Développement d'applications Web avec J2EE

Servlets et JSP

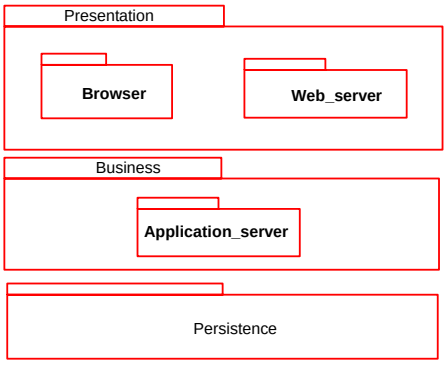
1538

Architecture d'une application Web J2EE

Servlets
Java Server Pages

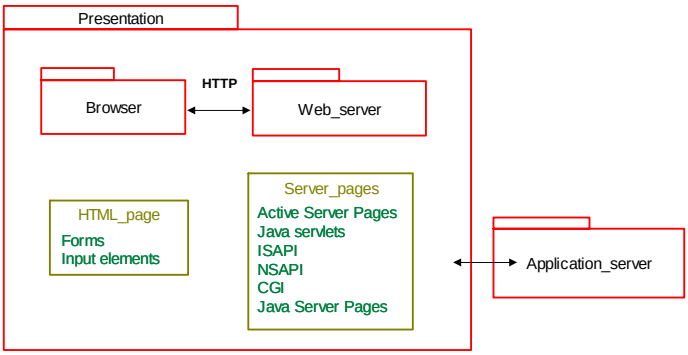
Architecture des applications Web

Il s'agit d'un style d'architecture client / serveur trois tiers



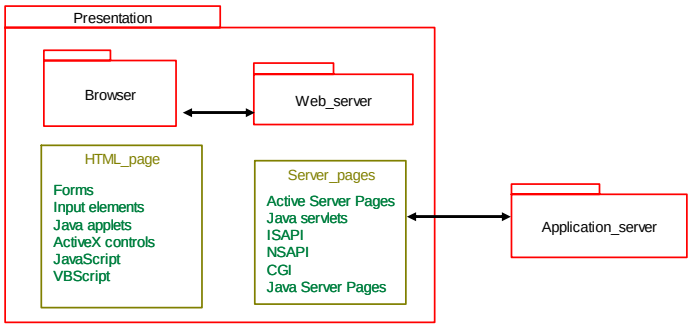
Client léger

Les composants majeurs du tier présentation résident sur le serveur Web

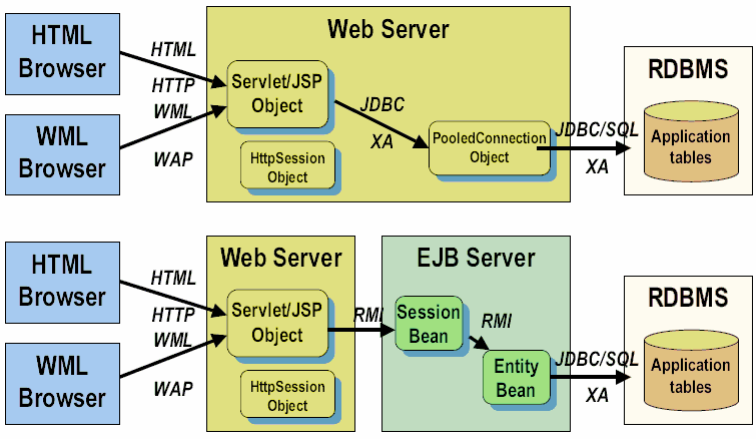


Client lourd

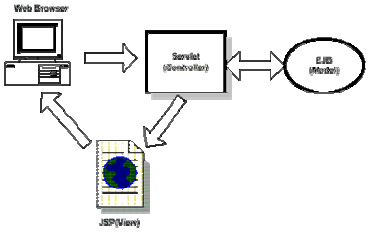
Des scripts et des modules sont exécutés sur le *browser*



Architecture d'une application Web J2EE



Pattern MVC 2



Le traitement des pages Web est vu comme la combinaison de trois fonctions :

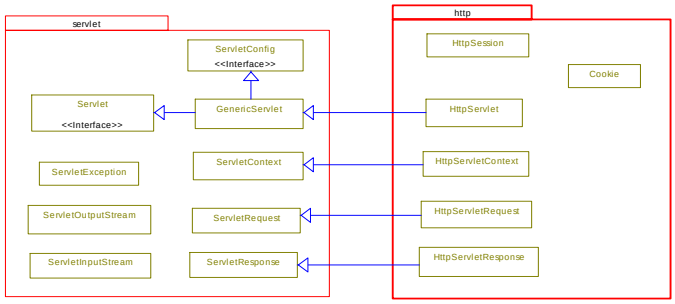
1) Perception de la requête entrante
2) Recherche ou génération de la réponse
3) Envoi de cette réponse à l'appelant

Architecture d'une application Web J2EE

Servlets
Java Server Pages

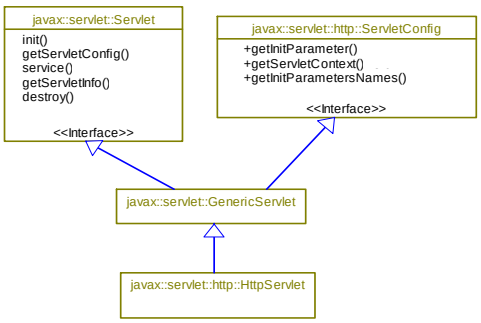
L'API Servlet

L'API Servlet est implémentée dans les packages javax.servlet et javax.servlet.http



Principales classes de l'architecture Servlet

Les deux classes principales de l'architecture Servlet sont les classes GenericServlet et HttpServlet



