center" class="entetetableau">
<td>ID</td>
<td>Pseudonyme</td>
<td>Nom</td>
<td>Pr&eacute;nom</td>
<td>Inscription</td>
</tr>
<%
for(int i=0;i<listeutilisateursalon.size();i++)
{
    //récupérer l'objet dans la liste
    Utilisateur utilisateur=(Utilisateur)listeutilisateursalon.get(i);
    if(i%2==0)out.println("<tr align=\"center\" class=\"ligneclaire\">");
    else out.println("<tr align=\"center\" class=\"lignefoncee\">");
    out.println("<td>"+utilisateur.getId_utilisateur()+"</td>");
    out.println("<td>"+utilisateur.getPseudonyme()+"</td>");
    out.println("<td>"+utilisateur.getNom()+"</td>");
    out.println("<td>"+utilisateur.getPrenom()+"</td>");
    out.println("<td>");
    //gestion de l'inscription pour l'utilisateur en cours
    out.println("<input type=\"checkbox\"
name=\"inscriptionutilisateur_"+utilisateur.getPseudonyme()+"\"");
    if(utilisateur.getInscrit()==1)
    {
        out.println(" checked=\"checked\"/>");
    }
    else
    {
        out.println(" />");
    }
    out.println("</tr>");
}
%>
<tr><td align="center" colspan="5"><input type="button"
onclick="javascript:validerInscriptionUtilisateurSalon(<%= id_salon%>)"
value="Valider" class="bouton"/></td></tr>
</table>
</form>
```

Le service d'affichage avec état est désormais opérationnel. Les utilisateurs inscrits au salon en cours sont bien marqués et la case à cocher correspondante affiche l'état. Nous pouvons maintenant gérer la partie inscription des utilisateurs aux salons.

Il serait possible à chaque clic de souris sur la case à cocher de l'utilisateur, d'envoyer l'information à une action et de fermer la fenêtre. Cette technique envisageable n'est pas très ergonomique pour l'administrateur qui devra réaliser plusieurs opérations à chaque inscription. Une interface plus adaptée permet de sélectionner plusieurs utilisateurs (plusieurs choix d'inscription) et de valider l'ensemble en une seule étape.

Le scénario de programmation est alors le suivant :

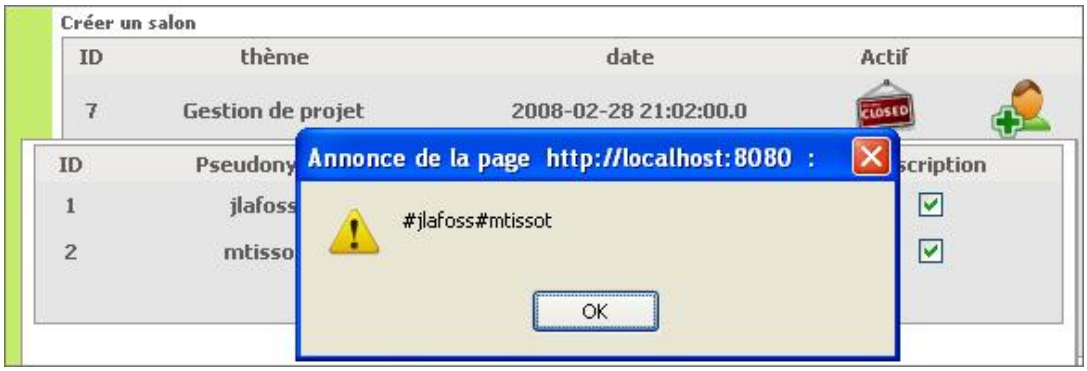
- L'administrateur sélectionne plusieurs inscriptions à la fois.

- L'administrateur clique sur le bouton de validation. Ce bouton déclenche une fonction JavaScript qui va récupérer toutes les cases cochées et les id des utilisateurs associés. Cette même fonction réalise alors une sérialisation des données et poste l'ensemble des informations à une action qui pourra réaliser la totalité des inscriptions en une seule étape.

La sérialisation de données est très utilisée en programmation Web. Cette technique permet d'envoyer plusieurs informations dans la requête HTTP GET ou POST avec un seul paramètre sous une syntaxe spécifique. Pour notre exemple, nous allons utiliser une variable nommée inscription qui aura par exemple pour valeur : *inscription=2#4#6* pour les utilisateurs d'id 2, 4 et 6.

Nous commençons le codage de la fonction de validation JavaScript par le parcours des éléments du formulaire de type checkbox (case à cocher).

```
//fonction qui permet d'envoyer sous forme serialisee
les inscriptions utilisateur
function validerInscriptionUtilisateurSalon(id_salon)
{
    //inscription des utilisateurs sous forme serialisee
    var inscription="";
    //recuperer toutes les cases a cocher
    var elts=document.forms["formulaire"].elements;
    //parcourir chaque element de la page
    for(var i=0;i<elts.length;i++)
    {
        //pas une case a cocher
        if(elts[i].type!="checkbox")
        {
            continue;
        }
        //case a cochee et validee
        if(elts[i].checked)
        {
            //la case a cocher doit contenir le terme
inscriptionutilisateur_
            var nom=elts[i].name;
            var pseudonyme=nom.substring(nom.indexOf("_")+1,nom.length);
            //l'id de l'utilisateur est correct on va le serialiser
            if(pseudonyme!=null && pseudonyme!="")
            {
                inscription+="#"+pseudonyme;
            }
        }
    }
    alert(inscription);
}
```



Nous allons maintenant optimiser cette méthode afin de vérifier que les cases à cocher sont bien en rapport avec les utilisateurs et sérialiser l'ensemble avant l'envoi des données en Ajax. La nouvelle fonction JavaScript est présentée ci-après :

```
//fonction qui permet d'envoyer sous forme serialisee
```

```
les inscriptions utilisateur
function validerInscriptionUtilisateurSalon(id_salon)
{
    //inscription des utilisateurs sous forme serialisee
    var inscription="";
    //recuperer toutes les cases a cocher
    var elts=document.forms["formulaire"].elements;
    //parcourir chaque element de la page
    for(var i=0;i<elts.length;i++)
    {
        //pas une case a cocher
        if(elts[i].type!="checkbox")
        {
            continue;
        }
        //case a cochee et validee
        if(elts[i].checked)
        {
           //la case a cocher doit contenir le terme inscriptionutilisateur_
           var nom=elts[i].name;
           var pseudonyme=nom.substring(nom.indexOf("_")+1,nom.length);
           //l'id de l'utilisateur est correct on va le serialiser
           if(pseudonyme!=null && pseudonyme!="")
           {
               inscription+="#"+pseudonyme;
           }
        }
    }

    //envoyer les inscriptions en ajax
    if(id_salon!=null)
    {
        //envoyer les donnees en POST
        $.ajax(
        {
           type: "POST",
           url: "inscriptionutilisateursalon.do",
           dataType: "html",
           data: "id_salon="+id_salon+"&inscription="+inscription,
           timeout : 4000,
           error: function(){
           },
           beforeSend : function()
           {
           },
           success: function(html)
           {
                //reponse correcte
                if(html==1)
                {
                    //fermer la fenetre ouverte
                    $("#listeutilisateursalon_"+id_salon).slideUp("slow");
                }
                //erreur lors des inscriptions
                else
                {
                    alert("Erreur lors des inscriptions");
                }
           }
        });
    }

}
```

Le service d'inscription des utilisateurs aux salons est quasiment terminé, il ne reste plus qu'à coder l'action, la fonction du modèle et la vue JSP pour l'appel de l'URL *inscriptionutilisateursalon.do*.

```
<!-- ===action qui permet d'inscrire des utilisateurs
a un salon indiqué=== -->
<action
```

```
    path="/inscriptionutilisateursalon"
    name="FormUtilisateur"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="chatbetaboutique.GestionUtilisateurAction"
    parameter="inscriptionutilisateursalon"
    >
<forward name="inscriptionutilisateursalon"
path="/vues/utilisateur/inscriptionutilisateursalon.jsp" redirect="false"/>
</action>

//inscrire les utilisateurs au salon précisé
public ActionForward inscriptionutilisateursalon(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
{
    //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
    ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
    //créer le modèle
    UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
    //fermer la datasource
    this.ds=null;
    //récupérer le salon en paramètre
    String id_salon=(String)request.getParameter("id_salon");
    //récupérer les validations
    String validation=(String)request.getParameter("inscription");
    //decouper les validations
    String[] tabutilisateur=validation.split("#");
    //inscrire les utilisateurs
    if(id_salon!=null && !id_salon.equals("") && tabutilisateur!=null)
    {
        int resinstruction=utilisateurmodele.inscrireUtilisateurSalon
(id_salon,tabutilisateur);
        request.setAttribute("resinstruction",resinstruction);
        return mapping.findForward("inscriptionutilisateursalon");
    }
    //en cas d'erreur
    return mapping.findForward("accueil");
}
```

```
/**********************************************************
 * inscrire les utilisateurs au salon
**********************************************************/
public int inscrireUtilisateurSalon(String id_salon,String[] tabutilisateur)
{
   int resinstruction=1;
   PreparedStatement requetea=null,requeteb=null;
   try
   {
   //ouvrir une connexion
   connection=ds.getConnection();
   //toujours supprimer les inscriptions pour ce salon
   requetea=connection.prepareStatement("DELETE FROM utilisateursalon WHERE
id_salon=?");
   requetea.setString(1,(String)id_salon);
   resinstruction=requetea.executeUpdate();

   //inscrire les utilisateurs reçus, on utilise pas resinstruction car on
n'a pas forcement d'utilisateur
      for(int i=1; i<tabutilisateur.length;i++)
      {
      requeteb=connection.prepareStatement("INSERT INTO
utilisateursalon(pseudonyme,id_salon) VALUES (?,?)");
      requeteb.setString(1,(String)tabutilisateur[i]);
      requeteb.setString(2,(String)id_salon);
      requeteb.executeUpdate();
      }
```

```
   }
   catch(Exception e)
   {
      System.out.println("Erreur dans la classe UtilisateurModele.java
fonction inscrireUtilisateurSalon");
   }
   finally
   {
      try
      {
         //fermer la connexion
         if(requetea!=null)OutilsBaseDeDonnees.fermerConnexion(requetea);
         if(requeteb!=null)OutilsBaseDeDonnees.fermerConnexion(requeteb);
         if(connection!=null)OutilsBaseDeDonnees.fermerConnexion(connection);
      }
      catch(Exception ex)
      {
         System.out.println("Erreur dans la classe UtilisateurModele.java
fonction inscrireUtilisateurSalon");
      }
   }
   //retourner le résultat de la suppression
   return resinstruction;
}

<%
//resultat de l'instruction
int resinstruction=(Integer)request.getAttribute("resinstruction");
%>
<%= resinstruction %>
```

Il existe dans les précédentes lignes de code deux éléments essentiels au bon fonctionnement de l'ensemble. Le premier, présent dans la fonction *inscrireUtilisateurSalon()*, permet de supprimer toutes les inscriptions au salon indiqué avant de réaliser les nouvelles inscriptions. Comme cela, si l'administrateur décide de décocher toutes les cases d'un salon par exemple, la fonction de suppression sera alors déclenchée mais la boucle de création ne réalisera aucune opération, la suppression totale des inscriptions sera donc bien réalisée.

Enfin la page JSP */vues/utilisateur/inscriptionutilisateursalon.jsp* permet uniquement de retourner l'état de l'action d'inscription qui sera lu dans la fonction JavaScript Ajax en retour pour l'affichage.

14. Gestion des connexions utilisateur et interactivité

Le service d'administration du chat est presque terminé, il reste à développer un service d'affichage des utilisateurs connectés au différents salons. Dans l'étape précédente nous avons développé un service Ajax pour récupérer la liste des utilisateurs inscrits à un salon donné. Nous pouvons développer sur des bases identiques des formulaires dynamiques pour afficher les utilisateurs actuellement connectés.

Le développement commence par la modification de la page */vues/salon/listesalon.jsp* avec l'insertion de balises * * cachées et d'une classe CSS adaptée.

```
...
out.println("<td><a href='javascript:listeConnexionSalon
(\""+salon.getId_salon()+"\");'><img src=\"img/utilisateurconnecte.png\"
border=\"0\" align=\"absmiddle\" title=\"Connexion\"/></a>
<div id=\"listeconnexionsalon_"+salon.getId_salon()+"\"\"
class=\"listeconnexionsalon\"> </div></td>");
...


.listeutilisateursalon,.listeconnexionsalon
{
display:none;
position:absolute;
height:5cm;
width:15cm;
overflow:auto;
z-index:1000;
margin-left:-500px;
border-style:solid;
```

```
border-top-width:1px;
border-right-width:2px;
border-bottom-width:2px;
border-left-width:1px;
border-color:#A8A6A7;
background-color:#fff;
}
```

```
.listeutilisateursalon,.listeconnexionsalon
{
display:none;
position:absolute;
height:5cm;
width:15cm;
overflow:auto;
z-index:1000;
margin-left:-500px;
border-style:solid;
border-top-width:1px;
border-right-width:2px;
border-bottom-width:2px;
border-left-width:1px;
border-color:#A8A6A7;
background-color:#fff;
}
```

Nous pouvons désormais passer au développement de la fonction JavaScript Ajax qui permet de retourner la liste des utilisateurs connectés pour le salon indiqué et d'afficher le résultat dans la balise <*div/*> adaptée.

```
//fonction qui permet d'afficher la liste des connexions au salon
function listeConnexionSalon(id_salon)
{
    //fermer la boîte en cours
    if($("#listeconnexionsalon_"+id_salon).css("display")=="block")
    {
        $("#listeconnexionsalon_"+id_salon).slideUp("slow");
        return;
    }
    //fermer toutes les boîtes qui ont la classe css
    $(".listeconnexionsalon").hide();
    //détruire leur formulaire
    $(".listeconnexionsalon").html("");

    //récupérer la liste des connexions au salon indique
    if(id_salon!=null)
    {
        //envoyer les donnees en POST
        $.ajax(
        {
            type: "POST",
            url: "listeconnexionsalon.do",
            dataType: "html",
            data: "id_salon="+id_salon,
            timeout : 4000,
            error: function(){
        },
            beforeSend : function()
            {
            },
            success: function(html)
            {
                //mettre le résultat dans la balise <div/> concernee
                $("#listeconnexionsalon_"+id_salon).html(html);
                $("#listeconnexionsalon_"+id_salon).slideDown("slow");
            }
        });
    }

}
```

Il ne reste plus qu'à développer l'action avec sa déclaration, le modèle et la vue JSP.

```
<!-- ===action qui permet de retourner la liste des connexions
pour un salon indiqué=== -->
<action
    path="/listeconnexionsalon"
    name="FormUtilisateur"
    scope="request"
    validate="false"
    input="/accueil.do"
    type="chatbetaboutique.GestionUtilisateurAction"
    parameter="listeconnexionsalon"
    >
<forward name="listeconnexionsalon"
path="/vues/utilisateur/listeconnexionsalon.jsp" redirect="false"/>
</action>
```

```
//afficher la liste des connexions pour le salon
public ActionForward listeconnexionsalon(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
{
    //récupérer la datasource du plug-in dans un attribut
présent dans le contexte de la servlet
    ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
    //créer le modèle
    UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
    //fermer la datasource
    this.ds=null;

    //récupérer le salon en paramètre
    String id_salon=(String)request.getParameter("id_salon");
    //récupérer la liste de toutes les connexions pour ce salon
    if(id_salon!=null && !id_salon.equals(""))
    {
    //retourner la liste des connexions
    ArrayList listeconnexionsalon=
(ArrayList)utilisateurmodele.getListeConnexionSalon(id_salon);
    //retourner la liste des connexions
    request.setAttribute("listeconnexionsalon",listeconnexionsalon);
    //retourner le numéro du salon
    request.setAttribute("id_salon",id_salon);
    //vider par sécurité
    listeconnexionsalon=null;
    //retourner sur la page d'affichage des connexions
    return mapping.findForward("listeconnexionsalon");
    }
    //en cas d'erreur
    return mapping.findForward("accueil");
}
```

```
/*********************************************************
 * liste des connexions au salon
*********************************************************/
public ArrayList getListeConnexionSalon(String id_salon)
{
    //statement
    PreparedStatement requete=null;

    try
    {
        //ouvrir une connexion
        connection=ds.getConnection();
        //enregistrements
        requete=connection.prepareStatement("SELECT * FROM utilisateur,
utilisateursalon WHERE utilisateur.pseudonyme=utilisateursalon.pseudonyme
AND utilisateursalon.connecte=1 AND utilisateursalon.id_salon=? ORDER BY
utilisateur.pseudonyme,utilisateur.id_utilisateur");
```

```
        requete.setString(1,(String)id_salon);
        rs=requete.executeQuery();
        //exécuter la requête
        if(rs!=null)
        {
        //utilisation de la librairie qui permet de
transformer un ResultSet en Objet Java
        BeanProcessor bp=new BeanProcessor();
        listeutilisateur =
(ArrayList<Utilisateur>)bp.toBeanList(rs,Utilisateur.class);
        }
    }
    catch(Exception e)
    {
        System.out.println("Erreur dans la classe
UtilisateurModele.java fonction getListeConnexionSalon");
    }
    finally
    {
        try
        {
            //fermer la connexion
            if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
            if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
        }
        catch(Exception ex)
        {
            System.out.println("Erreur dans la classe
UtilisateurModele.java fonction getListeConnexionSalon");
        }
    }

    //retourner la liste des utilisateurs connectés
    return listeutilisateur;
}
```

