Applications J2EE

Sommaire

1. Rappels sur l'environnement d'une application WEB J2EE
Packaging d'une application WEB dans une archive WAR
Déploiement sur un serveur TOMCAT
Rappels sur les Servlets, JSP
Rappels sur les Taglibs : utilisation dans les JSP

1. Environnement d'une application WEB J2EE

Packaging d'une application WEB dans une archive WAR

La packaging c'est la procédure d'inclure l'ensemble des ressources d'une application (HTML/XML, Image, Audio, Vidéo, Feuilles de Styles CSS, JSP, fichier de propriétés lié à l'internationalisation, classes Java, fichiers de configuration, librairies Jars tiers), dans une seule et même *structure* cohérente.

L'avantage d'utiliser des fichier WAR, qui sont des archives simples de types Zip/Jar, c'est qu'en général le Container d'application peut installer l'application sans même relancer le moteur du serveur.

Un des désavantages de cette approche est que toute modification d'une ressource entraîne la re-création d'un fichier WAR que l'on doit réinstaller ensuite sur la partie serveur.

La deuxième stratégie consiste toujours à produire un fichier WAR, mais ce dernier sera extrait pour remettre en place la structure des répertoires au sein de l'environnement serveur. Toute modification ultérieure peut être alors effectuée dans les répertoires concernés sur les ressources à mettre à jour.

En général cette deuxième solution est plus utilisée dans des phases de développement ou de debug, la première solution etant plus orientée « production ».

Etapes de création:

- Créer l'arborescence des répertoires contenant les ressources de l'application.
- Créer un répertoire WEB-INF, à vocation privée, contenant les ressources de haut niveau (librairies dans le sous répertoire *lib*, classes dans le sous répertoire *classes*, fichier de configuration XML).
- Créer le fichier WAR par la commande: `jar cvf myApp.war`, à l'intérieur du répertoire principal.

Remarques :

- L'Export WAR dans Eclipse marche parfaitement, il est à utiliser par défaut…
- On peut placer les pages JSP dans un sous répertoire de WEB-INF, ce qui empêche les clients Web de les accéder directement (voir plus bas model MVC2).
- On peut utiliser l'utilitaire Ant, qui utilise un fichier de configuration XML décrivant les tâches propres à une cible. Le fichier déclare le nom du projet en en-tête, ainsi qu'un certains nombre de tâches à prendre en charge par l'outil (d'autres tâches peuvent être ajoutées par le biais de classes héritant de o*rg.apache.tools.ant.Task*).

  On déclare ensuite les propriétés de :
  - Répertoire (différents path sont utilisés), des classpath requis pour la compilation,
  - La tâche de compilation en soit (qui *dépend* de la précédente tâche),
  - La tâche de production du fichier WAR (qui *dépend* de la tâche de compilation).
  - La tâche de nettoyage des répertoires, une fois le WAR produit.

Déploiement sur un serveur TOMCAT

- Tomcat fourni des outils d'administration pour les Tests et la mise en Production (déploiement, démantèlement, Liste des Applications, démarrage - arrêt, …).

  L'application Manager (path */manager*), utilisé par les utilisateurs de rôle *manager*. (défini dans le doc *$CATALINA_HOME/conf/tomcat-users.xml*)
  [`<user username="tom" password="angel" roles="manager" />`]

- Tomcat fourni une tâche Ant *InstallTask* qui upload et déploie le fichier WAR.

- Un fichier WAR peut être extrait simplement par des commandes :

```
$shell>mkdir myAppDirectory
$shell>cp myApp.war myAppDirectory
$shell>cd myAppDirectory
$shell>jar xvf myApp.war
```

- Le déploiement le plus simple consiste à copier le fichier War dans le répertoire des applications de Tomcat (webapps).

  A la suite de quoi, Tomcat prendra en charge :

  - L'extraction de l'archive.
  - La gestion des informations de déploiement fournie.
  - La copie des fichiers dans le répertoire de l'application. (installation physique de l'ensemble des ressources).

Servlets :

- Composants serveur 100% Java,
- Gestion du cycle de vie (*Init-Loaded*) des processus CGI plus optimisée que par certaines autres technologies (temps de chargement restreint par économies de ressources systèmes sur le chargement de processus).

  Les servlets, une fois chargées par l'*init()*, sont résidentes dans la VM du moteur de servlets. Leur rôle dès lors, consiste à répondre aux requêtes HTTP en associant directement la méthode correspondante (Post, Get, Trace, …).

- Les servlets sont une technologies serveur ayant des spécifications qui lui permet d'être *bindée* (rattachée) à d'autre protocole que HTTP. (cas beaucoup plus rare).

- Intégration d'API existantes :
  Threading, Connexions Réseaux, RMI, Accès File-System, JDBC, JNDI, XML,…

- Classes et Packages de base :

  - javax.servlet.* : Indépendante du protocole, racine *GenericServlet*.
  - javax.servlet.http.* : Spécifiques à http, racine *HttpServlet*.

- Méthodes première du cycle de vie : *Init(ServletConfig config)*, et *destroy()*
- Méthodes générique de requête : *service(ServletRequest req, ServletResponse res)*

- Méthodes propres au mécanismes de réponse HTTP:

doPost :	Réponse a une requête *Post* HTTP.
doGet. :	Réponse a une requête *Get* HTTP.
doPut :	envoi de fichier client vers le serveur.
doDelete :	suppression de document sur le serveur.
doHead :	uniquement les headers de réponses.
doOptions :	permet de connaître les services (doXXX) du serveur.
doTrace :	debug coté client par envoi de trace

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

public class servletTest extends HttpServlet
{
      public void init() {}

      public void doPost(HttpServletRequest req, HttpServletResponse res) {}

      public void doGet(HttpServletRequest req, HttpServletResponse res)
      {
            res.setContentType("text/html");

            try
            {
                  PrintWriter out                    = res.getWriter();

                  out.println("<html><head><title>Test Servlet</title></head>");
                  out.println("<body>Test simple de Servlet</body></html>");