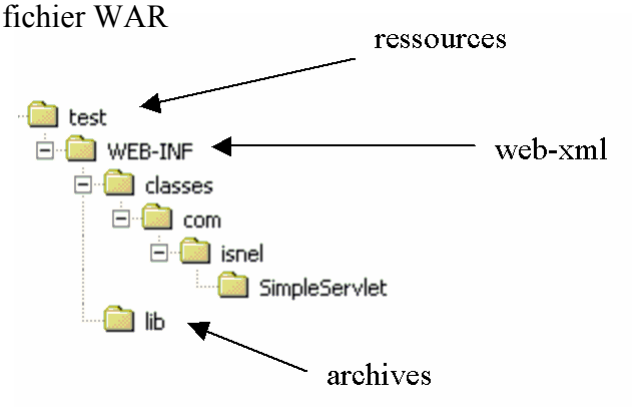
Exemple simple de servlet

```
public class SimpleServlet extends HttpServlet {

        public void doGet(HttpServletRequest request, HttpServletResponse
response)  throws ServletException, IOException {
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                out.println("<html>");
                out.println("<head><title>Servlet simple</title></head>");
                out.println("<body>");
                out.println("<p>Votre adresse est : " +
request.getRemoteAddr()+ "</p>");
                out.println("</body></html>");
                out.close();}}
```


Déploiement sur le serveur Web

L'application est assemblée et déployée sous forme d'un fichier WAR



Répertoire WEB-INF

Se place au niveau de la *racine* de l'application.

Contient les ressources utilisées par le serveur Web pour déployer et contrôler l'application

- descripteur de déploiement web.xml
- classes de l'application (le répertoire classes constitue le *classpath* de l'application)
- classes utilitaires (.jar dans lib)

Fichier de configuration web.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>
   <servlet>
      <servlet-name>SimpleServlet</servlet-name>
      <servlet-class>com.isnel.servlet.SimpleServlet</servlet-class>
    </servlet>

   <servlet-mapping>
      <servlet-name>SimpleServlet</servlet-name>
      <url-pattern>/ip</url-pattern>
   </servlet-mapping>

</web-app>
```

Requêtes et réponses HTTP

La plupart des clients Web utilisent le protocole HTTP pour communiquer avec un serveur Web J2EE.

La méthode GET permet de demander un document au serveur à partir d'une localisation spécifique. Des données peuvent être transmises dans l'URL spécifiée.

Avec la méthode POST les données sont transmises dans le *corps* de la requête. Cette méthode est généralement utilisée avec des formulaires à remplir.

Requêtes et réponses HTTP (2)

Soit l'URL http://host:80/?x=a&y=b

Le navigateur l'interprète de la manière suivante :

- http:// : utiliser HTTP
- host : contacter un ordinateur de nom host
- :80 : se connecter sur le port 80 de cet ordinateur
- /?x=a&y=b : paramètres x et y transmis au serveur web

Message envoyé au serveur web :

```
GET / HTTP/1.1
Accept: image.gif, image.jpeg, …
Accept-language: en-us
…
```

Traitement le serveur Web

Un serveur Web J2EE convertit toute requête HTTP en un objet **HttpServletRequest** qu'il transmet à la servlet identifiée par l'URL

La servlet instancie en retour un objet **HttpServletResponse** que le serveur convertit en une réponse HTTP renvoyée au client.

Association URL / Servlet

Le serveur Web utilise des « URL patterns » pour associer une servlet à une requête HTTP (servlet mapping)

Ces URL patterns sont définis dans le descripteur de déploiement web.xml requis par le serveur web pour le déploiement d'une application web

Exemple de pattern : /uri *.do

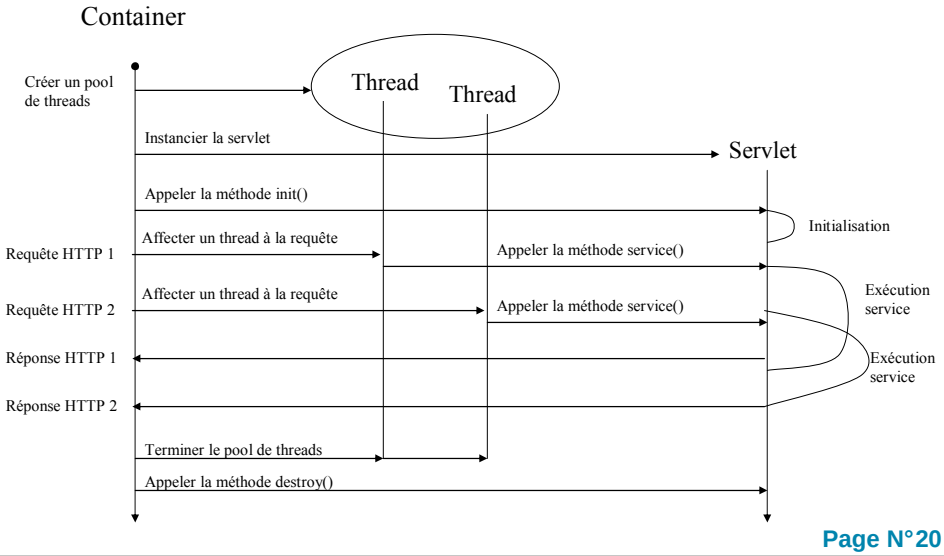
Cycle de vie d'une servlet

Le cycle de vie d'une servlet est le suivant :

- 1. La servlet est chargée et instanciée par le conteneur (serveur Web)
- 2. La méthode **init()** de la servlet est appelée par le conteneur avant que toute requête ne soit traitée
- 3. La servlet traite ensuite les requêtes. Chaque requête est traitée au moyen de la méthode **service()**
- 4. La servlet est enfin détruite par le garbage collector lorsque l'application Web qui la contient se termine. Juste avant la destruction de la servlet, sa méthode **destroy()** est appelée.

La méthode service est toujours appelée dans un nouveau thread. Cette méthode doit donc être réentrante (thread-safe)

Cycle de vie d'une servlet (2)



Méthodes doGet() et doPost()

Une classe spécialisant GenericServlet doit implémenter la méthode :
void service (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException

Une classe spécialisant HttpServlet n'a pas besoin d'implémenter la méthode service(). En effet, la classe HttpServlet contient déjà une implémentation de cette méthode.

Lorsque la méthode service() est invoquée, elle lit dans la requête HTTP le type de la méthode utilisée et détermine quelle méthode de la servlet doit être appelée.

- Si la méthode HTTP est GET, la méthode appelée est doGet().
- Si la méthode HTTP est POST, la méthode appelée est doPost()

Contexte d'une application web

Le contexte d'une application web est un objet de classe ServletContext qui est partagé par toutes les servlets de cette application.

Cet objet peut être obtenu via la méthode getServletContext().

Contexte d'une application web (2)

Les deux principales méthodes de ServletContext sont :

- setAttribute(nom, objet) qui associe un objet à un nom dans le ServletContext
- getAttribute(nom), qui retourne l'objet référencé

ServletContext permet également à une servlet d'accéder aux ressources statiques définies dans l'application en utilisant les méthodes getResource() et getResourceAsStream()

Contexte d'une application web (3)

Exemple :

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

ServletContext context = getServletContext();

      String userName = (String) context.getAttribute("USERNAME");

//     Si l'attribut est absent, création d'un nouvel attribut
       if (userName == null) {
          userName = new String("Aristote");
          context.setAttribute("USERNAME", userName);
       }
```

