http://www.labo-sun.com
labo-sun@supinfo.com

LABORATOIRE SUPINFO DES TECHNOLOGIES

Servlets & JavaServerPages

DEVELOPPEMENT WEB

Auteurs : Cyril Joui & Hélène Semere
Version n° 2.0 – 18 février 2005
Nombre de pages : 45

Paris Academy of Computer Science
SUPINFO Paris
Ecole Supérieure d'Informatique

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Table des matières

1. Introduction

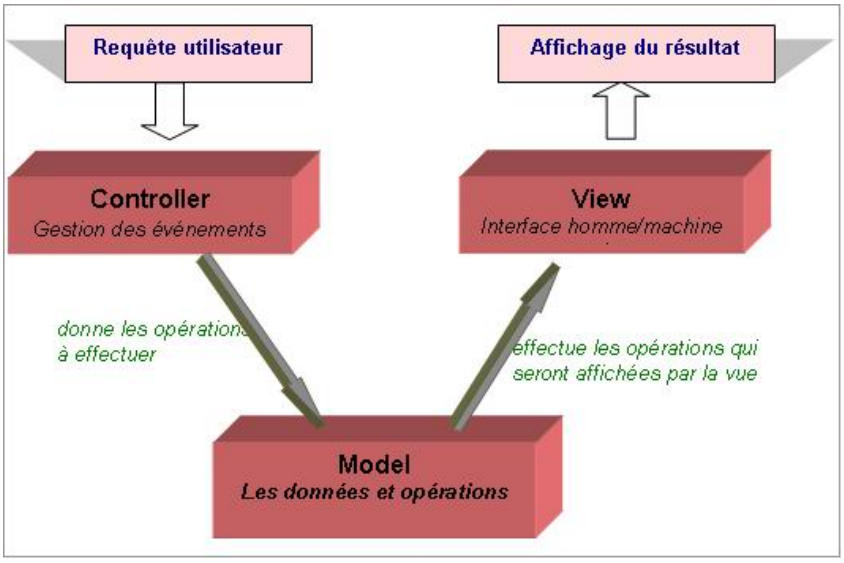
1.1. Présentation

De nos jours, les sites Internet ne se limitent plus à des contenus statiques, identiques d'un utilisateur à l'autre. Il devient nécessaire de pouvoir mettre à disposition des internautes un contenu personnalisé variant en fonction de leurs besoin, leur permettant ainsi d'accéder à l'information demandée et de pouvoir interagir avec celle-ci et celles des autres utilisateurs.

Les Servlets et les JSP constituent la solution offerte par Sun en ce qui concerne la création d'applications Web dynamique.

Contrairement aux langages de scripts simples où se mélangent couches métier, présentation et accès aux données, les Servlets/JSP permettent ici une architecture plus claire, détachée et modulaire en incitant le programmeur à séparer ces couches.

Pour une programmation optimisée, une application Web doit être fondée sur une architecture MVC (Model-View-Controller).



La couche présentation (View) correspond aux pages Web affichées au niveau du client (browser, applet, application propriétaire …).

Ces pages peuvent être statiques ou dynamiques. Dans le dernier cas, elles sont le résultat d'un code exécuté côté serveur. La programmation de cette couche peut être réalisée par un serveur d'application J2EE à l'aide des composants suivants :

- Les Servlets Java
- Les Java Server Pages
- Les JavaBeans
- Les Tag Librairies

2. Les serveurs d'applications J2EE

2.1.Présentation

Il existe de nombreux serveurs d'applications mais chacun d'eux présente des avantages et des inconvénients. Ils sont, en effet, adaptés à différentes utilisations.

Certains sont exclusivement spécialisés dans le développement de Servlet/JSP (Tomcat, Resin, Jetty, WebSphere Express et Weblogic Express) tandis que d'autres englobent l'ensemble des spécifications J2EE (JBoss, Jonas, Oracle 9iAS, WebSphere Application Server, Weblogic Platform).

Lors du choix d'un serveur applicatif, on va évidemment prendre en compte son prix. Si la majorité d'entre eux sont payants, il en existe tout de même qui sont tout à fait compétitifs en open source ; Tomcat, Jetty, JBoss, Jonas ou Orion.

Robuste et stable, Tomcat est le serveur standard pour le développement Servlet/JSP. Nous l'avons donc choisi pour les différents exemples du cours.

2.2.Fonctionnement

2.2.1. Les requêtes

2.2.1.1. Une requête HTTP de base

Une requête HTTP est effectuée par un navigateur Web (appelé aussi client) auprès d'un serveur Web afin de récupérer le contenu d'une page Web.

Les différentes étapes du processus sont les suivantes :

- Le navigateur se connecte au serveur
- Le navigateur envoie sa requête au serveur
- Le serveur cherche la page demandée (la réponse)
- Le serveur commence à envoyer le contenu de la page « réponse » (ou une page d'erreur)
- Le navigateur affiche la page au fur et à mesure qu'il la reçoit
- Une fois le transfert terminé, le navigateur ferme la connexion

Remarque : Dans le cas d'une requête simple (vers une page html par exemple) aucun moteur de servlet n'intervient pour la génération du résultat.

2.2.1.2. Une requête sur un serveur Java

Le serveur Web doit s'accompagner d'un module supplémentaire qui gèrera les servlets. Celui-ci est appelé « moteur de servlets » ou « conteneur de servlets ». Ce module est soit intégré directement au serveur Web soit sous forme de moteur indépendant. Nous expliquerons ci-après la différence entre ces deux technologies. Dans notre exemple, nous allons prendre le cas où notre moteur de servlets est indépendant (afin de mieux comprendre la distribution des tâches entre les deux composants).

Les différentes étapes du processus sont les suivantes :

- Le navigateur se connecte au serveur et envoie sa requête
- Le serveur détecte que l'on veut un accès à une servlet et transfert donc la requête au moteur de servlets
- Le moteur vérifie que la servlet est instanciée
- Si c'est le premier appel, le moteur crée une instance de la servlet et appelle la méthode ***init()*** de celle-ci
- Le moteur appelle ensuite la méthode ***service()*** de la servlet. Cette méthode reçoit deux objets en paramètres :
    - Le premier représentant la requête du client
    - Le deuxième représentant la réponse à donner à ce client
- Le moteur retourne le résultat généré au serveur web qui le renvoie au client.
- Le client affiche la page et ferme la connexion

Lors des appels suivants aucune instance ne sera créée puisque le moteur utilise toujours la même. Dans le cas de plusieurs appels synchrones (et d'une servlet générique) le moteur créé pour chaque appel un thread particulier qui générera le résultat pour l'appel associé.

2.2.1.3. Moteur de Servlets

○ Mode Autonome

Un moteur de servlet autonome est un moteur qui est totalement indépendant, c'est-à-dire qu'il contient également un serveur Web. Toutes les requêtes passent par le moteur de servlet, qu'elles appellent une page statique basique ou une servlet.

Ce mode est surtout utilisé lorsque l'on utilise uniquement des servlets ou que l'on souhaite les tester (lors de développement en local par exemple).

2.2.1.4. Mode Lié au serveur web

Le moteur de servlet est le plus souvent lié au serveur Web. En effet, dans la majorité des cas, les servlets sont couplées avec les JSP ou tout autre type de pages accessibles via ce serveur Web. Le fait que le moteur soit indépendant du serveur Web permet d'optimiser les temps de réponses car le moteur n'est sollicité que pour le traitement des servlets (et non pour toutes les requêtes).

Cependant la plupart des moteurs de servlet peuvent être utilisés aussi bien en mode autonome qu'en mode lié.

2.2.2. Configuration

2.2.2.1. Apache Tomcat

Pour indiquer quels dossiers sont partagés sur le serveur vous devez créer un « Context Path » sur le fichier de configuration du serveur : « **server.xml** ».

Ce fichier permet de spécifier une multitude d'informations concernant le paramétrage du serveur, cependant nous nous attarderons que sur les spécifications de base pour l'utilisation de servlet et JSP.

Il vous permet notamment de configurer différents noms d'hôtes (« host ») afin de spécifier différents paramètres pour chacun d'eux.

Voici le contenu d'une balise host de base pour la configuration de localhost (hôte par défaut).

```
<Host name="localhost" debug="0" appBase="webapps"
unpackWARs="true" autoDeploy="true"
xmlValidation="false" xmlNamespaceAware="false">

<Logger className="org.apache.catalina.logger.FileLogger"
directory="logs" prefix="localhost_log." suffix=".txt"
timestamp="true"/>

      <Context path="/localhost" reloadable="true"
docBase="C:\www_root\localhost\"
workDir=" C:\www_root\localhost\work" />

<Context path="/localhost2" reloadable="true"
docBase="C:\www_root\localhost2\"
workDir="C:\www_root\localhost2\work" />

</Host>
```

L'élément principal à retenir ici est la balise : « **Context** ». En effet c'est elle qui va permettre de définir des « Context Path » et donc de partager des dossiers de votre disque dur.

Dans l'exemple ci-dessus nous définissons 2 « **Context** » qui sont mappés sur des dossiers locaux.

Le premier : localhost est mappé sur : « C:\www_root\localhost\ ».

Le second : localhost2 est mappé sur : « C:\www_root\localhost2\ ».

Vous pourrez accéder à ces deux « context » depuis votre navigateur web via les adresses : http://localhost:port/localhost/ et http://localhost:port/localhost2/.

Le port est défini dans le fichier de configuration du serveur (par défaut il est égal à : 8080).

2.2.2.2. Sun Application Server

Une fois installé, le serveur vous crée un premier serveur virtuel. Un dossier de paramétrage est alors créé ; il porte le nom suivant : **https-[NomDevotreMachine.Utilisateur.com].** Cette structure peut changer en fonction des paramètres que vous avez choisis lors de votre installation. Nous utiliserons **supinfo.labosun.com** pour nos exemples. Nous allons nous attacher au dossier config.

Dans celui-ci vous avez le fichier : **default-web.xml** qui correspond au fichier par défaut utilisé pour les applications Web de votre serveur virtuel. Vous avez également un fichier **server.xml** qui va nous être utile pour configurer l'équivalent des « **context path** » vus précédemment.

Pour indiquer au serveur un mappage de dossier, il faut ajouter la balise suivante dans la balise **<VS …></VS>** :

```
<WEBAPP uri="/pathMap" path="[Lecteur]:[Chemin]"enabled="true"/>
```

Voici ce que cela donne pour notre exemple :

```
<VSCLASS id="vsclass1" objectfile="obj.conf" rootobject="default"
acceptlanguage="off">
  <PROPERTY name="docroot" value="d:/Sun/WebServer6.1/docs"/>
  <VS id="https-supinfo.labosun.com" connections="ls1" mime="mime1" aclids="acl1"
urlhosts="supinfo.labosun.com" state="on">
      <PROPERTY name="docroot" value="d:/Sun/WebServer6.1/docs"/>
      <USERDB id="default"/>
      <SEARCH>
        <WEBAPP uri="/search" path="d:/Sun/WebServer6.1/bin/https/webapps/search"
enabled="true"/>
```

```
      </SEARCH>
      <WEBAPP uri="/localhost" path="C:\www_root\localhost" enabled="true"/>
      <WEBAPP uri="/localhost2" path="C:\www_root\localhost2" enabled="true"/>
    </VS>
  </VSCLASS>
```

Nous avons, comme pour l'exemple précédent, mappé localhost et localhost2 aux deux dossiers respectifs : « **C:\www_root\localhost** » et « **C:\www_root\localhost2** ».

2.2.3. Compilation de servlets et tests

Nous allons dans cette partie utiliser une servlet la plus basique qui puisse exister. Elle affichera simplement un « Hello World » lors de son appel par le client.

