Aide mémoire servlets et JSP

Nathanaël COTTIN

http://www.ncottin.net

Version 1.2

Comprendre

les servlets

et Java Server Pages

simplement

par la pratique

Table des matières

Introduction

Ce document récapitule les informations essentielles à la conception de sites web faisant appel à la technologie Java côté serveur par l'utilisation de servlets et de scripts JSP.

Les servlets constituent la réponse Java aux programmes CGI (« Common Gateway Interface »). Exécutées sur un serveur d'applications, elles interceptent les requêtes des navigateurs web et génèrent des réponses (souvent en DHTML) construites dynamiquement à l'aide du langage Java.

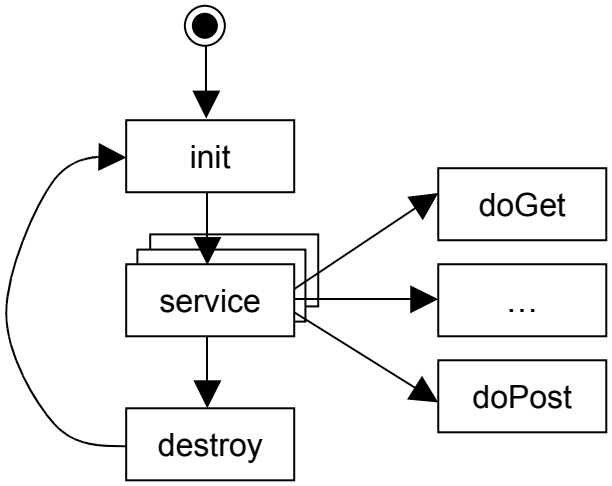
Les JSP sont l'extension des servlets et se rapprochent du modèle J2EE qui sépare :

- Les contenus statiques et dynamiques
- La présentation et la logique métier, notamment grâce aux JavaBeans.

Servlets

Les servlets font usage de la technologie Java pour générer des pages HTML dynamiques (DHTML). Elles permettent l'intégration de l'ensemble des fonctionnalités de J2EE, contrairement à d'autres langages web et notamment PHP.

Cycle de vie



Les méthodes « init() » et « destroy() » ne sont appelées qu'une seule fois par le conteneur de servlets, respectivement la première fois que la servlet est appelée et au moment où le conteneur (donc le serveur d'applications) termine son exécution, contrairement aux méthodes « service() » et « doXXX() », appelées à chaque invocation de la servlet.

Exemple

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

public class Hello extends HttpServlet {

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

      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head>");
      out.println("<title>");
      out.println("Test de servlet");
      out.println("</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>Hello World!!!</h1>");
      out.println("<p>This is a test servlet</p>");
```

```
        out.println("</body>");
        out.println("</html>");
    }
}
```

Gestion des paramètres

Les paramètres peuvent être indiqués :

- Directement dans l'URL de la requête (réécriture d'URL) :

```
http://serveur/servlet?param1=valeur1&param2=valeur2
```

- Transmis par le biais d'un formulaire (principalement GET, POST et POST/multipart, utilisé pour les « uploads » de fichiers sur le serveur).

Les paramètres en entrée sont récupérés par la servlet à l'aide de la méthode « getParameter » :

```
String request.getParameter(String paramName)
```

Gestion des informations de session

Les servlets permettent de simuler un mode « connecté » entre les pages consultées par l'utilisateur. En effet, le web n'offre par défaut aucun suivi de session. Cette fonctionnalité est supportée par les servlets à l'aide de l'objet « HttpSession » obtenu à l'aide de `request.getSession()`.

Les méthodes « `public void putValue(String name, Object value)` » et « `public Object getValue(String name)` » permettent respectivement d'attribuer (i.e. modifier) et consulter une valeur de session identifiée par son nom.

JSP

Définition

La technologie JSP simplifie la mise en œuvre de sites faisant appel à Java côté serveur. Il s'agit de code DHTML dans lequel s'insèrent des scriptlets.

Les pages JSP sont transformées « à la volée » par un traducteur interne au serveur d'applications lors de leur premier appel. Cette traduction permet de générer, compiler puis instancier un objet Java correspondant. Ce dernier est comparable, dans son fonctionnement, à une servlet pour laquelle la méthode « service » a été redéfinie à l'aide des instructions JSP.

Le code DHTML est ainsi interprété par le traducteur afin de le retourner au client à l'aide de commandes similaires à l'instruction « out.println() » des servlets. Le code Java n'est soumis à aucun traitement avant compilation de la classe Java résultante.

Résumé des directives

page	Informations relatives à la page courante
include	Fichier à inclure littéralement. Ce fichier peut donc lui-même contenir du code JSP

Les directives JSP s'écrivent généralement sous la forme

```
<%@ <nomDirective> <attributs> %>
```

Le balisage XML équivalent est

```
<jsp:<directive>:<nomDirective> <attributs> />
```

Directives de type « page »

```
<%@ page language="java" %>
<%@ page info="description" %>
<%@ page import="paquetage1.*" %>
<%@ page imports="paquetage1.*, paquetage2.classe" %>
<%@ page extends="classe" %>
<%@ page session="true|false" %>
<%@ page buffer="none|tailleKo" %>
<%@ page autoFlush="true|false" %>
<%@ page errorPage="cheminRelatifPageErreur" %>
<%@ page isErrorPage="true|false" %>
<%@ page contentType="text/html;charset=ISO-8859-1" %>
<%@ page isThreadSafe="true|false" %>
```

Directives de type « include »