Paramètres d'initialisation (1)

Les paramètres d'initialisation d'une servlet doivent être définis avec des balises <init-param> dans le fichier web.xml

Exemple :

```
<servlet>
    <servlet-name>ParamsServlet</servlet-name>
    <servlet-class>com.isnel.InitServlet.ParamsServlet</servlet-class>
    <init-param>
                <param-name>premierParametre</param-name>
                <param-value>valeurDuPremierParametre</param-value>
    </init-param>
</servlet>
```


Paramètres d'initialisation (2)

La récupération des paramètres d'initialisation peut se faire dans les méthodes init(), doGet() ou doPost() de la servlet :

Exemple :

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

ServletConfig config = getServletConfig();

for (Enumeration e = config.getInitParameterNames(); e.hasMoreElements();) {
        String p = (String) e.nextElement();
        out.println("<p>" + p + " : " + config.getInitParameter(p));
    }
```

HttpServletRequest (1)

Chaque requête traitée par une servlet est un objet de classe HttpServletRequest

L'interface d'une requête permet d'accéder à toutes les informations incluses dans la requête HTTP :

- Cookies
- Headers
- Paramètres
- Caractéristiques (méthode utilisée, scheme http ou https, …)
- L'URI (URI, contextPath, servletPath, …)
- Les informations sur le client, sa localisation
- Attributs (en général, références à des beans ou value objects)

HttpServletRequest (2)

Si l'URL d'une requête est de la forme générale :

http://[host]:[port][request path]?[query string]

Request path est composé :

- d'un « context path » : / suivi du nom de la racine de l'application Web
- d'un « servlet path » : / suivi de l'alias du composant activé par la requête (URL pattern)

HttpServletRequest (3)

Exemple de récupération des paramètres

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

ServletConfig config = getServletConfig();

out.println("<p>Type mime de la requête :" + request.getContentType() + "</p>");
out.println("<p>Protocole de la requête :" + request.getProtocol() + "</p>");
out.println("<p>liste des parametres de la requete : </p>");
for (Enumeration e = request.getParameterNames(); e.hasMoreElements();) {
        Object p = e.nextElement();
        out.println("<p>  nom : " + p + " valeur : " + request.getParameter("" +
p) + "</p>");}
```


HttpServletResponse

La réponse générée par une servlet est un objet de classe HttpServletResponse

L'interface d'une réponse permet d'y enregistrer :

- des en-têtes (HTTP headers) dont le plus important est le content-type (généralement text/html pour une page HTML et text/xml pour un document XML)
- des cookies (cf. slide xxx)
- un code erreur
- la redirection vers une autre ressource (méthode sendRedirect()), …

Noter que toutes les méthodes permettant de manipuler les headers ou les cookies doivent être exécutées avant la première écriture dans le buffer de sortie.

Gestion de session

Le protocole HTTP est *stateless*. Autrement dit, il ne permet pas de lier deux requêtes successives d'un même client.

Le suivi de session consiste à stocker les informations concernant le client sur le serveur, et à les retrouver à chaque requête grâce à un identifiant présent dans la requête et sur le serveur.

Il existe plusieurs façons standards de passer un identifiant en paramètre dans la requête :

- en ajoutant à l'URL des variables d'identification
- en passant en paramètre un champ de formulaire caché
- en utilisant les cookies
- en utilisant un objet HttpSession

Récriture d'URL

La méthode de récriture d'URL consiste à passer en paramètre dans l'URL un identificateur unique de session.

Si par exemple l'identificateur de session est *xxx* , les URL transmises dans les pages web du serveur pourraient être :

Lien

Ainsi, avec cette technique, toutes les pages du client doivent contenir une URL dont l'ID de session a été ajouté dynamiquement, ce qui signifie que toutes les pages doivent être dynamiques.

Champs cachés de formulaire

La technique consiste à passer en paramètre un champ de formulaire contenant l'identifiant de session; elle est similaire à la récriture d'URL, à la différence près que les données sont envoyées par la méthode POST (plus discrète) :

Exemple :

<Input type="hidden" name="id" value="674684641">

Cette méthode implique par contre l'utilisation systématique d'un formulaire pour passer d'une page à une autre.

Cookies (1)

Les cookies sont de petits fichiers de type texte (4 Ko max) stockés sur le disque du client, permettant de mémoriser des couples (clés,valeur)

Les cookies sont automatiquement envoyés dans les en-têtes HTTP lors de chaque requête du client.

Lorsque le serveur désire créer un cookie sur le navigateur du client, il lui suffit d'envoyer des instructions dans les en-têtes de la réponse HTTP.

Java fournit une classe permettant de gérer de façon transparente les cookies, il s'agit de la classe Cookie.

Cookies (2)

Il est possible de stocker dans un cookie un identifiant de session de la manière suivante :

```
// création du cookie
C = new Cookie("id",“compteur");
// définition de la limite de validité
C.setMaxAge(24*3600);   (secondes)
// envoi du cookie dans la réponse HTTP
response.addCookie(C);
```

La récupération des cookies par le serveur se fait à l'aide de request.getCookies() et getName(), getValue() de Cookie

Objet HttpSession (1)

Un objet HttpSession permet de mémoriser sur le serveur des informations relatives à un client.