Voici son code :

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.GenericServlet;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public final class HelloServlet extends GenericServlet {

	public void service (ServletRequest req, ServletResponse res)
			throws IOException {
		res.setContentType("text/html");
		PrintWriter pageWriter = res.getWriter();
		pageWriter.println("<html>");
		pageWriter.println("<body>");
		pageWriter.println("Hello World");
		pageWriter.println("</body>");
		pageWriter.println("</html>");
	}
}
```

2.2.3.1. Compilation

Si vous souhaitez exécuter la compilation à partir d'une ligne de commande et du compilateur **javac** du JDK vous devrez vérifier que votre variable d'environnement : **CLASSPATH** contient bien un chemin pointant vers les fichiers JAR des servlets standard (librairies).

Ensuite vous n'avez qu'à exécuter la commande :

```
javac HelloServlet.java
```

Le compilateur vous crée un fichier de classe qu'il faudra mettre à un endroit précis en fonction du moteur de servlets que vous utilisez.

2.2.3.2. Déploiement

Pour utiliser votre servlet il faut maintenant la déployer.

- ***Apache Tomcat et Sun Application Server***

Nous allons déployer la servlet basique pour l'exemple.

Tout d'abord, il faut que vous ayez au moins un « context path » sur votre serveur (nous utiliserons le context : localhost).

Voici l'arborescence que l'on doit utiliser pour faire fonctionner les servlets :

/ => Racine du context
/.classpath => Fichier de définissions des chemins aux librairies
/WEB-INF/classes => Dossier qui contient les packages et servlets
/WEB-INF/lib => Dossier qui peut contenir des librairies externes
/WEB-INF/web.xml => Fichier de configuration de l'application
/work/ => Dossier de travail pour le moteur de servlet

Nous plaçons donc notre fichier : **HelloServlet.class** dans le dossier : /WEB-INF/classes/.

Nous devons maintenant indiquer au moteur de servlet que nous souhaitons utiliser ce context en tant que « web-application ». Cela se fait via le fichier **web.xml**.

Voici ce que doit contenir au minimum notre fichier pour que notre servlet puisse fonctionner :

```
<?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>Web.xml de base</display-name>
    <description>
      Fichier xml de définition d'une application Web-apps de base.
    </description>

    <!-- Définition des servlets présentes dans l'application -->

    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>HelloServlet</servlet-class>
    </servlet>

    <!-- Mappage de noms pour les servlets -->

    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/servlet/Hello</url-pattern>
    </servlet-mapping>

</web-app>
```

Nous pouvons remarquer que nous spécifions que c'est une web-app par le biais de la balise : **<web-app></web-app>**

Nous indiquons alors que cette web-app contient la servlet HelloServlet qui correspond à la classe **HelloServlet**(.class).

Et finalement nous « mappons » le path : *servlet/Hello* à la servlet. Cela nous permet alors d'accéder au résultat de la servlet via l'adresse : http://localhost:8080/localhost/servlet/Hello

Remarque : il faut bien respecter l'ordre des balises, cela signifie que vous ne pouvez pas mettre une balise <**servlet**> suivie d'une balise <**servlet-mapping**> puis à nouveau une balise<**servlet**>. En effet, il n'est pas certain que tous les serveurs parsent correctement ce genre de fichier.

3. Les Servlets

3.1. Présentation

Les servlets représentent l'alternative dans le monde Java à la programmation des CGI (Common Gateway Interface). C'est une classe Java qui, chargée dynamiquement, permet d'accroître les capacités d'un serveur Web et de répondre à des requêtes dynamiquement. Le fait qu'elles s'exécutent dans une machine virtuelle Java (JVM) sur le serveur, leur assure une portabilité complète et une sécurité accrue.

Contrairement aux applets, les servlets n'imposent pas au client d'avoir le support de Java car elles s'exécutent côté serveur.

3.1.1. Avantages

Les servlets ont énormément d'avantages comparés aux simples CGI. En effet, les servlets peuvent être toutes gérées par des threads séparés au sein du même processus ou par des threads dans plusieurs processus répartis sur différents serveurs.

L'autre gros avantage des servlets est qu'elles sont portables d'un système d'exploitation à l'autre mais aussi d'un serveur Web à l'autre.

Voici une liste non exhaustive des avantages :

- Efficacité
  - Semi compilées
  - Résidentes
  - Multithreads
  - Gestion du cache
  - Connexions persistantes (pour les bases de données par exemple)

- Pratique
  - Gestion des cookies
  - Suivi de session
  - Simplification pour la manipulation du protocole HTTP
  - Portables (sur tous les systèmes d'exploitations supportant Java)
  - Il existe un grand nombre de solutions gratuites

- Puissance
  - Communication bidirectionnelle avec le serveur Web
  - Échange de données via URI
  - Partage de données entre servlets
  - Chaînage de servlet (inclusion de servlet dans une autre …)

3.1.2. Inconvénients

Le seul inconvénient des servlets est qu'elles sont limitées en matière d'interface graphique car elles s'exécutent côté serveur. Cependant il est tout à fait possible de les « coupler » à des technologies telles que Flash, Applet … Afin de rendre le contenu et les données beaucoup plus interactives.

3.2.Implémentation de base

3.2.1. Structure fondamentale

Nous avons vu précédemment comment installer et configurer un serveur Web et un moteur de servlets. Nous allons maintenant nous attarder sur le concept même des servlets avec leur conception proprement dite.

3.2.1.1. Interface Servlet

Pour qu'une classe représente une servlet, il faut impérativement qu'elle implémente l'interface **Servlet** (du package **javax.servlet**) directement ou indirectement. Cette interface oblige à implémenter les méthodes suivantes :

- La méthode ***public void init(ServletConfig cfg)***
- La méthode ***public void service(ServletRequest req, ServletResponse res)***
- La méthode ***public void destroy()***
- La méthode ***public ServletConfig getServletConfig()***
- La méthode ***public String getServletInfo()***

Nous étudierons ces méthodes en détail ci-dessous.

Remarque : les trois premières méthodes représentent le cycle de vie d'une servlet.

L'utilisation de l'interface **Servlet** n'est pas très courante. En effet différentes classes abstraites ont été développées afin de simplifier la tâche au développeur.

3.2.1.2. Classe GenericServlet

Cette première classe est la plus basique. Elle appartient elle aussi au package **javax.servlet**. Il vous suffit tout simplement, dans une classe fille, d'implémenter la méthode : ***public voidservice(ServletRequest req, ServletResponse res)***.

Voici un exemple de servlet implémentant cette classe :

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.GenericServlet;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public final class HelloServlet extends GenericServlet {

	public void service(ServletRequest req, ServletResponse res)
			throws IOException {
		res.setContentType ("text/html");
		PrintWriter pageWriter = res.getWriter();
		pageWriter.println("<html>");
		pageWriter.println("<body>");
		pageWriter.println("Coucou");
		pageWriter.println("</body>");
		pageWriter.println("</html>");
	}
}
```

3.2.1.3. Classe HttpServlet

Cette classe est un peu plus évoluée et davantage orientée développement Web. Elle est cependant inclue dans le package : **javax.servlet.http**. De même que pour la classe précédente, nous pouvons simplement implémenter la méthode : ***public void service(ServletRequest req, ServletResponse res)*** cependant une alternative s'ouvre à nous. Comme le protocole HTTP permet de transmettre des données de différentes manières (GET ou POST), cette classe possède les méthodes s'y rapportant. Cela permet d'implémenter les méthodes indépendamment des méthodes que l'on souhaite utiliser.

- La méthode ***public doGet(HttpServletRequest req, HttpServletResponse res)*** est appelée lors d'une requête de type GET.
- La méthode ***public doPost(HttpServletRequest req, HttpServletResponse res)*** est appelée lors d'une requête de type POST.

Vous pouvez également trouver d'autres méthodes ***public doXxx(HttpServletRequest req, HttpServletResponse res)*** avec Xxx = (Put, Delete, Options …)

Remarque : si vous ne souhaitez pas différencier le type de méthode utilisée pour la requête, vous pouvez soit utiliser la classe **GenericServlet**, soit n'implémenter que la méthode ***service()*** dans votre servlet.

De plus vous pouvez remarquer que nous utilisons l'objet **Http[ServletResponse/ServletRequest]** à la place de **ServletResponse/ServletRequest**.

3.2.2. Cycle de vie

Nous avons pu voir que le moteur de servlet n'utilise qu'une seule instance par servlet et que chaque requête cliente a pour résultat un nouveau thread qui est transmis à ***doGet()*** ou à ***doPost()*** ou bien ***service()*** (selon les cas). Nous allons ici examiner comment les servlets sont initialisées, utilisées et détruites. Ces étapes s'appellent le « Cycle de vie » d'une servlet.

3.2.2.1. La méthode init()

La méthode ***init()*** est appelée uniquement lors du premier appel à la servlet (soit via un appel client soit lors du démarrage du serveur en fonction de votre configuration serveur). Par conséquent, elle est employée pour effectuer les opérations de paramétrage et d'initialisation de la servlet.

Il existe deux prototypes de la méthode ***init()***.

- Le premier ne prend aucun argument et est utilisé lorsque la servlet n'a pas à lire de paramètre variant d'un serveur à l'autre.

La définition de celle-ci ressemble à ceci :

```
public void init() throws ServletException {
      // Initialisation Code
}
```

- La seconde version de ***init()*** est utilisée lorsque le servlet a besoin d'accéder aux paramètres de configuration du serveur afin de s'adapter au serveur sur laquelle elle est lancée. On peut très bien penser récupérer les paramètres de connexion à une base de données par exemple, un fichier de mots de passes ou bien d'autres choses encore.

Le second prototype ressemble à ceci :

```
public void init(ServletConfig cfg) throws ServletException {
      super.init(cfg);
      // Initialisation Code
}
```

Cette méthode admet un argument de type **ServletConfig** qui permet de récupérer les valeurs des différents paramètres de configuration grâce à la méthode ***getInitParameter()*** qui demande en entrée le nom du paramètre et retourne, en sortie, sa valeur.

La première ligne du corps de la méthode fait appel à ***super.init()***. Cette ligne ne doit pas être oubliée, en effet la méthode ***init()*** de la classe parent enregistre l'objet **ServletConfig** à un endroit précis dans la servlet qui est utilisé par la suite.

Remarque : si vous décidez de surcharger la méthode ***init()***, n'oubliez surtout pas l'appel à ***super.init()*** car vous risquez d'avoir des problèmes …

3.2.2.2. La méthode service

Cette méthode est appelée pour chaque requête reçue. Cette méthode vérifie le type de requête et appelle automatiquement soit ***doGet()***, ***doPost()***, ***doTrace()*** … Dans une servlet, qui doit traiter les requêtes POST et GET de la même manière, vous pourriez être tenté de surcharger ***service()*** mais ce n'est pas la bonne solution ! En effet, cela risquerait de nuire à l'évolution de votre servlet. Il vaut mieux dans ce cas appeler ***doPost()*** via ***doGet()*** ou inversement.

Grâce à cette méthode, vous pourrez, par la suite, surcharger les méthodes ***doPut()***, ***doTrace()***, ***doOptions()*** soit directement dans la servlet soit dans les « servlets filles ».

Voici le prototype de cette méthode :

```
public void service(ServletRequest req, ServletResponse res) throws IOException
```

3.2.2.3. La méthode destroy

Le serveur peut demander la suppression de l'instance de la servlet, soit par demande explicite de l'administrateur (redémarrage du serveur par exemple), soit parce que la servlet demeure inactif pendant une trop longue période. Le serveur appelle alors la méthode ***destroy()*** de la servlet afin que celle-ci effectue toutes les opérations de destruction. Vous pouvez retrouver des opérations de fermeture de connexion à une base de données, d'interruption de threads lancés en tâches de fond, fermer un fichier…

Remarque : un serveur Web peut très bien s'interrompre à cause d'un bug ou autre problème technique (une personne peut très bien débrancher la prise de courant !) c'est pour cela qu'il ne faut pas faire confiance à 100% à l'appel automatique de cette méthode et faire des sauvegardes d'éléments principaux durant l'exécution de la servlet.