```
<%@ include file="chemin_fichier_local" %>
```

Le fichier indiqué par « chemin_fichier_local » est inséré tel quel à l'endroit de l'appel à la directive « include ».

Il est également possible d'utiliser la directive « jsp:include » :

```
<jsp:include page="URL_relative" flush="true" />
```

Ou, alternativement :

```
<jsp:include page="URL_relative" flush="true">
   <jsp:param name="paramName" value="paramValue" />
</jsp:include>
```

Dans ce cas, les ressources incluses n'ont accès qu'à l'objet « out » (de type `JspWriter`) JSP et ne peuvent faire usage des cookies. De plus, celles-ci ne peuvent contenir du code JSP général.

Enfin, la première directive d'inclusion est exécutée au moment de la traduction du JSP en code Java (inclusion statique) alors que la seconde directive est exécutée au moment du traitement de la requête (inclusion dynamique).

Autres directives

La directive « taglib » permet de définir des balises personnalisées dans des bibliothèques de balises :

```
<%@ taglib uri="tagLibrairie" prefix="tagPrefix" %>
```

La commande « forward » permet de rediriger la page courante (JSP) vers une seconde page tout en conservant les informations de session collectées jusqu'alors (par les beans notamment) :

```
<jsp:forward page="autre_page.jsp" />
```

Ou alternativement (en mettant à jour la valeur d'un paramètre) :

```
<jsp:forward page="URL_relative">
   <jsp:param name="paramName" value="paramValue" />
</jsp:forward>
```

L'action « plugin » est utilisée pour générer des balises HTML de type <OBJECT> ou <EMBED> en fonction du navigateur client. Elle a pour effet le téléchargement du plug-in Java (si nécessaire) suivi de l'exécution de l'applet ou du bean spécifié dans la balise :

```
<jsp:plugin
   type="bean|applet"
   name="objectName"
   code="objectCode"
   codebase="objectCodebase"
   hspace="hspace"
   vspace="vspace"
   width="width"
   jreversion="jreversion"
   nspluginurl="url"
   iepluginurl="url">

   <jsp:params>
      <jsp:param name="param1" value="paramValue1" />
      <jsp:param name="paramN" value="paramValueN" />
   </jsp:params>

   <jsp:fallback>Texte alternatif si plugin non chargé
   </jsp:fallback>
</jsp:plugin>
```

Scriptlets

Définition et intégration

Les scriptlets permettent d'insérer du code Java au sein du code DHTML des pages JSP à l'aide des délimiteurs « <% » et « %> » :

```
<% code Java %>
```

Il est également possible d'utiliser des balises XML. Les scriptlets sont alors déclarées par :

```
<jsp:scriptlet> code Java </jsp:scriptlet>
```

Le code rédigé est recopié tel quel par le traducteur Java des scripts JSP. Ces derniers autorisent également la déclaration de variables de classe et de méthodes permettant d'agréger les traitements de présentation. Dans ce cas, les délimiteurs « <%! » et « %> » sont employés.

Exemples

```
<%
String str = "Boucle ";
for (int i=0; i<10; i++) {
  out.println(str + i + "<br/>");
}
%>
```

Le code précédent utilise la variable « out » définie par le traducteur de pages JSP en code Java intégral. Il est strictement équivalent à :

```
<%
for (int i=0; i<10; i++) {
%>
   Boucle <%= i %><br/>
<%
}
%>
```

L'affichage d'une variable ou d'une expression (l'évaluation d'une méthode par exemple) se fait ainsi indifféremment par `<% out.println(nomVariable); %>`, `<%= nomVariable %>` ou en utilisant l'opération « getProperty » appropriée lorsqu'il s'agit d'un bean.

L'équivalent XML est `<jsp:expression>nomVariable</jsp:expression>`.

Rédaction des commentaires

Les commentaires JSP ont leur propre syntaxe :

```
<%-- Début de commentaire JSP
Fin de commentaire JSP --%>
```

Les informations situées entre les balises de début et fin de commentaire ne sont pas intégrées lors de la traduction de la page JSP. Par contre, les commentaires HTML apparaîtront comme tels lors de la génération de la page HTML. Rappelons que les commentaires en langage HTML s'écrivent comme suit :

```
<!-- Début de commentaire HTML
Fin de commentaire HTML -->
```

Les commentaires Java demeurent valides au sein des scriptlets.

Utilisation avancée

Déclaration de méthodes et variables de classe

L'ensemble du code Java (et des instructions DHTML) est placé par le traducteur de scripts JSP au sein de la méthode « _jspService », similaire à la méthode « service » des servlets. De fait, les variables définies dans le script JSP sont déclarées comme locales à cette méthode par le traducteur ayant généré l'objet Java correspondant au JSP.

Il est toutefois permis de :

- Déclarer des méthodes utilisables par les scriptlets du JSP.

  Par exemple, la méthode « isEmpty » qui vérifie si un paramètre est renseigné ou non :