La durée de vie d’un objet HttpSession peut être spécifiée dans le descripteur de déploiement

```
<session-timeout>x minutes</session-timeout>
```

La méthode getSession(boolean x) de ***HttpServletRequest*** avec x = true crée une session relative au client de la requête

Gestion de session (2)

La méthode *getSession()* permet de retrouver la session relative à ce client (l'identification est faite par défaut par cookies ou sinon par récriture d'URL).

Les méthodes *setAttribute* (), getAttribute () et removeAttribute () de l'objet *HttpSession* permettent d'y gérer des informations

Ces méthodes n'ont pas à être protégées contre les accès concurrents

Si une servlet génère une URL (par exemple pour une redirection), utiliser la méthode encodeURL(String url) de HttpServletResponse pour générer cette URL

Enchaînement des servlets

Il est possible de faire communiquer des servlets par l'objet RequestDispatcher obtenu à partir de l'objet ServletContext.

Les méthodes forward et include servent respectivement à la transmission du contrôle ou à l'intégration de résultats de requêtes vers d'autres servlets.

Exemple :

```
public void doGet( HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
        RequestDispatcher rd = getServletContext()
                .getRequestDispatcher("/servlet") ;
        rd.include(  req , res ) ;
        res.getWriter().println( «  Fin ») ; }
```

Listeners (1)

Il est possible de contrôler les événements se produisant dans le cycle de vie d'une application Web ou d'une servlet en utilisant des objets « listeners »

La classe de ces objets doit implémenter une interface Listener telle que ServletContextListener ou HttpSessionListener

Quand une opération d'un objet listener est invoquée, il lui est passé un événement avec l'information associée

Listeners (2)

Objet	Evénement	Interfaces et classes
Web context	Initialisation and destruction	`ServletContextListener` `ServletContextEvent`
	Attribut ajouté, retiré, ou remplacé	`ServletContextAttributeListener` `ServletContextAttributeEvent`

Session	Création, invalidation, activation, passivation, et timeout	`HttpSessionListener,`
		`HttpSessionActivationListener`
		`HttpSessionEvent`

	Attribut ajouté retiré, ou remplacé	`HttpSessionAttributeListener` `HttpSessionBindingEvent`

Filtrage des requêtes et des réponses (1)

Un filtre peut intercepter une requête avant l'appel de la servlet ou intercepter la réponse de la servlet afin de les transformer (en-tête et corps).

Un filtre doit implémenter l'interface Filter dont l'opération doFilter() prend en paramètre la requête, la réponse et une chaine d'autres objets (instance de FilterChaine).

Cette méthode peut transformer la requête et la réponse et invoquer le cas échéant un successeur dans la chaîne de filtres

Filtrage des requêtes et des réponses (2)

Une caractéristique importante des filtres est de pouvoir être associé à toute servlet de manière transparente à celle-ci, dans le descripteur de déploiement.

Un filtre peut par exemple être utilisé pour valider une requête avant sa transmission à une servlet, ou tracer le réponse retournée …

Servlets et bases de données

Dans une architecture multi-tiers il n'est pas normal que les servlets accèdent directement à une base de données.

Ce sont les composants métiers qui accèdent aux bases de données, les échanges de données avec les servlets se faisant sous forme de beans (data transfert objects, value objects).

Il est toutefois important de savoir comment les composants métiers d'une application web peuvent utiliser les services d'une base de données.

Datasources (1)

JDBC 2.0 Standard Extension a introduit des propriétés très utiles pour l'utilisation d'une base de données :

- Support de JNDI pour les datasources JDBC
- Connection Pooling
- Rowsets
- Distributed Transactions

Les classes correspondantes apparaissent sous javax, en particulier dans le package javax.sql

DataSources (2)

La classe DataSource fournit une alternative à la classe DriverManager pour assurer la connection à une base de données.

Elle représente une connection physique à une base de données. Les fournisseurs de pilotes proposent tous une implémentation de l'interface DataSource.

L'utilisation d'un objet DataSource est obligatoire pour pouvoir utiliser un pool de connection et les transactions distribuées.

Une fois créé un objet de type DataSource peut être enregistré dans un service de nommage. Il suffit alors d'utiliser JNDI pour obtenir une instance de classe DataSource.

DataSources (3)

Pour une application Web, la datasource est généralement déclarée dans le descripteur web.xml ou le fichier de configuration de l'application utilisé par le serveur Web

Exemple de déclaration dans le fichier web.xml :

```
<resource-ref>
  <res-ref-name>jdbc/lib</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

DataSources (4)

Une instance de datasource peut alors être retrouvée en utilisant l'interface JNDI :

```
try {
        InitialContext ic = new InitialContext();
        Context envCtx = (Context) ic.lookup("java:comp/env");
        DataSource ds = (DataSource) envCtx.lookup("jdbc/lib");
        con = ds.getConnection();
    } catch (Exception ex) {
        throw new Exception("Couldn't open connection to
                                database: " + ex.getMessage()); }
```


Architecture J2EE
Servlets

Java Server Pages

La technologie JSP

La technologie JSP permet de créer des composants Web statiques et dynamiques.

JSP exploite toutes les possibilités des servlets mais avec une approche mieux adaptée pour générer le contenu statique des pages Web

La technologie JSP offre en particulier :

- Un langage pour décrire le traitement des requêtes et la génération des réponses
- Un langage pour accéder aux objets du côté serveur
- Des mécanismes permettant d'étendre le langage des JSP

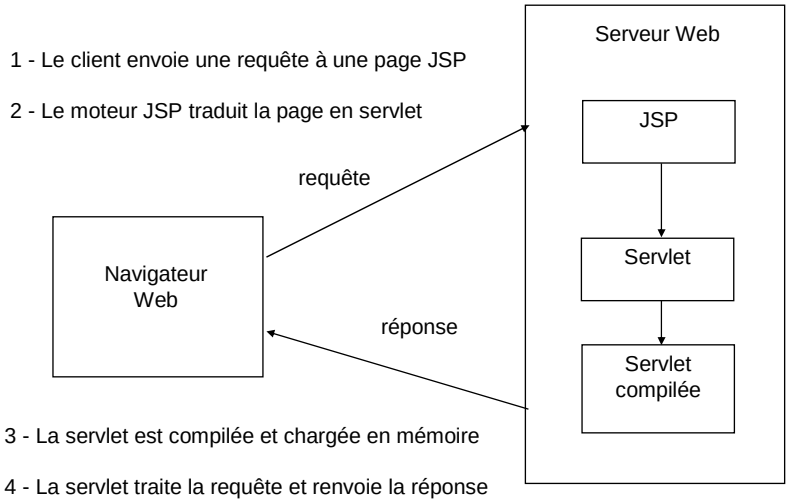
Qu'est-ce qu'une JSP ?

Une page JSP est un document qui contient deux types d'informations :

- Informations statiques décrites en HTML, SVG, WML, XML, …
- Éléments dynamiques JSP

Les éléments JSP peuvent être décrits dans deux syntaxes : soit standard, soit XML

Les étapes d'une requête JSP



Composants d'une JSP

Ces composants sont :

- *les directives*
- *les scripts*
- *les objets implicites*
- *les actions* correspondant à des balises (tags) standards ou spécifiques

Les commentaires JSP

Syntaxe : <%-- commentaire --%>

Contrairement aux commentaires HTML (de la forme <!-- commentaire -->), les commentaires JSP ne sont pas retournés au browser

Directives

Les directives fournissent des informations globales au sujet de la page

Syntaxe : <%@ directive option %>

Les directives sont utilisées au moment de la compilation des JSP (et non pas à la réception des requêtes)

Il existe trois directives possibles :

- page : informations relatives à la page
- include : identifie des fichiers à inclure
- taglib : indique que la page utilise une bibliothèque de balises

La directive page

Quelques attributs possibles :