                  out.flush();
                  out.close();
            }
            catch(Exception e) {}
      }
}
```

Fonctionnalités importantes :

- Chaînage de Servlets (redirection de requêtes et contenu par entrées-sorties).
- Gestion des logs sur les requêtes clientes.
- Génération de documents multiples, identifiés par la réponse *mime-typées* (documentaire : XLS, PDF, WML,… ; médias : GIF, WBMP, …, etc…) .

  Le doc généré étant interprété coté client, les spécifications de l'environnement de ce dernier sont à prendre en compte (User-Agent, …).

- Utilisation de *mapping* (alias) servant dans la résolution des requêtes clientes.
- Gestion mono/multi-instances des servlets hébergées
  (classe partagées permettant de mettre en place des pool de ressources).
- Gestion d'une *Map* (key/value) d'attributs pour le *Scope* de la requête (*HttpServletRequest*: *getAttribute(String), setAttribute(String, Object)*).
- Suivi de Session utilisateur
  Le protocole HTTP est *stateless* : entre chaque requête, il ne garde PAS d'état dit *conversationnel* entre ses clients et le serveur. Diverses stratégies ont été mise en œuvre pour recréer ce contexte (champ cachées de formulaire, URL, Cookies), ceci afin de pouvoir identifier les clients s'adressant au serveur.

  Les servlets fournissent un objet spécifiques prôpre à la manipulation des Session clientes : **HttpSession**. Cette classe manipule en interne une table de hashage dans laquel on peut fixer les objets accompagnant la session du client.

  Objet retourné par *HttpRequest.getSession(true)* (appelé en tête de méthode).

  Remarques :
  L'expiration de Session (par l'utilisateur, méthode *invalidate(),* ou bien le serveur) donne lieu à la suppression des données anciennement maintenues.

- Support de Session sécurisées par le standard SSL.
  (dépend de l'implémentation serveur).
- Possibilité d'effectuer des connexions persistantes optimisant les charges serveur sur les traitements de requêtes redondantes :

  (HttpResponse.setContentLength(), puis écriture en sortie de l'ensemble des documents afférents à la page renvoyée).

- Variables d'environnement du protocole HTTP (*HttpRequest.getXXX()*):

Server_Name	Server_Software	Server_Protocol	**Remote_Addr**
Server_Port	**Request_Method**	**Path_Info**	**Content_Type**
Path_Translated	Script_Name	**Document_Root**	**Http_User_Agent**
Query_String	**Remote_Host**	**Auth_Type**	**Remote_User**
Content_Length	Http_Accept	**Http_Referer**	

- Gestion des codes de statuts sur les requêtes http :

Intervalle	Signification de la réponse
100-199	Information
200-299	Requête de client réussie (200 :OK, …)
300-399	Requête de client redirigée, une autre action est nécessaire
400-499	Requête de client incomplète(401 :unauthorized, 404 : not found…)
500-599	Erreur du serveur (503 : service unavailable…)

- En-Têtes http (*HttpServletResponse.setHeader(name, value)):*

En-Tête	Utilisation
Cache-Control	http 1.1, précise les propriétés de cache du document.
Pragma	Equivalent http 1.0 de **Cache-Control**, sans *no-cache*
Connection	*Keep-alive* (connexion persistante), *close*(fermée)
Retry-After	Nombre de secondes avant de re-tenter la requête
Expires	Date de delai maximum avant le changement de la page
Location	Redirection concernant le doc cherché
WWW-Authenticate	Authentification par login-password demandée par le serveur
Content-Encoding	Encodage simple/multiple(';') des données renvoyées.

- Solutions Serveurs :

  - Apache Tomcat
  - WebSphere
  - WebLogic
  - Glassfish
  - …

Evolution de la technologie 2.1 vers 2.2 :

- Le buffering dans les réponses.
- Méthode **IsSecure()** permettant d'indiquer si la requête a été transmise par HTTPS.
- Méthode **getHeaders()** permettant de récupérer l'ensemble des headers.
  Méthode **getAttribute(), getAttributeNames(), setAttribute(), removeAttribute()** dans la classe **HttpSession**.

- Informations de configuration inclues dans la description des Applications Web :

  • Paramètres d'initialisation du ServletContext
  • Configuration des Session
  • Définitions des Servlet & JSP (déclaration, mapping, …)
  • Liste des pages 'Welcome' & Error
  • Sécurité

Exemple de fichier de déploiement XML

```
<?xml version=""1.0" encoding="UTF-8" ?>
<!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>
        <display-name>A Simple Application</display-name>
        <context-param>
                <param-name>Webmaster</param-name>
                <param-value>webmaster@mycorp.com</param-value>
        </context-param>
        <servlet>
                <servlet-name>catalog</servlet-name>
                <servlet-class>com.mycorp.CatalogServlet</servlet-class>
                <init-param>
                        <param-name>catalog</param-name>
                        <param-value>Spring</param-value>
                </init-param>
        </servlet>
        <servlet-mapping>
                <servlet-name>catalog</servlet-name>
                <url-pattern>/catalog/*</url-pattern>
        </servlet-mapping>
        <session-config>
                <session-timeout>30</session-timeout>
        </session-config>
        <mime-mapping>
                <extension>pdf</extension>
                <mime-type>application/pdf</mime-type>
        </mime-mapping>
        <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
                <welcome-file>index.html</welcome-file>
                <welcome-file>index.htm</welcome-file>
        <welcome-file-list>
        <error-page>
                <error-code>404</error-code>
                <location>/404.html</location>
        </error-page>
</web-app>
```

Evolution de la technologie 2.2 vers 2.3 :

- Modèle évènementiel:

- Notification des changements dans *ServletContext* (niveau application) ou *HttpSession.*
Cycle de vie : *ServletContextListener : contextInitialized(ServletContextEvent)*
Attributs de l'objet: *ServletContextAttributesListener*.

HttpSession :
Cycle de vie: *HttpSessionListener*, et attributs de l'objet: *HttpSessionAttributesListener*.

```
<listener>
  <listener-class>myListener</listener-class>
<listener>
```

Déclaration du listener dans web.xml

- Filtering de Servlet : Effectuer un postProcessing de la requête entrante !

* Interface *javax.servlet.Filter.doFilter(ServletRequest, ServletResponse, FilterChain),* appelée pour chaque *mapping* d'URL correspondant au filtre.

* Méthode propre au cycle de vie de l'objet : *init(FilterConfig)* ; *destroy()*

* Accède le servlet context pour y effectuer tout traitement (d'enrichissement, logs, …).

* Intégrer la classe du *filter* dans *WEB-INF/classes* ou sous la forme de jar dans *WEB-INF/lib*.

```
<filter>
  <filter-name>filtre1</filter-name>
  <filter-class>MyFilter</filter-class>
  <init-param>
    <param-name>logFile</param-name>
    <param-name>log.txt</param-name>
  </init-param>
</filter>
```

Déclaration du filtre web.xml

```
<filter-mapping>
  <filter-name>filtre1</filter-name>
  <servlet-name>myFilteredServlet</filter-name>
</filter-mapping>
```

```
<filter-mapping>
  <filter-name>filtre1</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Utilisation du filtre

- ***ServletContext*** : Ajout des methodes ***getServletContextName()***, et ***getResourcePaths()***.
- ***HttpRequest*** : Ajout d'attributs d'erreurs, méthod ***getQueryString(), getParameterMap()***
- ***HttpResponse*** : Ajout: ***resetBuffer()***, correction: ***flush(), setStatus(), sendRedirect/Error()***
- ***HttpSession*** : Ajout de l'interface ***HttpSessionActivationListener***
- Sécurité : Ajout d'attributs pour les connexions SSL, spécification sur les gestions d'erreurs.

- Le Container assure le déploiement des: Params d'initialisation, Session, Servlets, Mappings (Servlet, MIME), Filtres, applications lifecycle, *Welcome/Error* files.

Jsp :

- Pages de script :
  - Séparant le contenu statique HTML, de la logique : lisibilité claire.
  - Interprétées coté serveur par le Runtime du Container.
  - Intégrant des Tags au contenu dynamiquement généré.
  - Les Tags sont scindées en 3 catégories : *Directives, Actions,* et *Scripts*.
  - Les valeurs littérales des attributs de tags peuvent être des tags.
  - Permet de définir des librairies de balises (TagLib).
  - Compilées sous la forme de servlets, (les JSP héritent des Servlets).
  - Proposant un modèle utilisant des données objects de type JavaBeans.
  - Offrant des suivis de Sessions clientes avec gestion d'attributs objets.
  - Totalement intégrées à J2EE comme interface cliente Web légère.

```
<%@ page language="java" contentType="text/html" %>
<html>
 <body bgcolor= "black">
  <jsp:useBean id="myClock" class="java.util.Date" />
  <jsp:getProperty name="myClock" property="date" />
  <% if (myClock.getHours() < 12) { %> Bonjour !
  <% } else if (myClock.getHours() < 17) { %> Bon après-midi !
  <% } else { %> Bonsoir ! <% } %>
  <jsp:useBean id="myFormBean" class="ex.simple.simpleFormBean" >
   <jsp:setProperty name=" myFormBean" property="*" />
  </jsp:useBean>
  Le Bean contient, grâce au setProperty(*) sur les valeurs du formulaire passées, les paramètres suivants :
  <ul>
   <li>Nom : <jsp:getProperty name="myFormBean" property="name"/>
   <li>Mail  : <jsp:getProperty name="myFormBean" property="mail"/>
   …
  </ul>

  On peut aussi accéder un tableau concernant les paramètres de la requête HTTP:
  <%
   String[] tabParam = request.getParameterValues("key");
   if (tabParam != null)
    for (int i = 0; i != tabParam.length; i++)
     out.println("<br>" + tabParam[i]);
  %>
 </body>
</html>
```