Voici le prototype de cette méthode :

```
public void destroy()
```

3.2.3. Lire la requête

La lecture des données de la requête se fait via l'instance de l'objet **HttpServletRequest** passé en paramètre des méthodes ***doGet()***, ***doPost()*** …

Depuis la requête, il est possible de récupérer tous les paramètres passés au serveur web par le client :

Il existe quatre méthodes pour cela :

- La méthode ***getParameter(String name)*** : retourne la valeur d'un paramètre à partir de son nom
- La méthode ***getParameterValues(String name)*** : retourne une liste de valeurs d'un paramètre à partir de son nom (utilisé par exemple dans une liste à choix multiples)
- La méthode ***getParameterMap()*** : retourne une liste d'associations de paramètres et de valeurs
- La méthode ***getParameterNames()*** : retourne la liste de tous les noms des paramètres passés au serveur web

Cet objet permet également de récupérer des informations concernant le client :

- La méthode ***getRemoteAddr()*** : retourne l'adresse IP du client
- La méthode ***getRemoteHost()*** : retourne le nom complet de l'hôte client

De la même façon, nous pouvons récupérer des informations concernant le serveur :

- La méthode ***getServerName()*** : retourne le nom complet du serveur
- La méthode ***getServerPort()*** : retourne le numéro du port utilisé par le serveur

Vous pouvez également accéder à l'ensemble des variables d'environnement du serveur (comme pour les scripts CGI).

Voici un tableau récapitulatif des méthodes à utiliser :

Variable d'environnement CGI	**Méthodes d'une servlet HTTP**
SERVER_NAME	***getServerName()***
SERVER_PROTOCOLE	***getProtocole()***
SERVER_PORT	***getServerPort()***
REQUEST_METHOD	***getMethod()***
PATH_INFO	***getPathInfo()***
PATH_TRANSLATED	***getPathTranslated()***
SCRIPT_NAME	***getServletPath()***
DOCUMENT_ROOT	***getServletContext().getRealPath("/")***
QUERY_STRING	***getQueryString()***
REMOTE_HOST	***getRemoteHost()***
REMOTE_ADDR	***getRemoteAddr()***
AUTH_TYPE	***getAuthType()***
REMOTE_USER	***getRemoteUser()***
CONTENT_TYPE	***getContentType()***
CONTENT_LENGTH	***getContentLength()***
HTTP_ACCEPT	***getHeader("Accept")***

HTTP_USER_AGENT	***getHeader("User-Agent")***
HTTP_REFERER	***getHeader("Referer")***

3.2.4. Écrire la réponse

La réponse à la requête se fait via l'instance de l'objet : **HttpServletResponse** passée en paramètre des méthodes ***doGet()***, ***doPost()*** … (Nous noterons l'instance de cet objet : « res »).

3.2.4.1. Type de contenu

La première chose à faire est d'indiquer à la servlet le type de données qu'elle va renvoyer. En effet vous pouvez très bien retourner du html, une image générée…

La méthode : ***setContentType()*** vous permettra d'effectuer cela. Elle prend en paramètre le type de contenu. Voici quelques exemples d'utilisation de cette méthode :

```
res.setContentType("text/html")  // page HTML
res.setContentType("text/xml")   // page XML
res.setContentType("image/jpeg") // image JPEG
```

Il est possible de renvoyer tout type de contenu en spécifiant exactement le bon type MIME[1].

3.2.4.2. Écriture

L'écriture des données de type texte dans la réponse se fait par l'objet : **PrintWriter** que l'on peut récupérer par la méthode : ***getWriter()***.

```
PrintWriter out = res.getWriter();
out.println("<HTML>");
out.println("<BODY>");
out.println("Bonjour le monde");
out.println("</BODY>");
out.println("</HTML>");
out.close();
```

Si votre réponse renvoie des données binaires, il est préférable d'utiliser l'objet **ServletOutputStream** dont une instance est retournée par la méthode : ***getOutputStream()***

Voici un exemple pour retourner une image à partir d'une image locale :

```
res.setContentType("image/gif");
ServletOutputStream out = res.getOutputStream();
FileInputStream fis = new FileInputStream (new File ("./logo.gif"));
while (fis.available() > 0) {
      fis.read (buf, 0, buf.length);
      out.write(buf);
      out.flush();
}
fis.close();
out.close();
```

[1] Multipurpose Internet Mail Extensions

3.3.Implémentation avancée

Nous avons vu, dans la partie précédente, les bases pour le développement de servlet. Nous allons à présent nous attarder sur des concepts plus avancés.

3.3.1. Les cookies

3.3.1.1. Principe

Les cookies sont de petites informations textuelles envoyées par le serveur et renvoyé par le client, sans modification de sa part, à chaque visite ultérieure. Cette technique peut être très intéressante pour garder une trace des visiteurs (savoir s'il est déjà venu sur le site ou non …).

3.3.1.2. Avantages

Les cookies permettre de garder un lien entre le client et sa session sur un serveur par le biais d'un identifiant de session par exemple.

L'utilisation des cookies est très simple à mettre en place et permet de gérer un ensemble de fonctionnalités intéressantes pour le visiteur.

3.3.1.3. Inconvénients

Les cookies sont transmis à chaque requête, cela peut alourdir les demandes et réponses du client si ces cookies sont utilisés abusivement.

L'utilisation de cookies n'est pas à préconiser pour une utilisation sécurisée, en effet les valeurs de chaque paramètre sont passées en clair dans la requête et peuvent être récupérées par un pirate assez facilement.

Les cookies ne sont pas toujours acceptés par le client, cela peut poser un énorme problème lorsqu'un système n'a pas de moyen en parallèle mis en place.

3.3.1.4. Création de cookies

Les servlets ont leur propre API de cookies. Les cookies sont représentés par l'objet **Cookie**. Voici une description de cette classe :

- Le constructeur ***Cookie(String name, String value)*** : constructeur qui prend en paramètre le nom du cookie et sa valeur assignée.

Une fois l'objet cookie instancié, vous pouvez spécifier certains attributs de celui-ci (ce sont des « setters ») ou récupérer une information d'un cookie via le « getter » correspondant :

- La méthode ***String getComment()*** : retourne le commentaire
- La méthode ***setComment(String)*** : assigne un commentaire

- La méthode ***String getDomain()*** : retourne le domaine
- La méthode ***setDomain(String domainPattern)*** : assigne un domaine

- La méthode ***int getMaxAge()*** : retourne la durée en secondes avant que le cookie soit périmé
- La méthode ***setMaxAge(int lifetime)*** : indique la durée en secondes avant que le cookie soit périmée, la valeur 0 indique que le cookie doit être supprimé.

- La méthode ***String getName()*** : retourne le nom du cookie
- La méthode ***setName(String name)*** : assigne un nom au cookie

- La méthode ***String getPath()*** : retourne le chemin d'un cookie
- La méthode ***setPath(String path)*** : assigne un chemin au cookie (permet d'indiquer que le cookie n'est utilisé que pour certains niveaux dans le site)

- La méthode ***boolean getSecure()*** : retourne **true** si le cookie est transmis que pour les connexions sécurisées
- La méthode ***setSecure(boolean secureFlag) :*** positionne le drapeau d'indication de transmission.

- La méthode ***String getValue()*** : retourne la valeur du cookie
- La méthode ***setValue(String cookieValue)*** : assigne ou modifie la valeur du cookie

Une fois le cookie paramétré il faut le rajouter à l'en-tête http : Set-Cookie. Pour cela vous avez juste à appeler la méthode ***addCookie()*** de l'objet de réponse de votre servlet.

Voici un exemple de création de cookie :

```
Cookie cookieDejaVenu = new Cookie("dejaVenu", "1");
cookieDejaVenu.setMaxAge(60 * 60 * 24 * 365); // 1 an
res.addCookie(cookieDejaVenu);
```

3.3.1.5. Lecture des cookies

Les cookies reçus sont stockés dans un tableau que l'on peut récupérer par l'instance de l'objet **HttpServletResquest** via la méthode ***getCookies()***. Celle-ci renvoie un tableau dont la longueur représente le nombre de cookie envoyé (si la longueur est égale à nulle, aucun cookie n'a été envoyé). Pour obtenir un cookie en particuliers à partir de son nom vous devez parcourir le tableau en appelant pour chaque élément la méthode ***getName()*** jusqu'à temps d'arriver sur le cookie qui vous intéresse.

Une fois récupérer vous avez juste à appeler la méthode ***getValue()*** pour récupérer son contenu.

Voici un exemple de création de cookie :

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
      // Initialisation
      PrintWriter out = res.getWriter();
      res.setContentType ("text/html");
      Cookie[] tabCookies = req.getCookies();
      Cookie cookie;

      // Boucle qui affiche tous les cookies
      for (int i = 0; i < tabCookies.length; i++) {
            cookie = tabCookies [i];
            out.println("Cookie : " + i + "<br>" +
                        " - Nom : " + cookie.getName() + "<br>" +
                        " - Valeur : " + cookie.getValue() + "<br>");

      }
      out.flush();
      out.close();
}
```

Si vous voulez supprimer un cookie, il suffit d'attribut une durée de vie à 0 à votre objet cookie :