```
<%@ page language="java"%>
<%@ page import="java.util.*, java.net.*" %>
<%@ page contentType="text/plain; charset=ISO-8859-1" %>
<%@ page session="true | false " %>
<%@ page errorPage="URL" %>
<%@ page isErrorPage="true | false" %>
<%@ page isThreadSafe="true | false" %>
```


La directive include

Valeurs possibles :

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

pour se référer à la racine de l'application :

```
<%@ include file="/nom.html" %>
```

L'attribut file peut référencer une page HTML ou JSP, mais la ressource doit se trouver dans la même application

L'insertion se fait au moment de la traduction de la JSP en servlet (à la différence de l'action <jsp:include>)

Scripts JSP

Permettent d'insérer du code java dans une page HTML

Il existe pour les scripts trois types de composants :

- Les déclarations
- Les expressions
- Les scriplets

Déclarations

Sont employées pour déclarer des variables et des méthodes

Les déclarations ne sont évaluées que lorsque la JSP est chargée pour la première fois, avant l'évaluation des expressions et des scriplets

Les éléments déclarés peuvent ensuite être utilisés dans d'autres déclarations, des expressions ou des scriplets

Syntaxe : <%! déclaration1; declaration2; … %>

La portée des variables est la page

Expressions

Les expressions JSP sont des éléments dont le texte, après évaluation par le conteneur au moment des requêtes, est remplacé par le résultat de l'évaluation de cette expression

Le résultat de l'évaluation doit être un String

Syntaxe : <%= expression %>

Scriplets

Les scriplets peuvent contenir pratiquement toutes les instructions du langage référencé par la directive language.

Elles sont évaluées lors du traitement des requêtes et ont accès à tous les composants JSP

Le code des scriplets se retrouve dans le corps de la méthode _jservice() de la servlet générée

Syntaxe : <% code %>

Les objets implicites

Dans une expression JSP ou une scriplet il est possible d'accéder à des objets définis implicitement :

- **request**, qui représente l'objet HttpServletRequest
- **response**, qui représente l'objet HttpServletResponse
- **session**, qui représente l'objet HttpSession
- **out**, qui représente l'objet response.getWriter()
- **application**, qui représente l'objet ServletContext
- **config**, qui représente l'objet ServletConfig
- **pageContext**, qui permet de retrouver des infos relatives au contexte de la servlet générée
- **page**, qui représente la page elle-même (*this* de la servlet)
- **exception**, qui permet d'accéder à une exception lancée par la JSP et non attrapée. Cet attribut n'est accessible que dans les JSP dont l'attribut isErrorPage a la valeur true

Traitement des erreurs JSP

Les exceptions qui ne peuvent être traitées fans la JSP provoquent la transmission de la requête ainsi que de l'exception à une page spécifique. Cette page doit comporter une directive page avec l'attribut isErrorPage = true

Exemple de contenu pour une page error.jsp :

```
<%@ page isErrorPage = "true" %>
  … <p> Exception : <%= exception.getMessage() %>  </p>
```

La JSP susceptible de lever une exception doit comporter la directive **<%@ page errorPage = "error.jsp" %>**

Configuration des JSP

L'élément jsp-config est un sous-élément de web-app du descripteur web.xml qui est utilisé pour fournir une information globale pour les JSP d'une application.

Un élément jsp-config a deux sous-éléments : **taglib** and **jsp-property-group**, qui précisent respectivement des informations pour le mapping des taglibs et des groupes de JSP

Un groupe JSP définit une collection de propriétés qui s'appliquent à un ensemble de JSP.

Les propriétés qui peuvent être précisées dans un élément jsp-property-group sont en particulier :

- un URL Pattern
- la possibilité d'évaluation ou non des expressions EL
- la possibilité d'évaluation ou non des expressions JSP
- l'encodage des informations dans la page
- l'inclusion automatique d'un Prelude et d'un Coda

Les balises JSP

Une balise JSP référence, en utilisant la syntaxe XML, une action qui peut être insérée dans une page JSP.

Exemple :

```
<jsp:include page="url" flush='true"
   <jsp:param … />
</jsp:include>
```

Les actions sont exécutées au moment de la réception des requêtes.

Balises JSP standards

Trois balises prédéfinies concernent l'utilisation des JavaBeans :

<jsp:useBean>,
<jsp:setProperty>
<jsp:getProperty>

D'autres concernent les tâches génériques comme :

- <jsp:include> pour l'inclusion **dynamique** de composants dans une JSP
- <jsp:forward> pour le chaînage des JSP
- <jsp:param> pour le traitement des paramètres
- <jsp:plugin> pour le téléchargement d'une applet ou d'un JavaBean,
- …

Balise <jsp:useBean>

L'action correspondant à la balise <jsp:useBean> recherche une instance de JavaBean et la crée si nécessaire, avec la portée et l'identificateur indiqués

Exemple :