Exemple

- Tomcat supporte uniquement le Java, (*JRun* et *Resin* supporte aussi le JavaScript).
- Le Bean ***myFormBean*** affecte automatiquement au sein du Bean tous les champs (**property="*"**) passés par le formulaire référant ayant strictement le même nom. (invocation dynamique par l'utilisation de l'API de réflexivité en Java).
- On a toujours 2 moyens d'invoquer les services des objets utilisés dans les pages:
  - Par un tag get : **<jsp:getProperty name="myClock" property="date" />**
  - Par une expression : **<%= myClock.getDate() %>**

Directives : Ajoute des informations globales (indiffèrement des requêtes), à la page.

- Syntaxe: **<%@ directiveName attr1="value1" attr2="value2" %>**

<%@ page … %>

Inclue des propriétés à l'environnement de la page (type de script, page d'erreur, cache, …).

Attribut	Valeur	Description
autoFlush	true	- Flush la réponse si le buffer alloué est plein. - Si la valeur est *false*, et que le buffer est plein, alors une exception de type IOException est lancée.
buffer	8kb	- Donne la taille du buffer de la page courante. - La valeur doit être exprimée en kiloBytes *kb*, ou bien valoir *none* (ne pas utiliser le buffering)
contentType	text/html	- mime-type de l'objet réponse retourné
errorPage	Aucune	- Page ou URI forwardée si une exception est lancée
extends	Aucune	- Classe Java dont l'implémentation de la JSP hérite. - Cette classe doit implémenter l'interface *JspPage* ou *HttpJspPage* du package *javax.servlet.jsp.package*. - Déconseillé car, empêche les optimisations possibles agencées par le Container, lors de la génération de la page, à l'aide d'une super classe dédiée.
import	Aucune	- Import stricte au sens java de package ou de classe (éventuellement séparé par une virgule dans le tag). - Nécessaire aux objets de scriptlets de la page.
info	Aucune	- Valeur littérale décrivant la page, utilisée par l'interface d'administration du Container.
isErrorPage	false	- Initialise la JSP avec l'exception source produite. - Ne concerne que les pages de traitement d'erreurs.
isThreadSafe	true	- Permet au container de multi-threader la page pour servir des requêtes en parallèle (Recommandé !). - Avoid JSp declaration, ensure object are thread-safe
language	java	- Langage du script utilisé
session	true	- La page est incluse dans la Session du client Web.

<%@ include … %>

Fait référence à un fichier *static* inclus lors de la traduction de la page.

Attribut	Valeur	Description
file	Aucune	- Une page ou une URI relative à inclure.

<%@ taglib … %>

Déclare une librairie de Tags aux actions pré-définies.

Attribut	Valeur	Description
prefix	Aucune	- Nom déterminant les tags spécifiques d'une librairie.
uri	Aucune	- Identifiant couplé au fichier tld de la configuration XML ou bien valeur littérale relative du fichier de la librairie (.jar, .tld).

Scripts :

<% … %>	Déclaration d'un code de type *Scriptlet*, (*if*, *else*, accolades, …). Les variables de scriptlets ne sont **pas** partagées entre les requêtes clientes.
<%= … %>	Expression fournissant une valeur de retour inclue dans le corps de la réponse
<%! … %>	Déclaration de variables d'instances et de méthodes dans l'implémentation physique de la JSP. Le Runtime a recours à deux méthodes semblables au cycle de vie des servlets : *jspInit*() et *jspDestroy*(), que l'on peut redéfinir, et dans lesquels on peut initialiser des objets puis les terminer proprement. Remarque : Les variables déclarées de la sorte sont partagées par toutes les requêtes pointant sur la page (problème potentiel de multi-threading si plusieurs requêtes sont traitées en même temps). Il vaut mieux déclarer les variables avec un tag de scriptlet **<% … %>**, ces variables ne sont pas partagées entre les requêtes. Tag adapté à la déclaration de méthode *stateless* (pas d'argument extérieur) ! (accessibles par des variables déclarées dans des tags de scriptlet **<% … %>**).

Commentaires :

<%-- XXX --%>	Mis en remarque, sur une ou plusieurs lignes.

Caractères d'échappement : pour utiliser les expressions littérales **"**<%**"** ou bien **"**%>**"** on doit insérer un '\' (backslash) avant le dernier caractère : **"**<\%**"** ou bien **"**%\> **"**. De même certaines valeurs d'attributs de tags XML, ' (quote) - **"** (double quote) - '\' (backslash), peuvent être échappées avec la caractère '\'.

Actions :

- Tags XML effectuant des services au sein de la page lors de son interprétation. (Partage de ressources (JDBC, JNDI, …), traitement de données, relais, …)
- Comporte, soit un tag de début - un corps (d'autres tags d'actions) - et un tag de fin, soit un seul tag court <… />.
- Framework permettant l'implémentation d'action dédiées dans des librairies (ici "jsp").

<jsp:useBean>

- Recherche par le Runtime d'un objet bean pour le nom et le scope donné.
- Déclaration de la référence **typée** de la variable de script.

- Si l'objet est trouvé, on l'affecte à la variable, cast si nécessaire selon le type spécifié.

- Si: l'objet n'est pas trouvé, *beanName* et *class* absents: lance un *InstantiationException*.

- Si l'objet n'est pas trouvé, et que l'attribut *class* détermine une classe non-abstraite avec un constructeurs public sans argument, alors un nouvel objet est instancié et associé à la variable de script selon le scope donné (si attribut incorrect alors *InstantiationException*).

- Si l'objet n'a toujours pas été trouvé ou instancié, on tente de le créer grâce à l'attribut *beanName* (s'il existe) avec la méthode *instantiate()* du package *java.beans.Beans*.

- Enfin, si le tag contient un corps d'actions imbriquées, elles sont traitées et disposent de la variable créée pour les actions éventuelles (telles que **<jsp:setProperty>**, par exemple).

Remarques :
- *class* ou *type* doit être spécifié, si les deux le sont une vérification de type est lancée.
- *beanName* doit être combiné à l'attribut *type*, mais il est invalide combiné avec *class*.

Attribut	Dynamique	Description
beanName	yes	- Nom du Bean par la méthode *Beans.instantiate()*. Optionnel
class	non	- Nom de classe servant à instancier l'objet.
Id	non	- Référence du bean utilisé en variable du scope. Obligatoire
scope	non	- "porté" ou "étendue" de vie de la variable (*page* par défaut) (valeurs possibles : *page, request, session, application*)
type	non	- Un type valide pour le Bean (classe, super-classe, interface)

<jsp:getProperty>

- Accède la propriété donnée d'un composant, renvoie le résultat dans la page créée.

Attribut	Dynamique	Description
name	non	- L'identifiant du bean utilisé. Obligatoire
property	non	- Nom de la propriété accédée, utilisation de *getXXX()*.

<jsp:setProperty>

- Affecte une ou plusieurs propriété à un composant bean.