```
<%@ page import="java.util.ArrayList" %>
<%@ page import="boiteoutils.Utilisateur" %>
<%
//liste des connexions
ArrayList listeconnexionsalon=
(ArrayList)request.getAttribute("listeconnexionsalon");
//numéro du salon
String id_salon=(String)request.getAttribute("id_salon");
%>
<table border="0" id="tableaubordure" cellspacing="0" cellpadding="0">
<tr align="center" class="entetetableau">
<td>ID</td>
<td>Pseudonyme</td>
<td>Nom</td>
<td>Pr&eacute;nom</td>
</tr>
<%
for(int i=0;i<listeconnexionsalon.size();i++)
{
    //récupérer l'objet dans la liste
    Utilisateur utilisateur=(Utilisateur)listeconnexionsalon.get(i);
    if(i%2==0)out.println("<tr align=\"center\" class=\"ligneclaire\">");
    else out.println("<tr align=\"center\" class=\"lignefoncee\">");
    out.println("<td>"+utilisateur.getId_utilisateur()+"</td>");
    out.println("<td>"+utilisateur.getPseudonyme()+"</td>");
    out.println("<td>"+utilisateur.getNom()+"</td>");
    out.println("<td>"+utilisateur.getPrenom()+"</td>");
    out.println("</tr>");
}
%>
</table>
```

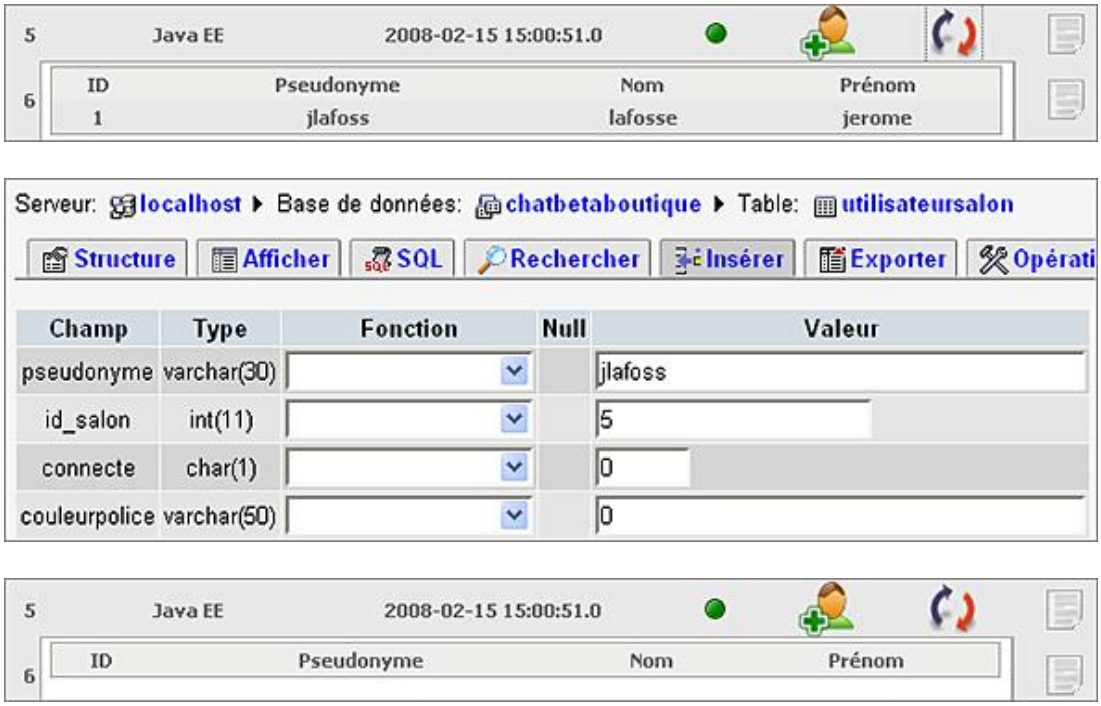
Ce service est opérationnel, nous voyons par exemple que pour le salon d'identifiant *5* qui a pour thème *Java EE*, l'utilisateur *jlafoss* est actuellement connecté. Ce système bien que très utile n'est pas fonctionnel et ergonomique. En effet, si l'utilisateur se déconnecte du salon et que l'administrateur n'actualise pas sa page Web, il ne verra pas le départ de l'utilisateur. Nous allons donc ajouter de l'ergonomie à ce service en utilisant un thread JavaScript qui est en fait un objet *timer*. Ce timer va déclencher à intervalles de temps régulier la fonction de listing pour prendre en compte les connexions et déconnexions utilisateurs.

```
//temps d'attente entre deux rafraichissements
var tempsattente=1500;
//timer
var timer=null;
//fonction qui permet d'afficher la liste des connexions au salon
function listeConnexionSalon(id_salon)
{
    //détruire le timer
    if(timer!=null)
    {
        clearTimeout(timer);
    }
    //fermer la boîte en cours
    if($("#listeconnexionsalon_"+id_salon).css("display")=="block")
    {
        $("#listeconnexionsalon_"+id_salon).slideUp("slow");
        return;
    }
    //fermer toutes les boîtes qui ont la classe css
    $(".listeconnexionsalon").hide();
    //détruire leur formulaire
    $(".listeconnexionsalon").html("");

    //declencher la fonction Ajax qui permet de retourner
la liste des connexions
    getListeConnexionSalon(id_salon);
}

//recuperer la liste des connectes en Ajax
function getListeConnexionSalon(id_salon)
{
    //récupérer la liste des connexions au salon indique
    if(id_salon!=null)
    {
        //envoyer les donnees en POST
        $.ajax(
        {
            type: "POST",
            url: "listeconnexionsalon.do",
            dataType: "html",
            data: "id_salon="+id_salon,
            timeout : 4000,
            error: function(){
        },
            beforeSend : function()
            {
            },
            success: function(html)
            {
                //mettre le résultat dans la balise <div/> concernee
                $("#listeconnexionsalon_"+id_salon).html(html);
                $("#listeconnexionsalon_"+id_salon).show();
            }
        });
    }
    //creer un Timer/thread qui va declencher la fonction par intervalle
```

```
    timer=setTimeout("getListeConnexionSalon("+id_salon+")",tempsattente);
}
```

La fonction JavaScript *getListeConnexionSalon()* permet de récupérer avec l'objet *timer*, par intervalles réguliers, la liste des utilisateurs connectés au salon. Cet objet va déclencher toutes les 1.5 secondes la même fonction pour rafraîchir les connexions. Enfin, la fonction *listeConnexionSalon()* commence par détruire, s'il existe, l'objet *timer* lors des fermetures des blocs <*div/*>.

Nous pouvons tester ce service en utilisant deux navigateurs et en affichant un salon. Dans le premier navigateur, nous voyons la liste des connectés et dans le second nous réalisons la déconnexion au salon avec *PhpMyAdmin* par exemple, l'affichage du résultat doit être quasiment instantané.



Le service d'administration de gestion du chat est maintenant terminé. Le paragraphe suivant de ce chapitre est consacré aux technologies WEB 2.0 et aux différents services associés.

Web 2.0

1. Présentation

Actuellement, les protagonistes de l'Internet parlent beaucoup de technologies Web 2.0. Derrière ce terme se cache un ensemble de technologies et d'outils axés sur l'ergonomie des interfaces. Le terme a été inventé par un membre de la société O'Reilly pour désigner les nouvelles technologies et la renaissance du Web. Le but est d'utiliser les technologies de l'Internet tout en se rapprochant des interfaces homme machine attractives des logiciels installés sur les machines personnelles.

La définition exacte du Web 2.0 n'est toujours pas très claire, cependant il est admis qu'un site Web 2.0 possède les caractéristiques suivantes :

- La saisie, modification et suppression des informations du site sont simples.
- Le site est totalement utilisable avec un navigateur standard.
- L'outil utilise des standards de technologie.

Du point de vue technologique, l'infrastructure Web 2.0 est assez complexe, elle inclut à la fois le ou les serveur(s), la syndication (accessibilité partagée) de contenu, la messagerie et les applications. Un site Web 2.0 est désigné comme tel s'il utilise les technologies suivantes :

- Les feuilles de style CSS.
- Les pages XHTML.
- La syndication RSS (usage des données du site dans un autre contexte).
- L'utilisation d'URL spécifiques pour le référencement.
- Une Application Internet Riche (*Rich Internet Application*) utilisant la technologie Ajax.
- L'étiquetage (métadonnées : utilisation de mots-clés ou nuages de mots-clés pour les recherches dans un contenu, le but étant d'interconnecter les éléments entre eux).
- Utilisation de l'approche HTTP et SOAP.

Il n'existe pas d'accord unanime sur la définition et le sens du Web 2.0, le terme peut ainsi désigner des concepts radicalement différents suivant les personnes. Par exemple, certains associent le terme Web 2.0 pour des sites XHTML valides et bien formés. D'autres parlent de Web 2.0 avec l'utilisation abusive d'Ajax qui peut rendre les pages Web particulièrement longues au chargement. Pour résumer, les technologies qui se cachent derrière ce terme sont surtout JavaScript, DHTML et Ajax avec un respect des standards et du code valide syntaxiquement.

Dans cette partie du cours nous allons aborder quelques services Web 2.0 pour faire évoluer notre système de gestion du chat BetaBoutique.

2. Tableaux redimensionnables

Il est parfois très utile de bénéficier de tableaux redimensionnables en DHTML avec l'utilisation de la souris. Nous allons mettre en place ce service avec l'utilisation d'un plug-in pour la bibliothèque JQuery. Ce plug-in nommé *gridcolumnsizing*, nécessite l'utilisation de la librairie de base JQuery et fonctionne sur les tableaux HTML.

Nous commençons notre mise en place en copiant les librairies du plug-in *gridcolumnsizing* dans le répertoire */javascript/jquery/plugin* de notre application.

Ensuite, nous devons installer ces librairies dans notre page d'en-tête afin d'inclure les fichiers.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
```

```
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html>
<head>
<title>Administration - BetaBoutique</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta name="description" content="BetaBoutique">
<!-- feuilles de style -->
<link rel="stylesheet" type="text/css" href="css/styles.css"
title="defaut" />
<link rel="stylesheet" type="text/css" href="css/styles_admin.css"
title="defaut" />
<!--  librairie personnelle -->
<script type="text/javascript" src="javascript/boiteoutils.js"></script>
<!--  librairie JQuery -->
<script type="text/javascript" src="javascript/jquery/jquery.js"></script>
<!--  plug-in pour les redimensionnements de colonnes des tableaux -->
<script type="text/javascript" src="javascript/jquery/plugin/
gridcolumnsizing/jquery.dimensions.pack.js"></script>
<script type="text/javascript" src="javascript/jquery/plugin/
gridcolumnsizing/jquery.cookies.pack.js"></script>
<script type="text/javascript" src="javascript/jquery/plugin/
gridcolumnsizing/jquery.iutil.pack.js"></script>
<script type="text/javascript" src="javascript/jquery/plugin/
gridcolumnsizing/jquery.idrag.js"></script>
<script type="text/javascript" src="javascript/jquery/plugin/
gridcolumnsizing/jquery.grid.columnSizing.pack.js"></script>
<script type="text/javascript" src="javascript/jquery/plugin/
gridcolumnsizing/jquery.tabs.pack.js"></script>
</head>
<body>
```

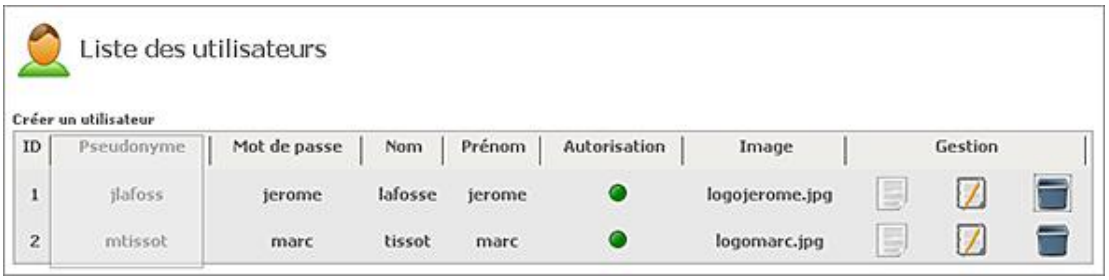
La mise en place nécessite la déclaration du service Web 2.0 en JavaScript dans la page d'en-tête par exemple.

```
<!--  outil de redimensionnement -->
<script>
$(document).ready(function(){
  //pour les en-tetes de colonnes des tableaux
  $("#tableaubordure").columnSizing({
        viewGhost : true,
        viewResize : true,
        opacity : 0.5,
        dleft : 0,
        dtop : 0,
        title : "Redimensionner la colonne",
        minWidth : 50,
        tableWidthFixed : false,
        cookies : false,
        cssHandler :
        {
        position: "relative",
        right:"-3px",
        float:"right",
        height:"20px",
        cursor:"col-resize",
        borderRight:"2px solid #fff",
        marginRight:"2px",
        borderLeft:"1px solid #555",
        },
        cssDragLine :
        {
        borderRight:"4px solid #777",
        cursor:"col-resize"
        },
        cssDragArea :
        {
        border:"1px solid #777",
        backgroundColor:"#eee",
        cursor:"col-resize"
```

```
        }
        });
});
</script>
```

Le script précédent est simple, il permet d'appliquer au chargement de l'application, à chaque balise HTML qui possède un identifiant nommé *tableaubordure*, le service de redimensionnement avec les styles CSS. Désormais, notre tableau qui permet d'afficher la liste des utilisateurs est initialisé avec le redimensionnement automatique. Nous pouvons tester notre service en déplaçant la souris sur les différentes colonnes.

```
...
<table border="0" id="tableaubordure" cellspacing="0" cellpadding="0">
...
```



3. Champs redimensionnables

De la même façon que nous avons utilisé un service de redimensionnement pour les tableaux HTML, nous pouvons gérer les champs des formulaires. Pour cela nous utilisons le plug-in JQuery nommé *resizehandle* qui permet des redimensionnements sur les champs de type <*textarea.../*> et <*input.../*>.

Nous commençons la mise en place de ce service avec la copie des librairies dans le répertoire */javascript/jquery/plugin/resizer*. Cette librairie contient deux scripts JavaScript (pour la gestion de l'horizontale et de la verticale), deux images et une feuille de style CSS.

Nous incluons ces fichiers dans la page */vues/outils/entete.jspf*.

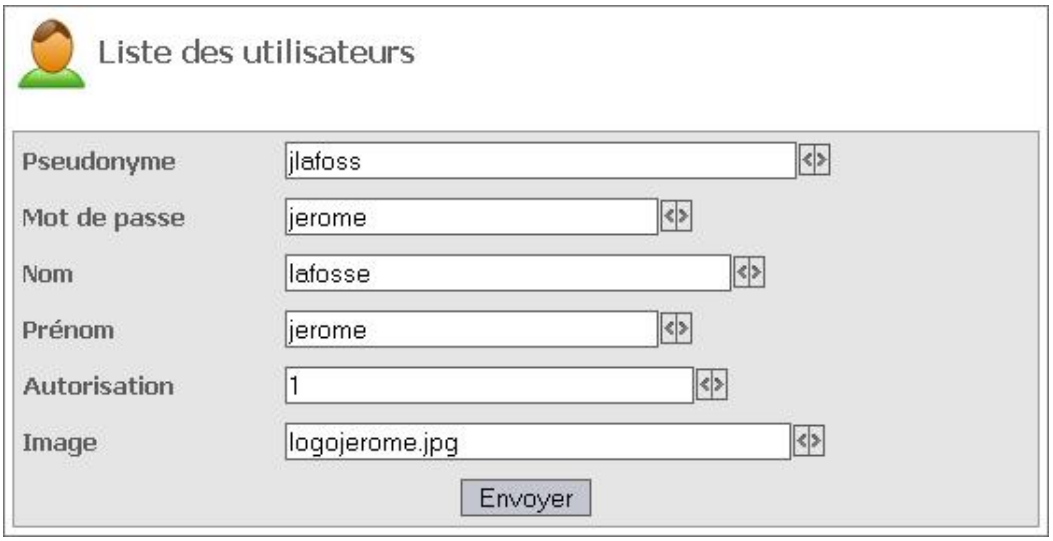
```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html>
<head>
...
<!--  pour les redimensionnements -->
<script type="text/javascript" src="javascript/jquery/plugin/
resizer/resizehandle.js"></script>
<script type="text/javascript" src="javascript/jquery/plugin/
resizer/resizehandleH.js"></script>
</head>
<body>
...
```

L'installation est quasiment terminée, il ne reste plus qu'à déclarer les champs de la page qui seront redimensionnables et à insérer la feuille de style CSS.

```
<!--  feuilles de style pour resizer -->
<link rel="stylesheet" type="text/css"
href="javascript/jquery/plugin/resizer/resize.css" />

<!--  outil de redimensionnement -->
<script>
$(document).ready(function(){
  ...
  //redimension forcee a l'horizontale
  $(".input").resizehandleH(98,350);
  //textearea redimensionnable
  $("textarea").resizehandle();
});
```

```
</script>
```

Dans cette déclaration, tous les champs qui sont associés à la classe CSS *input* auront le service de redimensionnement horizontal avec 98 pixels au minimum et 350 pixels au maximum. Enfin, chaque champ de type <*textarea*> sera redimensionnable verticalement.