```
<jsp:useBean id="myBean" scope="session" class="com.isnel.HelloBean"/>
```

Le seul attribut obligatoire est *id* qui permet de définir le bean dans la portée indiquée (page par défaut)

La valeur de ces attributs ne peut pas être modifiée au moment de l'exécution de la JSP

Balises <jsp:setProperty> et <jsp:getProperty>

Ces balises permettent de modifier et de lire la valeur d'une propriété d'un bean

Exemple :

```
<html>
<body>
        <jsp:useBean id="myBean"  class="com.isnel.HelloBean"/>
        <jsp:setProperty name="myBean" property="*"/>
                Hello,
        <jsp:getProperty name="myBean" property="nom"/> !
</body>
</html>
```


Portée des beans

L'attribut "scope" de <jsp:useBean> définit la portée d'un bean

- page : le bean est disponible pendant la génération de la page en réponse à une requête. Il est un attribut de pageContext. N'est propagé ni par <jsp:include> ni par <jsp:forward> (scope par défaut)
- request : le bean est un attribut de request propagé par <jsp:include> ni par <jsp:forward>.
- session : le bean est un attribut de session
- application : le bean est un attribut de servletContext

Balises <jsp:include> et <jsp:forward>

La balise <jsp:include> permet d'inclure dans une JSP des composants Web statiques ou le produit de composants dynamiques

Syntaxe :

```
<jsp:include page="url" flush='true"
   <jsp:param … />
</jsp:include>
```

La balise <jsp:forward> permet de transmettre la requête en cours à une autre ressource résidant sur le même serveur (ressource statique, JSP ou servlet)

Balises <jsp:plugin>

La balise permet à une JSP de générer le code nécessaire pour demander au navigateur le téléchargement et l'exécution d'une applet ou d'un JavaBean.

Exemple :

```
<jsp:plugin type="applet"
code="VM/VM_Presentation/VM_Applet.class"
codebase="applet" jreversion="1.5"
archive="vm.jar" width="400" height="200">
        <jsp:params>
             <jsp:param name="model_name"
value="VM.VM_Business.F_VM" />
                …
        </jsp:params>
        <jsp:fallback>Impossible de charger l'applet
VM_Applet.class  </jsp:fallback>
</jsp:plugin>
```

JSP Standard Tag Library (JSTL)

Les scriptlets consistent à imbriquer du code java dans du code HTML, ce qui tend à compliquer fortement l'écriture et la maintenance des JSP.

L'usage des accolades dans cet exemple illustre cette difficulté :

```
<% if (user.getRole() == "member")) { %>
<p>Welcome, member!</p>
<% } else { %> <p>Welcome, guest!</p> <% } %>
```


JSP Standard Tag Library (2)

La librairie JSTL 1.1 (JSP 2.0) comporte cinq librairies de balises (core, format, xml, sql and function).

- *core* fournit des actions permettant de gérer les données à partir de variables locales (scoped variables) et de définir des itérations et des actions conditionnelles. Elle permet également de créer et de gérer des URL
- *format* permet de formater les données, typiquement des nombres et des dates, en prenant en compte leur l'internationalisation.
- *XML* permet de manipuler des données représentées en XML
- *SQL* sert à définir des datasources et des requêtes SQL.
- *functions* fournit des actions pour manipuler les chaînes de caractères

JSP Standard Tag Library (3)

Les expressions utilisées dans les JSP sont dites "*request-time attribute values"* et sont le seul mécanisme permettant de spécifier dynamiquement la valeur des attributs

Exemple :

```
<jsp:setProperty name="user" property="timezonePref"
        value='<%= request.getParameter("timezone") %>'/>
```

Les balises JSTL offrent un autre mécanisme pour spécifier dynamiquement la valeur des attributs en utilisant le langage EL (*Expression Language*).

Expression Langage (EL)

EL fournit des identifiants, des accesseurs et des opérateurs pour retrouver et manipuler des données résidant dans le serveur web.

EL est partiellement basé sur EcmaScript et XPath.

Les expressions EL sont délimitées avec le caractère $ et des accolades comme dans cet exemple :