Attribut	Dynamique	Description
name	non	- L'identifiant du bean utilisé. Obligatoire
property	non	- Nom de la propriété, ou '*' pour toute les propriété ayant la même valeur littérale que les paramètres de la requête HTTP. - affectée(s) par *setXXX(..)*. Obligatoire - Tout **Type** objet valide, Primitive et *Array* (+ index) inclus.
param	non	- Nom exact d'un paramètre de la requête HTTP à utiliser. - Si la valeur est omise, les noms de paramètres HTTP et de propriété du bean doivent être égaux (CaseSensitive).
value	**oui**	- Fournie une valeur spécifique pour l'affectation. - Hors formulaire dynamique, **pas d'utilisation combinée** possible de l'attribut *param* ci-dessus. - Conversion, par *valueOf(String)*, vers un type Primitif *(Boolean, Byte, Character, Double, Float, Integer, Long)*.

<jsp:include>

- Flush le buffer courant de la page(sous réserve de l'attribut *flush*="false"; Déconseillé !)
- Intègre le contenu d'un composant (JSP, Servlet, texte statique, …)
- Reprend l'interprétation de la page courante.
- La cible dispose de tous les paramètres et headers originelles de la requête, elle peut en rajouter grâce aux tag **<jsp:param>** imbriqués dans le tag **<jsp:include>**.

Attribut	Dynamique	Description
page	**oui**	- Une page ou URI relative à inclure. Obligatoire
flush	non	- Flush la sortie standard de la requête avant la fin. (La seule valeur autorisée officiellement est *true* !)

<jsp:forward>

- Termine le cycle de vie de la page courante et, redirige le processus de contrôle de la requête vers un composant Web spécifié en attribut.
- Si un début de réponse a déjà été renvoyé au client le forward échoue ! il lance alors une exception de type *IllegalStateException*.
- La cible dispose de tous les paramètres et headers originelles de la requête, elle peut en rajouter grâce aux tag **<jsp:param>** imbriqués dans le tag **<jsp:forward>**.

Attribut	Dynamique	Description
page	**oui**	- Une page ou URI relative à inclure. Obligatoire

<jsp:param>

- Ajoute un paramètre lors de redirection **<jsp:include>** ou **<jsp:forward>**, ou bien dans le cadre d'un tag **<jsp:params>** complétant les paramètres d'un tag HTML <Applet>.

Attribut	Dynamique	Description
name	non	- Nom du paramètre. Obligatoire
value	**oui**	- Valeur du paramètre. Obligatoire

<jsp:params>

- Ne s'utilise qu'à l'intérieur d'un tag **<jsp:plugin>**, (ajout de paramètres à un tag <Applet>

<jsp:plugin>

- Inclus un tag <Object> ou <Embed> (selon navigateur) d'Applet destiné au java plug-in.
- Peut imbriquer les tags: **<jsp:params>** pour l'ajout de paramètre HTML, ou bien **<jsp:fallback>** message **<jsp:fallback>** pour les navigateurs ne supportant pas les applets.

Attribut	Dynamique	Description
archive	non	- Charge des librairie selon le *codebase* spécifié
code	non	- Le nom de la classe chargé de l'*init*. Obligatoire
codebase	non	- URL du répertoire contenant les classes de l'Applet
height	non	- La hauteur, en pixels ou %, de la zone de l'Applet.
hspace	non	- L'espace inséré à gauche et à droite de la zone.
iepluginurl	non	- L'URL de référencement du java plug-in pour IE.
jreversion	non	- La version du Runtime requis pour l'interpretation.
name	non	- Le nom de l'applet (communication inter-applet).
nspluginurl	non	- L'URL de référencement du java plug-in pour Netscape.
title	non	- Info-bulle de l'Applet
type	non	- Type d'objet embarquée : *applet* ou **un *bean ?***
vspace	non	- L'espace inséré en haut et en bas de la zone.
width	non	- La largeur, en pixels ou %, de la zone de l'Applet.

Variables implicites à l'environnement JSP et HTTP:

Nom	Type	Informations
application	*Javax.servlet. ServletContext*	- Accès aux ressources partagées (JDBC, JNDI, logs, …). - Gestion des attributs ayant un 'scope' de valeur "application". - Information sur le Container (Info, Version, Contexte, …). - Objet 'singleton' partagé dans la VM. - Méthodes usuelles de communication avec le Container : *getAttribute(String), getAttributeNames()* *getMimeType(String), getResource(String)* *log(String), log(String, Throwable)* *removeAttribute(String), setAttribute(String, Object)*
config	*Javax.servlet. ServletConfig*	- Utile au Container pour initialiser les JSP et Servlets avec des informations issues du descripteur de déploiement XML. - Méthodes usuelles : *getInitParameter(String name), getInitParameterNames()* *getServletContext(), getServletName()*
exception	*Java.lang.Throwable*	- Uniquement les pages d'erreurs à l'attribut *isErrorPage=true*.
out	*Javax.servlet.jsp. JspWriter*	- Descripteur de sortie standard de la réponse renvoyée. - Méthodes usuelles: *clear(), clearBuffer(), print(), println(), close(), flush(), getBufferSize(), getRemaining(), isAutoFlush()*.
Page	*Java.lang.Object*	- Variable très peu utilisée, elle symbolise la page JSP produite.
pageContext	*Javax.servlet.jsp. PageContext*	- Permet d'accéder aux attributs de *Scope* et de la *page*. - Associé à chaque requête/réponse. - Méthodes usuelles: *findAttribute(String), forward(String),* *getAttribute(String), getAttribute(String, int),* *getAttributeNamesInScope(int), getAttributesScope(String),* *getException(), getRequest(), getResponse(), getServletConfig(),* *getServletContext(), getSession(),* *include(String),* *removeAttribute(String), removeAttribute(String, int),* *setAttribute(String, Object), setAttribute(String, Object, int)* - Variables usuelles: public static final int PAGE_SCOPE = 1; public static final int REQUEST_SCOPE = 2; public static final int SESSION_SCOPE = 3; public static final int APPLICATION_SCOPE = 4;

request	*Javax.servlet.http. HttpServletRequest*	- Infos sur la requête courante (paramètres, attributs, en-têtes et variables d'environnement http, cookies). - Méthodes usuelles (hors variables http, accessible par *getVar()*): *getAttribute(String), getAttributeNames(), getAuthType(), getCharacterEncoding(), getContentType(), getContextPath(), getCookies(), getHeader(String), getHeaderNames(), getHeaders(String), getLocale(), getLocales(), getParameter(Str), getParameterNames(), getParameterValues(), getSession(),getSession(boolean), isSecure(), isUserInRole(String), removeAttribute(String), setAttribute(String, Object),* Ex : (cf Servlets: variables d'env http) request.getHeader("User-Agent") request.getMethod(), request.getRemoteAddr()
response	*Javax.servlet.http. HttpServletResponse*	- Permet de définir la réponse courante (en-tête, code d'états, ajout de cookie, suivi de session,) - Méthodes usuelles: *addCookie(Cookie), addDateHeader(String, long), addHeader(String, String), addHeader(String, int), encodeRedirectUrl(String), encodeUrl(String), flushBuffer(), getBufferSize(), getCharacterEncoding(), getLocale(), getWriter(), isCommited(), reset(), sendError(int), sendError(int, String), sendRedirect(String), setBufferSize(int), setContentLength(int), setContentType(String), setDateHeader(String, long), setHeader(String, [String; int]), setLocale(Locale), setStatus(int)*
session	*Javax.servlet.http. HttpSession*	- Permet d'accéder les données associées à la session (scope=session), sauf si l'attribut *session* de la page vaut *false*. - Méthodes usuelles: *getAttribute(String), getAttributeNames(), getCreationTime(), getId(), getLastAccessedTime(), getMaxInactiveInterval(), invalidate(), isNew(), removeAttribute(String), setAttribute(String, Object), setMaxInactiveInterval(int)*

Les Taglibs : utilisation dans les JSP

- Actions de traitement spécifiques de données.
- Accès complet à l'environnement intrinsèque JSP: requête, réponse, variable de scope.
- Spécifié depuis la version 1.1 des JSP (« *mécanisme d'extension de balises* »).
- Structurées sous la forme de JavaBeans (get/set + propriété).
- Groupées dans des bibliothèque ayant un nom de package.
- Préfixes de TagLib interdits : *jsp, jspx, java, javax, servlet, sun* et *sunw*.