```
c1.setMaxAge(0);
```

3.3.2. Les sessions

3.3.2.1. Présentation

Nous avons vu précédemment comment garder des informations chez le client par le biais de cookie. Cependant ce mécanisme est très limité, c'est donc pour cela que le système de session a été mis en place.

Lorsque l'on parle de session, l'activité se passe côté Serveur et non Client (contrairement aux cookies). Les sessions sont très utilisées dans les applications web et plus particulièrement dans les sites e-business … Elles permettent de stocker des informations pour chaque client côté serveur.

3.3.2.2. Avantages

Les sessions sont exécutées côté serveur ce qui leur permet d'être beaucoup plus sécurisées que les cookies.

Un seul identifiant est nécessaire pour lier le serveur et le client car les données sont stockées sur le serveur.

Les sessions sont simples d'utilisation grâce à l'API de suivi de session des servlets.

3.3.2.3. Inconvénients

Un lien doit être mis en place entre le client et le serveur (identifiant de session) soit par cookie soit par le mécanisme de réécriture d'URL.

3.3.2.4. API de suivi de session

Les servlets possèdent une API pour la gestion des sessions. L'utilisation de sessions dans les servlets est très simple car il suffit de récupérer l'objet session lié à la requête actuelle. Si aucun objet de session n'a encore été créé, on peut décider de le créer automatiquement.

La classe décrivant une session est : **HttpSession**.

Voici une description de chacun des méthodes de celle-ci :

- La méthode ***Object getValue()*** ou ***Objet getAttribute(String name)*** : retourne la valeur d'un attribut de la session précédemment ajouté.
  Remarque : la méthode ***getValue()*** est à remplacer par ***getAttribute()*** depuis la version 2.2 de l'API des servlets.

- La méthode ***putValue(String name, Object value)*** ou ***setAttribute(String name, Object value)*** : associe un objet à un nom d'attribut de la session.
  Remarque : la méthode ***putValue()*** est à remplacer par ***setAttribute()*** depuis la version 2.2 de l'API des servlets.

- La méthode ***removeValue(String name)*** ou ***removeAttribute(String name)*** : supprime un attribut à partir de son nom.
  Remarque : la méthode ***removeValue()*** est à remplacer par ***removeAttribute()*** depuis la version 2.2 de l'API des servlets.

- La méthode ***String[] getValueNames()*** ou ***Enumeration getAttributeNames()*** : retourne les noms de tous les attributs de la session.
  Remarque : la méthode ***getValueNames()*** est à remplacer par ***getAttributeNames()*** depuis la version 2.2 de l'API des servlets.

- La méthode ***String getId()*** : retourne l'identifiant unique de la session (il peut être utilisé comme une clé).

- La méthode ***boolean isNew()*** : retourne **true** si la session vient d'être créées (nouvelle session) ou **false** dans le cas d'une session déjà existante.

- La méthode ***invalidate()*** : invalide l'ensemble de la session et libère tous les objets qui y sont associés.

3.3.2.5. Gestion de la session

Tout d'abord il vous faut récupérer l'objet **session** associé à la requête cliente, pour appeler la méthode : ***getSession()*** de l'objet **HttpServletRequest**.

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

Pour que la session soit créée automatiquement (dans le cas où elle n'existerait pas), nous passons **true** en paramètre à la méthode ***getSession()***.

Remarque : Si on passait **false** en paramètre, la méthode ***getSession()*** nous retournerait un objet **null**.

Avec cet objet **session**, vous allez pouvoir lui ajouter des objets (méthode ***setAttribute()***), en retirer (méthode ***removeAttribute()***) ou en lire (méthode ***getAttribute()***).

3.3.3. L'interface SingleThreadModel

Nous avons vu qu'une servlet n'a qu'une seule instance qui est 'attaquée' par plusieurs threads à la fois (1 thread par client). Ceci peut parfois être gênant lorsque l'on veut, par exemple, qu'un objet de connexion à une base de données ne soit pas partagé parmi les threads, la modification d'un fichier précis …

Une première solution serait d'affecter l'attribut **synchronized** à la méthode ***service()*** de la servlet.

Cependant il existe un meilleur moyen de gérer cela. En effet, il existe l'interface **SingleThreadModel** qui permet d'indiquer au moteur de servlets que l'on souhaite qu'une instance de la servlet ne soit « attaquée » que par un **Thread** à la fois.

Cette interface ne comporte aucune méthode à implémenter, elle ne représente qu'un drapeau pour le moteur de servlet.

Finalement cette interface est pratique car elle permet en quelques sortes de sécuriser les accès concurrents, cependant ce n'est pas la meilleure solution. L'idéal est de créer dans le cas d'une

connexion à la base de données, une gestion de pool de connexions multithreadées. En effet, il se peut que la base de données n'accepte qu'un nombre restreint de connexions (qui peut être dépassé par le nombre de clients accédant à la servlet).

3.3.4. Utilisation du ServletContext

On appelle contexte associé à la servlet (**ServletContext**) l'environnement au sein duquel la servlet s'exécute. L'objet **ServletContext** est créé par le containeur de servlet et permet d'accéder à des informations rattachées à l'environnement.

L'objet créé est partagé par toutes les servlets du containeur. Cela permet donc de partager des informations entre les servlets.

3.3.4.1. Description de l'objet ServletContext

Voici une description des méthodes principales de l'objet **ServletContext** :

- La méthode ***Object getAttribute(String name)*** : retourne l'attribut associé à name
- La méthode ***setAttribute(String name, Object o)*** : assigne un objet à un nom d'attribut
- La méthode ***removeAttribute(String name)*** : supprime un attribut du context
- La méthode ***Enumeration getAttributeNames()*** : retourne la liste des attributs du contexte
- La méthode ***int getMajorVersion()*** : retourne la dernière version supportée par l'environnement
- La méthode ***int getMinorVersion()*** : retourne la première version minimale supportée

3.3.4.2. Utilisation l'objet ServletContext

L'accès au context se fait par le biais de la méthode ***getServletContext()*** de la servlet. L'objet récupéré est du type **ServletContext**.

4. Les Java Server Pages

4.1. Présentation

Nous venons donc de voir les diverses utilisations d'une servlet avec leurs avantages et leurs inconvénients.

Les servlets, classes java étendant la classe **javax.servlet.http.HttpServlet**, mettent l'accent sur le code java. Les pages JSP sont, elles, plutôt proches du code html.

En effet, une page JSP ressemble beaucoup à une page html à laquelle on a ajouté du code Java encapsulé dans des balises (<% %>). Elle va cependant réutiliser des fonctionnalités de la classe **Servlet** telles que la variable request de la classe **HttpSerlvetRequest** ou la variable out de la classe **javax.servlet.ServletOutputStream**.

Au contraire des servlets une page JSP va nous permettre de séparer clairement deux types de code (le traitement de la requête et la génération du flux HTML).

Il est très aisé de programmer à l'aide d'une page JSP ; en effet, il n'est pas nécessaire d'avoir une grande connaissance de l'API sous-jacente.

Par ailleurs, contrairement aux servlets, le webmaster va pouvoir modifier le code HTML sans avoir à toucher au code Java.

Comme nous venons de le voir, une page JSP peut à la fois contenir du code Java et du code html.

Le HTML va composer la structure statique de la page tandis que le code JSP composera les éléments dynamiques de la page. Ces éléments peuvent être de 3 types :

- Eléments de scripting
- Directives
- Eléments d'action

4.2. Eléments de base du scripting

Il existe 4 types d'éléments de scripting dans une page JSP : les scriplets, les commentaires, les déclarations et les expressions.

Les scriplets, <% *mon code java* %>, sont des blocs de code Java intégrés au sein d'une page HTML. Ces caractères vont permettre d'indiquer le langage à utiliser pour traiter le code encapsulé. Les variables déclarées dans cette portion de code ne seront connues que du code de la page qui suivra.

Exemple :

```
<%String s = "Hello!"; %>
```

Les commentaires, <%-- mes commentaires --%>, comme leur nom l'indique vont vous permettre d'insérer des commentaires dans votre code. Vous pouvez évidemment utiliser les balises de commentaire HTML, mais l'avantage des balises JSP est que les commentaires ne seront pas visibles des utilisateurs.

Les déclarations, <%! %>, vont permettre de déclarer des méthodes et des variables d'instance, connues de toute la page JSP, afin qu'elles soient intégrées dans la servlet résultante. Elles peuvent être placées puis utilisées n'importe où dans la page JSP.

Les expressions, <%= %>, sont évaluées lorsqu'une page JSP est demandée et leurs résultats sont convertis en une chaîne transmise à l'objet out implicite. Aucun point-virgule n'est requis à la fin d'une expression.

Exemple:

```
Message :<%= s %>
```

Affichage:

Hello!

4.3.Les objets implicites en JSP

Lorsque vous allez coder en java à l'intérieur de vos scriplets, vous aurez accès à une liste d'éléments, appelés objets implicites, qui vous permettront d'interagir avec l'environnement de la servlet d'exécution.

- **L'élément request** fait référence à l'objet **javax.servlet.http.HttpServletRequest** associé à la requête. Il va permettre d'accéder aux paramètres POST, GET, ou les en-têtes des requêtes http tels que les cookies.
- **L'élément response** fait référence à l'objet **javax.servlet.http.HttpServletResponse** envoyé avec la réponse du client. Il n'est pas souvent utilisé, puisqu'on lui préfère le flux de sortie out.
- **L'élément out** fait référence à l'objet **javax.servlet.jsp.JspWriter** est un flux de sortie permettant d'envoyer la réponse au client.
- **L'élément config** fait référence à l'objet **javax.servlet.ServletConfig** de la page.
- **L'élément pageContext** fait référence au contexte de la page, représenté par l'objet **javax.servlet.jsp.PageContext**. Il va permettre d'accéder à différents attributs de la page.
- **L'élément application** fait référence à l'objet **javax.servlet.ServletContext**. Il va permettre d'enregistrer des données globales tout comme les variables d'instance. La différence est que ces données vont être partagés par toutes les servlets du moteur de servlets.
- **L'élément page** fait référence à une instance de la page elle-même tout comme le fait **this** dans une classe java.

- **L'élément session** fait référence à l'objet **javax.servlet.http.HttpSession**.
- **L'élément exception** fait référence à l'objet **java.lang.Throwable**. Il permet d'attraper les exceptions.

4.4.Exemple d'utilisation avec un formulaire

On souhaite créer un formulaire très simple permettant d'entrer des données dans un champ de texte puis de les afficher dans une page suivante que l'on accèdera à l'aide d'un bouton.

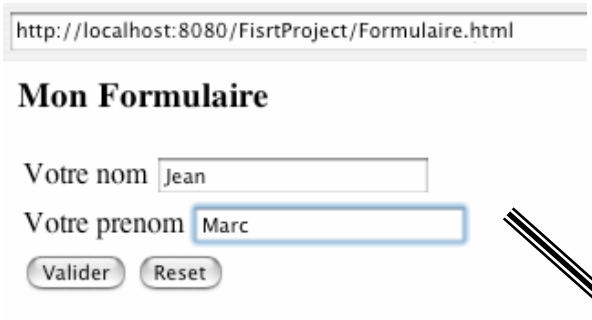


Figure 1 : **Formulaire.html**

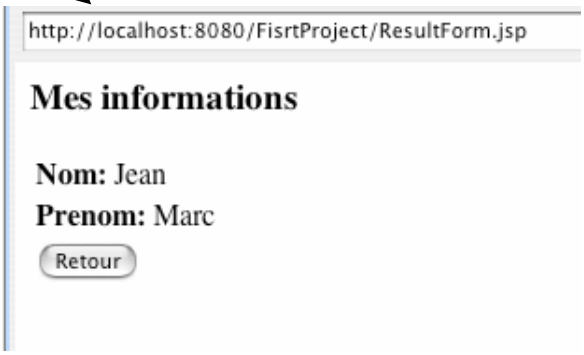


Figure 2 : **ResultForm.jsp**

Voici le code correspondant aux pages JSP Formulaire.html et ResultForm.jsp :

FirstJSP.html :

Nous ne attarderons pas sur cette partie de code puisque ce fichier ne contient que du code HTML.

```
<html>
<head>
	<title>Mon Formulaire</title>
</head>
<body>
	<h1>Mon Formulaire</h1>
	<form action="ResultForm.jsp" method="post">
```

```
		<table><tbody>
			<tr><td>
				Votre nom <input type="text" name="nom">
			</td></tr>
			<tr><td>
				Votre prenom <input type="text" name="prenom">
			</td></tr>
			<tr><td>
				<input type="submit" name="submit" value="Valider">
				<input type="submit" name=submit value="Reset">
			</td></tr>
		</tbody></table>
	</form>
</body>
</html>
```

Result .jsp :

Cette partie de code est plus intéressante ; on peut voir ici comment récupérer des données envoyées par un formulaire.

Ainsi, grâce à la méthode *getParameter(String s)* de la variable request, on peut récupérer la valeur des variables passées en paramètre.

```
<html>
<head>
<title>Résultats de mon formulaire</title>
</head>
<body>
<h3>Mes informations</h3>

<%
if (request.getParameter("submit") != null) {

	if (request.getParameter("submit").equals("Valider")) {

%>
		<table><tbody>
			<tr><td><b>Nom:
</b><%out.println(request.getParameter("nom"));%></td></tr>
			<tr><td><b>Prenom:
</b><%out.println(request.getParameter("prenom"));%></td></tr>
			<tr><td><form action="Formulaire.html"><input type="submit"
value="Retour"></form></td></tr>
			</tbody>
		</table>
<%

} else {
%>
	<jsp:forward page="Formulaire.html"/>
<%
 }
}

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

4.5. Eléments d'action : Inclusions de scripts et balises « *jsp:* »

Un élément d'action est constitué de balises que l'on peut intégrer dans une page JSP.