```
<%!  private boolean isEmpty(String str) {
        return (str == null) || str.equals("");
     }
%>
```

- De déclarer des variables de classe.

  L'exemple suivant déclare les variables privées non statiques « monEntier » et « maChaine » :

```
<%!  private int monEntier = 0;
     private String maChaine = null;
%>
```

L'équivalent XML de ce type de déclaration est :

```
<jsp:declaration>
     private int monEntier = 0;
     private String maChaine = null;
</jsp:declaration>
```

Note : l'intérêt de recourir à l'utilisation de variables de classe semble beaucoup moins évident que la déclaration de méthodes. En effet, les variables définies au sein

des scriptlets sont reconnues par l'ensemble des scriptlets situées en aval dans la page JSP.

Initialisation et destruction de page JSP

Conformément aux méthodes « init() » et « destroy() » des servlets, JSP met en œuvre les méthodes « jspInit() » et « jspDestroy() » ayant pour signature :

```
public void jspInit();
public void jspDestroy();
```

Ces méthodes ont un comportement identique à leurs équivalents servlets.

Liste des objets définis par défaut par le traducteur de JSP

Objet	Utilisation
request	De type `HttpServletRequest`
response	De type `HttpServletResponse`
pageContext	Fournit l'ensemble des objets implicites JSP
session	Session HTTP correspondant à la requête (`HttpSession`)
application	Valeur retournée par `getServletConfig().getContext()`, permettant de manipuler des attributs communs à tous les JSP : • `void setAttribute(String, Object)` • `Object getAttribute(String)`
out	Flux de sortie, de type `JspWriter` (similaire à `PrintWriter`)
config	Valeur retournée par `getServletConfig()`
page	Remplace le mot-clé « `this` »
exception	Erreur : • Transmise à la page d'erreur (si « errorPage » est spécifié dans la page ayant généré l'ereur). • Récupérée par la page d'erreur (uniquement si la propriété « isErrorPage » est positionnée à « true »).

Page d'erreur

Les erreurs non capturées par les scriptlets peuvent être redirigées vers une page d'erreur via la directive « <%@ page errorPage="cheminRelatifPageErreur" %> ». Cette page d'erreur peut être ou non une page JSP. Seul un chemin relatif est toléré.

Dans le cas où la page d'erreur est un script JSP, l'attribut « isErrorPage » doit explicitement être indiqué de sorte que la variable intitulée « exception » (de type `Throwable`) puisse être utilisée. Il est également possible de faire appel à l'attribut « javax.servletjsp.jspException » de la requête (via l'objet implicite « request »).

Lorsqu'une exception est levée, le tampon de sortie est vidé avant affichage de la page d'erreur. Néanmoins, si la page source de l'exception non capturée autorise l'autoFlush, une erreur HTTP 500 (voir codes HTTP en annexe) peut survenir si une réponse partielle a déjà été retournée au client.

JavaBeans

Définition

Un JavaBean est un composant Java permettant de séparer la logique métier de la présentation des données. Il s'agit d'une classe Java (souvent déclarée « finale ») qui dispose des propriétés suivantes :

- Il doit être déclaré « public » afin d'être accessible depuis les scripts JSP.
- Le seul constructeur disponible est public sans argument. Ce dernier doit initialiser les valeurs par défaut des attributs (à « null » par exemple).
- Les méthodes publiques « set » et « get » sont définies pour chaque attribut, selon le principe des DesignPatterns. Par exemple, un attribut « att » doit disposer des méthodes « `public void setAtt(String val)` » et « `public String getAtt()` ».

Déclaration

Les JavaBeans s'intègrent au sein des JSP en les déclarant en en-tête à l'aide de la directive « useBean » :

```
<jsp:useBean id="beanId" class="beanClass" scope="session" />
```

La portée du bean (« scope ») peut être :

- « application » : le bean est persistant et partagé entre tous les clients (et par conséquent les pages JSP) qui en font usage.
- « session » : le bean est spécifique à chaque client. Il permet de transmettre des informations de session d'une page à l'autre. Il suffit de déclarer le même bean dans la seconde page pour que celle-ci soit en mesure de récupérer les informations de session enregistrées par le bean.
- « request » : le bean est détruit lorsque le client demande une nouvelle page. Le passage des paramètres de suivi de session doit ainsi être réalisé manuellement (par formulaire avec champs cachés ou par récriture d'URL).
- « page » : rarement employée, cette portée est limitée à la page en cours (aucune information de session ne peut être récupérée) : ce bean est accessible à n'importe quel endroit de la page.

Cette directive doit être complétée par l'instruction suivante :

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

Cette dernière permet de limiter éventuellement les informations que le bean doit mettre à jour. Par défaut, le caractère générique « * » permet de spécifier que le bean récupère autant d'informations que possible. Cette déclaration de mise à jour doit être indiquée dans le code JSP : dans le cas contraire, les attributs du bean ne seront pas mis à jour.

Une restriction cependant : l'utilisation des JavaBeans au sein des pages JSP ne permet pas de différentier directement (sans recourir à l'objet implicite « request ») la méthode d'envoi HTTP (GET, POST, etc.).

Utilisation

Lorsqu'un bean est déclaré et porte d'identifiant « beanId », il peut être utilisé directement au sein du code DHTML ou depuis une scriptlet.

Chaque bean dispose d'un ensemble d'attributs (et des méthodes « set » et « get » correspondantes). Lors de l'appel d'un script JSP avec envoi d'informations, les attributs déclarés dans les beans sont mis à jour (par appel interne des méthodes « set »).

Lorsqu'une valeur d'attribut n'est pas mentionnée par la requête, deux cas sont envisageables (la portée « page » est ici hors de propos) :

- La portée du bean est « application » ou « session » : l'ancienne valeur de l'attribut est conservée (la requête ne mettant pas à jour cet attribut).
- La portée du bean est « request » : dans ce cas, la valeur de l'attribut est réinitialisée (souvent à « null », mais toute autre valeur peut être spécifiée par le constructeur, appelé lors de la création de l'instance du bean).

Utilisation directe

En règle générale, l'utilisation directe n'est pas conseillée car il est souvent nécessaire d'effectuer des vérifications (validation) avant affichage. Néanmoins, l'affichage de l'attribut « att » du bean « beanId » utilise la syntaxe suivante :