Ex :

```
<%@ taglib uri="/WEB-INF/lib/mytaglib.jar" prefix="pref" %>
<%@ taglib uri="/WEB-INF/lib/mytaglib.tld" prefix="pref" %>
```

Développer des Tags :

- Un tag spécifique repose sur une classe appelée « *gestionnaire de balise* » qui est un bean possédant un *Accessor* « *set* » pour chacun des attributs de la balise. Cette classe doit, en outre, implémenter une des deux interfaces suivantes : *Tag* et *BodyTag* (ces interfaces font parties du package *javax.servlet.jsp.tagext*).

  Tag: définit les méthodes à implémenter pour toute tag d'action sans corps:
  BodyTag: hérite de *Tag*, ajoute des méthodes propres au contenu imbriqué d'un tag.

  L'API Java fournie deux classes standards, *TagSupport* et *BodyTagSupport*, implémentant par défaut les 2 interfaces *Tag* et *BodyTag*. Il est conseillé de les utiliser.

TagSupport

public void setPageContext(PageContext pageContext); Affectation du Contexte de Page qui permet d'accéder à de nombreuses propriétés (cf « variables implicites à l'environnement JSP et HTTP).

public int doStartTag() *throws JspException*; Notification de début de Tag (parser SAX), permet d'implémenter des vérifications sur les attributs, et un éventuel traitement de corps de Tag : cette méthode doit renvoyer la constante *SKIP_BODY* si le tag ne doit pas traîter de corps imbriqué, ou bien *EVAL_BODY_INCLUDE* dans le cas contraire (le Conteneur de JSP traite tout de même le contenu du corps, mais de façon moins pratique qu'avec *BodyTag*).

public int doEndTag() *throws JspException*; Méthode ré-écrite la plus couramment, renvoie soit *EVAL_PAGE* signifiant au Conteneur de poursuivre le traitement de la page, ou bien *SKIP_PAGE* pour l'arrêter (utile pour les balises effectuant un *forward* de la page par exemple).

BodyTagSupport

public int doStartTag() *throws JspException*; Renvoie par défaut *EVAL_BODY_INCLUDE* : le Conteneur doit traiter le corps de Tag, et mettre à disposition le résultat au gestionnaire de balise.

Exemple de gestionnaire de Balise simple, héritant de TagSupport :

```
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HelloTag extends TagSupport
{
  private String desc = “world”;

  public void setDesc(String newDesc) {desc = newDesc;}

  public int doEndTag()
  {
    try {pageContext.getOut().println(“Hello ” + desc);}
    catch (Exception e) { System.out.println(“Tag Hello: ” + e.toString());}

    return EVAL_PAGE;
  }
}
```

Déployer des Tags :

- Les TagLibs doivent contenir un fichier XML appelé TLD : Tag Library Descriptor.
  (Système d’*alias* entre les tags d’actions spécifiques et les classes de logique des balises).

- Il doit décrire aussi de façon exhaustive l’ensemble des attributs de chaque tag.

- Il est conseillé de packager l’ensemble des classes et le fichier TLD dans un fichier jar.

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

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>test</shortname>

  <tag>
    <name>hello</name>
    <tagclass>com.test.HelloTag</tagclass>
    <bodyContent>empty</bodyContent>
    <attribute>
      <name>desc</name>
    </attribute>
  </tag>
</taglib>
```

Exemple de fichier XML TLD de configuration (*myLib.tld* placé dans */WEB-INF/tlds/*)

```
  <taglib>
    <taglib-uri>/WEB-INF/tlds/myTag.tld</taglib-uri>
    <taglib-location>/WEB-INF/tlds/myTag.tld</taglib-location>
  </taglib>
```

Référencement de la librairie de tag dans la configuration web.xml du Container

```
<%@ taglib uri="/WEB-INF/tlds/myLib.tld" prefix="test" %>
<html>
  <body bgcolor="white">
    <test:hello desc="world" />
  </body>
</html>
```

Utilisation des Tags dans une page JSP

Librairie de Tag JSTL

- Spécification et implémentation de références fournie par Sun.
- Désir d'unifier la logique de développement des tags, qui ont tendance à proliférer.
- Met en place des librairies de tags : Core, Format, SQL, XML.
- Défini un Langage d'Expression (EL) fournissant des calculs standards.
- Enrichi le modèle manipulé par des *Collections* de type *Map*, des informations Web.
- Sert de base au nouvelles spécifications JSF (Java Server Faces).
- Tomcat met à disposition ces fonctionnalités.

JSP 2.0 Expression Language - Basic Arithmetic

Error, like div by zero, are handled gracefully.

EL Expression	Result
\${1}	1
\${1 + 2}	3
\${1.2 + 2.3}	3.5
\${1.2E4 + 1.4}	12001.4
\${-4 – 2}	-6
\${21 * 2}	42
\${3/4}	0.75
\${3 div 4}	0.75
\${3/0}	Infinity
\${10%4}	2
\${10 mod 4}	2
\${(1==2) ? 3 : 4}	4

JSP 2.0 Expression Language - Basic Comparisons

The following comparison operators are supported:

- Less-than (< or lt)
- Greater-than (> or gt)
- Less-than-or-equal (<= or le)
- Greater-than-or-equal (>= or ge)
- Equal (== or eq)
- Not Equal (!= or ne)

Numeric

EL Expression	Result
\${1 < 2}	true
\${1 lt 2}	true
\${1 > (4/2)}	false
\${1 > (4/2)}	false
\${4.0 >= 3}	true
\${4.0 ge 3}	true
\${4 <= 3}	false
\${4 le 3}	false
\${100.0 == 100}	true
\${100.0 eq 100}	true
\${(10*10) != 100}	false
\${(10*10) ne 100}	false

Alphabetic

EL Expression	Result
\${'a' < 'b'}	true
\${'hip' > 'hit'}	false
\${'4' > 3}	true

Extrait du serveur Tomcat

Objets implicites (déjà exposé, cf JSP)

- 11 *Map* structurent l'accès au informations et Meta-Informations de la page.
- Certaines d'entre elles permettent d'accéder les structures de type tableaux.
- L'exemple présente l'utilisation de ces *Map* par le biais de l'EL.
  On dispose de 2 grammaires différentes pour y accéder : '.' ou bien, l'opérateur [« »]

JSP 2.0 Expression Language - Implicit Objects

The following implicit objects are available (not all illustrated here):

- pageContext - the PageContext object
- pageScope - a Map that maps page-scoped attribute names to their values
- requestScope - a Map that maps request-scoped attribute names to their values
- sessionScope - a Map that maps session-scoped attribute names to their values
- applicationScope - a Map that maps application-scoped attribute names to their values
- param - a Map that maps parameter names to a single String parameter value
- paramValues - a Map that maps parameter names to a String[] of all values for that parameter
- header - a Map that maps header names to a single String header value
- headerValues - a Map that maps header names to a String[] of all values for that header
- initParam - a Map that maps context initialization parameter names to their String parameter value
- cookie - a Map that maps cookie names to a single Cookie object.

Change Parameter

foo = bar Soumettre la requête

EL Expression	Result
\${param.foo}	bar
\${param["foo"]}	bar
\${header["host"]}	localhost:8080
\${header["accept"]}	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, */*
\${header["user-agent"]}	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)

Extrait du serveur Tomcat

JSP 2.0 Expression Language - Functions

An upgrade from the JSTL expression language, the JSP 2.0 EL also allows for simple function invocation. Functions are defined by tag libraries and are implemented by a Java programmer as static methods.

Change Parameter

foo = JSP 2.0 Soumettre la requête

EL Expression	Result
\${param["foo"]}	JSP 2.0
\${my:reverse(param["foo"])}	0.2 PSJ
\${my:reverse(my:reverse(param["foo"]))}	JSP 2.0
\${my:countVowels(param["foo"])}	0

Extrait du serveur Tomcat