Voici ci-dessous la description des actions JSP standard.

- **jsp:useBean**

La balise **useBean** va permettre de séparer la partie traitement du code de la partie présentation. En effet, on va avoir des classes java, qui vont nous permettre de créer les JavaBeans. Ceux-ci vont récupérer différentes données de la page JSP, les traiter et éventuellement retourner des données que l'on pourra afficher dans la page JSP.

Un JavaBean est un objet représenté une classe Java qui doit respecter les règles suivantes :

- Elle doit comporter au moins un constructeur sans arguments déclaré en public
- Aucuns des attributs ne doivent être publics
- Chaque attribut (property) doit avoir un getter et un setter.
- Pour les attributs booléens, on doit remplacer ***getXXX()*** par ***isXXX()***.

//SES ATTRIBUTS….
La balise **useBean** permet d'instancier un composant **JavaBean** qui pourra être appelé dans la page JSP.

MonBean.java

```
package monpackage;
public class MonBean {
	private int nb;
	private boolean empty;

	public MonBean() {
	}
	public int increm(int i) {
		return nb + 10;
	}
	public int getNb() {
		return nb;
	}
	public void setNb(int nb) {
		this.nb = nb;
	}
	public boolean isNb() {
		return empty;
	}
}
```

TestBean.jsp

```
<jsp:useBean id="testB" class="monpackage.MonBean"/>
```

Ici, un objet de type **monpackage.MonBean** vient d'être créé et affecté à la variable **testB**.

Remarque : Il est nécessaire de placer votre classe dans un package.

- **Les balises jsp:setProperty et getProperty**

Ces balises vont permettre de récupérer ou modifier les valeurs d'une instance de JavaBean.

Il y a 3 manières de faire :

Une étoile pour **property** permet d'attribuer automatiquement à chaque attribut de l'objet bean les valeurs récupérer d'un formulaire par exemple.

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

On peut indiquer directement le nom de la propriété à laquelle on veut attribuer une valeur.

```
<jsp:setProperty name="testB" property="nomProp" param="nomParam"/>
```

On peut indiquer directement le nom de la propriété à laquelle on veut attribuer une valeur et, de plus indiquer la valeur grâce à l'attribut value.

```
<jsp:setProperty name="testB" property="nomProp" value="Jean"/>
```

Pour récupérer les valeurs assignées grâce aux balises <jsp :setProperty…>, on agira de la manière suivante :

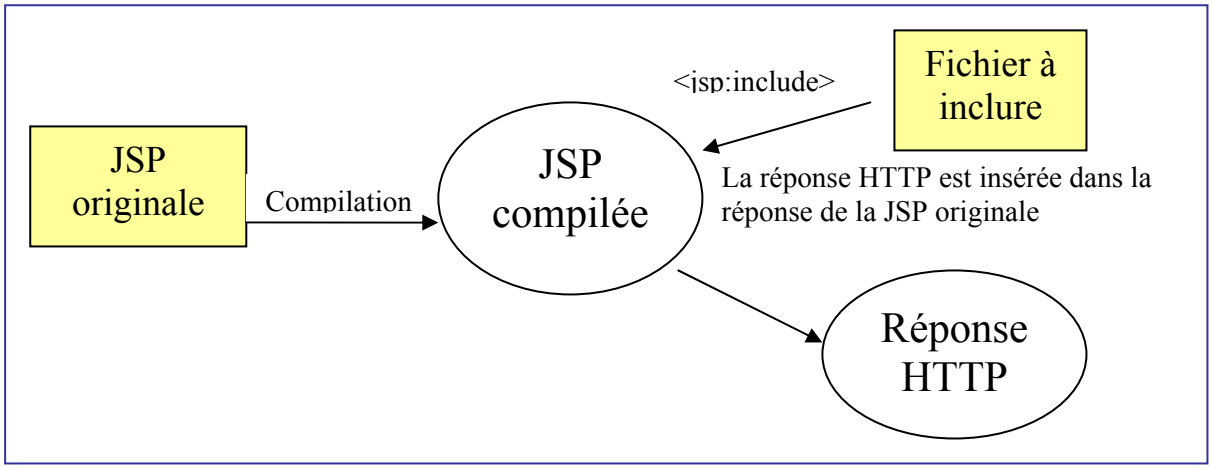
```
<jsp:getProperty name="testB" property="nomProp"/>
```

On peut, par la suite, utiliser cette instance de JavaBean dans des balises scriplets et faire appel à ses méthodes. Ainsi, au lieu d'utiliser la balise <jsp:getProperty…>, on pourra utiliser la syntaxe suivante :

```
<%= testB.getNomProp() %>
```

- **Les balises jsp:include et param**

La balise **include** est utilisée pour intégrer des ressources statiques ou dynamiques dans une page JSP. Elle va permettre d'inclure des fichiers au moment de la requête.



Il est aussi possible d'inclure un fichier au moment de la traduction de la page, juste avant la compilation (4.4.2).

Afin de passer des informations à la ressource à inclure il suffit d'ajouter la ligne suivante :

```
<jsp:include page="Chemin relatif de la page" flush="true">
      <jsp:param name="PARAM1" value="VALUE1"/>
      <jsp:param name="PARAM2" value="VALUE2"/>(...)
</jsp:include>
```

- **jsp:forward**

Cette balise est utilisée pour passer le contrôle de la requête à une autre ressource dynamique ou statique.
Afin de passer des informations à la ressource à inclure, il suffit de procéder de la même manière que la balise include, avec la balise param.

Exemple :

```
<jsp:forward page="Redirect.jsp">
            <jsp:param name="redir" value="ca marche"/>
</jsp:forward>
```

Ainsi, dans la page Redirect.jsp on va pouvoir récupérer la valeur de « redir » à l'aide de la variable **request** (***request.getParameter("redir")***).

- **Les balises jsp:plugin et jsp:fallback**

La balise plugin est utilisée pour générer des tags HTML <object> ou <embed> permettant d'indiquer au navigateur client le chargement du plugin correspondant à la balise.

La balise fallback va permettre de spécifier le message à montrer si le client ne supporte pas l'affichage d'applet.

4.6.Directives

Les directives sont des messages envoyés au container JSP qui vont indiquer au container la façon dont il doit transcrire une page en servlet. Ces directives sont repérables grâce aux caractères <%@ %>.

Il existe 3 types de directives ; les directives de pages, les directives d'inclusions et les balises personnalisés.

4.6.1. Directives de pages

Ces directives se présentent sous la forme suivante : **<%@page... %>.** Elles prennent en paramètre un certain nombre de paramètres listés ci-dessous:

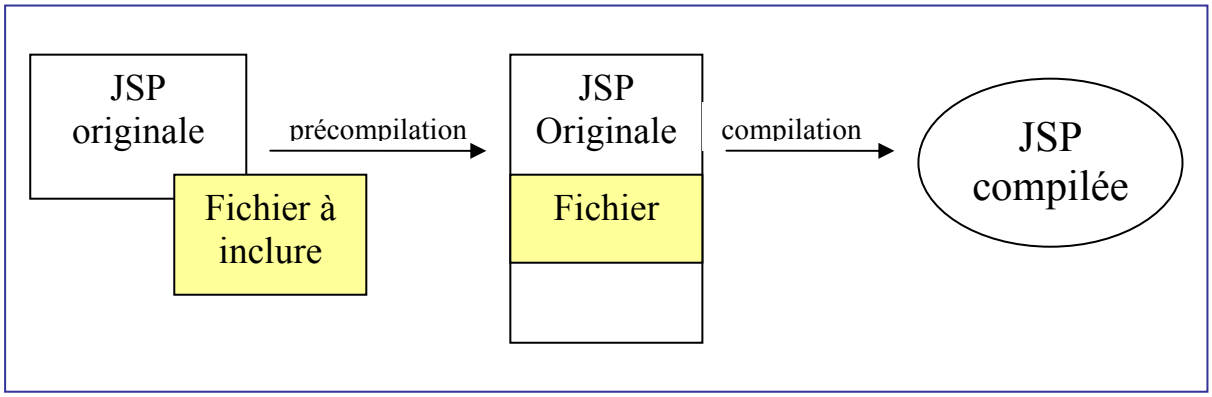
Paramètre	Rôle du paramètre	Valeurs possibles
langage	Indique le langage utilisé	« **java** »
info	Le texte contenu dans ce paramètre, contenant des informations sur la page sera récupérable grâce à la méthode *servlet.getServletInfo()*	*dépend du container de la jsp*
contentType	Indique le type MIME(Multipurpose Internet Mail Extension) de la page ainsi que le jeu de caractères utilisés	**text/html;charset=ISO-8859-1**
extends	Indique la classe mère de la servlet générée. Cet attribut est à manipuler avec précaution.	« **class** »
import	Indique les inclusions de package ou de classes. Plusieurs localisations peuvent être spécifiées, séparées à l'aide de virgule. <%@page import="java.util.*"%> <%@page import="java.util.*,java.io.*"%> Il correspond à l'import dans une classe java.	**Aucune**

buffer	Par défaut, le contenu d'une page JSP est bufferisé pour accroître les performances.	**8192b**
autoFlush	Indique si le tampon doit être vidé lorsqu'il est plein.	« **true** »/« false »
session	Indique si la page courante peut accéder aux données stockées dans la session.	« **true** »/« false »
isThreadSafe	Indique si le modèle SingleThreadModel (vu au point 3.3.3) est utilisé ou non.	« **true** »/« false »
errorPage	Toute exception non gérée dans le code sera remontée à la classe mère de la JSP.	« **url** »
isErrorPage	Indique au conteneur J2EE que la servlet générée sert de page d'erreur.	« **false** »

* les valeurs en gras sont les valeurs par défaut

4.6.2. Directives d'inclusions

Ces directives permettent d'inclure le contenu d'un autre fichier dans la page JSP courante à l'aide de la syntaxe suivante : <%@ include file="UrlRelative" %>.
Ce type d'inclusion est réalisé avant la compilation de la JSP en servlet



4.6.3. Directives des tag-libs

Ces directives définissent l'adresse et le préfixe d'une librairie de balises (tags) pouvant être utilisés dans la page.

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

Elles permettent d'aller créer de nouveaux tags de la forme :

```
<tagPrefix:NomTag attribut1="valeur" ... %> ... </tagPrefix:NomTag>
```

Nous étudierons plus précisément ces directives dans le chapitre suivant (4.7).

4.7.Les balises personnalisées (Taglib)

4.7.1. Présentation

Comme nous l'avons expliqué précédemment, une page JSP est une page de code qui peut contenir du HTML et du code Java. Malgré la puissance offerte par le langage Java, il arrive souvent que l'on réécrive plusieurs fois la même chose au sein d'une même page JSP (boucles, tests, récupération d'objets en session).

Afin d'éviter ce type de perte de temps, un mécanisme permettant d'effectuer ces tâches répétitives à l'aide de balises personnalisées a été mis en place. Ce mécanisme qui définit le fonctionnement de nouvelles balises s'appelle **taglib**.

Permettant d'isoler le code Java dans un fichier .java plutôt que dans un fichier .jsp, ces **taglib** ont pour but de faciliter l'organisation d'une réalisation d'un site Web en JSP. Ceci est en effet fondamental pour le développement et surtout la maintenance puisqu'on va permettre d'isoler le travail des Web designers de celui des développeurs.

Ainsi, on va avoir un fichier TLD qui va décrire les attributs de la balise et un fichier Java qui va décrire les actions de la balise.

4.7.2. L'API JSP Custom Tag

La balise personnalisée doit être reliée à un **Tag handler** qui va définir ses actions. Un tag handler est une classe Java appelée chaque fois que le container de JSP rencontre une balise personnalisée.