```
<jsp:getProperty name="beanId" property="att" />
```

Lorsque le paramètre « att » n'a pas été transmis, le résultat de cet appel sera « null » (conversion du pointeur« null » en chaîne de 4 caractères valant « null »).

La mise à jour de cet attribut emploie la directive « jsp:setProperty » :

```
<jsp:setProperty name="beanId" property="att" value="valeur" />
```

Le caractère générique « * » peut remplacer « att » afin d'indiquer que l'ensemble des propriétés du bean doivent être mises à jour. Dans ce cas, le dernier paramètre n'est pas mentionné :

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

Le serveur d'applications utilise le principe de « réflexion » (ou « miroir ») Java pour appeler les méthodes de type « set » déclarées par le bean. Consulter le paquetage « java.lang.reflect » de l'API Java pour en savoir davantage.

Utilisation au sein des scriptlets

Il est possible d'appeler l'ensemble des méthodes déclarées publiquement par le bean, comme l'indique l'exemple suivant :

```
<%
String valeur = beanId.getAtt();
if (valeur == null) {
   // La requête ne mentionne pas le paramètre "att"
}
else {
   // La requête mentionne effectivement le paramètre "att"
}
%>
```

Exemple simple de mise en œuvre

Cet exemple propose de remplir un formulaire puis d'afficher les informations saisies par l'utilisateur à l'aide d'un script JSP et d'un bean « SimpleBean » (qui devra être compilé). Pour simplifier, seul le nom de l'utilisateur est récupéré.

Construction du bean

Le bean « simpleBean » est déclaré comme faisant partie du paquetage « test » afin d'être compatible avec le serveur d'applications Tomcat. A l'attribut privé « name » sont associées les méthodes « setName » et « getName ». Enfin, le constructeur sans argument est indiqué (même s'il est dans ce cas facultatif). Le code du bean proposé est donc conforme à la définition d'un JavaBean.

```
package test;
// La définition d'un paquetage est nécessaire sous Tomcat

public final class SimpleBean {
   private String name = null;      // Attribut privé

   public SimpleBean() {
      // Il n'est pas nécessaire d'indiquer le constructeur dans ce cas
      // car l'attribut à déjà été initialisé
   }

   public void setName(String value) {
      this.name = value;
   }

   public String getName() {
      return this.name;
   }
}
```

Création de la page de formulaire « index.html »

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

<!-- L'appel à la page JSP suppose que la connexion est déjà établie,
par exemple sous "localhost" en indiquant le port 8080 -->

<br/><br/>
<form action="display.jsp" method="post">
  Nom : <input type="text" name="name">
  <!-- Le champ s'appelle "name" afin que l'attribut correspondant
  soit mis à jour par le bean via "setName(String)" dans la page JSP -->
  <br/><input type="submit" value="Envoyer">
</form>

</body>
</html>
```

Création de la page « display.jsp »

Première version

Le paramètre « name » fourni par l'envoi du formulaire est récupéré par le bean « simpleBean ». Il est ensuite affiché deux fois : en utilisant un affichage direct par la directive « jsp:getProperty » puis à l'aide de la chaîne de caractères « nameStr » :

```
<%@ page language="java" %>

<jsp:useBean id="simpleBean" class="test.SimpleBean" scope="request" />
<jsp:setProperty name="simpleBean" property="*" />

<html>
<head>
</head>
<body>

<!-- Affiche 2 fois le nom, à l'aide de 2 techniques différentes -->

Nom saisi : <jsp:getProperty name="simpleBean" property="name" />
<%
// Le nom a été mis à jour automatiquement par le bean (cf setProperty)
String nameStr = simpleBean.getName();
%>
<br/>
Nom saisi: <%= nameStr %>

<br/><br/>
<form action="display.jsp" method="post">
  Nouveau nom : <input type="text" name="name">
  <br/><input type="submit" value="Envoyer">
</form>
```

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

Seconde version

Le paramètre « name » fourni par l'envoi du formulaire est récupéré par le bean « simpleBean ».