```
package jsp2.examples.el;

import java.util.*;

/**
 * Defines the functions for the jsp2 example tag library.
 *
 * <p>Each function is defined as a static method.</p>
 */
public class Functions {
    public static String reverse( String text ) {
        return new StringBuffer( text ).reverse().toString();
    }

    public static int numVowels( String text ) {
        String vowels = "aeiouAEIOU";
        int result = 0;
        for( int i = 0; i < text.length(); i++ ) {
            if( vowels.indexOf( text.charAt( i ) ) != -1 ) {
                result++;
            }
        }
        return result;
    }

    public static String caps( String text ) {
        return text.toUpperCase();
    }
}
```

Package CORE

- Ecrire sur la sortie standard de la requête par le descripteur *out* de type JspWriter.
- Gérer le cycle de vie des variables liées à un Scope (Création, Destruction).
- Utiliser les propriétés de composants JavaBeans et gérer les collections d'objets.
- Se servir de structure de type *Map* (*HashTable*, *HashMap*, …).
- Intégrer des opérateurs conditionnels (*if-then*, et *switch-case-default*).
- Gérer les exceptions générées au sein des pages interprétées.
- Implémenter le design pattern *Iterator* pour des structures, des int, ou bien des *String*.
- Manipuler des URL (ajouter des paramêtres, …)
- Importer des ressources
- Rediriger des réponse.

Action	Description
<c:catch>	Catch une exception lancée dans le corps de l'action
<c:choose>	Equivalent d'un opérateur *switch*.
<c:forEach>	Design Pattern *Iterator*, agit sur une *Collection* d'objet, ou un nombre de fois.
<c:forTokens>	Equivalent du *StringTokenizer*, itère les *tokens* d'une String.
<c:if>	Opérateur conditionnel *if*.
<c:import>	Import d'une URL dans l'interprétation de la page.
<c:otherwhise>	Choix par défaut d'un opérateur **<c:choose>** (*switch*)
<c:out>	Ecrit sur la redirection standard de sortie (*JspWriter* out).
<c:param>	Définit un paramètre pour une redirection **<c:import>** ou **<c:url>**.
<c:redirect>	Redirige la réponse de la requête vers une URL donnée par **<c:param>**.
<c:remove>	Suppression d'une variable pour un scope donné.
<c:set>	Déclaration d'une variable pour un scope donné.
<c:url>	Permet de créer et de définir une URL.
<c:when>	Agit comme opérateur *case* pour l'opérateur **<c:choose>** (*switch*).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Set Attributes for HTML Elements</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <%-- Because the following form has no action, this JSP
           page will be reloaded when the form is submitted --%>
      <form>
         <table>
            <tr>
               <td>Enter Text for the Submit button:</td>
               <td><input name='buttonText' type='text'></td>
            </tr>
         </table>

         <p><input type='submit'
                   value='<c:out value="${param.buttonText}" default="Submit"/>'/>
      </form>
   </body>
</html>
```

Formulaire Simple

```
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

<%-- Create a color preferences bean and store it in page scope --%>
<jsp:useBean id='bean' class='beans.ColorPreferences'/>

<%-- If no bgcolor or fgcolor request parameters (first loaded), set to null --%>
<c:set target='${bean}' property='background' value='${param.bgcolor}'/>

<%-- idem for foreground with value inside the action's body, it can be done ! --%>
<c:set target='${bean}' property='foreground'><c:out value='${param.fgcolor}'/></c:set>

<table align='center'>
   <tr>
      <td colspan='2' align='center'></td>
         <%-- Create a table with the bean's background property color --%>
        <table border='3' align='center' bgcolor='<c:out value="${bean.background}"/>'>
            <tr>
               <td>
               <%-- Create table data with the bean's foreground property color --%>
               <font size='7' color='<c:out value="${bean.foreground}"/>'>JSTL</font>
               </td>
            </tr>
         </table><p>
      </td>
   </tr>
   <tr>
      <%-- The following form has no action: submit -> JSP page reload --%>
      <form>
         <table align='center'>
            <tr>
               <td>Background Color:</td>
               <td>
                  <%-- Create the HTML select element for background color --%>
                  <select name='bgcolor'>
                     <option value='white'>white</option>
                     <option value='black'>black</option>
                  </select>
               </td>
            </tr>
            <tr>
               <td>Foreground Color:</td>
               <td>
                  <%-- Create the HTML select element for foreground color --%>
                  <select name='fgcolor'>
                     <option value='white'>white</option>
                     <option value='black'>black</option>
                  </select>
               </td>
            </tr>
            <tr>
               <table align='center'>
                  <tr><td><input type='submit'></td></tr>
               </table>
            </tr>
         </table>
      </form>
   </tr>
</table>
```

Exemple de tableau dynamique

Package FORMAT

- Renseigner des *ResourceBundle* utilisé pour les clés de messages recherchés.
- Renseigner une variable *Locale* gérant les règles de formatage et de parsage.
- Atteindre les messages cherchés par clés.
- Formater et parser des nombres, des numéraires, des pourcentages, des dates.
- Gérer l'encodage d'une requête.

Action	Description
<fmt:bundle>	Fixe un contexte de localisation (**<fmt:message>** imbriqués ou bien formatages)
<fmt:setBundle>	Fixe un contexte de localisation (**<fmt:message>** ou bien formatages)…….
<fmt:setLocale>	Fixe une variable *Locale* (utilisé par **<fmt:message>** ou bien des formatages).
<fmt:formatDate>	Formate une *Date* selon la *Locale* renseignée.
<fmt:formatNumber>	Formate un nombre, un numéraire, un pourcentage selon la *Locale* renseignée.
<fmt:message>	Utilise un message *String* d'un *ResourceBundle*.
<fmt:param>	Fourni un paramètre pour une action **<fmt:message>**
<fmt:parseDate>	Parse une *Date* selon une *Locale* donnée
<fmt:parseNumber>	Parse un nombre, un numéraire, un pourcentage selon la *Locale* renseignée.
<fmt:requestEncoding>	Fixe l'encodage de la requête pour une page JSP
<fmt:setTimeZone>	Fixe le fuseau horaire utilisé par les formatages de date et de temps
<fmt:timeZone>	Fixe le fuseau horaire utilisé par les formatages imbriqués de date et de temps

```
<%@ page contentType='text/html; charset=UTF-8' %>
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
<%@ taglib uri='http://java.sun.com/jstl/fmt'  prefix='fmt'%>
<c:set var='number' value='456789.1234'/>
<font size='5'><c:out value='Formatting this number: ${number}'/></font>
<p><table border='1'>
   <tr>
      <th><font size='5'>Country</font></th>
      <th><font size='5'>Formatted Currency</font></th>
   </tr>
   <tr><td><font size='5'>France (EURO)</font></td>
      <fmt:setLocale value='fr-FR'/>
      <td><font size='5'>
         <fmt:formatNumber value='${number}' type='currency' currencyCode='EUR'/></font>
      </td>
   </tr>
   <tr><td><font size='5'>France (French Franc)</font></td>
       <fmt:setLocale value='fr-FR'/>
       <td><font size='5'>
           <fmt:formatNumber value='${number}' type='currency'currencyCode='FRF'/></font>
       </td>
   </tr>
   <tr><td><font size='5'>Germany (Deutsche Mark)</font></td>
       <fmt:setLocale value='de-DE'/>
       <td><font size='5'>
           <fmt:formatNumber value='${number}' type='currency'currencyCode='DEM'/></font>
       </td>
   </tr>
   <tr><td><font size='5'>United States</font></td>
       <fmt:setLocale value='en-US'/>
       <td><font size='5'>
           <fmt:formatNumber value='${number}' type='currency' currencyCode='USD'/></font>
       </td>
   </tr>
</table>
```

format-currencies\index.jsp

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

   <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
   <%@ taglib uri='http://java.sun.com/jstl/fmt'  prefix='fmt'%>
   <%@ taglib uri='http://java.sun.com/jstl/core_rt' prefix='c_rt' %>