Cette classe doit implémenter une interface du package **javax.servlet.jsp.tagext** où dériver de l'une des classes de ce package.

Les 3 interfaces les plus importantes sont **Tag**, **IterationTag** et **BodyTag**.

4.7.2.1. L'interface Tag

```
public interface Tag {

	int doEndTag();
	int doStartTag();
	Tag getParent();
	void release();
	void setPageContext(PageContext pc);
	void setParent(Tag t);
}
```

L'interface Tag est donc l'interface à implémenter lors de la création d'un tag handler. Les différentes méthodes qu'elle implémente vont permettre, tour à tour l'initialisation, l'évaluation puis la fermeture de notre balise. Leur comportement suit un cycle de vie, décrit ci-dessous.

4.7.2.2. Le cycle de vie d'un Tag Handler

Tout d'abord, il faut savoir que dans une page JSP, une balise n'est instanciée qu'une seule fois. Cela signifie que si vous faites appel à une même balise plusieurs fois dans une page, le container fera toujours appel à la même instance du tag handler (classe Java correspondant à la balise).

Le cycle de vie d'un tag handler est contrôlé par le container de JSP :

1. Dans un premier temps, le container de JSP obtient une instance du tag handler à partir du pool, ou en crée une nouvelle.

2. Il appelle ensuite la méthode ***setPageContext()*** à laquelle il passe un objet **PageContext** représentant la page JSP qui contient le custom tag.

3. Le container appelle ensuite la méthode ***setParent()*** à laquelle il passe un objet **Tag** (tag du parent) ou un objet **null**.

4. Le container de JSP définit ensuite tous les attributs du custom tag. Les attributs sont traités comme les propriétés dans un Javabean (**getters**, **setters**)

5. Le container appelle ensuite la méthode ***doStartTag()***. Celle-ci peut retourner les valeurs :
   a. **Tag.SKIP_BODY** : Il n'y a pas d'évaluation du corps de la balise, c'est à dire que le contenu entre la balise d'ouverture et de fermeture ne sera pas affiché.
   b. **Tag.EVAL_BODY_INCLUDE** : Le container traite le corps du tag en tant que JSP et il est affiché dans la page JSP (ce corps peut être constitué de JSP, de HTML, et même d'autres tags).

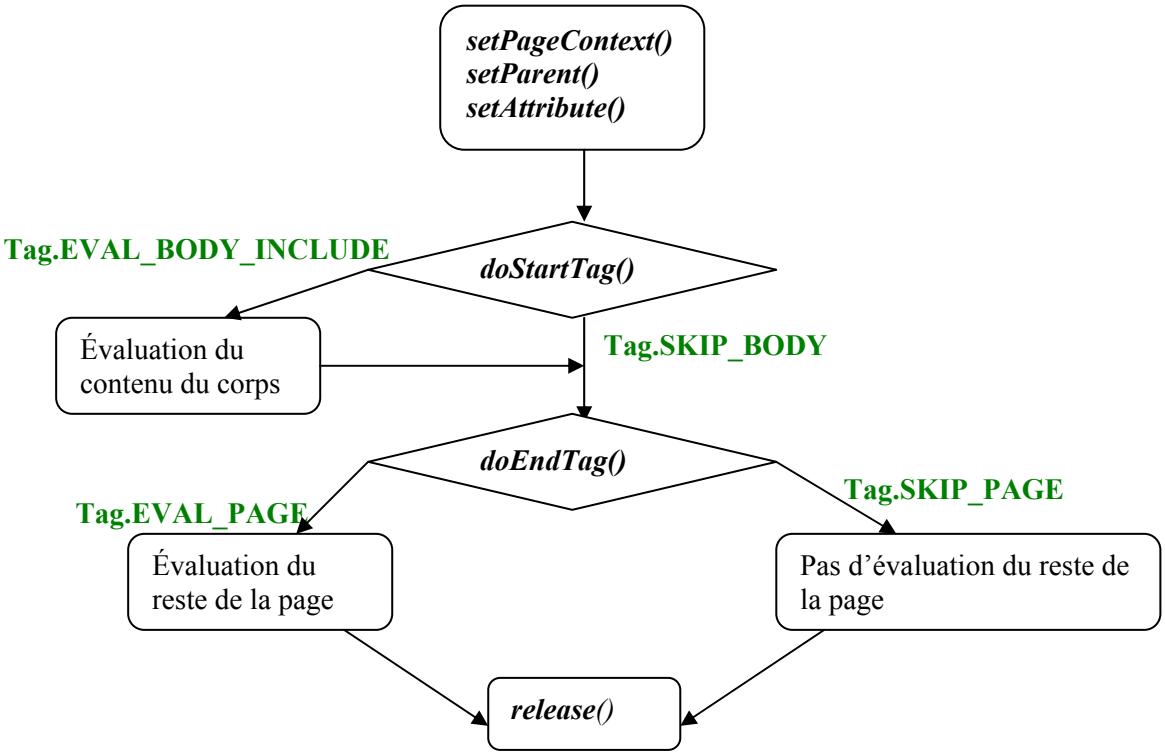
6. Quelle que soit la valeur retournée précédemment, le container appelle ensuite la méthode ***doEndTag()***.Cette méthode retourne la valeur
   a. **Tag.SKIP_PAGE** : pas de traitement pour le reste de la page JSP, le contenu de la page se trouvant après la balise de fermeture ne sera pas affiché.
   b. **Tag.EVAL_PAGE** : Le reste de la page JSP est traité normalement.

7. La dernière méthode appelée est la méthode ***release()***. C'est au sein de cette méthode qu'il est possible d'écrire du code de libération de la mémoire (fermeture des connexions …).

On notera bien que cette méthode n'est pas forcément appelée à chaque méthode ***doEndTag()*** ! En effet, le container ne fera appel à cette méthode que lorsque l'objet Tag handler ne sera plus utilisé, juste avant de supprimer sa référence définitivement.

8. Enfin, le container de JSP retourne une instance du tag handler dans le pool.

Schéma du cycle de vie du Tag handler (1)



Pour le moment, les méthodes de notre Tag handler nous limitent dans nos actions. Par exemple, on ne peut évaluer le corps de notre balise (son contenu) qu'une seule fois, et, si l'on peut afficher des données, on ne peut pas récupérer le corps de la balise et le modifier.
On ne peut que récupérer des données mises en paramètre de la balise.

Exemple de balise implémentant directement l'interface Tag :
Nous allons créer un Tag handler correspondant à une balise messColor. Cette balise va afficher le message ***Hello World Tag !*** avec une couleur définie en paramètre.

```
public class MonTag implements Tag {
	private PageContext pageContext;
	private Tag parent;
	private String color;

	public void setColor(String color) {
		this.color = color;
	}

	public void setPageContext(PageContext arg0) {
		pageContext = arg0;
	}
	public void setParent(Tag arg0) {
		parent = arg0;
	}
	public Tag getParent() {
		return parent;
	}
	public int doStartTag() throws JspException {
		JspWriter out = pageContext.getOut();
		try {
			out.println("<font color="+color+">Hello World Tag !</font>");
```

```
            } catch (IOException e){

            }
            return SKIP_BODY;
      }
      public int doEndTag() throws JspException {
            return SKIP_PAGE;
      }
      public void release() {
      }
}
```

4.7.2.3. L'interface IterationTag

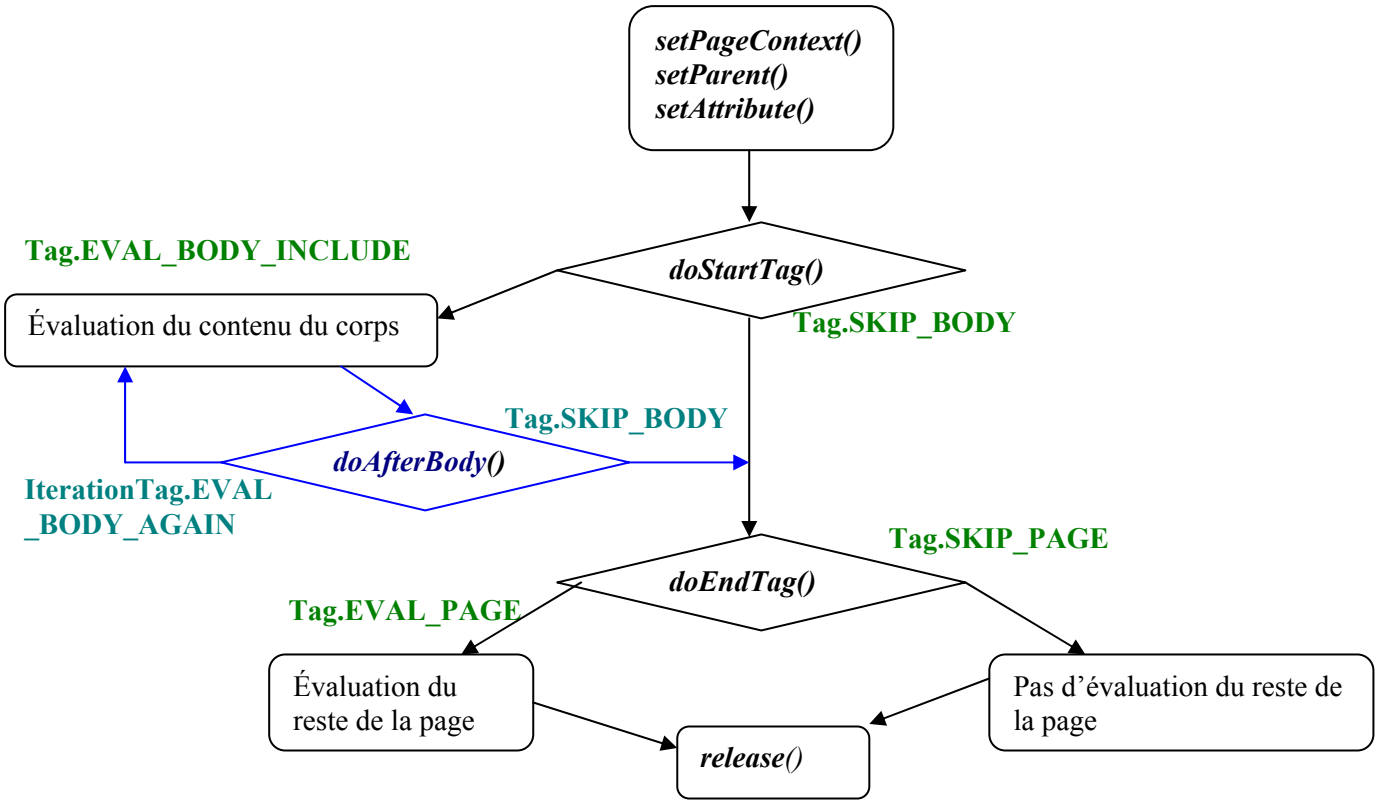
L'interface **IterationTag,** qui étend l'interface **Tag,** permet d'afficher plusieurs fois le contenu d'un tag.

```
public interface IterationTag {

      int doAfterBody();
}
```

La méthode ***doAfterBody()*** est appelée après la méthode ***doStartTag()*** et peut retourner **Tag.SKIP_BODY** ou **Tag.EVAL_BODYAGAIN**.

Elle va permettre la réévaluation du corps de la balise plusieurs fois. Si ***doAfterBody()*** retourne l'entier **Tag.SKIP_BODY** la méthode ***doEndTag()*** est directement appelée, si l'entier **Tag.EVAL_BODYAGAIN** est retourné la méthode ***doAfterBody()*** est rappelée**.**

Schéma du cycle de vie du Tag handler(2)



4.7.2.4. L'interface BodyTag

L'interface **BodyTag,** qui étend l'interface **IterationTag,** permet de récupérer le corps de la balise.