Si une valeur a été transmise, celle-ci est affichée. Dans le cas contraire, le script redirige vers la page d'erreur « error.jsp » via la directive JSP « <%@ page errorPage="error.jsp" %> » :

```
<%@ page language="java" %>

<%@ page errorPage="error.jsp" %>

<jsp:useBean id="simpleBean" class="test.SimpleBean" scope="request" />
<jsp:setProperty name="simpleBean" property="*" />

<%!
// Ajout de la méthode "isEmpty" pour valider le paramètre "name"
private boolean isEmpty(String str) {
  return (str == null) || str.equals("");
}
%>

<%
String nameStr = simpleBean.getName();
if (isEmpty(nameStr)) {
  throw new Exception("Veuillez saisir un nom");
}
%>

<html>
<head>
</head>
<body>

Nom saisi: <%= nameStr %>

<br/><br/>
<form action="display.jsp" method="post">
  Nouveau nom : <input type="text" name="name">
  <br/><input type="submit" value="Envoyer">
</form>
</body>
</html>
```

Création de la page d'erreur « error.jsp »

Première version

La page d'erreur affiche le texte du message indiqué par la page ayant généré l'erreur.

```
<%@ page language="java" %>
<%@ page isErrorPage="true" %>
```

```
<html>
<head>
</head>
<body>
<h1>Erreur :
<span style="color:red;">
<!-- Utilisation de la variable "exception" déclarée automatiquement
lors de la génération du code Java correspondant à la page JSP -->
<%= exception.getMessage() %>
</span>
</h1>

<br/><br/>
<form action="display.jsp" method="post">
  Nouveau nom : <input type="text" name="name">
  <br/><input type="submit" value="Envoyer">
</form>
</body>
</html>
```

Seconde version

Cette version propose d'afficher la pile des erreurs, notamment en cas de débogage.

```
<%@ page language="java" %>
<%@ page isErrorPage="true" import= "java.io.*" %>

<html>
<head>
</head>
<body>
<h1>Erreur :
<span style="color:red;">
<!-- Utilisation de la variable "exception" déclarée automatiquement
lors de la génération du code Java correspondant à la page JSP -->
<%
StringWriter sw = new StringWriter();
PrintWriter pw = new PrintWriter(sw);
excetion.printstackTrace(pw);
out.println(sw);
%>
</span>
</h1>

<br/><br/>
<form action="display.jsp" method="post">
  Nouveau nom : <input type="text" name="name">
  <br/><input type="submit" value="Envoyer">
</form>
</body>
</html>
```

Utilisation de la directive « include »

Considérons le bloc de code HTML suivant :

```
<br/><br/>
<form action="display.jsp" method="post">
  Nouveau nom : <input type="text" name="name">
  <br/><input type="submit" value="Envoyer">
</form>
```

Ce bloc est commun à l'ensemble des pages JSP. Il serait judicieux de l'extraire dans un fichier tiers « form.dat » (par exemple) qui sera ensuite intégré au sein des pages JSP à l'aide de la commande « `<%@ include file="form.dat" %>` ».

Connexion aux bases de données

La création d'applications nécessite généralement une communication avec une base de données. Ceci suppose d'une part l'installation de la base de données et d'autre part l'exécution d'un démon sur la machine hébergeant la base.

L'intégration des appels à la base de données au sein du code java des JSP est réalisé à l'aide de la directive :

```
<%@ page import="java.sql.*" %>
```

Connexion d'une page JSP à une base de données

Les JSP doivent ensuite se connecter à la base choisie via JDBC (« Java DataBase Connectivity »), l'équivalent Java des pilotes ODBC. L'annexe 4 traite spécifiquement de la connexion à une base MySQL sous Tomcat. L'idée est d'instancier le pilote approprié (il s'agit par exemple de la classe « org.gjt.mm.mysql.Driver ») dans une scriptlet à l'aide de l'instruction :

```
<%
Class.forName("ClasseDuDriver").getInstance();
%>
```

L'interface « java.sql.Connection » permet d'établir la connexion avec la base de données (à l'aide du service « java.sql.DriverManager »). Elle indique de plus l'ensemble des tables qui composent la base ainsi que les propriétés de ces tables (description) :

```
<%
String urlBase = "jdbc:TypeBase://Hôte/NomBase?user=UtilisateurParDéfaut";
Connection c = DriverManager.getConnection(urlBase);
%>
```

La machine hôte est souvent référencée comme étant « localhost ». La portion d'adresse débutant par « ?user= » est facultative.

Exécution d'une requête SQL

Le langage SQL (« Simple Query Language ») est le langage standard de manipulation de bases de données (création, suppression, sélection, mise à jour).

L'exécution d'une requête à l'aide de l'interface « java.sql.Statement » suppose que la connexion avec une base de données est établie.

Dans une première phase, la requête est préparée. Deux alternatives sont proposées : effectuer une préparation statique ou une préparation dynamique.