   <%-- Create a scoped variable based on the locale request parameter --%>
   <c:choose>
      <c:when test='${empty param.locale}'>
         <c_rt:set var='locale' value='<%= java.util.Locale.getDefault() %>'/>
      </c:when>
      <c:otherwise><c:set var='locale' value='${param.locale}'/></c:otherwise>
   </c:choose>

   <%-- Set the locale according to the locale request parameter --%>
   <fmt:setLocale value='${locale}'/>

   <%-- Create a scoped variable that contains the current date --%>
   <jsp:useBean id='now' class='java.util.Date'/>

   <%-- Show formatted number --%>
   <font size='5'>Formatted date:
      <c:choose>
         <c:when test='${not empty param.pattern}'>
            <fmt:formatDate value='${now}' pattern='${param.pattern}'/>
         </c:when>
         <c:otherwise><fmt:formatDate value='${now}'/></c:otherwise>
      </c:choose>
   </font>

   <%-- Current form does not specify an action-> JSP page will be reloaded on submit --%>
   <form>
      <table>
         <tr>
            <td>Format this date:</td>
            <td><c:out value='${now}'/></td>
         </tr>
         <tr>
            <td>With this locale:</td>
            <td>
            <select name='locale'>
               <c_rt:forEach var='thisLocale'
                     items='<%= java.text.DateFormat.getAvailableLocales()%>'>
                  <option
                  <c:if test='${locale == thisLocale}'>selected</c:if>
                  <c:out value='>${thisLocale}' escapeXml='false'/>
                  </option>
               </c_rt:forEach>
            </select>
            </td>
         </tr>
         <tr>
            <td>With this pattern:</td>
            <td><input type='text' name='pattern' size='50'
                    value='<c:out value="${param.pattern}"/>'/>
            </td>
         </tr>
      </table>
      <p><input type='submit'/>
   </form>
</body>
```

format-dates

Package SQL

- Renseigner une *DataSource*.
- Exécuter des requêtes vers une base et utiliser les résultats.
- Effectuer des requêtes de type *Update*.
- Gérer les propriétés de Transaction.
- Préparer des requêtes précompilés (*PreparedStatement*).

Action	Description
<sql:dateParam>	Renseigne un paramètre de Date pour une **<sql:query>** ou un **<sql:update>**.
<sql:param>	Renseigne un paramètre simple pour une **<sql:query>** ou un **<sql:update>**.
<sql:query>	Exécute une requête *Query* vers une Base.
<sql:setDataSource>	Fixe une DataSource (actions: **<sql:query>**, **<sql:update>**, **<sql:transaction>**).
<sql:transaction>	Englobe une transaction avec des **<sql:query>** ou **<sql:update>** imbriqués.
<sql:update>	Exécute une requête *Update* vers une Base.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Accessing Database Queries</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
      <%@ taglib uri='http://java.sun.com/jstl/sql'  prefix='sql'%>

      <sql:query var='customers'>
         SELECT * FROM CUSTOMERS
      </sql:query>

      <%-- Access the rowCount property of the query --%>
      <p>There are <c:out value='${customers.rowCount}'/> rows
       in the customer query. Here they are:</p>

      <%-- Create a table with column names and row data --%>
      <p><table border='1'>
         <tr>
            <c:forEach var='columnName'
                     items='${customers.columnNames}'>
               <th><c:out value='${columnName}'/></th>
            </c:forEach>
         </tr>

         <c:forEach var='row' items='${customers.rowsByIndex}'>
            <tr>
               <c:forEach var='rowData' items='${row}'>
                     <td><c:out value='${rowData}'/></td>
               </c:forEach>
            </tr>
         </c:forEach>
      </table>
   </body>
</html>
```

Accessing-queries\indes.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Executing Database Transactions</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c'  %>
      <%@ taglib uri='http://java.sun.com/jstl/sql' prefix='sql'%>
      <%@ taglib uri='http://java.sun.com/jstl/fmt' prefix='fmt'%>

      <sql:query var='fromBalance'>
         SELECT BALANCE FROM ACCOUNTS WHERE CUST_ID = ?
         <sql:param value='${param.fromCustomer}'/>
      </sql:query>

      <c:set var='fromBalance'
           value='${fromBalance.rows[0].balance}'/>

      <sql:query var='toBalance'>
         SELECT BALANCE FROM ACCOUNTS WHERE CUST_ID = ?
         <sql:param value='${param.toCustomer}'/>
      </sql:query>

      <c:set var='toBalance'
           value='${toBalance.rows[0].balance}'/>

      <c:catch var='transactionException'>
         <sql:transaction>
            <sql:update>
               UPDATE ACCOUNTS SET BALANCE = ? WHERE CUST_ID = ?
                <sql:param value='${fromBalance - param.amount}'/>
                <sql:param value='${param.fromCustomer}'/>
            </sql:update>

            <sql:update>
               UPDATE BADTABLENAME
                              SET BALANCE = ? WHERE CUST_ID = ?
               <sql:param value='${toBalance + param.amount}'/>
               <sql:param value='${param.toCustomer}'/>
            </sql:update>
         </sql:transaction>
      </c:catch>

      <c:if test='${transactionException != null}'>
         <font size='4' color='red'>
            Transaction Failed! Please make sure that account
            you are withdrawing from has sufficient funds
            for the withdrawl.
         </font>
      </c:if>

      <c:import url='index.jsp'/>
   </body>
</html>
```

Transactions\transfer_funds.jsp

Package XML

- Parser un document XML
- Transformer un document par XSLT
- Fixer un identifiant système pour résoudre les entités externes;
- appliquer des filtres SAX à un document XML; ….

Action	Description
<x:choose>	Version XML de **<c:choose>**.
<x:forEach>	Version XML de **<c:forEach>**.
<x:if>	Version XML de **<c:if>**.
<x:otherwhise>	Version XML de **<c:otherwise>**.
<x:out>	Version XML de **<c:out>**.
<x:param>	Version XML de **<c:param>**, ajoute un paramètre à une action **<x:transform>**.
<x:parse>	Parse un document XML.
<x:set>	Version XML de **<c:set>**.
<x:transform>	Transforme un document XML
<x:when>	Version XML de **<c:when>**.