```
<c:out value="Hello ${user.firstName} ${user.lastName}"/>
```

Variables locales EL

L'API JSP permet à travers l'action <jsp:useBean> de mémoriser et de retrouver des objets dans différentes portées (Page, Request Session et Application scope).

JSTL étend cette possibilité en fournissant des actions supplémentaires pour mémoriser et retrouver des objets dans ces différentes portées.

De plus, EL permet de considérer ces objets comme des variables locales (scoped variables).

Tout identifiant apparaissant dans une expression EL qui ne correspond pas aux objets implicites EL est supposé référencer un objet stocké dans l'une des portées des JSP

Objets implicites EL (1)

Ne pas confondre ces objets implicites avec ceux des JSP (qui sont seulement au nombre de 9). Un seul objet implicite est commun : pageContext. Les autres objets implicites JSP sont toujours accessibles dans les expressions EL

Category	Identifier	Description
JSP	`pageContext`	The `PageContext` instance corresponding to the processing of the current page
Scopes	`pageScope`	A `Map` associating the names and values of page-scoped attributes
	`requestScope`	A `Map` associating the names and values of request-scoped attributes
	`sessionScope`	A `Map` associating the names and values of session-scoped attributes
	`applicationScope`	A `Map` associating the names and values of application-scoped attributes
Request parameters	`param`	A `Map` storing the primary values of the request parameters by name
	`paramValues`	A `Map` storing all values of the request parameters as `String` arrays
Request headers	`header`	A `Map` storing the primary values of the request headers by name
	`headerValues`	A `Map` storing all values of the request headers as `String` arrays
Cookies	`cookie`	A `Map` storing the cookies accompanying the request by name
Initialization parameters	`initParam`	A `Map` storing the context initialization parameters of the Web application by name

Objets implicites (2)

Tous les objets implicites EL (sauf pageContext) sont des maps.

Les quatre premiers permet de rechercher des identifiants dans les différentes portées sans reposer sur le processus séquentiel utilisé par défaut par les JSP

Les cinq suivants représentent les paramètres et headers des requêtes. Comme HTTP permet à ceux-ci d'etre multi-valués, il y a une paire de maps pour chacun.

L'objet implicite cookie fournit un accès aux cookies.

Le dernier objet implicite initParam représente les paramètres d'initialisation du contexte de l'application spécifiés dans le fichier web.xml

Accesseurs

EL fournit deux opérateurs pour accéder aux propriétés des objets ou aux éléments des collections.

L'opérateur . permet d'accéder aux propriétés des objets en utilisant les conventions des java beans

Par exemple, l'expression ${user.address.city} retourne la propriété city de l'objet adress d'un objet user.

L'opérateur [] permet de retrouver les éléments d'une collection en utilisant un index (collections List) ou une clé (collections Map). La valeur de l'indice ou de la clef peuvent être le résultat de l'évaluation d'une expression.

Les accesseurs ne lèvent pas d'exceptions comme en java lorsqu'ils sont appliqués à null, mais retournent null.

Opérateurs

Category	Operators
Arithmetic	+, -, *, / (or div), % (or mod)
Relational	== (or eq), != (or ne), < (or lt), > (or gt), <= (or le), >= (or ge)
Logical	&& (or and), || (or or), ! (or not)
Validation	empty

L'opérateur empty est très utile pour tester si une expression retourne null, si une collection est vide ou si un String est de longueur 0

Librairie de balises JSP

Une librairie de balises JSP est un document XML (.tld) qui contient les informations générales et spécifiques à chaque balise

Elle associe une classe gestionnaire (TagHandler) à chaque balise

Elle permet à des outils de prendre connaissance des balises disponibles

La directive taglib

La directive taglib permet d'indiquer que la JSP utilise une bibliothèque particulière de balises, identifiée de manière non ambiguë par son URI et un préfixe précisant la bibliothèque d'où elle provient

La syntaxe de cette directive est :

<%@ taglib uri = "uri de la bibliothèque" prefix = "prefixe" %>

Exemple :

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


Librairies standards JSP

Area	Subfunction	Prefix
Core	Variable support	c
	Flow control	
	URL management	
	Miscellaneous	
XML	Core	x
	Flow control	
	Transformation	
I18n	Locale	fmt
	Message formatting	
	Number and date formatting	
Database	SQL	sql
Functions	Collection length	fn
	String manipulation	

Core Tag library

Area	Function	Tags	Prefix
Core	Variable support	remove set	c
	Flow control	choose when otherwise forEach forTokens if	
	URL management	import param redirect param url param	
	Miscellaneous	catch out	

Balise <c:set>

<c:set var="*name*" scope="*scope*" value="*expression*"/>

Permet de créer et d'initialiser des variables locales dans la portée indiquée (page, si la portée n'est pas indiquée)

Exemples :

<c:set var="timezone" scope="session" value="CST"/>
<c:set var="square" value="${param['x'] * param['x']}"/>

La valeur de la variable locale peut aussi être spécifiée dans le corps de la balise :

<c:set var="timezone" scope="session">CST</c:set>

Balise <c:out>

<c:out value="*expression*" default="*expression*" escapeXml="*boolean*"/>

Cette action évalue l'expression spécifiée et imprime le résultat.

La valeur de l'attribut optionnel default est imprimée si l'expression vaut null ou un String vide.

Les actions <c:set> et <c:out> peuvent être imbriquées pour fournir une valeur à une variable locale.

Exemple :

```
<c:set var="timezone" scope="session">
        <c:out value="${cookie['tzPref'].value}" default="CST"/>
</c:set>
```


Balises <c:if> et <c:catch>

<c:if test="expression" var="name" scope="scope" />

Les attributs optionnels var et scope permettent de mémoriser le résultat du test.

Exemple d'utilisation de <c:if> avec <c:catch> :

```
<c:catch var="exception">
        …
</c:catch>
<c:if test="${exception != null}">
        …
</c:if>
```

Balise <c:choose>

```
<c:choose>
        <c:when test="expression">
                body content
        </c:when>
        ...
        <c:otherwise> body content </c:otherwise>
</c:choose>
```

La balise <c:otherwise> est optionnelle, mais il doit y avoir au moins une balise <c:when>

Balise <c:forEach> (1)

<c:forEach var="*name*" items="*expression*" varStatus="*name*" begin="*expression*" end="*expression*" step="*expression*"> *body content* </c:forEach>

Permet d'itèrer sur une collection :

```
<c:forEach var="customer" items="${customers}"> Customer:
      <c:out value="${customer}"/>
</c:forEach>
```

Les attributs begin, end et step permettent de restreindre les éléments de la collection pris en compte :

```
<table>
      <tr><th>Value</th>
      <th>Square</th></tr>
      <c:forEach var="x" begin="0" end="10" step="2">
            <tr><td><c:out value="${x}"/></td> <td>
            <c:out value="${x * x}"/></td></tr>
       </c:forEach>
 </table>
```

Balise <c:forEach> (2)

L'attribut varStatus crée une variable locale instance de LoopTagStatus dont les propriétés sont les suivantes :

Property	Getter	Description
current	`getCurrent()`	The item (from the collection) for the current round of iteration
index	`getIndex()`	The zero-based index for the current round of iteration
count	`getCount()`	The one-based count for the current round of iteration
first	`isFirst()`	Flag indicating whether the current round is the first pass through the iteration
last	`isLast()`	Flag indicating whether the current round is the last pass through the iteration
begin	`getBegin()`	The value of the `begin` attribute
end	`getEnd()`	The value of the `end` attribute
step	`getStep()`	The value of the `step` attribute

Balise <c:url> (1)

<c:url value="*expression*" context="*expression*" var="*name*" scope="*scope*">
<c:param name="*expression*" value="*expression*"/>
...
</c:url>

Cette balise est utilisée pour :

- générer des URL absolues commençant par le path du contexte de la JSP
- la récriture d'une URL pour la gestion de session
- l'encodage du nom et de la valeur des paramètres d'une requête

L'attribut value fournit une base pour l'URL qui peut être précédée explicitement par la valeur de l'attribut context. Par défaut, c'est le contexte de la JSP fixé au moment du déploiement qui est utilisé.

Balise <c:url> (2)

La balise <c:param> permet d'ajouter des paramètres à l'URL générée

Exemple d'utilisation :