La préparation statique revient à spécifier la requête directement sous forme d'une chaîne de caractères :

```
<%
int id = 10;
String query = "SELECT * FROM MA_TABLE WHERE `id` = '" + id + "'";

Statement stmt = c.createStatement();
ResultSet rs = stmt.executeQuery(query);
%>
```

De fait, appeler cette même requête avec un identifiant différent (« id ») revient à recréer entièrement la requête et demander sa ré-exécution. Pour pallier ce défaut, le paquetage « java.sql » fournit la possibilité de créer une requête pour laquelle certaines valeurs sont omises :

```
<%
PreparedStatement ps = c.prepareStatement(query) ;
ps.setInt(1, 10);

Statement stmt = c.createStatement();
ResultSet rs = stmt.executeQuery();
%>
```

Note : l'instruction « executeQuery » doit être remplacée par « executeUpdate » dans le cas où la requête SQL est de type « INSERT » ou « UPDATE ».

Conclusion

Les JSP (et les JavaBeans) sont en règle générale plus intuitifs que les servlets, cependant le recours aux servlets peut s'avérer utile dans certains cas :

- Besoin de contrôle total du code pour éviter les lignes de code « inutiles » générées par le serveur d'applications lors de la transformation des JSP en code Java.
- Envoi de données qui ne sont pas au format HTML (le contenu d'un fichier « zip » par exemple), c'est-à-dire dont le type de contenu (« content-type ») n'est pas « text/html ».

Les JSP permettent en outre de séparer au maximum la logique métier (traitements) de la présentation. La possibilité d'inclure textuellement des fichiers annexes permet de conserver une certaine cohérence dans la présentation des informations.

Enfin, les JSP (et les servlets) ouvrent la conception web à l'ensemble de la technologie Java côté serveur décrite par J2EE et notamment aux systèmes distribués par le biais des EJB (« Enterprise Java Beans ») ou par le recours aux solutions de type CORBA.

Annexe 1 : utilisation des servlets et JSP sous Tomcat

Installation

L'exécutable Tomcat peut être téléchargé depuis l'URL « http://jakarta.apache.org ».

Avant de procéder à son installation, un JDK (1.2 ou supérieur) doit être installé et la variable d'environnement `JAVA_HOME` définie. Cette dernière indique le chemin menant au dossier racine du JDK (« C:\jdk1.4.2 » sous Windows, par exemple).

Mise en route

Tomcat écoute le port 8080 (« alternate HTTP ») par défaut. Les URL permettant de se connecter avec le serveur d'applications sont donc de la forme « http://serveur:8080/ ».

Les tests en mode local sont de fait exécutés avec « http://localhost:8080/ » ou « http://127.0.0.1:8080/ ». Pour y accéder, il faut :

- Démarrer Tomcat.
- Ouvrir un navigateur web et indiquer l'URL « http://localhost:8080/ ».

Ces deux dernières URL ouvrent la page d'accueil de Tomcat, à partir de laquelle il est possible d'accéder à un ensemble d'exemples avec leur code source.

Remarque d'utilisation générale

L'ensemble des sources (notamment des JSP) doit être inscrit dans le dossier « webapps » situé sous la racine d'installation de Tomcat. Il est fortement recommandé de créer un sous-dossier pour chaque projet (servlets ou JSP). Chaque sous-dossier doit être de la forme suivante :

```
webapps/
      mon_dossier/
            (mon_sous_dossier/) mes_sources (.java, .jsp)
            WEB-INF/
                  classes/
                  (web.xml)
```

Lorsqu'une modification d'un fichier source demande une compilation manuelle, Tomcat doit être redémarré pour que celle-ci soit prise en compte. De plus, des problèmes liés à la mise en œuvre de caches par le navigateur web peuvent survenir. Il est conseillé de fermer également toutes les fenêtres de navigateur ouvertes (afin de vider le cache) avant de procéder à de nouveaux essais.

Servlets sous Tomcat

Chaque servlet est compilée manuellement. Le résultat de cette compilation (fichier portant l'extension « class ») doit être placé dans le dossier « WEB-INF/classes ».

Cette phase de compilation demande l'intégration des paquetages dérivant de « javax.servlet ».

Deux solutions sont envisageables :

- Installer un JSDK (et non un JDK).
- Utiliser la librairie « servlet-api.jar » de Tomcat : dans ce cas, il faut penser à l'ajouter au CLASSPATH avant de demander la compilation de la servlet. Cette mise à jour peut être réalisée temporairement au moment de la compilation à l'aide de l'instruction (sous environnement Windows) :

```
set TOMCAT_HOME=C:\Program Files\Apache Software Foundation\Tomcat 5.0
set CLASSPATH=%TOMCAT_HOME%\common\lib\servlet-api.jar;%CLASSPATH%
```

La compilation n'est cependant pas suffisante pour pouvoir exécuter la servlet : celle-ci doit en effet être déclarée dans le fichier « web.xml » situé sous le dossier « WEB-INF » évoqué précédemment (attention : il ne s'agit pas du fichier « web.xml » du dossier « conf » de Tomcat !).