4. Bulles d'aide

Il est assez courant d'avoir besoin de créer des textes, images, boutons ou autre avec des bulles d'aide. Nous pouvons créer ceci avec des styles CSS de façon simple.

```
a.info
{
position:relative;
text-decoration:none
}
a.info:hover
{
color:#BEBCBC;
z-index:10;
cursor:help;
}
a.info span
{
display:none;
z-index:10;
}
a.info:hover span
{
display:block;
position:absolute;
z-index:10;
top:2em;
left:-130px;
width:220px;
border-style:solid;
border-left-width:1px;
border-top-width:1px;
border-right-width:2px;
border-bottom-width:2px;
border-top-color:#999999;
border-left-color:#999999;
border-right-color:#666666;
border-bottom-color:#666666;
padding-left:10px;
padding-right:10px;
padding-top:3px;
```

```
padding-bottom:5px;
margin-left:10px;
background-color:#F6F6F6;
font-family:tahoma, verdana, arial, sans-serif;
font-size:11px;
color:#686667;
font-weight:normal;
text-align:left;
}
```

Cette définition de styles permet de déclarer une classe nommée *info* qui sera intégrée dans un lien HTML. Chaque balise *<a href=””...>* qui contient un élément avec la classe *info* pourra bénéficier de bulles d'aide. Nous pouvons par exemple ajouter une bulle d'aide sur chaque lien du menu de navigation.

```
<table border="0" cellspacing="0" cellpadding="0" name="menugauche"
 id="menugauche" style="background: url('img/fondmenu.gif')
repeat-x;"  height="500">
<tr>
  <td width="100%" valign="top"><br/><span style="margin-left:
30px;"><b>ADMINISTRATION</b></span>
   <ul class="menucategorie">
       <li><a href="etatserveur.do" class="info"><img
src="img/fleche.gif" border="0" align="absmiddle"/><span>Cliquez ici
pour afficher le service de gestion du serveur</span> Gestion
du serveur</a></li>
       <li><a href="listeutilisateur.do" class="info"><img
src="img/fleche.gif" border="0" align="absmiddle"/><span>Cliquez ici
pour afficher le service de gestion des utilisateurs</span> Gestion
des utilisateurs</a></li>
   </ul>
  </td>
</tr>
</table>
```

Le lien possède la classe *info*, tout ce qui sera inclus à la suite de cette classe dans une balise ** sera alors considéré comme le texte de l'aide et sera alors déclenché au survol de la souris utilisateur.

5. Menu contextuel

Il est parfois utile d'utiliser un menu contextuel pour gérer les clics droits sur des images (pour afficher un copyright) ou liens (pour afficher un menu d'actions). Le plug-in *contextmenu* de JQuery permet de gérer ces clics et d'afficher un contenu adapté en conséquence. Comme d'habitude nous copions d'abord les librairies dans le répertoire adapté */javascript/jquery/plugin/contextmenu*. Ensuite, nous insérons les librairies dans la page d'en-tête de notre projet.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
```

```
<html>
<head>
<title>Administration - BetaBoutique</title>
...
<!--  pour les clics droits -->
<script type="text/javascript" src="javascript/jquery/plugin/
contextmenu/jquery.contextmenu.r2.packed.js"></script>
<script type="text/javascript" src="javascript/jquery/plugin/
contextmenu/contextmenu.js"></script>
</head>
<body>
...
```

Maintenant, nous pouvons coder la partie JavaScript pour la gestion du clic droit. Le fichier */javascript/jquery/plugin/contextmenu/contextmenu.js* possède le code suivant :

```
//au chargement de la page
$(document).ready(function(){
 //declencher la fonction pour la gestion du clic droit
 gestionClicDroit();
});

//pour le style du menu contextuel
$.contextMenu.defaults({

    menuStyle : {
      width : "110px",
    },

    shadow: true
});

//gerer les clics droits sur les liens de contexte
function gestionClicDroit()
{
      $('.contextmenu').contextMenu('clicdroit', {

        bindings: {

          'consulter': function(t) {
            document.location.href=
"consulterutilisateur.do?id_utilisateur="+t.id;
          },

          'modifier': function(t) {
            document.location.href=
"modifierutilisateur.do?id_utilisateur="+t.id;
          },

        }

      });
}
```

Ce code permet de lancer la méthode *gestionClicDroit()* au lancement de l'application. Cette méthode permet d'ajouter un menu contextuel sur chaque balise HTML qui utilise la classe *contextmenu*. Cette fonction est associée à un bloc d'affichage nommé *clicdroit*. Ensuite, deux actions sont définies consulter et modifier. Pour améliorer l'ergonomie, nous pouvons ajouter le style CSS suivant avec la classe *contextmenu* qui permet d'afficher l'icône d'aide (le point d'interrogation) lors des survols de la souris.

```
.contextmenu
{
cursor:help;
}
```

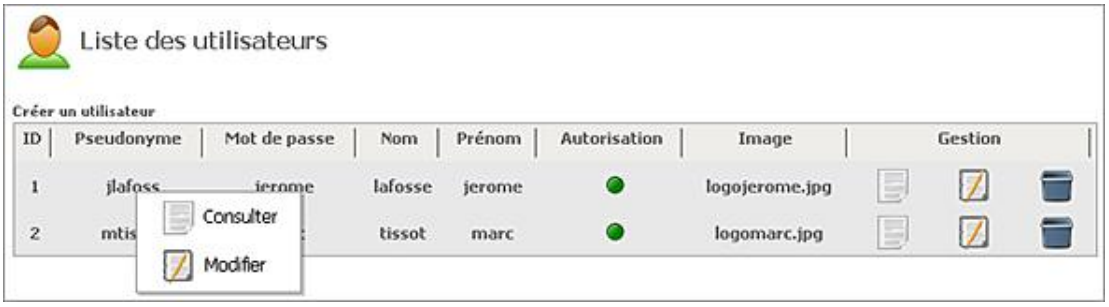
Les styles sont insérés ainsi que le code de gestion, il reste maintenant à coder l'apparence de la boîte du menu contextuel. La page *listeutilisateur.jsp* est détaillée ci-après :

```
<%
...
   out.println("<td><span class=\"contextmenu\"
id=\""+utilisateur.getId_utilisateur()+"\">"+utilisateur.getPseudonyme()
+"</span></td>");
...
%>
</table>
<%@ include file="../outils/piedpage.jspf" %>
<!--  gestion du clic droit -->
<div class="contextMenu" id="clicdroit" style="display:none">
  <ul>
    <li id="consulter"><img src="img/consulter.png"/> Consulter</li>
    <li id="modifier"><img src="img/modifier.png"/> Modifier  
</li>
  </ul>
</div>
```

Nous remarquons que la balise HTML ** associée à la classe *contextmenu* permet de gérer le menu qui est inséré et caché en bas de page. Le paramètre *id* de la balise ** est lié au paramètre *t.id* du fichier JavaScript afin de récupérer l'identifiant à traiter.

```
'consulter': function(t) {
            document.location.href=
"consulterutilisateur.do?id_utilisateur="+t.id;
         },
```

Le service est désormais disponible. Nous pouvons tester l'utilisation des liens *consulter* et *modifier* pour un utilisateur donné, en réalisant un clic droit sur son pseudonyme.



6. Les arrondis

Avec la montée en puissance des technologies Web 2.0, il y a de plus en plus de sites qui utilisent des arrondis. Le plug-in *corner* de la librairie JQuery permet de gérer des arrondis sur des blocs HTML de type * *. L'utilisation de cette librairie est identique à celle des étapes précédentes. Nous commençons par copier le répertoire adapté */javascript/jquery/plugin/corner*. Ensuite, nous déclarons le script JavaScript dans notre page d'en-tête afin d'utiliser ce service.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html>
<head>
<title>Administration - BetaBoutique</title>
...
<!--  pour les arrondis -->
<script type="text/javascript" src="javascript/jquery/plugin/
corner/jquery.corner.js"></script>
</head>
<body>
```

Nous pouvons ensuite ajouter des arrondis sur des blocs par simple JavaScript. Pour cela, nous allons insérer un arrondi sur le tableau HTML qui a comme id *tableaubordure*.

```
<script>
```

```
$(document).ready(function(){
  ...
  //coins arrondis
  $("#tableaubordure").corner();
});
</script>
```

7. Aide dynamique et remplissages génériques

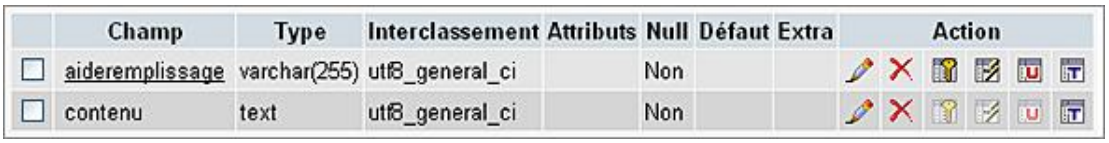
Lors du développement d'un site Internet même basique, il est toujours nécessaire d'avoir des pages d'aide pour bénéficier d'informations tout au long de la navigation. La réalisation des pages HTML est souvent très longue et laborieuse. De même, les mises à jour de textes statiques sont très rares étant donné que les utilisateurs ne sont pas des développeurs HTML. Pour éviter cela, il est souvent très utile de développer un système modulable et générique pour la gestion des pages d'aide. De même, il existe toujours des pages statiques dans un site Internet pour les conditions de vente, les informations diverses, les frais de port ou les explications sur l'entreprise. Il est alors tout à fait possible de créer un service de remplissage avec du contenu XHTML qui pourra être mis à jour par la suite.

Nous allons créer ce service d'aide et de remplissage générique sur le même principe que la gestion des utilisateurs ou des salons. Nous commençons le développement de ce service avec le codage de la classe *AideRemplissage*.

```
package boiteoutils;

import java.io.Serializable;

public class AideRemplissage implements Serializable
{
    //java 5.0 pour avoir un identifiant unique de la sérialisation
    private static final long serialVersionUID = 1L;

    //variables de classe
    private String aideremplissage=null;
    private String contenu=null;

/*********************************************************
*********************************************************/
public AideRemplissage() {
}
public String getAideremplissage() {
    return aideremplissage;
}
public void setAideremplissage(String aideremplissage) {
    this.aideremplissage = aideremplissage;
}
public String getContenu() {
    return contenu;
}
public void setContenu(String contenu) {
    this.contenu = contenu;
}

//fin de la classe
}
```

La structure de la table *aideremplissage* est la suivante : le champ *aideremplissage* est utilisé comme clé et le champ

contenu sera composé de texte brut ou XHTML.



	Champ	Type	Interclassement	Attributs	Null	Défaut	Extra	Action
☐	aideremplissage	varchar(255)	utf8_general_ci		Non			
☐	contenu	text	utf8_general_ci		Non			

Le développement n'est pas détaillé ici, la conception est toujours la même avec la déclaration des actions dans le fichier de configuration *struts-config.xml*, le codage de la classe composée des actions *chatbetaboutique.GestionAideRemplissage*, le codage du modèle *chatbetaboutique.AideRemplissageModele* et les différentes vues présentes dans le répertoire */vues/aideremplissage/*.

La partie administration des pages d'aide et de remplissage est réalisée. Nous pouvons passer au développement d'une page JSP nommée */vues/outils/aideremplissage.jsp* qui permet d'afficher un contenu dynamique à l'aide de la fonction *getContenuAideRemplissage()* du modèle.

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ page import="javax.sql.DataSource" %>
<%@ page import="modele.AideRemplissageModele" %>
<%
//récupérer la datasource du plug-in dans un attribut
présent dans le context de la servlet
DataSource ds=(DataSource)this.getServletContext().getAttribute
("datasource");
//créer le modèle
AideRemplissageModele aideremplissagemodele=new AideRemplissageModele(ds);
//fermer la datasource
ds=null;
//récupérer la page d'aide ou de remplissage de manière dynamique
StringBuffer
contenu=aideremplissagemodele.getContenuAideRemplissage
((String)request.getParameter("aideremplissage"));
%>
<html>
<head>
<title>Aide Remplissage</title>
</head>
<body>
<%= contenu %>
</body>
</html>
```

```
/*********************************************************
 * récupérer une page d'aide ou de remplissage
*********************************************************/
public StringBuffer getContenuAideRemplissage(String aideremplissage)
{
    PreparedStatement requete=null;
    //contenu de l'aide ou du remplissage
    StringBuffer contenu=new StringBuffer();

    //récupérer la page d'aide ou de remplissage indiquée
    if(aideremplissage!=null && !aideremplissage.equalsIgnoreCase(""))
    {
        try
        {
            //ouvrir une connexion
            connection=ds.getConnection();
```

```
            requete=connection.prepareStatement("SELECT contenu
FROM aideremplissage WHERE aideremplissage.aideremplissage=?");
            requete.setString(1,(String)aideremplissage);
            rs = requete.executeQuery();
            //exécuter la requête
            if(rs!=null)
            {
                if(rs.next())
                {
                    //récupérer la aide
                    if(rs.getString("contenu")==null)contenu.append("");
                    else contenu.append(rs.getString("contenu"));
                }
            }
        }
        catch(Exception e)
        {
            System.out.println("Erreur dans la classe
AideRemplissageModele.java fonction getContenuAideRemplissage");
        }
    finally
        {
            try
            {
                fermer la connexion
                if(rs!=null)OutilsBaseDeDonnees.fermerConnexion(rs);
                if(requete!=null)OutilsBaseDeDonnees.fermerConnexion
(requete);
                if(connection!=null)OutilsBaseDeDonnees.fermerConnexion
(connection);
            }
            catch(Exception ex)
            {
                System.out.println("Erreur dans la classe
AideRemplissageModele.java fonction getContenuAideRemplissage");
            }
        }
    }

    //retourner le contenu de l'aide ou du remplissage
    return contenu;
}
```

Le plug-in JTip de la librairie JQuery permet d'afficher un contenu dynamique à partir d'un système générique et ergonomique. La librairie est copiée dans le répertoire */javascript/jquery/plugin/jtip*. L'inclusion de cette librairie est réalisée dans le fichier d'en-tête */vues/outils/entete.jspf*.

```
<!--  pour l'aide et les remplissages -->
<script type="text/javascript"
src="javascript/jquery/plugin/jtip/jtip.js"></script>
<link type="text/css" rel="stylesheet" media="all"
href="javascript/jquery/plugin/jtip/jtip.css"/>
```

Le service est opérationnel, nous pouvons modifier notre page de navigation */vues/outils/navigation.jspf* afin d'utiliser une aide dynamique.

```
...
<li><a href="listeutilisateur.do" class="info"><img
src="img/fleche.gif" border="0" align="absmiddle"/> Gestion
des utilisateurs</a><a href="vues/outils/aideremplissage.jsp?
aideremplissage=aideutilisateur&width=400" class="jTip"
id="aidedynamique" name="Aide et Remplissage"><img
src="img/aide.gif" align="absmiddle" border="0"
style="margin-left:20px;"/></a>
</li>
...
```

La classe *jTip* utilisée sur un lien permet d'activer le module avec en paramètres, le nom de l'aide ou du remplissage à afficher (*aideutilisateur*), la taille de fenêtre (*width=400*), l'id de la fenêtre (*aidedynamique*) et le titre qui va apparaître

dans la fenêtre avec le paramètre *name*.

Nous pouvons ainsi ajouter aussi bien en partie front-office qu'en administration des icônes d'aide dynamique, dont le contenu est présent dans une base de données.

8. Éditeur de texte évolué

La technologie DHTML permet actuellement d'utiliser des éditeurs de texte riche, en anglais RTE (*Rich Text Editor*). Ces objets JavaScript sont plus ou moins lourds, bien développés et utiles.

Le RTE *TinyMCE* est un éditeur de type barre d'outils 100% XHTML/XML qui permet de réaliser les différentes étapes de mise en forme dans des pages HTML. http://tinymce.moxiecode.com/.

Nous pouvons installer cet éditeur pour gérer les contenus des pages d'aide ou de remplissage. Nous copions l'archive *tinymce* dans le répertoire */javascript* de l'application. Ensuite, nous éditons les fichiers de création */vues/aideremplissage/creeraideremplissag.jsp* et de modification */vues/aideremplissage/modifieraideremplissage.jsp* puis nous ajoutons le code adapté pour l'inclusion de l'éditeur DHTML afin de gérer le champ de type *textarea* de la page en mode RTE.

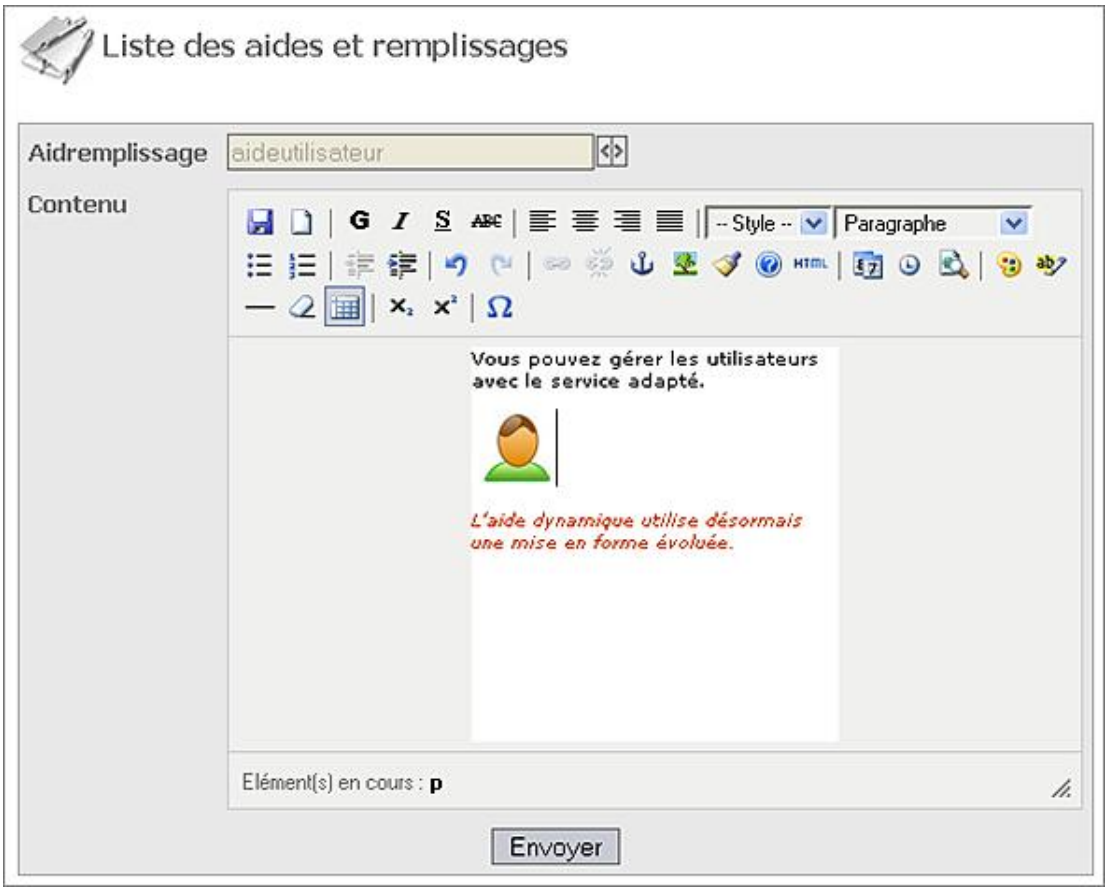
```
<!--  inclure l'en-tete de la page -->
<%@ include file="../outils/entete.jspf" %>