```
<body>
   <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
   <%@ taglib uri='http://java.sun.com/jstl/xml'  prefix='x' %>

   <c:import var='rolodex_xml' url='rolodex.xml'/>  <%-- Import the XML file --%>

   <x:parse var='document' xml='${rolodex_xml}'/>   <%-- Parse the XML file --%>

   <p>There are <x:out select='count($document//contact)'/> contacts in the rolodex.<p>

   <x:out select='count($document//contact/phone[@type="work"])'/> have a work phone and
   <x:out select='count($document//contact/phone[@type="home"])'/> have a home phone.<p>

   <%-- For each contact in the rolodex... --%>
   <x:forEach select='$document//contact'>
      <table>
         <tr>
            <td>First Name:</td>
            <td><x:out select='firstName'/></td>
         </tr>
         <tr>
            <td>Last Name:</td>
            <td><x:out select='lastName'/></td>
         </tr>
         <tr>
            <td>Work Phone:</td>
            <td><x:out select='phone[@type="work"]'/></td>
         </tr>

         <%-- Home phone is optional, check to see if it exists before processing it --%>
         <x:if select='phone[@type="home"]'>
            <tr>
               <td>Home Phone:</td>
               <td><x:out select='phone[@type="home"]'/></td>
            </tr>
         </x:if>
      </table><p>
   </x:forEach>
</body>
```

Simple Parsing

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rolodex>
   <contact>
      <firstName>Anna</firstName>
      <lastName>Keeney</lastName>
      <company>BSC, Inc.</company>
      <phone type="work">716-873-9644</phone>
      <phone type="home">716-834-8772</phone>
   </contact>
   <contact>
      <firstName>Lynn</firstName>
      <lastName>Seckinger</lastName>
      <company>Sabreware, Inc.</company>
      <phone type="work">716-219-2012</phone>
   </contact>
</rolodex>
```

Fichier de contenu XML

```
   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
      <%@ taglib uri='http://java.sun.com/jstl/xml'  prefix='x' %>

      <%-- Import the XML file and XSLT stylesheet --%>
      <c:import var='rolodex_xml' url='rolodex.xml'/>
      <c:import var='rolodex_xsl' url='rolodex.xsl'/>

      <%-- Perform the transformation --%>
      <x:transform xml='${rolodex_xml}' xslt='${rolodex_xsl}'/>
   </body>
```

Simple transform

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <!-- Generate HTML for the document root -->
   <xsl:template match='/'>
      <table border='1'>
         <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Company</th>
            <th>Email</th>
            <th>Work Phone</th>
            <th>Home Phone</th>
         </tr>
         <xsl:apply-templates/>
      </table>
   </xsl:template>

   <!-- Create a table row for each item and apply templates -->
   <xsl:template match='contact'>
      <tr><xsl:apply-templates/></tr>
   </xsl:template>

   <!-- Create table data for each item and apply templates -->
   <xsl:template match='contact/*'>
      <td><xsl:apply-templates/></td>
   </xsl:template>
</xsl:stylesheet>
```

Fichier de transformation XSL

Architecture de communication entre les composants : chaînages et redirections

- **Chaînage (*Forward*)**

  Renvoie de la requête vers une autre ressource serveur, sans en avertir le client : la barre d'adresse du navigateur indique toujours la même ressource.

  Permet de distribuer à plusieurs composant la requête avant que la réponse ne soit générée (orientée Design Pattern *Chain of Responsability*)

  Les objets ayant enrichi le scope de la requête sont envoyés à la nouvelle ressource.

  Plus efficaces qu'une redirection.

  Fonctionne si rien n'a été renvoyé par la sortie standard (sinon *IllegalStateException*)

```
HttpServletRequest.setAttribute(String, Object);
HttpServletRequest.getRequestDispatcher(strRedirect).forward(Request, Response)
```

- **Redirection**

  La réponse renvoyée au client Web lui indique qu'il doit charger une autre URI.

  La requête originale, et son scope, termine son cycle de vie.

  Plus lent qu'un *Forward.* (nécessite une nouvelle requête).

```
HttpServletResponse.sendRedirect(HttpServletResponse.encodeRedirectURL(String))
```

  La méthode *encodeRedirectURL* permet de conserver les ID de Session.

Internationalisation des applications

- Tenir compte des langues et habitudes des clients selon leur localisation géographique.
- Impact sur les données bruts : (textes, graphiques, …).
- les formats de données : (formats de Dates, d'heures, de nombres…).
- Internationalisation : modèle de gestion de ressources dynamique pour chaque région.
- Localisation : choix des ressources à utiliser selon la provenance une variable *locale*.
- Abréviations usuelles: I18N (Internationalisation), L10N (Localisation).

API :

- *java.util.Locale* symbolise une région géographique (code langue + code pays).
- Exemple : java.util.Locale usLocale = new Locale("en", "US");

code langue ISO 639	Langue		code pays ISO 3166	Pays
af	Africain		DK	Danemark
da	Danois		DE	Allemagne
de	Allemand		FR	France
el	Grec		GR	Grèce
en	Anglais		MX	Mexique
es	Espagnol		NZ	Nouvelle Zélande
fr	Français		ZA	Afrique du Sud
ja	Japonais		GB	Royaume Uni
pl	Polonais		US	Etats-Unis
ru	Russe			
sv	Suédois			
zh	Chinois			

Cf :
http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt
http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

- Des classes de formatage :
  - java.text.NumberFormat: Nombres (séparateur de virgules, de milliers…)
  - java.text.DateFormat: Dates (ordre des éléments, …).

- Ces ressources sont couplées généralement à une instance de la classe PropertyResourceBundle, héritant de la classe java.util.ResourceBundle, qui permet de charger depuis des fichiers textes (nommées avec les valeurs de Langue et de Pays) des lignes de propriétés structurées sous la forme clé=valeur.

Exemple de contenu du fichier exBundle_fr_FR.properties:

```
Msg_welcome=Bienvenue sur le site
Company_logo=/images/logo.gif
```

L'accès à une clé est le suivant:

```
java.util.Locale locale             =  HttpRequest.getLocale() ;
java.util.ResourceBundle bundle = java.util.ResourceBundle.getBundle("exBundle", locale);
String msg                          = bundle.getString("Msg_welcome");
```

Remarque:

- Le navigateur Web envoie un en-tête « Accept-Language » dans la requête http qui suit cette nomenclature (il peut utiliser un ou plusieurs code, séparés par des virgules, selon les préférences de la configuration du browser). Le container de servlet permet ensuite de récupérer ces informations par les méthodes :

```
java.util.Locale preferredLocale     =  HttpRequest.getLocale();
java.util.Enumeration enumLocale  = HttpRequest.getLocales();
```

- *NumberFormat* utilise par défaut la *Locale* du Système d'exploitation, cependant on peut lui spécifier une Locale en paramètre :

```
java.util.Locale preferredLocale     =  HttpRequest.getLocale();
java.text.NumberFormat nf           =  java.text.NumberFormat.getNumberInstance(preferredLocale);
String localFormatedNumber          = nf.format(10000000.00);
```

Méthodes de *NumberFormat*:

- *getPercentInstance()* (nbr décimaux),
- *getCurrencyInstance()* (valeurs monétaires),
- *getDateTimeInstance*(date et heure), et *getTimeInstance*(heure seule).

- *DateFormat* utilise une Locale mais aussi un style de mise en forme (*DEFAULT, SHORT, MEDIUM, LONG, FULL*) :

```
import java.util.*;
import java.text.*;
Locale myLocale        =  HttpRequest.getLocale();
DateFormat df          = java.text.DateFormat.getDateInstance(DateFormat.SHORT, preferredLocale);
String localDate       = df.format(new java.util.Date());
```

Format SHORT :
16/10/03 pour une *Locale* française,
2003-10-16 pour une Locale suédoise.

Format Full :
16 octobre 2003 pour une *Locale* française.

- Dans le cas ou le Runtime ne trouve pas le fichier correspondant à la Locale spécifiée, il cherche alors un fichier proche (même langue, et pas forcément le même pays), et sinon il prend le bundle indépendamment des deux critères.