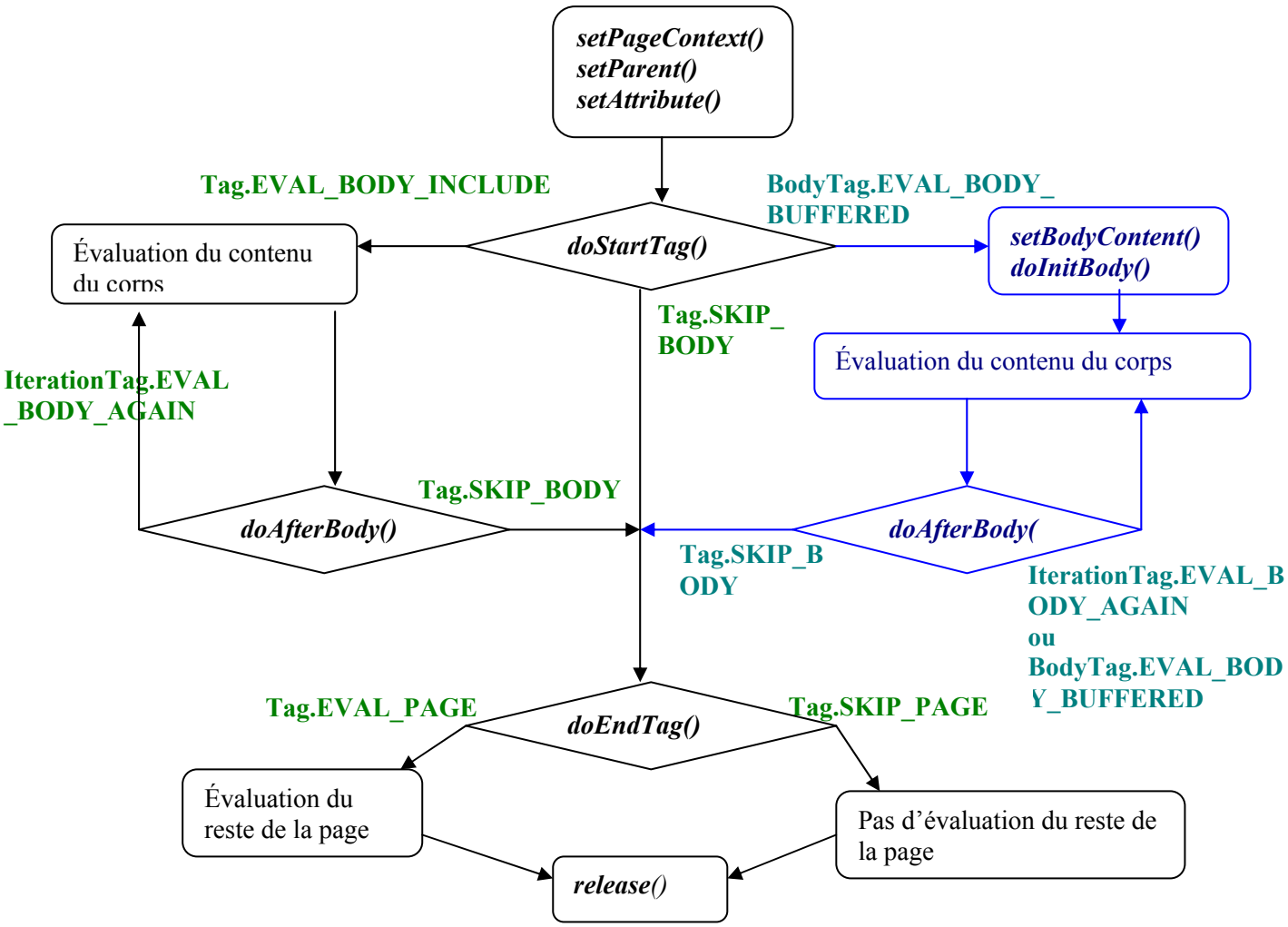
```
public interface BodyTag {

	void doInitBody();
	void setBodyContent(BodyContent b);
}
```

Cette interface implémente deux méthodes supplémentaires :

La méthode ***setBodyContent(BodyContent b)***, appelée après la méthode ***doStartTag()*** va initialiser le **BodyContent**, et donc permettre de récupérer le corps de la balise et de le modifier.

La méthode ***doInitBody()*** va permettre de vérifier que cet objet BodyContent a bien été initialisé et retourner une JspTagException si ce n'est pas le cas.

Schéma du cycle de vie du Tag handler(3)



4.7.2.5. Création d'un custom tag

On peut diviser les customs Tags en deux catégories ;

- **Les balises sans traitement de corps**

Il s'agit des balises comme les sauts de ligne (balise <br> en HTML) qui ne vont pas nécessiter de modification de contenu.

Pour créer une balise ne nécessitant pas de modification de contenu, on va créer une classe étendant la classe **TagSupport** qui elle-même implémente l'interface **IterationTag** :

```
public class TagSupport {

	int doEndTag();
	int doStartTag();
	static Tag findAncestorWithClass(Tag from, Class classe);
	String getId();
	Tag getParent();
	Object getValue(String k);
	Enumeration getValues();
	void removeValue(String k);
	void setId(String id);
	void setPageContext(PageContext pageContext);
	void setParent(Tag t);
	void setValue(String k, Object o);
}
```

L'avantage d'étendre la classe TagSupport et non l'interface InterationTag est que l'on pourra ne redéfinir que les méthodes dont on a besoin de modifier.

En effet, on a pu remarquer dans les exemples de classe implémentant les interfaces Tag, IterationTag ou BodyTag, que l'on est obligé de redéfinir toutes les méthodes des interfaces ce qui est assez contraignant.

Voici les implémentations par défaut de cette classe **TagSupport :**

- La méthode ***doStartTag()*** retourne l'entier SKIP_BODY.
- La méthode ***doAfterBody()*** retourne l'entier SKIP_BODY.
- La méthode ***doEndTag()*** retourne l'entier EVAL_PAGE.

Exemple de balise sans traitement de corps :

Nous allons, dans cet exemple, créer une balise affichant le message "Tag sans corps" avec un font color défini. Pour cela, nous allons créer une classe Java (tag handler) **TagSansCorps**, étendant la classe **TagSupport**.

TagSansCorps.java

```
public class TagSansCorps extends TagSupport {
	private String couleur;
	public void setCouleur(String couleur) {
		this.couleur = couleur;
	}
	public int doEndTag() throws JspException{
		JspWriter out = pageContext.getOut();
		try {
			out.println("<font color="+couleur+">Tag sans corps</font>");
		} catch (Exception e) {
			System.out.println(e.getLocalizedMessage());
		}
		return EVAL_PAGE;
	}
}
```

La classe TagSansCorps possède donc un attribut couleur qui va définir la couleur de font de notre message.

- **Les balises avec traitement de corps**

Il s'agit des balises comme des mises en gras ou en couleur (balises ou bien en html) qui vont modifier un contenu.

Pour ce type de balises, on va créer une classe étendant la classe **BodyTagSupport** qui elle-même étend la classe **TagSupport** et implémente l'interface **BodyTag** :

```
public class BodyTagSupport {

	int doAfterBody();
	int doEndTag();
	void doInitBody();
	int doStartTag();
	BodyContent getBodyContent();
	JspWriter getPreviousOut();
	void release();
	void setBodyContent(BodyContent b);
}
```

Ainsi, pour récupérer le corps de la balise, vous allez récupérer le BodyContent de la classe BodyTagSupport.

La classe BodyContent :

Cette classe hérite de **javax.servlet.jsp.JspWriter**. Elle représente le corps du custom tag (lorsqu'il existe).

Le **BodyContent** d'un custom tag peut être récupéré à partir de la méthode ***getBodyContent()*** de l'interface bodytag à partir du paramètre en argument de la méthode.

String content = bodyContent.getString() ;

Enfin, l'écriture vers la page JSP se fait à l'aide d'un **PrintWriter** joint au corps du tag

JspWriter out = bodyContent.getEnclosingWriter() ;

Voici les implémentations par défaut de cette classe **BodyTagSupport :**

- La méthode ***doStartTag()*** retourne l'entier EVAL_BODY_BUFFERED.
- La méthode ***doAfterBody()*** retourne l'entier SKIP_BODY.
- La méthode ***doEndTag()*** retourne l'entier EVAL_PAGE.

Exemple de balise de traitement de corps :

La classe UpperTag.java que l'on a utilisé dans l'exemple de fichier .tld correspond justement à une balise de traitement de corps puisqu'elle doit permettre de récupérer un corps, le mettre en majuscule et le renvoyer.

Cette classe UpperTag va donc étendre la classe BodyTagSupport :

UpperTag.java

```
public class UpperTag extends BodyTagSupport {
	private int nb = 0;
	private String couleur = "";

	public int doAfterBody() throws JspException {
		nb--;
		if (nb >= 0) {
			BodyContent bc = getBodyContent();
```

```
                        String s = bc.getString().toUpperCase();

                        JspWriter out = getBodyContent().getEnclosingWriter();

                        try {
                                out.println("<br><font color="+couleur+">Votre contenu: "
+ s+"</font>");
                                bc.clearBody();
                                return EVAL_BODY_BUFFERED; // le renvoi de
//EVAL_BODY_BUFFERED va provoquer le rappel de cette méthode doAfterBody()
                        } catch (IOException io) {

                        }
                }
                return SKIP_BODY;
        }
        /**
         * @param couleur The couleur to set.
         */
        public void setCouleur(String couleur) {
                this.couleur = couleur;
        }
        /**
         * @param nb The nb to set.
         */
        public void setNb(String nb) {
                this.nb = Integer.parseInt(nb);
        }
}
```

4.7.2.6. Rôle et format du fichier TLD (Tag Librairy Descriptor)

Le fichier TLD est un fichier XML qui va permettre de définir les différentes balises personnalisées.

Syntaxe d'un fichier TLD :

Balise	Rôle de la balise
<taglib>(obligatoire)	Constitue l'élément racine du fichier
<tlib-version> (obligatoire)	version de la librairie de balise
<jsp-version> (obligatoire)	version des spécifications JSP dont dépend la librairie de balise
<short-name>	nom court pour la librairie
<info>	informations pour la documentation
<uri>	lien vers une source additionnelle de documentation identifiant de manière unique la librairie
<tag>	englobe la définition d'une balise personnalisée

- ***L'élément tag***

L'élément tag va permettre de définir les différentes caractéristiques de notre balise personnalisée à l'aide des éléments suivants :

Nom de l'élément	Rôle de d'élément
name (obligatoire)	identifiant pour le tag
tagclass (obligatoire)	Nom de la classe correspondant à la balise

teiclass	Nom de la classe d'info concernant la tagclass
body-content • Empty :pas de contenu pour le tag • JSP : le corps peut être constitué d'éléments JSP • tagdependent : le corps doit être interprété par ce tag	type de contenu
info	descriptions, infos
Attribute • name (obligatoire) : identifiant • required : (true,false) • rtextprvalue:la valeur de cet attribut peut être déterminé au moment de la requête(true,false)	

○ *Exemple de fichier TLD*

Dans cet exemple, nous avons une classe Java UpperTag.java qui va permettre de mettre un texte en majuscule. Nous souhaitons donc définir la balise correspondante dans notre fichier Desc.tld.