<!--  inclure l'éditeur syntaxique -->
<script language="javascript" type="text/javascript"
src="javascript/tinymce/jscripts/tiny_mce/tiny_mce.js"></script>
<script language="javascript" type="text/javascript">
tinyMCE.init({
        mode : "exact",
        elements : "aideremplissage.contenu",
        theme : "advanced",
        language : "fr",
        plugins : "table,save,advhr,advimage,advlink,emotions,
iespell,insertdatetime,preview,zoom,flash,searchreplace,print,
contextmenu,paste,directionality,fullscreen",
        _advanced_buttons1_add_before : "save,newdocument,separator",
        theme_advanced_buttons2_add : "separator,insertdate,
inserttime,preview,zoom,separator,forecolor,backcolor",
        theme_advanced_toolbar_location : "top",
        theme_advanced_toolbar_align : "left",
        theme_advanced_statusbar_location : "bottom",
        content_css : "css/styles_admin.css",
        plugi2n_insertdate_dateFormat : "%d-%m-%Y",
        plugi2n_insertdate_timeFormat : "%H:%M:%S",
        theme_advanced_resizing : true,
        theme_advanced_resize_horizontal : false,
        paste_auto_cleanup_on_paste : true,
        entities : "",
        entity_encoding : "numeric"
});
</script>
<br/><div class="titre"><a href="listeaideremplissage.do"><img
src="img/aideremplissage.jpg" border="0" align="absmiddle"/> Liste
des aides et remplissages</a></div>
```

```
<html:form action="/validercreeraideremplissage" method="POST">
<table border="0" id="tableau" cellpadding="0" cellspacing="0" width="50%">
<tr><td>Aideremplissage</td><td><html:text
property="aideremplissage.aideremplissage"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td valign="top">Contenu</td><td><html:textarea
property="aideremplissage.contenu"/></td></tr>
<tr><td align="center" colspan="2"><html:submit value="Envoyer"
styleClass="bouton"/></td></tr>
</table>
</html:form>

<!--  inclure le pied de page -->
<jsp:include page="../outils/piedpage.jsp" />
```

La mise en place est simple, il suffit d'inclure les librairies et de donner le même nom à l'éditeur (attribut *elements*) qu'au champ de type *<textarea/>* (attribut *name*) qui va recevoir celui-ci. Il est aussi possible d'utiliser plusieurs options de l'éditeur comme le type de barre d'outils, le style de l'éditeur, les plug-ins utilisés, l'encodage des entités XML, le type d'encodage et surtout, lier l'éditeur à notre feuille de style avec le paramètre *content_css*. Avec cette option nous pourrons ainsi appliquer nos propres styles au contenu.

La librairie de l'éditeur n'est pas incluse de manière globale dans l'en-tête JSPF de notre projet comme pour les autres JavaScript car cette bibliothèque est lourde. Il n'est donc pas nécessaire d'inclure partout cette bibliothèque alors qu'elle sera uniquement utilisée dans le formulaire de création et de modification du contenu d'une aide ou d'un remplissage.

Nous pouvons maintenant réaliser une aide complexe avec des styles ou des images et disposer ainsi d'un contenu riche inséré avec un éditeur 100% XHTML.



9. Date et calendrier

La gestion des champs de type date est parfois contraignante du fait des différentes syntaxes utilisées. En effet, certains utilisateurs vont saisir *jj/mm/aaaa*, d'autres *jj-mm-aaaa* ou encore *aaaa-mm-jj*. L'utilisation d'un calendrier DHTML est donc plus souple et ergonomique. La librairie calendar (http://www.dynarch.com/projects/calendar/) permet d'utiliser plusieurs formulaires de type date au sein de la même page HTML. Nous allons procéder à l'installation de la librairie */javascript/calendar* ainsi qu'à l'inclusion des fichiers dans la page */vues/outils/entete.jspf*.

```
<!--  pour les calendriers -->
<link rel="stylesheet" type="text/css" media="all"
href="javascript/calendar/calendar-win2k-cold-1.css">
<script type="text/javascript"
src="javascript/calendar/calendar.js"></script>
<script type="text/javascript"
src="javascript/calendar/calendar-fr.js"></script>
<script type="text/javascript"
src="javascript/calendar/calendar-setup.js"></script>
```

Nous allons ensuite installer un calendrier dynamique sur le champ date du formulaire de création */vues/salon/creersalon.jsp* et de modification d'un salon */vues/salon/modifiersalon.jsp*.

```
<!--  inclure l'en-tete de la page -->
<%@ include file="../outils/entete.jspf" %>

<br/><div class="titre"><a href="listesalon.do"><img
src="img/salon.jpg" border="0" align="absmiddle"/> Liste
des salons</a></div>

<html:form action="/validercreersalon" method="POST">
<table border="0" id="tableau" cellpadding="0" cellspacing="0" width="50%">
<tr><td>Th&egrave;me</td><td><html:text property="salon.theme"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Date</td><td><html:text styleId="salon.date" property="salon.date"
styleClass="input" errorStyleClass="inputerreur"/></td></tr>
<tr><td>Actif</td><td><html:text property="salon.actif" styleClass="input"
errorStyleClass="inputerreur"/></td></tr>
<tr><td align="center" colspan="2"><html:submit value="Envoyer"
styleClass="bouton"/></td></tr>
</table>
</html:form>
<script type="text/javascript">
    Calendar.setup({
        inputField     :    "salon.date", // id of the input field
        ifFormat       :    "%Y-%m-%d %H:%M:%S",   //
format of the input field
        showsTime      :    "true",
        timeFormat     :    "24"
    });
</script>

<!--  inclure le pied de page -->
<jsp:include page="../outils/piedpage.jsp" />
```

L'identifiant du champ est précisé avec l'attribut *styleID* et le code JavaScript associé (*inputField*). Le format de la date est précisé avec le paramètre *ifFormat*. Dans notre cas, nous utilisons le format TimeStamp pour être en accord avec notre type dans la base de données. Pour utiliser plusieurs calendriers dans la même page, il suffit de réaliser autant de portions de code JavaScript que de calendrier.

L'utilisation du calendrier nécessite la présence de la balise *id* dans le champ HTML. Avec l'utilisation de la taglib Struts, il est donc nécessaire de préciser ce paramètre avec l'attribut *styleID*.

10. Effets d'attente

Il est parfois nécessaire de réaliser des effets d'attente sur des formulaires Ajax afin d'indiquer un chargement, une modification du contenu ou autre. Nous allons ajouter un effet d'attente sur le formulaire de gestion des connexions utilisateurs aux salons. Pour rappel, la fonction JavaScript *listeConnexionSalon()*, présente dans le fichier *boiteoutils.js*, permet de récupérer en Ajax toutes les X secondes la liste des utilisateurs actuellement connectés.

Nous allons modifier la fonction JavaScript *getListeConnexionSalon()* afin de réaliser une animation d'attente. Pour cela, nous utilisons la fonction *beforeSend()* afin de placer une animation d'attente dans la balise <*div*/> concernée avec un fond noir, un effet de transparence et une animation. Ensuite, nous réalisons l'opération inverse dans la fonction *success()* qui est exécutée à la fin de l'opération. Nous augmentons également le temps de rafraîchissement à 4 secondes.

```
//recuperer la liste des connectes en Ajax
function getListeConnexionSalon(id_salon)
{
    //récupérer la liste des connexions au salon indique
    if(id_salon!=null)
    {
        //envoyer les donnees en POST
        $.ajax(
        {
            type: "POST",
            url: "listeconnexionsalon.do",
            dataType: "html",
            data: "id_salon="+id_salon,
            timeout : 4000,
            error: function(){
            },
        beforeSend : function()
        {
        //effet d'ouverture noire
        $("#listeconnexionsalon_"+id_salon).css("background-color","#000");
        $("#listeconnexionsalon_"+id_salon).fadeTo("slow",0.33);
        //animation d'attente
        $("#listeconnexionsalon_"+id_salon).html('<img
```

```
src="img/ajax-loader.gif" style="margin-top:50px;"/>');
        },
        success: function(html)
        {
        //effet de fermeture noire
        $("#listeconnexionsalon_"+id_salon).css("background-color",
"#E8E8E8");
        $("#listeconnexionsalon_"+id_salon).fadeTo("slow",1);
        //mettre le résultat dans la balise <div/> concernee
        $("#listeconnexionsalon_"+id_salon).html(html);
        $("#listeconnexionsalon_"+id_salon).show();
        }
        });
    }

    //creer un Timer/thread qui va declencher la fonction par intervalle
    timer=setTimeout("getListeConnexionSalon("+id_salon+")",tempsattente);
}
```

Nous remarquons l'affichage d'une image animée au format .gif pendant le chargement des données en Ajax.

11. Feuilles de style dynamiques

Parfois, il est nécessaire de proposer sur un site plusieurs tailles de polices, différentes couleurs ou encore mieux, deux compositions graphiques différentes. Actuellement avec le langage JavaScript, il est possible de changer de feuille de style à la volée. Nous allons utiliser pour cela la librairie JavaScript */javascript/styleswitcher*.

```
<!--  pour les feuilles de style -->
<script type="text/javascript" src="javascript/styleswitcher.js"></script>
```

Lors de la déclaration d'une feuille de style, le paramètre *title* permet de donner un nom à la feuille et ainsi de la référencer.

```
<!-- feuilles de style -->
<link rel="stylesheet" type="text/css" href="css/styles.css"
title="defaut" />
<link rel="alternate stylesheet" type="text/css" href="css/tableau.css"
title="tableau" />
```

La déclaration ci-dessus permet d'insérer une feuille de style par défaut nommée *styles.css*. Par contre, il existe une seconde feuille de style nommée *tableau* qui sera chargée en cas de besoin (*rel="alternate stylesheet"*). Dans cette feuille de style, nous modifions uniquement la classe *tableaubordure* pour proposer une autre apparence. Nous pourrions changer l'image de fond, la taille de toutes les polices, réaliser une mise en forme différente pour l'impression ou modifier la totalité des styles.

```
#tableaubordure{
margin-top:2px;
border-style:solid;
border-width:1px;
border-color:#000;
```

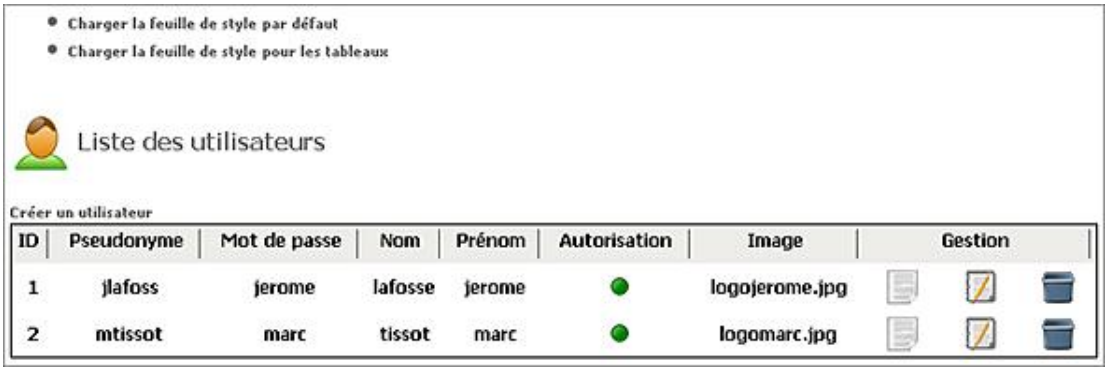
```
background-color:#fff;
color:#000;
font-family:tahoma, verdana, arial, sans-serif;
font-size:12px;
font-weight:bold;
width:98%;
}
#tableaubordure td{
padding-top:3px;
padding-right:3px;
padding-bottom:3px;
padding-left:3px;
}
```

Nous terminons par l'ajout de liens dynamiques dans la page d'en-tête */vues/outils/entete.jspf*.

```
<ul>
   <li><a href="JavaScript:chargerFeuilleStylesCss('defaut');">Charger
la feuille de style par défaut</a></li>
   <li><a href="JavaScript:chargerFeuilleStylesCss('tableau');">Charger
la feuille de style pour les tableaux</a></li>
</ul>
```

Enfin, le code de la fonction JavaScript *chargerFeuilleStylesCss()* présente dans le fichier */javascript/boiteoutils.js* est très simple.

```
//changeant dynamiquement de feuille de style
function chargerFeuilleStylesCss(nom)
{
  if(nom!=null && nom!="")
  {
      setActiveStyleSheet(nom);
  }
}
```



Il existe également une feuille de style qui est utilisée par défaut pour les impressions en CSS. Cette feuille de style est chargée lors des impressions ou des aperçus avant impression. Cette feuille est distinguée par le paramètre *media* qui a pour valeur *print*.

Nous allons réaliser une feuille de style pour l'impression (*impressions.css*) par simple copier-coller de la feuille principale *styles.css* et en modifiant l'apparence du bloc en-tête.

```
<!-- feuilles de style -->
<link rel="stylesheet" type="text/css" href="css/styles.css"
title="defaut" />
<link rel="alternate stylesheet" type="text/css"
href="css/tableau.css" title="tableau" />
<link rel="stylesheet" type="text/css" href="css/impression.css"
media="print" />

#en-tete
{
height:150px;
border-style:solid;
border-left-width:0px;
```

```
border-top-width:0px;
border-right-width:0px;
border-bottom-width:1px;
background-color:#FFF;
align:left;
display:none;
}
```

Cet exemple simple permet de cacher le bloc d'en-tête à l'impression avec la directive *display:none*. Il est possible de modifier la taille des polices, les couleurs, l'apparence des blocs, leurs positions... Si un aperçu avant impression est réalisé avec un navigateur, c'est la feuille de style *impression.css* qui va se charger automatiquement et qui va enlever le bloc d'en-tête, donc l'image du haut de la partie administration.

Cette technique est très largement utilisée sur Internet pour imprimer des fiches articles, des modes d'emploi, des conditions de vente ou autre.

La vue d'écran présente l'affichage du site avec la feuille de style *impression.css* lors d'un aperçu avant impression.

En résumé

Ce chapitre a présenté le framework Java EE Struts et sa mise en place dans le cadre du projet *BetaBoutique*. La première partie a présenté les intérêts de l'utilisation d'un framework dans le cadre du développement d'un projet d'entreprise.

Dans un deuxième temps, le guide a présenté en détail le framework Struts avec son modèle, et son installation.

La troisième partie a concerné le développement du système d'administration du chat *BetaBoutique* au travers d'exemples concrets.

La quatrième partie a détaillé l'utilisation des formulaires en Struts avec les JavaBean de formulaires, le fichier de configuration *struts-config.xml* et les actions associées.

Le paragraphe suivant a introduit la gestion des vues en Struts avec les taglibs spécifiques, l'accès aux données et les accès multilingues.

La sixième partie a concerné la validation des données qui reste un élément essentiel des développements Internet. Pour cela, Struts fournit un ensemble de solutions souples et simples à mettre en oeuvre aussi bien du côté serveur que du côté client.

La partie suivante a présenté en détail le contrôleur Struts, les classes Actions et les méthodes associées.

La septième partie a été consacrée au développement du module d'administration *BetaBoutique*. Tout le développement fut détaillé de la mise en place du pool de connexion à la source de données en passant par le codage des actions, vues et modèles, jusqu'aux optimisations JavaScript.

Le dernier paragraphe a été consacré aux techniques Web 2.0 et à l'ergonomie qui sont actuellement utilisées sur Internet. Nous avons retrouvé ainsi les différentes techniques qui permettent de rendre un site plus agréable et professionnel.

Gestion des traces et des logs

1. Présentation

La conception de projets nécessite la mise en place de traces lors des développements, débogages mais aussi en phase de production. La plupart des projets utilisent des API de journalisation réalisées en interne et plus ou moins performantes. Les programmeurs placent des traces dans le code en utilisant la sortie standard et la classe statique *System* au moyen du code suivant :

```
System.out.println("Une trace dans la console Java");
```

Les outils de journalisation offrent de nombreux avantages aux programmeurs. Le service le plus utilisé est évidemment la mise au point du code lors des développements. Ils permettent également d'enregistrer les messages, d'envoyer des emails, de gérer des niveaux de traces ou autre, mais surtout par l'intermédiaire d'un fichier de configuration, de gérer à tout moment les traces. En effet, un outil de journalisation permet d'activer ou désactiver à tout moment certains messages en fonction de nos besoins, sans être obligé de reprendre tout le code.

2. L'API LOG4J

La bibliothèque Log4J est très répandue dans le monde Java et notamment Java EE. Cette API de journalisation permet de gérer les traces utilisateurs en combinaison avec l'API *commons-logging*. Nous allons commencer la mise en place de la journalisation pour notre projet *BetaBoutique* en installant l'API LOG4J. Actuellement la version 1.3 est la plus utilisée en attendant la version 2.0 qui en est au stade expérimental (http://logging.apache.org/log4j/).

La première étape consiste à télécharger et à installer les archives au format *.jar* dans notre répertoire de librairies */WEB-INF/lib*. Nous utilisons l'application *chatbetaboutique* avec la partie d'administration développée à l'aide du framework Struts dans le chapitre précédent.

Les librairies suivantes sont alors copiées dans le répertoire */WEB-INF/lib* :

- *commons-logging.jar*
- *log4j-1.3alpha-8.jar*

3. Mise en place de L'API LOG4J

La librairie Log4J met à disposition du programmeur trois composants :

- Les **Loggers** qui permettent d'écrire les messages.
- Les **Appenders** qui permettent de sélectionner la destination des messages.
- Les **Layouts** qui permettent de mettre en forme les messages.

a. Logger

Le Logger est l'entité de base qui est utilisée pour la journalisation, il utilise la classe *org.apache.log4J.logger*. La déclaration d'un Logger est réalisée dans chaque classe qui doit utiliser le système de journalisation. Nous pouvons par exemple reprendre la classe *ServerModele* et ajouter la déclaration du Logger en début de fichier.