La commande de compilation suivante réalise ce travail, en supposant que les servlets sont créées dans le dossier « webapps/servlets/ » :

```
javac *.java –d ./WEB-INF/classes
```

Voici un exemple de fichier « web.xml » simplifié déclarant les servlets « Test » et « Hello » :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!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>Servlet examples</display-name>
      <description>Servlet examples</description>

      <servlet>
            <servlet-name>Test</servlet-name>
            <servlet-class>Test</servlet-class>
      </servlet>

      <servlet-mapping>
            <servlet-name>Test</servlet-name>
            <url-pattern>/Test</url-pattern>
      </servlet-mapping>

      <servlet>
            <servlet-name>Hello</servlet-name>
            <servlet-class>Hello</servlet-class>
      </servlet>

      <servlet-mapping>
            <servlet-name>Hello</servlet-name>
```

```
        <url-pattern>/Hello</url-pattern>
    </servlet-mapping>

</web-app>
```

JSP et JavaBeans sous Tomcat

Tomcat supporte les JSP v1.0 et v2.0. Contrairement aux servlets, la modification d'un fichier JSP ne demande pas de redémarrage de Tomcat. Cependant, toute modification d'un bean (suivie d'une compilation manuelle) entraîne nécessairement un redémarrage pour que celle-ci soit répercutée.

Pour qu'un bean soit reconnu par les pages JSP, ce dernier doit nécessairement faire partie d'un paquetage (« package test; » par exemple).

De manière similaire aux servlets, le résultat de la compilation des beans doit être placé dans le dossier « WEB-INF/classes ».

Enfin, le fichier « web.xml » n'est plus requis.

Les classes Java correspondant à la traduction des scripts JSP par le moteur Catalina sont générées dans un sous-dossier de « work/Catalina/localhost ». Le radical de chaque fichier JSP produit un code source Java nommé comme suit : « radical_jsp.java ». Par exemple, le fichier « index.jsp » sera traduit par Catalina pour produire le fichier « index_jsp.java ».

Lorsqu'un script JSP semble ne pas produire le résultat attendu, il convient :

- De vérifier que les beans sont compilés.
- De supprimer les fichiers Java (et classes pseudo-compilées à partir de ces fichiers sources) de ce sous-dossier afin de forcer Tomcat à les générer à nouveau.

Annexe 2 : description du protocole HTTP

HTTP (« HyperText Transfer Protocol ») est un protocole hypermédia déconnecté permettant d'échanger des informations indépendamment de la nature du matériel employé.

Les communications HTTP utilisent par défaut le port TCP 80 mais peuvent être réalisées par le biais d'autres ports de communication (tel que TCP 8080 ou « alternate HTTP »).

Son fonctionnement en mode déconnecté ne permet pas a priori de conserver une information autrement qu'en la transmettant au sein de chaque requête.

Annexe 3 : principaux codes d'état HTTP

Liste des codes HTTP les plus employés :

Code HTTP	Constante HttpServletResponse	Description
200	SC_OK	La requête a abouti
201	SC_CREATED	La requête a abouti et a créé une ressource sur le serveur
202	SC_ACCEPTED	La requête est acceptée mais n'est pas effectuée
204	SC_NO_CONTENT	La requête a abouti mais n'a produit aucun contenu nouveau
400	SC_BAD_REQUEST	La syntaxe de la requête est incorrecte
401	SC_UNAUTHORIZED	La requête exige une authentification HTTP
403	SC_FORBIDDEN	Le serveur interprète correctement la requête mais ne peut la traiter
404	SC_NOT_FOUND	La ressource demandée est inaccessible
500	SC_INTERNAL_SERVER_ERROR	Une erreur survenue au sein du serveur HTTP a fait échouer l'exécution de la requête
501	SC_NOT_IMPLEMENTED	Le serveur HTTP ne prend pas en charge les fonctionnalités nécessaires au traitement de la requête
503	SC_SERVICE_UNAVAILABLE	Le serveur HTTP est surchargé et ne peut traiter la requête

Ces codes d'erreur peuvent être retournés au client à l'aide de la méthode « sendError » de la classe « HttpServletResponse ».

Depuis les servlets ou les pages JSP, l'appel est effectué comme suit :

```
response.sendError(400, "Message")
```

Annexe 4 : JSP sous Tomcat et base de données MySQL

Cette annexe suppose qu'une base de données MySQL est (installée et) active sur la même machine que Tomcat.

Installation du paquetage « util.sql »

La librairie « util », disponible sur le site **http://www.ncottin.net**, propose un grand nombre de foctionnalités. Dans le cas présent, le paquetage « util.sql » fournit une classe « DbConnection » simplifiant l'interrogation d'une base de données.

La reconnaissance de ce paquetage par les scripts JSP se réalise comme suit :

1. Télécharger le paquetage sur le site « www.ncottin.net »
2. Installer l'archive dans le dossier « shared/lib » de Tomcat
3. Inclure la directive « <%@page import "util.sql.*" %> » dans les pages JSP.

Utilisation de la classe « DbConnection »

Cet exemple simple permet de compter le nombre d'entrées dans la table « TABLE » de la base de données « simpledb ». Bien sûr il existe l'instruction SQL « count(*) » mais l'objectif est de montrer l'intérêt de la classe « DbConnection »…

Requête statique

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

<%
DbConnection dbc = new DbConnection(
      "org.gjt.mm.mysql.Driver",
      "jdbc:mysql://localhost/simpledb");

String query = "SELECT * FROM `TABLE`";
ResultSet rs = dbc.execute(query);
if (dbc.isEmpty(rs)) {
      throw new Exception("EMPTY TABLE");
}

int i = 0;
dbc.prepareIteration(rs);
while (dbc.canIterate(rs)) {
      i++;
}
%>

<p>Nombre d'enregistrements : <%= i %></p>
```