Fichier Desc.tld :

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<taglib>
	<tlib-version>1.0</tlib-version>
	<jsp-version>1.2</jsp-version>
	<short-name>mesTags</short-name>
	<description>Mes premiers Tags</description>
	<tag>
		<name>Majuscule</name>
		<tag-class>UpperTag</tag-class>
		<attribute>
			<name>nb</name>
			<rtexprvalue>true</rtexprvalue>
			<required>false</required>
		</attribute>
		<attribute>
			<name>couleur</name>
			<rtexprvalue>true</rtexprvalue>
			<required>true</required>
		</attribute>
	</tag>
</taglib>
```

4.7.2.7. Rôle du descripteur de déploiement

Le descripteur de déploiement est un fichier XML qui va permettre au container de JSP l'accès au fichier TLD (en lui indiquant le nom, l'emplacement dans le répertoire...). Il est possible d'éliminer le rôle du descripteur de déploiement en ajoutant des informations sur le nom et l'emplacement du fichier TLD directement dans la page JSP. Toutefois cela diminue la flexibilité, car lors de changements, il convient de modifier tous les fichiers JSP utilisant les customs Tags.

Exemple de fichier web.xml :

```
<web-app id="WebApp_ID">
   <display-name>Td_Corriges</display-name>
   <welcome-file-list>
      <welcome-file>index.html</welcome-file>
      <welcome-file>index.htm</welcome-file>
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
         <taglib>
                  <taglib-uri>/MesTags</taglib-uri>
                  <taglib-location>/WEB-INF/tlds/Desc.tld</taglib-location>
         </taglib>
</web-app>
```

4.7.2.8. Syntaxe et utilisation d'un custom tag

Lorsque vous voudrez intégrer votre balise personnalisée dans une page JSP, il suffira de tout d'abord de 'déclarer' la balise en début de page comme suit :

```
<%@ taglib uri="MesTags" prefix="meslibs" %>
```

L'attribut **uri** permet de spécifier le chemin où se trouve le fichier TLD qui décrit la syntaxe de la balise, tandis que l'attribut **prefix** permet de définir une instance du custom tag.

Pour utiliser votre balise personnalisée, il suffit alors d'utiliser les expressions suivantes :

<PrefixeDuTag:NomTag attribut1="valeur"/>

<PrefixeDuTag:NomTag>body</PrefixeDuTag:Nomtag>

Exemple :

```
<meslibs:Majuscule nb="12" couleur="red">Hello World</meslibs: Majuscule >
```

4.7.3. La richesse des CustomTag

Les CustomTag présentent une grande richesse dans la programmation JSP. Là où l'on va pouvoir réellement profiter de leur puissance, c'est en réutilisant des taglibs prédéfinies, très complètes et parfaitement réutilisables.

Les JSTL (JavaServer Pages Standard Tag Library) sont des librairies de référence. Elles contiennent toute une série de balises personnalisées qui vont permettre, par exemple, d'itérer, exécuter des requêtes SQL, utiliser des documents XML.

Il existe, par ailleurs de nombreuses autres librairies toutes aussi puissantes les unes que les autres.

Voici quelques liens où vous trouverez les plus connues :

- http://jakarta.apache.org/taglibs/
- http://displaytag.sourceforge.net/install.html
- http://struts.apache.org/userGuide/

4.7.3.1. Les JavaServer Pages Server TagLib

Les JSTL présentent donc de nombreuses fonctionnalités. Nous allons présenter quelques exemples d'utilisation.

Tout d'abord, pour utiliser des balises de cette librairie, il faut utiliser un conteneur d'application implémentant au moins l'API servlet 2.3 et l'API JSP 1.2.

Tous les fichiers nécessaires à leur utilisation se trouvent dans le pack **Java Web Services Developer Pack v1.4.**

Une fois le dossier concerné(/jstl/) récupéré, il faudra placer :

- les TLD (présents dans /jstl/tld/) dans le répertoire /WEB-INF/tlds/
- les fichiers jstl.jar et standard.jar (présents dans /jstl/tld) dans le répertoire /WEB-INF/lib

Les JSTL présentent 4 librairies différentes :

- **La librairie Core** représente les fonctions de base et est associée au fichier TLD c.tld.
- **La librairie XML** représente les traitements XML et est associée au fichier TLD x.tld
- **La librairie I18n** représente les fonctions d'internationalisation de la page JSP et est associée au fichier TLD fmt.tld
- **La librairie Database** représente les traitements SQL est associée au fichier TLD sql.tld

La librairie Core

Cette librairie concerne les balises de base telles que des balises de condition, de gestion d'url ou d'itération.

- Les balises d'expression de langage : **out**, **set**, **remove** et **catch**.

La balise **out** permet d'envoyer un flux de sortie à la page JSP contenu dans l'attribut **value**. Elle prend, de plus pour attributs **default** (valeur si **value** est nulle) et **escapeXml** (booléen indiquant si les caractères particuliers **<**, **>**, **&**, **'**, **"**, **é**…- doivent être convertis en leur équivalent HTML **<**, **>**, **&**, **"**,…-, **true** par défaut).

On notera que cette balise doit contenir au moins l'attribut **value** et ne doit pas contenir de corps.

Exemple :

```
<coreTag:out value="Hello !" default="default value" escapeXml="true"/>
```

La balise **set** permet de stocker une variable dont on va définir la portée de visibilité.

Elle permet d'« instancier » une variable, à l'aide des attributs **var** (nom de la variable) et **value** (valeur à stocker) ou bien de modifier les valeurs d'un objet bean, à l'aide des attributs **target** (nom de la variable contenant le bean), **property** (nom de la propriété du bean) et **value**. C'est ensuite l'attribut **scope** qui va permettre de spécifier la portée de visibilité de la variable. Il peut prendre en paramètre les valeurs suivantes : **page**, **request**, **session** et **application**.

Exemple 1:

```
<coreTag:set value="Hello !" var="mess" scope="page"/>
```

Exemple 2:

```
<coreTag:set value="Hello !" target="monBean" property="nomB" scope="page"/>
```

Pour récupérer la valeur de la variable, on va pouvoir utiliser la balise **out** vue précédemment en utilisant la syntaxe suivante **${nomvariable}.**

Il est important de spécifier dans la directive de page <%@ page …%> l'attribut **isELIgnored** à la valeur **false.** En effet, étant par défaut à **true**, la page jsp interpréterait la chaîne de caractère "${nomvariable}" en tant que chaîne de caractère et non en tant que variable:

Exemple :

```
<coreTag:out value="${pageScope.mess}" default="default value" escapeXml="true"/>
```

Si on ne précise pas la portée de la variable dans l'attribut **value** (value="${mess}"), elle sera, par défaut recherchée dans la page, puis dans la requête, la session, et enfin dans l'application.

La balise **remove** permet de supprimer une variable. Il suffit pour cela de spécifier le nom de la variable dans l'attribut **var** et la portée de cette variable dans l'attribut **scope**.

La balise **catch**, comme son nom l'indique, permet d'attraper les exceptions. Le code risquant de lever une exception doit être encapsulé entre les balises **catch**. L'attribut **var** qui contient les informations concernant l'exception levée possède une propriété **message** qui va permettre d'afficher un message d'erreur.

Exemple :

```
<coreTag:catch var="Erreur">
      <%-- code risquant de lever une exception --%>
</coreTag:catch>
<coreTag:out value="${Erreur.message}"/>
```

- Les balises de condition et d'itération : **if**, **choose**, **forEach**, **forTokens**.
- Les balises de gestion d'url : **import**, **url**, **redirect**.

5. Interactions entre JSP et Servlet

5.1. Pourquoi faire des interactions entre les JSP et Servlet

Bien qu'elles présentent des avantages énormes (génération d'image en temps réel, compression de pages …) lorsque l'application nécessite beaucoup de programmation, les servlets ne sont pas vraiment adaptées à la génération de code HTML, c'est également pour cela que les JSP ont fait leur apparition.

Même si les technologies comme les JavaBeans, les Tag-Libs, les scriptlets ont fait leur apparition et ont montré leur puissance en ce qui concerne l'intégration de code java propre dans les pages de résultat, il reste toujours un problème.

Les JSP sont fournies pour ne fournir qu'un seul type de présentation. Si l'on souhaite créer une application complexe cherchant à présenter les informations de façon différentes en fonction des demandes du client cela devient de plus en plus complexe.

C'est alors qu'interviennent les transmissions de requêtes.

5.2. A partir d'une Servlet

Pour utiliser la transmission de requête ou l'inclusion de document externe il faut utiliser un **RequestDispatcher**. Cet objet s'obtient depuis un **ServletContext** et la méthode ***getRequestDispatcher()***.

Vous devez fournir une URL absolue du fichier vers lequel vous voulez transmettre votre requête. Si vous voulez obtenir un **RequestDispatcher** vers l'adresse : http://host/folder/page.jsp il vous faudra faire l'appel suivant :

```
RequestDispatcher dispat =
getServletContext().getRequestDispatcher("/folder/page.jsp");
```

Une fois que vous avez eu votre objet il ne vous reste plus qu'à appeler la méthode ***forward()*** ou ***include()*** de celui-ci. Vous devez fournir à celle-ci les objets **HttpServletRequest** et **HttpServletResponse**.

Si vous utilisez *include*, la ressource est insérée dans la servlet active.

Si vous utilisez *forward*, le flux est complètement redirigé vers la nouvelle ressource (l'appelant ne peut plus effectuer de sortie au client, cette tâche est transmise à la nouvelle ressource uniquement).

L'inclusion ou le forward vers une nouvelle ressource (page ou servlet) se fait ainsi :

```
// Inclusion
```

```
void service(HttpServletRequest req, HttpServletResponse res) {
	RequestDispatcher dispat =
getServletContext().getRequestDispatcher("/folder/page.jsp");
	dispat.include(req, res);
}
```

```
// Forward
void service(HttpServletRequest req, HttpServletResponse res) {
	RequestDispatcher dispat =
getServletContext().getRequestDispatcher("/folder/page.jsp");
	dispat.forward(req, res);
}
```

Remarques :

- L'inclusion d'une servlet dans une autre est synchrone, c'est à dire qu'elle ne rend la main qu'à la fin de la méthode ***service()*** de la servlet appelée
- La redirection est quand à elle asynchrone

5.3.A partir d'un script JSP

La transmission de requête dans une JSP est quasiment la même que celle pour une servlet. En effet même si ce concept est beaucoup moins utilisé il peut l'être lorsque la JSP découvre qu'une donnée transmise n'est pas correcte par exemple.

La méthode classique de redirection peut être utilisé avec les JSP, cependant il est préférable d'utiliser **jsp:forward** pour des raisons de commodités.

Voici le code qui permet de rediriger vers une autre page :

```
// Forward
<jsp:forward page="/pageForward.jsp"/>
```

5.4.Transmission des données supplémentaires

Lorsque vous faite une inclusion ou un « forward » vous passer la main à un nouveau script ou une servlet. Cela ne veux pas dire que les données que vous avez pu travailler dans le premier script sont passées à l'autre script.

La solution à cela est d'utiliser l'objet **HttpServletRequest** et la méthode ***setAttribute()***. La page de destination pourra alors accéder à ces informations via la méthode ***getAttribute()***.

Depuis la version 2.2 des servlets une nouvelle manière d'envoyer des informations a été mise en place. Vous pouvez directement mettre les paramètres dans l'url (méthode GET) que vous passer lors de la demande du **RequestDispatcher**.

Voici un exemple d'utilisation de cette méthode :

```
// Forward
void service(HttpServletRequest req, HttpServletResponse res) {
	String url = "/folder/page.jsp?param1=valeur1&param2=valeur2";
```

```
        RequestDispatcher dispat = getServletContext().getRequestDispatcher(url);
        dispat.forward(req, res);
}
```

6. Hébergeurs gratuits

http://developpeur.journaldunet.com/tutoriel/jav/011124jav_jspfreehost.shtml