```
package modele;

import boiteoutils.*;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.sql.DataSource;
import org.apache.log4j.Logger;
```

```
public class ServeurModele
{
    //variables de classe
    DataSource ds=null;
    Connection connection=null;
    ResultSet rs=null;
    //log4j
    private final static Logger logger=Logger.getLogger("modele");
...
```

L'obtention de l'instance du Logger est réalisée en appelant la méthode statique *Logger.getLogger()*. Cette méthode prend en paramètre un nom de Logger de notre choix ou la référence directe à la classe.

```
private final static Logger logger2=Logger.getLogger(ServeurModele.class);
```

Ensuite, il est nécessaire de gérer le niveau de journalisation ou de priorité des messages. Ceci permet de représenter l'importance du message à journaliser. La classe *org.apache.log4j.Level* permet de gérer ces niveaux de messages. Un message n'est alors journalisé que si sa priorité est supérieure ou égale au Logger effectuant la journalisation.

L'API Log4J définit cinq niveaux de logging classés par ordre d'importance :

- FATAL : ce niveau est utilisé pour une erreur grave pouvant provoquer l'arrêt de l'application.
- ERROR : ce niveau est utilisé pour une erreur qui empêche un fonctionnement important de l'application (requête SQL, copie de fichier...).
- WARN : ce niveau est utilisé pour un avertissement ou une trace.
- INFO : ce niveau est utilisé pour un message informatif.
- DEBUG : ce niveau très verbeux est utilisé pour des messages utilisés en phase de débogage.

La journalisation d'un message à un niveau donné se fait au moyen de la méthode *log(priorité,message)*. Par exemple, nous pouvons déclencher une trace d'erreur des deux manières suivantes :

```
logger.error("Erreur dans la classe ServeurModele.java
fonction ServeurModele");
logger.log(Level.FATAL,"Erreur dans la classe ServeurModele.java
fonction ServeurModele");
```

b. Appenders

Un Appender représente la cible d'un message, c'est-à-dire l'endroit où sera stocké ou affiché ce message. Les Appenders sont utilisés pour enregistrer les événements de journalisation. Ils sont représentés par l'interface *org.apache.log4j.Appender* et chaque Appender enregistre d'une manière spécifique les événements.

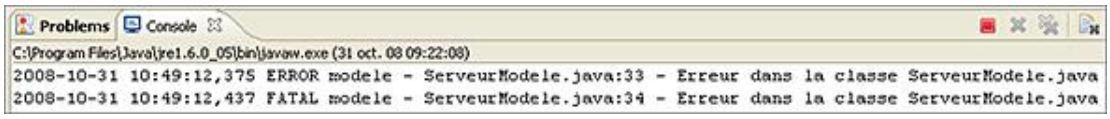
Il existe plusieurs types d'Appender afin de gérer les traces dans une base de données (*org.apache.log4j.jdbc.JDBCAppender*), par mail (*org.apache.log4j.net.SMTPAppender*), pour la console (*org.apache.log4j.ConsoleAppender*) ou encore dans un fichier (*org.apache.log4j.FileAppender*). Nous retrouvons donc des Appenders pour la console, le système de fichiers, les sockets, le démon Unix syslog ou les composants graphiques.

Nous pouvons reprendre notre classe précédente afin de tracer le message d'erreur dans la console.

```
/*********************************************************
 * constructeur
*********************************************************/
public ServeurModele(DataSource ds)
{
    PatternLayout layout=new PatternLayout("%d %-5p %c - %F:%L - %m%n");
    ConsoleAppender stdout=new ConsoleAppender(layout);
    logger.addAppender(stdout);

    logger.error("Erreur dans la classe ServeurModele.java
fonction ServeurModele");
```

```
    logger.log(Level.FATAL,"Erreur dans la classe
ServeurModele.java fonction ServeurModele");
    //récupérer le DataSource de la servlet
    this.ds=ds;
}
```

Le format défini avec le modèle (*PatternLayout*) permet de conserver l'heure et la date, le niveau d'erreur, le nom du fichier et le numéro de la ligne de code correspondante au message lui-même. Désormais si nous déclenchons un service qui utilise cette classe modèle, les traces sont affichées dans la console système.



La classe *PatternLayout* permet de gérer la mise en forme des messages en sortie. Cette classe permet d'informer les développeurs de données utiles mais demande parfois d'importantes ressources en fonction du détail des informations tracées.

c. Layouts

Les Layouts sont utilisés pour la mise en forme des événements de journalisation. Ils sont utilisés en accord avec les Appenders afin d'associer la cible de l'enregistrement avec la manière de tracer les données. Log4J fournit plusieurs Layouts comme :

- *org.apache.log4j.SimpleLayout* : qui permet de journaliser de façon simple les événements.
- *org.apache.log4j.PatternLayout* : qui permet de journaliser les messages en fonction d'un modèle ou motif.
- *org.apache.log4j.HTMLLayout* : qui permet de journaliser les événements au format HTML. Chaque journalisation produit un document HTML complet.
- *org.apache.log4j.XMLLayout* : qui permet de journaliser les événements au format XML en conjugaison avec un FileAppenders.
- *org.apache.log4j.net.SMTPAppender* : qui permet de journaliser les événements en les envoyant par email.

4. Configuration dynamique

La configuration précédente avec les Appenders dans les fichiers sources n'est pas très pratique et mélange le code source avec des éléments de configuration de la journalisation. De même, lors du développement, tous les messages seront affichés avec un niveau WARN par exemple et en production, seuls les messages de type ERROR devront être tracés.

Avec l'API Log4J, il existe trois méthodes pour configurer les Loggers. La première est la configuration par défaut que nous venons d'aborder avec une gestion dans les sources. La seconde configuration est réalisée à l'aide d'un fichier de propriétés avec le format habituel clé=valeur. Enfin le dernier type de configuration repose sur un fichier au format XML.

Nous allons utiliser la solution intéressante proposée à partir d'un fichier de propriétés. Dans un tel fichier, chaque Logger peut être configuré et les Appenders et les Layouts peuvent y être paramétrés. Nous pourrons ainsi aisément modifier le formatage des messages. Dans le fichier de configuration, chaque Appender doit avoir un nom afin de pouvoir y faire référence lors de la configuration des Loggers. Les Appenders sont préfixés par *log4j.appender*.

La déclaration d'un Appender est réalisée sous la forme suivante :

```
log4j.appender.NomAppender=ClasseAppender
```

Le paramètre *NomAppender* est le nom que nous souhaitons utiliser pour notre Appender et le paramètre *ClasseAppender* est la classe d'implémentation de l'Appender. Voici un exemple de configuration d'un Appender simple pour la console.

```
#CONSOLE est l'Appender de type ConsoleAppender
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.SimpleLayout
```

L'Appender est correctement configuré, nous devons maintenant paramétrer les Loggers pour qu'ils utilisent les Appenders. La configuration des Loggers est similaire à celle des Appenders, la forme est la suivante : *log4j.logger.nomdulogger=niveau, appender1, appender2...*

Le paramètre *nomdulogger* est le nom du Logger indiqué dans l'instruction *Logger.getLogger(nomdulogger)* et représente une structure pointée, identique à la notion de paquetage. Le paramètre *niveau* est le nom du niveau attribué au Logger (ERROR, DEBUG...). S'il n'est pas précisé, le niveau est hérité du nœud parent ou positionné à DEBUG. Le paramètre *appenderX* est le nom d'un Appender déclaré dans le fichier.

Pour mettre en place ce mécanisme nous allons créer un fichier nommé *log4j.properties* dans le paquetage ressources.

```
#définition du niveau et des Appender du rootLogger
(ordre : DEBUG - INFO - WARN - ERROR - FATAL)
log4j.rootLogger=ERROR, CONSOLE
#CONSOLE est l'Appender de type console
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
#definition du format des messages 2005-06-18 14:53:37 DEBUG
[Main] Hello World
log4j.appender.CONSOLE.layout.ConversionPattern=%d %-5p %c - %F:%L - %m%n
```

Nous allons donc pouvoir intercepter les traces de types ERROR et donc FATAL, car celles-ci sont plus basses dans la hiérarchie. Ce type de trace est associé à l'Appender nommé CONSOLE qui utilise la classe *org.apache.log4j.ConsoleAppender* et qui va donc afficher les traces dans la console Java avec le format défini par le Layout ou modèle.

```
public class ServeurModele
{
    //variables de classe
    DataSource ds=null;
    Connection connection=null;
    ResultSet rs=null;
    //log4j
    private final static Logger logger=Logger.getLogger("modele");

/*********************************************************
 * constructeur
*********************************************************/
public ServeurModele(DataSource ds)
{
    logger.error("Erreur dans la classe ServeurModele.java
fonction ServeurModele");
    logger.log(Level.FATAL,"Erreur dans la classe
ServeurModele.java fonction ServeurModele");

    //récupérer le DataSource de la servlet
    this.ds=ds;
}
```

Maintenant, pour que ce service soit opérationnel et que les Loggers déclarés dans les fichiers sources utilisent ce fichier de propriétés, il est nécessaire de le mettre en place dans une classe chargée au démarrage. Pour cela, nous ajoutons un attribut dans le fichier de configuration de l'application *web.xml*.

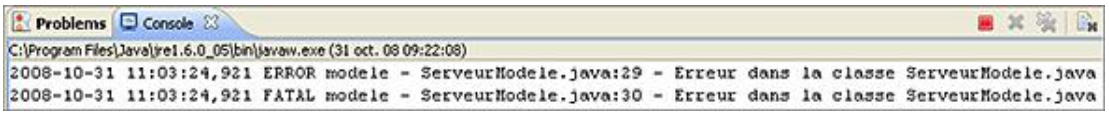
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
  <display-name>Application chatbetaboutique</display-name>
  <!-- chemin pour acceder au fichier de proprietes log4J -->
  <context-param>
       <param-name>log4jfichier</param-name>
       <param-value>WEB-INF/classes/ressources/log4j.properties</param-value>
  </context-param>
...
```

Enfin, nous terminons le paramétrage en précisant le fichier de configuration dans la classe de gestion des plug-ins ou toute autre classe lancée au démarrage de l'application.

```
package chatbetaboutique;

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

```
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.sql.DataSource;
import org.apache.log4j.PropertyConfigurator;
import org.apache.struts.action.ActionServlet;
import org.apache.struts.action.PlugIn;
import org.apache.struts.config.ModuleConfig;

public class PluginDataSource implements PlugIn
{
   //fonction appelée lors de la création du plug-in
   public void init(ActionServlet servlet,ModuleConfig
moduleConfig) throws ServletException
    {
   //récupérer les paramètres présents dans le fichier
de configuration web.xml
   String nomprojet=(String)servlet.getServletContext().getInitParameter
("urlapplication");
   String connecteurjdbc=(String)servlet.getServletContext().getInitParameter
("connecteurjdbc");
   //gestion de la journalisation
   String prefix=servlet.getServletContext().getRealPath("/");
   String log4jfichier=(String)servlet.getServletContext().getInitParameter
("log4jfichier");
   PropertyConfigurator.configure(prefix+log4jfichier);
   //initaliser le context
   Context initCtx=null;
...
```

Désormais, si nous déclenchons une page qui utilise la classe *ServeurModele*, les traces de niveau ERROR et FATAL sont affichées dans la console Java.

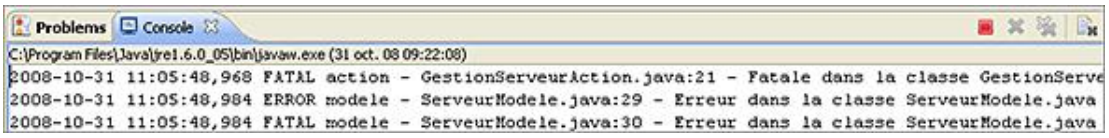


Toutes les traces de types ERROR et FATAL sont affichées dans la console étant donné que notre Logger est positionné au niveau de la racine (*log4j.rootLogger*).

Nous pouvons vérifier ceci en déclarant une trace dans l'action de la classe *GestionServeurAction*.

```
package chatbetaboutique;

import modele.ServeurModele;
import org.apache.log4j.Logger;
import org.apache.struts.action.*;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import javax.sql.DataSource;

public class GestionServeurAction extends Action{

    //variables de la classe
    DataSource ds=null;
    //log4j
    private final static Logger logger=Logger.getLogger("action");

    //gérer l'état du serveur
    public ActionForward execute(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response)throws
IOException, ServletException
    {
    logger.fatal("Fatale dans la classe
GestionServeurAction.java fonction execute");
    //récupérer la datasource du plugin dans un attribut
présent dans le context de la servlet
    ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
    //créer le modèle
```

```
    ServeurModele serveurmodele=new ServeurModele(ds);
    //fermer la datasource
    this.ds=null;
...
```



Maintenant, nous pourrions souhaiter journaliser les traces du paquetage Log4J nommé *modele* différemment du paquetage *root*. Nous allons par exemple changer de niveau de journalisation pour le paquetage *modele* afin d'avoir des traces simples et stocker les données dans un fichier de journalisation.

```
#définition du niveau et des Appender du rootLogger
(ordre : DEBUG - INFO - WARN - ERROR - FATAL)
log4j.rootLogger=ERROR, CONSOLE
#CONSOLE est l'Appender de type console
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
#definition du format des messages 2005-06-18 14:53:37 DEBUG
[Main] Hello World
log4j.appender.CONSOLE.layout.ConversionPattern=%d %-5p %c - %F:%L - %m%n

#logger pour le paquet modele, dans le fichier tracer que les WARN,
ERROR et FATAL
log4j.logger.modele=WARN, fichiermodele
#fichier
log4j.appender.fichiermodele=org.apache.log4j.RollingFileAppender
log4j.appender.fichiermodele.File=E:\\PROJETWEB\\chatbetaboutique\\modele.log
log4j.appender.fichiermodele.MaxFileSize=200KB
log4j.appender.fichiermodele.MaxBackupIndex=2
log4j.appender.fichiermodele.layout=org.apache.log4j.PatternLayout
log4j.appender.fichiermodele.layout.ConversionPattern=%d %-5p %c -
%F:%L - %m%n
```

Nous précisons sur quel paquetage Log4J nous souhaitons réaliser la journalisation par l'intermédiaire de l'instruction *log4j.logger.modele*. Ensuite, nous détaillons la configuration de l'Appender avec le type *org.apache.log4j.RollingFileAppender* (journalisation par fichier), le fichier de stockage des traces avec l'attribut *File*, la taille maximale du fichier avant rotation (la relation est utilisée pour conserver plusieurs fichiers de journalisation) avec le paramètre *MaxFileSize*, le nombre de fichiers conservés avec le paramètre *MaxBackupIndex*, le type de Layout utilisé (*org.apache.log4j.PatternLayout*) et enfin le modèle du Layout avec l'attribut *ConversionPattern*.