Requête dynamique

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

<%
DbConnection dbc = new DbConnection(
      "org.gjt.mm.mysql.Driver",
      "jdbc:mysql://localhost/simpledb");

String query = "SELECT * FROM `TABLE` WHERE `privilege` = '?'";
```

```
PreparedStatement ps = dbc.prepareQuery(query);
ps.setString(1, "admin");

ResultSet rs = dbc.execute(ps);
if (dbc.isEmpty(rs)) {
      throw new Exception("EMPTY TABLE");
}

int i = 0;
dbc.prepareIteration(rs);
while (dbc.canIterate(rs)) {
      i++;
}
%>

<p>Nombre d'enregistrements : <%= i %></p>
```

Annexe 5 : uploads de fichiers

La mise à jour de fichiers sur le serveur à l'aide d'une servlet ou d'une page JSP pose le problème de la réception des données. En effet, les requêtes traditionnelles (GET et POST) ne soumettent que le nom du fichier correspondant à un champ de type « file »…

Présentation du paquetage « util.more.web.form »

La librairie « util », disponible sur le site **http://util.ncottin.net**, propose un paquetage « util.more.web.form ». Ce dernier permet de réaliser (entre autres fonctionnalités) les uploads de fichiers.

Ce paquetage a l'avantage d'effectuer un contrôle sur les données reçues, de telle sorte qu'il est possible de refuser une requête dont le fichier soumis dépasse une taille maximale fixée par exemple. Toute requête dont un paramètre est inconnu peut être également refusée. Enfin, le nombre maximum de valeurs d'un champ multivalué (des cases à cocher par exemple) peut être indiqué.

Pour résumer, ce paquetage est indispensable à toute application Java sur le web.

La présentation des autres fonctionnalités de cette librairie est accessible à l'adresse http://util.ncottin.net.

Utilisation du paquetage « util.more.web.form »

Le paquetage « util.more.web.form » permet de recevoir des données issues de la soumission d'une requête HTTP. En plus de fournir les fonctionnalités de base attendues (réception des valeurs des champs notamment), ce paquetage permet de recevoir des fichiers (upload) et de contrôler les données attendues.

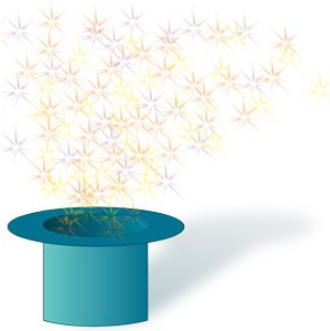
Ce contrôle peut s'effectuer sur les noms de champs attendus (afin que l'on ne soumette pas n'importe quels champs), les valeurs des champs (en cas de choix restreint de valeurs), et leur taille maximum (valable aussi bien pour un champ textuel – il s'agit de la longueur du champ – que pour un fichier – en limitant la taille maximale attendue, exprimée en octets).

Les requêtes reçues ne répondant pas aux contraintes spécifiées par la servlet (ou la page JSP) sont automatiquement rejetées.

Voici un exemple générique de description XML des contraints de validation de formulaires HTTP :

```
<formDesc   method="get|post|multipart|any"
            extraAllowed="true|false"
            tmpDirPath="..."
            bufferSize="...">
      <forms>     /* Restricted to listed forms (if present) */
            <form name="..."
                  method="get|post|multipart|any"
                  extraAllowed="true|false"
                  tmpDirPath="..."
                  bufferSize="..."/>
            ...
      </forms>
      <params>
            <param      name="..."
                        mandatory="true|false"
                        isFile="true|false">
                  <values>  /* If isFile="true" only */
                        <value>...</value>
                        ...
                  </values>
                  <minSize>...</minSize>
                  <maxSize>...</maxSize>
                  <maxOccur>...</maxOccur>
                  <regex>...</regex>
                  <validations>     /* Parameter validation constraints */
                        <validation id="..."/>
                        ...
                  </validations>
            </param>
      </params>
      <validations>     /* Form validation constraints */
            <validation id="..."/>
            ...
      </validations>
</formDesc>
```

Annexe 6 : description de la librairie « util »



Depuis plusieurs années, cette librairie et son API complète, régulièrement mises à jour, sont en libre téléchargement sur le site **http://util.ncottin.net**.

Cette librairie propose à la fois une simplification de certaines classes fournies par le JDK (gestion des fichiers, des chaînes de caractères, des flux XML via DOM, accès aux bases de données, …), une amélioration de certaines fonctionnalités du JDK ainsi que de nouvelle fonctions (découpage de chaînes de caractères avec choix de prise en compte de la casse, traducteurs de code source en HTML, uploads de fichiers sur un serveur web, …).

« util » est employée dans ce manuel pour réaliser une connexion à une base de données et exécuter des requêtes sur cette base d'une part, et pour poster des fichiers sur le serveur Tomcat (upload) d'autre part. Ce ne sont que quelques fonctionnalités parmi l'ensemble mis à disposition.

Veuillez vous référer à l'anne 5 pour quelques précisions concernant les uploads de fichiers faisant appel au paquetage « util.more.web.form ».