```
package modele;

import boiteoutils.*;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.sql.DataSource;
import org.apache.log4j.Logger;

public class ServeurModele
{
    //variables de classe
    DataSource ds=null;
    Connection connection=null;
    ResultSet rs=null;
    //log4j
    private final static Logger logger=Logger.getLogger("modele");
/*********************************************************
 * constructeur
*********************************************************/
public ServeurModele(DataSource ds)
{
    logger.debug("Debug dans la classe ServeurModele.java
fonction ServeurModele");
    logger.info("Info dans la classe ServeurModele.java
```

```
fonction ServeurModele");
    logger.warn("Warn dans la classe ServeurModele.java
fonction ServeurModele");
    logger.error("Error dans la classe ServeurModele.java
fonction ServeurModele");
    logger.fatal("Fatal dans la classe ServeurModele.java
fonction ServeurModele");

    //récupérer le DataSource de la servlet
    this.ds=ds;
}
```

Nous voyons, après avoir lancé un service qui utilise cette classe, qu'un fichier nommé *modele.log* est créé à la racine du projet et possède le contenu suivant :

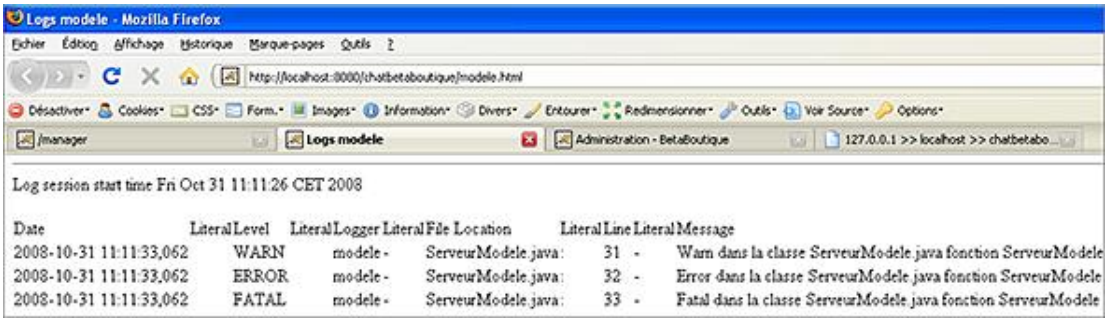
```
2008-10-31 11:09:37,468 WARN  modele - ServeurModele.java:31 -
Warn dans la classe ServeurModele.java fonction ServeurModele
2008-10-31 11:09:37,468 ERROR modele - ServeurModele.java:32 -
Error dans la classe ServeurModele.java fonction ServeurModele
2008-10-31 11:09:37,484 FATAL modele - ServeurModele.java:33 -
Fatal dans la classe ServeurModele.java fonction ServeurModele
```

Les traces de type WARN, ERROR et FATAL sont bien insérées dans le fichier *modele.log* sans les traces de types DEBUG et INFO et sans les traces de type *root* malgré la déclaration dans l'action *GestionServeurAction*.

Maintenant, nous pouvons encore améliorer la journalisation en utilisant une trace de type fichier HTML.

```
#définition du niveau et des Appender du rootLogger
(ordre : DEBUG - INFO - WARN - ERROR - FATAL)
log4j.rootLogger=ERROR, CONSOLE
#CONSOLE est l'Appender de type console
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
#definition du format des messages 2005-06-18 14:53:37 DEBUG
[Main] Hello World
log4j.appender.CONSOLE.layout.ConversionPattern=%d %-5p %c - %F:%L - %m%n

#logger pour le paquet MODELE, dans le fichier tracer que les WARN,
ERROR et FATAL
log4j.logger.modele=WARN, fichiermodele, htmlmodele
#fichier
log4j.appender.fichiermodele=org.apache.log4j.RollingFileAppender
log4j.appender.fichiermodele.File=E:\\PROJETWEB\\chatbetaboutique\\modele.log
log4j.appender.fichiermodele.MaxFileSize=200KB
log4j.appender.fichiermodele.MaxBackupIndex=2
log4j.appender.fichiermodele.layout=org.apache.log4j.PatternLayout
log4j.appender.fichiermodele.layout.ConversionPattern=%d %-5p %c - %F:%L
- %m%n
#html
log4j.appender.htmlmodele=org.apache.log4j.RollingFileAppender
log4j.appender.htmlmodele.File=E:\\PROJETWEB\\chatbetaboutique\\modele.html
log4j.appender.htmlmodele.MaxFileSize=300KB
log4j.appender.htmlmodele.MaxBackupIndex=2
log4j.appender.htmlmodele.layout=org.apache.log4j.HTMLLayout
log4j.appender.htmlmodele.layout.LocationInfo=true
log4j.appender.htmlmodele.layout.Title=Logs modele
log4j.appender.htmlmodele.layout.ConversionPattern=%d %-5p %c - %F:%-4L
- %m%n
```

La configuration est quasiment identique à celle d'un *RollingFileAppender*, c'est essentiellement la déclaration du Layout qui change. Cette journalisation aura pour effet de générer des traces de journalisation au format HTML directement consultables (*modele.html*).



Afin d'améliorer l'affichage de la page HTML, nous pouvons très bien développer une feuille de style CSS avec des couleurs, des images ou cadres pour les niveaux de messages et associer cette feuille aux pages HTML générées par Log4J.

Enfin, nous terminerons la mise en place de Log4J avec l'installation d'un système de journalisation par email. Pour cela, le paquetage Log4J SMTP est nécessaire, nous installons donc l'archive l*og4j-smtp-1.3alpha-8.jar* dans le répertoire */WEB-INF/lib* de l'application. La librairie Log4J utilise JavaMail pour le transfert des emails, il donc nécessaire d'installer également le paquetage *mail.jar*.

```
#définition du niveau et des Appender du rootLogger
(ordre : DEBUG - INFO - WARN - ERROR - FATAL)
log4j.rootLogger=ERROR, CONSOLE
#CONSOLE est l'Appender de type console
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
#definition du format des messages 2005-06-18 14:53:37 DEBUG
[Main] Hello World
log4j.appender.CONSOLE.layout.ConversionPattern=%d %-5p %c - %F:%L - %m%n

#logger pour le paquet MODELE, dans le fichier tracer que les WARN,
ERROR et FATAL
log4j.logger.modele=WARN, fichiermodele, htmlmodele, emailmodele
#fichier
log4j.appender.fichiermodele=org.apache.log4j.RollingFileAppender
log4j.appender.fichiermodele.File=E:\\PROJETWEB\\chatbetaboutique\\modele.log
log4j.appender.fichiermodele.MaxFileSize=200KB
log4j.appender.fichiermodele.MaxBackupIndex=2
log4j.appender.fichiermodele.layout=org.apache.log4j.PatternLayout
log4j.appender.fichiermodele.layout.ConversionPattern=%d %-5p %c - %F:%L
- %m%n
#html
log4j.appender.htmlmodele=org.apache.log4j.RollingFileAppender
log4j.appender.htmlmodele.File=E:\\PROJETWEB\\chatbetaboutique\\modele.html
log4j.appender.htmlmodele.MaxFileSize=300KB
log4j.appender.htmlmodele.MaxBackupIndex=2
log4j.appender.htmlmodele.layout=org.apache.log4j.HTMLLayout
log4j.appender.htmlmodele.layout.LocationInfo=true
log4j.appender.htmlmodele.layout.Title=Logs modele
log4j.appender.htmlmodele.layout.ConversionPattern=%d %-5p %c - %F:%-4L
- %m%n
#email
log4j.appender.emailmodele=org.apache.log4j.net.SMTPAppender
log4j.appender.emailmodele.Threshold=INFO
log4j.appender.emailmodele.BufferSize=100
log4j.appender.emailmodele.To=jerome@nouvellepage.com
log4j.appender.emailmodele.From=info@documentheque.com
log4j.appender.emailmodele.SMTPHost=localhost
log4j.appender.emailmodele.Subject=Log4J Message en ligne classe : modele
log4j.appender.emailmodele.layout=org.apache.log4j.PatternLayout
log4j.appender.emailmodele.layout.ConversionPattern=%5p [%t] (%F:%L)
- %m%n
```

L'outil Log4J permet de gérer la journalisation de façon précise et adaptée. Pour notre projet *BetaBoutique*, nous pouvons utiliser par exemple le fichier précédent ainsi que trois autres paquetages Log4J pour les actions, les vues et les JavaBean. Nous plaçons chaque définition avec le niveau WARN, nous réalisons des traces de débogage, de tests ou autre avec le niveau WARN et les erreurs graves avec le niveau ERROR.

Dès la mise en production, il faudra utiliser soit un second fichier de propriétés avec le niveau ERROR à la place de WARN, soit modifier le niveau dans le fichier *log4j.properties* afin de tracer uniquement les erreurs graves. Le service de génération des erreurs par email ou avec un fichier HTML permet alors de consulter les traces en ligne et en temps réel.

Par habitude avec des développements en MVC et Java EE, il faut utiliser quatre définitions de paquetages Log4J qui sont relatives aux modèles, aux actions, aux JavaBean et aux vues afin de tracer très précisément ce qui est souhaité pendant les phases de développement et production.

Ant : Another Neat Tool

1. Présentation

Ant est un projet OpenSource du consortium Apache-Jakarta. C'est un outil qui permet de compiler des sources Java, de les assembler dans les fichiers *.jar* et de les exécuter. Il est aussi possible de s'en servir pour automatiser certaines tâches.

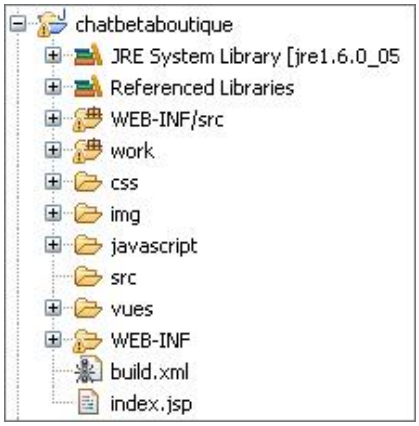
Ant est souvent comparé au célèbre outil *make* Unix. Écrit en Java, il est donc multiplates-formes et repose sur un fichier de configuration écrit en XML qui définit les différentes tâches qui devront être exécutées par l'outil (fichier *build.xml*). Le fichier de configuration contient un ensemble de cibles ou target. Chaque cible contient une ou plusieurs tâches et chaque cible peut avoir une dépendance envers une ou plusieurs autres cibles lors de l'exécution.

Ant est livré en standard avec la plupart des outils de développement (Eclipse ou NetBeans). Il est maintenant utilisé dans la plupart des projets pour construire les exécutables à partir de sources.

Ant repose sur Java, il est donc aisé de développer un plug-in pour une fonctionnalité précise. Il permet en effet de couvrir à peu près tous les besoins nécessaires au développement d'applications conséquentes : compilation, gestion de version, empaquetage, déploiement et archivage.

2. Utilisation

Le fichier de configuration pour les projets Ant est nommé *build.xml*. Nous allons créer un premier fichier d'exemple qui permet d'afficher un message simple dans la console Java. Sous Eclipse, la commande **Fenêtre - Afficher la vue - ANT** permet d'afficher l'éditeur XML pour Ant. Ensuite, dans la fenêtre affichée (icône fourmi), il est nécessaire de créer un nouveau fichier *build.xml* à la racine du projet.



Le fichier *build.xml* contient la description du processus de construction de l'application. Comme tout document XML, le fichier commence par le prologue.

```
<?xml version="1.0" ?>
```

Le principal élément de l'arborescence du document XML est représenté par le tag *<project/>* qui est la racine. À l'intérieur de ce tag, nous retrouvons la définition des éléments du projet :

- Les cibles ou targets qui sont les étapes de construction du projet.
- Les propriétés ou properties qui sont les variables qui contiennent les données utilisables par les éléments.
- Les tâches ou tasks qui sont les traitements à réaliser dans une cible donnée.

Le projet - project

Cette balise permet de définir la racine du projet dans le fichier de configuration *build.xml*. Ce tag possède plusieurs attributs :

- *name* : qui permet de préciser le nom du projet.

- *default* : qui permet de préciser la cible à exécuter par défaut.
- *basedir* : qui permet de préciser le répertoire qui servira de référence pour la localisation d'autres références.

```
<project name="AFFICHAGE" default="run" basedir=".">
```

Les cibles - target

La balise <*target/*> permet de définir une cible. Une cible est un ensemble de tâches à réaliser dans un ordre bien précis. L'ordre correspond aux tâches définies dans la cible elle-même.

Ce tag possède plusieurs attributs :

- *name* : qui permet de préciser un nom à la cible.
- *description* : qui contient une brève description de la cible.

Les tâches - task

Une tâche est un traitement qui est réalisé par l'intermédiaire d'une classe Java qui implémente l'interface *org.apache.ant.Task*. Une tâche est obligatoirement incluse dans une cible pour pouvoir être exécutée.

Ant fournit une liste très complète de tâches pour les traitements lors des développements :

- *echo* : afficher un message dans la console.
- *taskdef* : définir une tâche externe.
- *available* : définir une propriété.
- *java* : exécuter une application.
- *javac* : compiler les sources.
- *javadoc* : générer la documentation.
- *signjar* : signer un fichier jar.
- *gunzip* : décompresser une archive.
- *jar* : créer une archive au format .jar.
- *tar* : créer une une archive au format .tar.
- *zip* : créer une archive au format .zip.
- *war* : créer une archive au format .war.
- *exec* : exécuter une commande externe.
- *mail* : envoyer un email.
- *chmod* : modifier les droits sur un fichier.
- *copy* : copier un fichier.
- *delete* : supprimer un fichier.

- *mkdir* : créer un répertoire.
- *ant* : exécuter un autre fichier de build.
- *record* : enregistrer les traitements de l'exécution dans un fichier journal.

Les propriétés - property

La balise <*property/*> permet de définir une propriété qui pourra être utilisée dans le projet. Les propriétés sont souvent utilisées pour préciser une variable, un répertoire de sources, une version...

La définition par l'intermédiaire de propriété ou variable permet facilement de changer les valeurs des paramètres sans reprendre la totalité du fichier de construction *build.xml*.

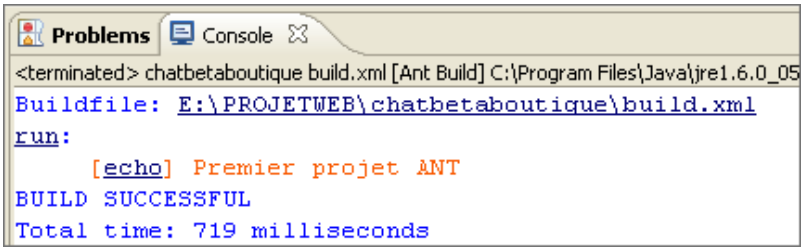
Ce tag possède plusieurs attributs :

- *name* : qui permet de donner un nom à la propriété.
- *value* : qui permet de donner une valeur à la propriété.
- *location* : qui permet de définir un fichier avec un chemin absolu.
- *file* : qui permet de préciser le nom d'un fichier qui contient la définition d'un ensemble de propriétés.

Nous pouvons éditer le fichier *build.xml* et insérer le code ci-dessous afin d'afficher un message dans la console.

```
<?xml version="1.0" ?>
<project name="affichage" default="run" basedir=".">
    <target name="run">
        <echo message="Premier projet ANT"/>
    </target>
</project>
```

Ce projet nommé *affichage* utilise une seule cible nommée *run* qui est lancée par défaut. Pour exécuter ce code, il est nécessaire d'ouvrir l'onglet Ant, d'ajouter le fichier *build.xml* et de lancer son exécution avec l'icône *run*.



Ant peut également s'utiliser en ligne de commande avec la syntaxe suivante :

```
ant options cible
```

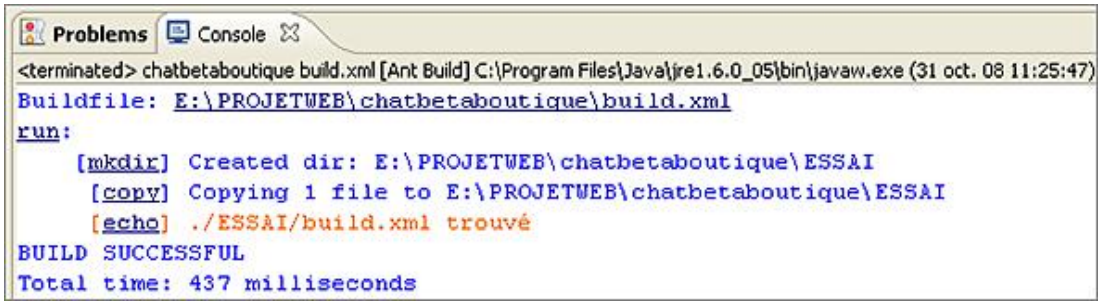
Par défaut, Ant recherche un fichier nommé *build.xml* dans le répertoire courant. Ce fichier peut être précisé avec l'option :

```
ant -buildfile monbuild.xml
```

Le fichier *build.xml* suivant permet de créer deux tâches appelées *run* et *clean*. La tâche *run*, exécutée par défaut, permet de créer un répertoire nommé *ESSAI*, d'y copier le fichier *build.xml* et de vérifier que celui-ci existe. La tâche *clean* permet de supprimer le répertoire créé ainsi que les fichiers de ce répertoire. Une tâche qui n'est pas celle par défaut est lancée en donnant son nom en paramètre.

```
<?xml version="1.0" ?>
<project name="fichier" default="run" basedir=".">
    <target name="run">
        <mkdir dir="ESSAI"/>
        <copy file="build.xml" todir="./ESSAI/"/>
        <available file="./ESSAI/build.xml" type="file"
property="build.xml.found"/>
        <echo message="./ESSAI/build.xml trouvé"/>
    </target>
    <target name="clean">
        <delete includeEmptyDirs="true" quiet="true">
        <fileset dir="ESSAI"/>
        </delete>
    </target>
</project>
```

En lançant Ant avec Eclipse, c'est la tâche par défaut qui est exécutée, soit *run*. Cette tâche va bien créer le répertoire *ESSAI* dans le répertoire courant ainsi que le fichier associé.



Maintenant, nous pouvons lancer la tâche *clean* qui permet de supprimer le répertoire. Pour cela, avec Eclipse, il faut déplier le fichier Ant et sélectionner la seconde tâche.

Nous pouvons également définir une propriété pour le nom du fichier *build.xml*. Cette technique permet ainsi de changer très facilement une valeur sans être obligé de tout vérifier dans le code source.

```
<?xml version="1.0" ?>
<project name="fichier" default="run" basedir=".">
 <property name="repertoiredestination" value="./ESSAICOPIE/"/>
 <target name="run">
    <mkdir dir="${repertoiredestination}"/>
    <copy file="build.xml" todir="${repertoiredestination}"/>
    <available file="${repertoiredestination}build.xml"
type="file" property="build.xml.found"/>
    <echo message="${repertoiredestination}build.xml trouvé"/>
 </target>
 <target name="clean">
    <delete includeEmptyDirs="true" quiet="true">
    <fileset dir="${repertoiredestination}"/>
    </delete>
 </target>
</project>
```

3. Génération de l'archive d'un projet Java EE

Nous allons maintenant utiliser l'outil Ant pour déployer notre projet *chatbetaboutique* de façon professionnelle à l'aide d'une archive qui sera directement exploitable sur n'importe quel serveur compatible Java EE. Le projet sera transformé en une archive *.war* (Web ARchive) qui pourra directement être déployée à l'aide d'un serveur. Pour cela,

nous procéderons par étape, en augmentant progressivement la capacité des fonctionnalités de notre fichier *build.xml*. La première tâche à réaliser lors des déploiements est la compilation des fichiers sources contenus dans les différents répertoires.

La tâche *<javac/>* permet de compiler les fichiers sources. Cette tâche possède les principaux attributs suivants :

- *srcdir* : qui permet de préciser le répertoire racine de l'arborescence des fichiers sources.
- *destdir* : qui permet de préciser le répertoire de destination des résultats compilés.
- *executable* : qui permet de préciser le chemin vers le compilateur Java utilisé.
- *fork* : qui permet de lancer la compilation dans une JVM dédiée ou non, par rapport à l'exécution de ANT.

Nous allons créer des propriétés afin d'améliorer la lisibilité et la maintenance du fichier *build.xml*.

```
<?xml version="1.0" ?>
<project name="FICHIER" default="run" basedir=".">

      <!-- librairies utilisées -->
      <property name="lib" value="./WEB-INF/lib"/>
      <property name="libtomcat" value="E:\Tomcat 6.0\lib"/>
      <!-- fichier sources -->
      <property name="src" location="./WEB-INF/src"/>
      <!-- fichier compilés -->
      <property name="classes" location="./WEB-INF/classes"/>
      <!-- Définition du classpath du projet -->
      <path id="classpath">
        <fileset dir="${lib}">
          <include name="*.jar"/>
        </fileset>
            <fileset dir="${libtomcat}">
              <include name="*.jar"/>
            </fileset>
      </path>

      <!-- tache1 : afficher la date de la génération du build ANT
et creer un fichier informatif-->
      <target name="init" description="Initialisation">
        <tstamp/>
        <buildnumber file="numerobuild.txt"/>
        <echo message="Generation numero : ${build.number} du ${TODAY}"/>
      </target>

      <!-- tache2 : générer les fichiers .class -->
      <target name="compil" depends="init" description="Compilation">
        <echo message="Compilation des classes en Java"/>
        <javac srcdir="${src}" destdir="${classes}"
executable="E:\jdk1.5\bin\javac" fork="yes">
            <classpath refid="classpath"/>
        </javac>
          <echo message="Copie des ressources dans
le repertoire des classes"/>
          <copy todir="${classes}/ressources">
              <fileset dir="${src}/ressources"/>
          </copy>
     </target>

      <!-- tache 0 : lancée par défaut -->
      <target name="run" depends="init, compil"
description="Generation complete">
        <echo message="Generation complete"/>
      </target>
</project>
```

Nous définissons les librairies nécessaires à la compilation du projet. Ensuite, nous précisons le répertoire des sources Java ainsi que le répertoire de destination. Enfin, nous précisons le classpath du projet en indiquant que toutes nos librairies .jar seront utilisées lors de la compilation.

La tâche *run* est lancée en premier et va déclencher dans l'ordre la tâche *init* suivie de la tâche *compil*. La tâche *compil* permet d'afficher dans la console et dans un fichier, le numéro de version de la compilation. Ensuite, tous les fichiers sources *.java* sont compilés en fichiers *.class*.

Nous allons maintenant créer un répertoire nommé */war* ainsi qu'une nouvelle tâche nommée *clean* qui permet de supprimer l'ancienne archive WAR du projet si elle existe.

```
<!-- tache3 pour nettoyer l'ancienne archive war-->
<target name="clean" description="Suppression de l'ancienne archive war">
    <echo message="Suppression de l'ancienne archive war"/>
    <delete file="./war/chatbetaboutiquefin.war"/>
</target>
```

Ensuite, nous passons à la tâche qui permet de générer une archive WAR Java EE pour le déploiement du projet. Cette tâche va générer une archive nommée *chatbetaboutiquefin.war* à partir du fichier de configuration */WEB-INF/web.xml*, sans inclure le fichier Ant *build.xml*, les fichiers sources (*/WEB-INF/src*) et les JSP compilées (répertoire *work*). En effet, le projet en phase de production n'a pas besoin des fichiers sources et les JSP seront recompilées avec le moteur JSP du serveur en production.

L'archive finale est nommée *chatbetaboutiquefin.war* afin de tester le déploiement sur le serveur et de faire la distinction avec le projet lui-même qui est nommé *chatbetaboutique*.

Il est également possible d'utiliser une tâche de communication Ant pour envoyer l'application WAR en FTP sur le serveur en production.

```
<ftp server="ftp.gdawj.com" userid="utilisateur" password="motdepasse">
 <echo message="Envoyer l'application en FTP"/>
 <fileset dir="./war/chatbetaboutiquefin.war" />
</ftp>
```

L'utilisation de la tâche FTP nécessite la mise en place de plusieurs librairies Java dépendantes. http://ant.apache.org/manual/OptionalTasks/ftp.html

Le fichier final est présenté ci-dessous :

```
<?xml version="1.0" ?>
<project name="FICHIER" default="run" basedir=".">
      <!-- librairies utilisées -->
      <property name="lib" value="./WEB-INF/lib"/>
      <property name="libtomcat" value="E:\Tomcat 6.0\lib"/>
      <!-- fichier sources -->
      <property name="src" location="./WEB-INF/src"/>
      <!-- fichier compilés -->
      <property name="classes" location="./WEB-INF/classes"/>
      <!-- Définition du classpath du projet -->
      <path id="classpath">
        <fileset dir="${lib}">
          <include name="*.jar"/>
        </fileset>
          <fileset dir="${libtomcat}">
            <include name="*.jar"/>
          </fileset>
      </path>
<!-- tache1 : afficher la date de la génération du build ANT
et creer un fichier informatif-->
      <target name="init" description="Initialisation">
        <tstamp/>
        <buildnumber file="numerobuild.txt"/>
        <echo message="Generation numero : ${build.number} du ${TODAY}"/>
      </target>
      <!-- tache2 : générer les fichiers .class -->
      <target name="compil" depends="init" description="Compilation">
    <echo message="Compilation des classes en Java"/>
        <javac srcdir="${src}" destdir="${classes}"
executable="E:\jdk1.5\bin\javac" fork="yes">
            <classpath refid="classpath"/>
        </javac>
          <echo message="Copie des ressources dans
le repertoire des classes"/>
          <copy todir="${classes}/ressources">
```

```
              <fileset dir="${src}/ressources"/>
          </copy>
     </target>
      <!-- tache3 pour nettoyer l'ancienne archive war-->
      <target name="clean" description="Suppression
de l'ancienne archive war">
        <echo message="Suppression de l'ancienne archive war"/>
        <delete file="./war/chatbetaboutiquefin.war"/>
      </target>
      <!-- tache4 : génération de l'archive WAR -->
      <target name="war" depends="compil,clean" description="Generation
de l'archive war">
    <echo message="Generation de l'archive war"/>
        <war destfile="./war/chatbetaboutiquefin.war" basedir="./."
webxml="./WEB-INF/web.xml" excludes="build.xml,WEB-INF/src/**,work/org/**">
            <manifest>
              <attribute name="Built-By" value="Jérôme Lafosse"/>
            </manifest>
        </war>
      </target>
      <!-- tache 0 : lancée par défaut -->
      <target name="run" depends="init, compil, clean,
war" description="Generation complete">
        <echo message="Generation complete"/>
      </target>
```

Déployer un projet Java EE

1. Présentation

Maintenant que le projet est correctement archivé et généré conformément au standard WAR, nous pouvons le déployer sur le serveur Java EE en production. Pour cela, nous devons définir un contexte. Avec Tomcat 6, cela peut être réalisé explicitement de plusieurs manières :

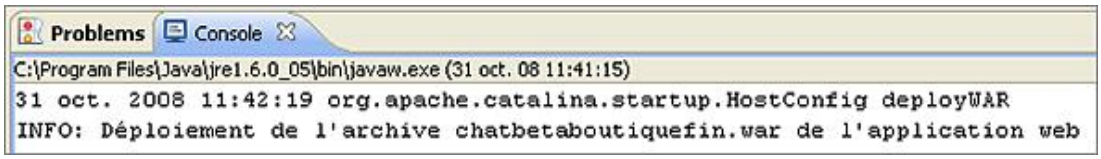
- Soit dans le fichier : *Tomcat 6/conf/server.xml*.
- Soit dans un fichier XML spécifique de configuration du contexte comme c'est le cas avec nos projets Eclipse définis par le plug-in Sysdeo : *Tomcat 6/conf/Catalina/localhost/nomduprojet.xml*.

  La syntaxe est la suivante : *Tomcat 6/conf/nom_engin/nom_hôte/nom_application*.
- Soit dans le fichier de l'application : *META-INF/context.xml*.

Pour déployer notre projet final *chatbetaboutiquefin.war* sur un serveur en production il existe plusieurs techniques :

- En réalisant un copier-coller de notre application dans le répertoire */webapps* de Tomcat.
- En déployant le projet avec l'outil manager de Tomcat.
- En utilisant les commandes HTTP de Tomcat.
- En utilisant l'outil Tomcat Client Deployer.

Nous pouvons tester le déploiement de notre projet en démarrant Tomcat et en copiant notre archive dans le répertoire */webapps* de Tomcat. Après quelques secondes d'attente, nous pouvons observer dans le fichier de journalisation de Tomcat */logs/catalin.out*, le déploiement automatique de notre application.



Nous voyons d'ailleurs la création de sa référence dans le manager de Tomcat.

Tomcat a décompressé l'archive *chatbetaboutique.war* afin de générer un répertoire */webapps/chatbetaboutique* avec la totalité des fichiers. De même, les pages JSP et les fichiers temporaires sont présents dans le répertoire *Tomcat 6/work/Catalina/localhost/chatbetaboutiquefin*.

L'archive est correctement déployée mais le projet n'est pas fonctionnel. En effet le déclenchement de l'URL suivante entraîne une erreur SQL : http://localhost:8080/chatbetaboutiquefin/etatserveur.do. Ceci est tout à fait normal, nous avons déployé un projet mais celui-ci ne possède pas de déclaration dans le serveur Tomcat. Les informations de configuration de ce projet ne sont pas présentes ni dans le fichier */conf/server.xml*, ni dans un fichier spécifique */conf/Catalina/localhost/nomduprojet.xml*, ni dans le fichier *META-INF/context.xml* de l'application.

Lors du développement nous avons utilisé le fichier */conf/Catalina/localhost/chatbetaboutique.xml* pour déclarer le projet et le pool de connexion JDBC à la base de données. Avec notre archive ces données ne sont pas renseignées, le projet ne sait donc pas comment se connecter à la base de données. Pour réaliser ces étapes, nous allons utiliser un fichier *META-INF/context.xml* dans notre application avec la déclaration suivante :

```
<Context path="/chatbetaboutiquefin" reloadable="true" docBase=".">
 <Resource name="jdbc_chatbetaboutiquemysql" auth="Container"
type="javax.sql.DataSource"
 username="root" password="" driverClassName="com.mysql.jdbc.Driver"
 url="jdbc:mysql://localhost:3306/chatbetaboutique"
maxActive="20" maxIdle="10"
 validationQuery="SELECT 1" defaultAutoCommit="true"
```

```
characterEncoding="UTF-8"/>
</Context>
```

Nous supprimons l'ancienne archive sur le serveur Tomcat et nous utilisons Ant pour réaliser une nouvelle archive de notre projet que nous déposons à nouveau sur le serveur en production. Cette opération nécessite un arrêt et un démarrage de Tomcat (ou redémarrage).

Le déploiement de l'archive par le serveur Tomcat aura pour effet de créer un fichier */conf/Catalina/localhost/chatbetaboutiquefin.xml* qui sera une copie du contenu du fichier *META-INF/context.xml* de l'application. De même la suppression de l'application aura pour conséquence de détruire le fichier de configuration */conf/Catalina/localhost/chatbetaboutiquefin.xml*.

2. Mise en production d'un projet Java EE

Nous avons regardé comment déployer correctement un projet Java EE à partir d'une archive WAR. Cependant, lors de la mise en production, nous souhaitons déployer notre projet à partir d'un nom de domaine.

Pour ce paragraphe, nous allons utiliser les deux noms de domaines suivants : www.gdawj.com et www.gdawj.net (*g*uide de *d*éveloppement d'*a*pplications *w*eb en *j*ava).

Pour accéder à ces URL en local, nous pouvons modifier le fichier hosts de la machine. Par exemple sous Windows : *C:\WINDOWS\system32\drivers\etc\hosts* ou */etc/host.conf* sous Linux.

```
# Copyright (c) 1993-1999 Microsoft Corp.
#
# Ceci est un exemple de fichier HOSTS utilisé par Microsoft TCP/IP
# pour Windows.
#
# Ce fichier contient les correspondances des adresses IP aux noms d'hôtes.
# Chaque entrée doit être sur une ligne propre. L'adresse IP doit être placée
# dans la première colonne, suivie par le nom d'hôte correspondant. L'adresse
# IP et le nom d'hôte doivent être séparés par au moins un espace.
#
# De plus, des commentaires (tels que celui-ci) peuvent être insérés sur des
# lignes propres ou après le nom d'ordinateur. Ils sont indiqué par le
# symbole '#'.
#
# Par exemple :
#
#      102.54.94.97     rhino.acme.com          # serveur source
#       38.25.63.10     x.acme.com              # hôte client x

127.0.0.1       localhost
127.0.0.1  www.gdawj.com
127.0.0.1  www.gdawj.net
```

Le fonctionnement peut être vérifié en saisissant l'adresse suivante dans un navigateur : http://www.gdawj.com:8080/. Nous devons alors obtenir la page d'accueil de notre serveur Tomcat étant donné que nous utilisons la bonne URL et le bon port.

Maintenant si nous voulons accéder à notre boutique directement depuis cette adresse, nous devons créer un hôte virtuel à la fin du fichier */conf/server.xml* de Tomcat.

```
<Host name="www.gdawj.com"
appBase="E:\Tomcat 6.0\webapps\chatbetaboutiquefin"
unpackWARs="true" autoDeploy="true" deployXML="false">
    <Alias>www.gdawj.net</Alias>
    <Context path="" reloadable="true" docBase=".">
        <Resource name="jdbc_chatbetaboutiquemysql" auth="Container"
type="javax.sql.DataSource" username="root" password=""
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/chatbetaboutique"
maxActive="20" maxIdle="10" validationQuery="SELECT 1"
defaultAutoCommit="true" characterEncoding="UTF-8"/>
    </Context>
</Host>
```

Cette définition est utilisée dans le cas d'un déploiement sans fichier *META-INF/context.xml* au sein de l'application. C'est le cas lorsque l'administrateur du serveur va déployer notre projet sur le serveur en production. Nous retrouvons la balise *<Host/>* avec le nom de domaine utilisé (*name*), le répertoire de base où se trouve l'application déployée

(*appBase*), le déploiement automatique de l'application (*autoDeploy*) et le décompactage des archives au format WAR (*unpackWARs*). L'attribut (*deployXML*) permet d'ignorer les fichiers *META-INF/context.xml* et de préciser que le contexte est déclaré à ce niveau.

La balise <*Alias/*> permet de ne pas créer un hôte virtuel pour chaque nom de domaine mais d'assigner la même configuration à plusieurs domaines. Enfin, nous retrouvons la déclaration courante du contexte avec une petite modification au niveau de l'attribut *path* qui est vide étant donné que nous n'accédons plus au site par une adresse de la forme http://localhost:8080/nomdemonprojet mais directement à partir du nom de domaine.

Nous pouvons redémarrer le serveur et accéder directement à l'application à partir des deux domaines.

Nous pouvons désormais optimiser l'ensemble afin de préciser l'attribut *workDir* qui est facultatif mais qui permet d'indiquer l'emplacement dans lequel Tomcat doit écrire les fichiers temporaires, tels que le code source et les Servlets compilées qu'il génère à partir des pages JSP. Si cet emplacement n'est pas précisé Tomcat crée les répertoires de travail dans le répertoire *Tomcat 6.0/work* et vérifie que chacun possède un nom unique pour éviter les conflits entre les différentes applications.

```
<Host name="www.gdawj.com" appBase="E:\Tomcat 6.0\webapps\chatbetaboutiquefin"
    unpackWARs="true" autoDeploy="true" deployXML="false"
workDir="E:\Tomcat 6.0\webapps\chatbetaboutiquefintemp">
    <Alias>www.gdawj.net</Alias>
    <Context path="" reloadable="true" docBase=".">
    <Resource name="jdbc_chatbetaboutiquemysql" auth="Container"
type="javax.sql.DataSource" username="root" password=""
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/chatbetaboutique"
maxActive="20" maxIdle="10" validationQuery="SELECT 1"
defaultAutoCommit="true" characterEncoding="UTF-8"/>
    </Context>
</Host>
```

Dans ce cas, l'application sera déployée dans le répertoire *chatbetaboutiquefin* et les fichiers temporaires dans le répertoire *chatbetaboutiquefintemp*.

Lors des déploiements avec la déclaration d'un hôte virtuel, parfois le premier démarrage ne fonctionne pas et indique que le répertoire portant le nom de l'application n'est pas trouvé. Il existe deux méthodes pour résoudre ce problème : soit réaliser un démarrage qui va créer le répertoire et ensuite un redémarrage, ou alors créer le répertoire portant le nom de l'application à la main avant le premier démarrage.

L'attribut *unpackWARs* permet de préciser si les fichiers WAR contenus dans le répertoire indiqué par *appBase* doivent être décompressés au démarrage du serveur. S'il possède la valeur *true*, l'hôte décompresse les fichiers WAR dans un répertoire portant le même nom que le fichier WAR sans l'extension. Par exemple, *chatbetaboutiquefin.war* est décompressé dans le répertoire *chatbetaboutiquefin*. Par contre, le serveur ne décompresse pas le fichier WAR s'il existe déjà un répertoire correspondant. Si nous souhaitons redéployer une application Web à partir d'un fichier WAR, nous devons supprimer ce répertoire.

En fait, lors de la mise en production d'un projet Java EE l'attribut *autoDeploy* doit être positionné à *false* car il indique à

Tomcat de mettre à jour automatiquement le serveur lorsque de nouveaux fichiers sont présentés dans le répertoire de l'application. La valeur par défaut est *true*, ce qui est très utile en phase de développement puisque nous voulons voir directement les modifications apportées à une Servlet ou page JSP. Ce système d'écoute permanent est lourd en terme de ressources. Pour un serveur en production, il est préférable de placer les fichiers et de recharger manuellement l'application avec le manager. De même l'attribut *liveDeploy* placé au niveau de la déclaration de l'hôte possède un fonctionnement similaire et doit également être positionné à *false* sur un serveur en production. Pour notre projet, le fichier de configuration en production sera donc le suivant :

```
<Host name="www.gdawj.com" appBase="E:\Tomcat 6.0\webapps\chatbetaboutiquefin"
 unpackWARs="false" autoDeploy="false" liveDeploy="false" deployXML="false"
workDir="E:\Tomcat 6.0\webapps\chatbetaboutiquefintemp">
    <Alias>www.gdawj.net</Alias>
    <Context path="" reloadable="true" docBase=".">
      <Resource name="jdbc_chatbetaboutiquemysql" auth="Container"
type="javax.sql.DataSource" username="root" password=""
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/chatbetaboutique"
maxActive="20" maxIdle="10" validationQuery="SELECT 1"
defaultAutoCommit="true" characterEncoding="UTF-8"/>
    </Context>
</Host>
```

3. Déploiement d'un projet Java EE à distance

Nous avons étudié jusqu'à maintenant plusieurs manières de déployer un projet Java EE à partir d'une archive WAR présente sur le serveur ou postée en FTP, avec SSH ou autre.

Il existe des possibilités pour déployer les projets à distance. Une possibilité intéressante est l'utilisation du manager Tomcat. Le formulaire HTML permet de parcourir notre disque dur à la recherche d'une archive WAR à l'aide du bouton **Parcourir** et de l'envoyer avec le bouton **Deploy**. Le projet est alors envoyé dans le répertoire */webapps* du serveur. Le déploiement est alors effectué comme expliqué précédemment.

Une autre solution consiste à utiliser l'outil Ant. Le fichier *build.xml* étudié dans le paragraphe précédent permet de générer correctement une archive WAR du projet *chatbetaboutique*. Une fois l'archive générée, il est possible de la déployer automatiquement sur un serveur Tomcat en utilisant Ant et le manager. Pour cela, il est nécessaire de déclarer une tâche externe avec la balise <*taskdef/*>. Pour commencer, il faut que le fichier *Tomcat 6/lib/catalina-ant.jar* soit accessible par Ant. Il faut utiliser pour cela une balise <*classpath/*>.

Nous créons une tâche externe pour le déploiement :

```
<!-- tache5 : déployer l'archive WAR -->
<property name="tomcat" location="E:/Tomcat 6.0"/>
<taskdef name="deploy" classname="org.apache.catalina.ant.DeployTask">
  <classpath>
    <path location="${tomcat}/lib/catalina-ant.jar" />
  </classpath>
</taskdef>
```

Ensuite, il est nécessaire d'ajouter une cible pour déployer l'application, cette cible utilise la tâche *deploy* déclarée précédemment, elle se connecte au manager de Tomcat avec un compte correct et déploie le projet avec les attributs nécessaires.

```
<!-- tache5 : déployer l'archive WAR -->
<property name="tomcat" location="E:/Tomcat 6.0"/>
<taskdef name="deploy" classname="org.apache.catalina.ant.DeployTask">
  <classpath>
    <path location="${tomcat}/lib/catalina-ant.jar" />
  </classpath>
</taskdef>

<target name="deployer" depends="war" description="Deployer l'archive war">
  <deploy url="http://localhost:8080/manager" username="admin"
password="admin"
  path="/chatbetaboutiquefin" war="./war/chatbetaboutiquefin.war"/>
```

```
</target>
```

Cette tâche permet donc d'utiliser la librairie *catalina-ant.jar* par l'intermédiaire de la balise <*classpath/*>. La balise <*deploy/*> permet de déployer le projet à distance, nous utilisons dans notre exemple le serveur http://localhost mais rien n'empêche d'utiliser un serveur distant. Nous précisons ensuite le compte pour se connecter au manager ainsi que l'archive WAR à déployer et le *path* sous lequel elle sera accessible.

Il est possible avec cette association de l'outil Ant et de la librairie *catalina-ant.jar*, d'utiliser toutes les tâches utiles à la gestion de projets Java EE, à savoir :

- *org.apache.catalina.ant.DeployTask* : permet de déployer un projet.
- *org.apache.catalina.ant.ReloadTask* : permet de recharger un projet.
- *org.apache.catalina.ant.ListTask* : permet de lister les applications déployées.
- *org.apache.catalina.ant.StartTask* : permet de démarrer une application.
- *org.apache.catalina.ant.StopTask* : permet d'arrêter une application.
- *org.apache.catalina.ant.UndeployTask* : permet de supprimer une application.
- *org.apache.catalina.ant.SessionsTask* : permet d'obtenir des informations sur la session.
- *org.apache.catalina.ant.RolesTask* : permet d'obtenir des informations sur les rôles.
- *org.apache.catalina.ant.ServerInfoTask* : permet d'obtenir des informations sur le serveur.
- *org.apache.catalina.ant.ResourcesTask* : permet d'obtenir des informations sur les ressources JNDI.

Cet exemple montre qu'il est relativement simple d'automatiser la création et le déploiement d'une application à partir des sources. Ant fait donc partie intégrante de la boîte à outils des administrateurs et développeurs de projets Java EE.

Optimisation de la mémoire

1. Présentation

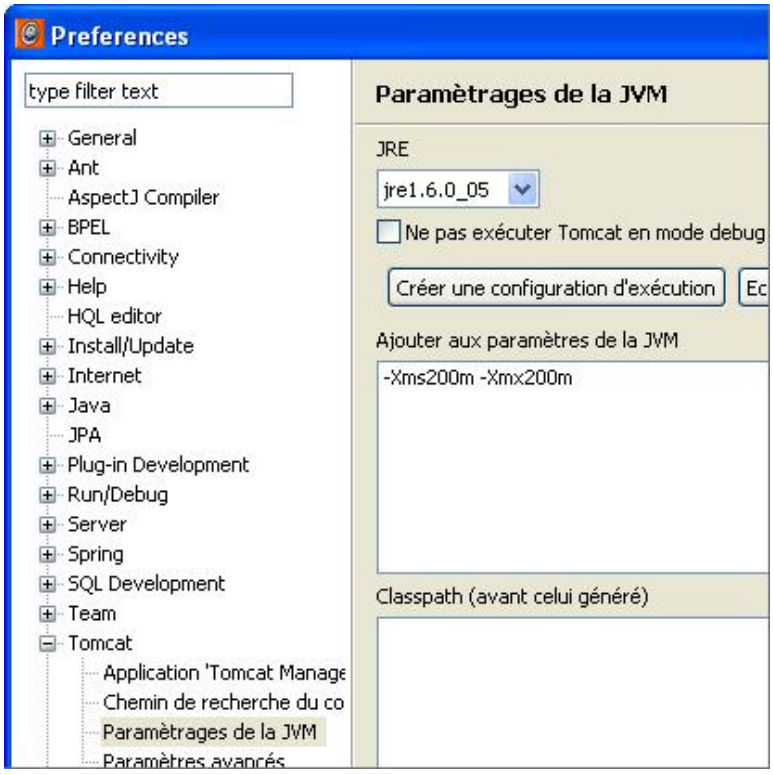
Les applications Java et plus particulièrement le serveur Tomcat sont gourmands en terme de mémoire vive (RAM). La version Tomcat 6.X améliore considérablement la gestion de la mémoire et l'utilisation du Garbage Collector ou ramasse-miettes afin d'éviter les fuites de mémoire et la persistance d'objets non utilisés dans la mémoire.

Avant d'allouer inutilement de la ressource mémoire à notre serveur, il est nécessaire de tester la montée en charge de celui-ci avec des outils de monitoring comme ceux présentés dans le chapitre consacré à Tomcat. Il n'est pas rare de constater que la montée en charge de la RAM est liée à une mauvaise fermeture de connexion au SGBD, à un pointeur de lecture d'un fichier ou un accès à des ressources diverses.

Le chapitre consacré à Tomcat explique en détail comment allouer et gérer la mémoire du serveur Java. Pour rappel, la page *status/état du serveur* du manager Tomcat permet d'afficher les informations sur l'utilisation de la mémoire.

Sous Windows, il est possible d'augmenter la mémoire allouée au serveur Tomcat en utilisant l'outil *Apache Tomcat Properties* et les options Java ou avec la fenêtre de préférences d'Eclipse.

La mémoire initiale réservée au serveur est paramétrée avec l'option *-Xms* et la mémoire maximale avec l'attribut *-Xmx*.



Sous Linux la mémoire allouée à Tomcat est paramétrée dans le fichier *catalina.sh*. Le code suivant permet d'allouer 2 Go de RAM au serveur.

```
JAVA_OPTS="-Xms2048m -Xmx2048m"
```

Ce principe d'allocation mémoire associé à la configuration de la JVM est très utilisé mais il est également parfois nécessaire de gérer la mémoire en dynamique du point de vue de la programmation. En effet, le code peut parfois être gourmand en terme de mémoire lors d'instanciations de nombreux objets, d'accès à des ressources importantes, de transformations d'images et d'utilisation de bibliothèques graphiques.

2. Gestion dynamique de la mémoire

La gestion dynamique de la mémoire et du Garbage Collector peut être réalisée en programmation. Dans notre projet *chatbetaboutique*, nous allons ajouter un attribut au fichier de configuration de l'application *web.xml*.

```
<!-- charge mémoire utilisée pour la limite (ex :
Total-free=15-6=9Mo de limite maxi à atteindre) -->
<context-param>
  <param-name>chargememoire</param-name>
  <param-value>6</param-value>
</context-param>
```

D'après la configuration affichée avec le manager de Tomcat, nous disposons dans notre exemple de 16 Mo de mémoire avec un maximum autorisé de 64 Mo. Cette charge mémoire correspond à un calcul mathématique qui est le maximum de charge mémoire utilisée sur la machine (Total - libre). Dans notre cas, la charge maximale autorisée est de 9 Mo de RAM. Si cette limite est atteinte, le Garbage Collector sera déclenché de manière forcée. Sur un serveur en production nous trouvons par exemple le calcul suivant : Total-libre=1290-600=690 Mo qui indique que pour 1 Go de RAM au total, le ramasse-miettes sera déclenché lorsque la mémoire aura atteint 690 Mo de charge.

Nous pouvons ensuite créer une classe statique nommée *OutilsGarbageCollector* placée dans le paquetage *boiteoutils* et qui permet de déclencher le Garbage Collector en fonction de la limite atteinte.

```
package boiteoutils;

public class OutilsGarbageCollector
{
//fixer la mémoire maxi à atteindre en méga octets
(500Mo soit une charge de Total-free=6-5=1Mo)
private static final int LIMITEMEMOIRE=5;

/**************************************************************************
 * fonction qui permet de vider le garbage collector en fonction d'une taille
**************************************************************************/
public static void verifierChargeGarbageCollector(double chargememoire)
{
    //récupérer les informations du système
    Runtime r=Runtime.getRuntime();
    //mémoire
    double memoirelibre=(r.freeMemory() / 1000000d);
    //limite
    double limitememoire;

    //utiliser notre limite ou celle par défaut
    if(chargememoire!=0) limitememoire=chargememoire;
    else limitememoire=LIMITEMEMOIRE;

    //si moins de mémoire libre que notre limite, alors vider la mémoire
    if(memoirelibre<limitememoire)
    {
        try
        {
            //appeler les méthodes finalize() des objets
            r.runFinalization();
            //vider la mémoire
            r.gc();
            System.gc();
        }
        catch(Exception e)
        {
            //logger
            System.out.println("Erreur lors du vidage de
la mémoire en mode automatique");
        }
    }
}

/**************************************************************************
 * fonction qui permet de vider le garbage collector de manière forcée
**************************************************************************/
public static void viderGarbageCollector()
```

```
{
    //récupérer les informations du système
    Runtime r=Runtime.getRuntime();

        try
        {
            //appeler les méthodes finalize() des objets
            try
            {
                //appeler les méthodes finalize() des objets
                r.runFinalization();
                //vider la mémoire
                r.gc();
                System.gc();
            }
            catch(Exception e)
            {
                //logger
                System.out.println("Erreur lors du vidage de
la mémoire en mode forcé runFinalization");
            }
        }
        catch(Exception e)
        {
            //logger
            System.out.println("Erreur lors du vidage de
la mémoire en mode forcé viderGarbageCollector");
        }
}

//fin de la classe
}
```

La première méthode permet de vider la RAM à l'aide du GarbageCollector à partir d'un maximum atteint correspondant à la valeur précisée dans le fichier de configuration *web.xml*. Pour cela nous vérifions la taille actuelle de la mémoire et en cas de besoin nous détruisons les objets non utilisés et nous lançons le Garbage Collector. La seconde méthode permet de toujours vider la RAM de manière forcée.

Maintenant nous pourrons vérifier cette mémoire et la vider au besoin dans une page JSP qui réalise un traitement complexe, une classe qui manipule d'importantes ressources ou alors un filtre qui sera déclenché sur toutes les URL.

```
<%@ page import="boiteoutils.OutilsGarbageCollector" %>
<%
//charge maxi de la mémoire autorisée
String chargememoire=getServletContext().getInitParameter("chargememoire");
//gérer la mémoire
OutilsGarbageCollector.verifierChargeGarbageCollector
(Double.parseDouble(chargememoire));
%>
```

Cette technique bien que radicale est très utilisée en production afin d'éviter les ralentissements des traitements voire même des saturations de mémoire. Cette solution nous permet en effet de bénéficier d'une plate-forme Internet conséquente, qui manipule d'importantes ressources sans risquer de saturer le serveur.

```
package chatbetaboutique;

import modele.UtilisateurModele;
import org.apache.struts.action.*;
import org.apache.struts.actions.MappingDispatchAction;
import boiteoutils.OutilsGarbageCollector;
import boiteoutils.Utilisateur;
import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import javax.sql.DataSource;


public class GestionUtilisateurAction extends MappingDispatchAction{
```

```
    //variables de la classe
    DataSource ds=null;

    //afficher la liste des utilisateurs
    public ActionForward listeutilisateur(ActionMapping mapping,
ActionForm form, HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
    {
       //récupérer la datasource du plugin dans un attribut
présent dans le context de la servlet
       ds=(DataSource)servlet.getServletContext().getAttribute("datasource");
       //créer le modèle
       UtilisateurModele utilisateurmodele=new UtilisateurModele(ds);
       //fermer la datasource
       this.ds=null;
       //retourner la liste des utilisateur
       ArrayList listeutilisateur=
(ArrayList)utilisateurmodele.getListeUtilisateur();
       //retourner la liste des utilisateurs
       request.setAttribute("listeutilisateur",listeutilisateur);
       //vider par sécurité
       listeutilisateur=null;

       //vider le garbage collector
       OutilsGarbageCollector.viderGarbageCollector();

       //retourner sur la page d'affichage des utilisateurs
       return mapping.findForward("listeutilisateur");
    }
}
```

En résumé

Ce chapitre a présenté les techniques avancées en Java EE. La première partie a présenté les intérêts de l'utilisation d'un outil de journalisation performant. En effet l'outil Log4J permet par l'intermédiaire des Logger, Appender et Layout de paramétrer de façon précise les traces d'un projet Java EE.

Dans un deuxième temps, l'outil Ant a été présenté pour la compilation, l'empaquetage et le déploiement des projets. Les balises *<projet/>* et *<target/>* ont été détaillées ainsi que les tâches afin de générer automatiquement une archive Java EE pouvant être directement déployée.

La troisième partie a concerné le déploiement d'un projet Java EE à partir d'une archive WAR sur un serveur en production. Le déploiement à partir d'un nom de domaine a été explicité dans cette étape ainsi que les balises XML nécessaires à la configuration. Un élément important a aussi été évoqué : le déploiement de projet Java EE à distance.

Le dernier paragraphe a développé un élément essentiel de la programmation de projet Java, à savoir la gestion et l'optimisation de la mémoire. La configuration de la mémoire du serveur Java EE a été présentée ainsi que la gestion dynamique de la mémoire en programmation à partir du fichier de configuration *web.xml* et de code Java.