```
<c:url var="url" value="/order.do" >
    <c:param name="bookId" value="${book.id}" />
</c:url>
<strong><a href="${url}"> Catalogue </a></strong>
```


Balise <c:import>

```
<c:import url="expression" context="expression" charEncoding="expression" var="name"
scope="scope">
<c:param name="expression" value="expression"/>
   ...
</c:import>
```

Cette action permet d'insérer dans une JSP le contenu d'une autre ressource web. Cette ressource n'est pas forcément locale comme c'est le cas avec <jsp:include>

Exemple :

```
<c:catch var="exception">
   <c:import url="ftp://ftp.example.com/package/README"/>
</c:catch>
<c:if test="${not empty exception}"> Sorry, the remote content is not currently available.
</c:if>
```

XML Tag library

Area	Function	Tags	Prefix
XML	Core	out parse set	x
	Flow control	choose when otherwise forEach if	
	Transformation	transform param	

Format Tag library

Area	Function	Tags	Prefix
I18n	Setting Locale	setLocale requestEncoding	fmt
	Messaging	bundle message param setBundle	
	Number and Date Formatting	formatNumber formatDate parseDate parseNumber setTimeZone timeZone	

Balise <fmt:setLocale>

<fmt:setLocale value="*expression*" scope="*scope*" variant="*expression*"/>

Cette action permet de spécifier la localisation de l'utilisateur, les préférences de langage spécifiées dans le browser étant alors ignorées

Exemple :

```
<fmt:setLocale value="fr_CA" scope="session"/>
```

Une locale par défaut peut être spécifiée dans le fichier web.xml

Exemple :

```
<context-param>
        <param-name>javax.servlet.jsp.jstl.fmt.fallbackLocale</param-name>
        <param-value>en</param-value>
</context-param>
```


Balise <fmt:formatDate>

<fmt:formatDate value="*expression*" timeZone="*expression*" type="*field*" dateStyle="*style*" timeStyle="*style*" pattern="*expression*" var="*name*" scope="*scope*"/>

Seul l'attribut value est requis. Sa valeur doit être une instance de java.util.Date

L'attribut type peut valoir time, date ou both

Les attributs dateStyle et timeStyle peuvent valoir default, short, medium, long et full

L'attribut pattern permet de spécifier un format suivant les conventions de SimpleDateFormat. Un exemple de pattern est "MM/dd/yyyy"

Balise <fmt:formatNumber>

<fmt:formatNumber value="*expression*" type="*type*" pattern="*expression*"
currencyCode="*expression*" currencySymbol="*expression*"
maxIntegerDigits="*expression*" minIntegerDigits="*expression*"
maxFractionDigits="*expression*" minFractionDigits="*expression*"
groupingUsed="*expression*" var="*name*" scope="*scope*"/>

Seul l'attribut value est requis. L'attribut type peut valoir number, currency ou percentage

Exemple :

<ftm:formatNumber value="15" type="currency" currencySymbol="€"/>

La balise <fmt:parseNumber> analyse une valeur numérique fournie par son attribut value ou son corps et fournit une instance de Number.

Balise <fmt:setBundle>

<fmt:setBundle basename="*expression*" var="*name*" scope="*scope*"/>

Cette action permet de spécifier le ResourceBundle à utiliser pour internationaliser les messages.

Exemple :

<fmt:setBundle basename="com.isnel.messages.BookstoreMessages" />

Un ResourceBundle par défaut peut être spécifié dans le fichier web.xml

Exemple :

```
<context-param>
   <param-name>javax.servlet.jsp.jstl.fmt.localizationContext</param-name>
   <param-value>com.isnel.messages.BookstoreMessages</param-value>
 </context-param>
```

Balise <fm:message>

Deux syntaxes possibles. Seul l'attribut key est requis.

<fmt:message key="expression" bundle="expression" var="name" scope="scope"/> ou

```
<fmt:message key="expression" bundle="expression" var="name" scope="scope">
<fmt:param value="expression"/>
        ...
</fmt:message>
```

Exemple

```
<fmt:message key="message.key">
      <fmt:param value="${mailbox.userName}"/>
      <fmt:param value="${mailbox.messageCount}"/>
</fmt:message>
```

"Bienvenue {0}, votre boîte de réception {1,choice, 0#ne comporte aucun message | 1#comporte un message | 1<comporte {1} messages}.

SQL Tag library

Area	Function	Tags	Prefix
Database		setDataSource	sql
	SQL	query dateParam param transaction update dateParam param	

Functions Tag library

Area	Function	Tags	Prefix
Functions	Collection length	`length`	`fn`

	String manipulation	`toUpperCase, toLowerCase`
		`substring, substringAfter, substringBefore`
		`trim`
		`replace`
		`indexOf, startsWith, endsWith, contains, containsIgnoreCase`
		`split, join`
		`escapeXml`	

Développement de balises spécifiques

Une librairie de tags se compose d'un descripteur (Tag Library Descriptor) et d'un ensemble de classes java (tag handlers) implémentant l'interface Tag

Un "tag handler" est tout d'abord un java bean qui implémente l'interface Tag ou l'une de ses spécialisations (BodyTag, …)

Les classes TagSupport et BodyTagSupport fournissent une implémentation par défaut de ces interfaces.

Toutes les classes nécessaires pour implémenter un tag handler sont contenues dans le package javax.servlet.jsp.tagext

Descripteur de taglib

Il s'agit d'un fichier XML (.tld) qui décrit les différents tags de la librairie. Ce fichier peut être inclus dans le jar qui contient les tag handlers (il possède alors le nom suivant : "META-INF/taglib.tld").

Afin de pouvoir utiliser une taglib dans un fichier JSP, il faut la déclarer avec la directive **taglib**.

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

ou si le fichier tld est dans le Jar :

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

Il est possible de définir la taglib dans le fichier **web.xml** :

```
<taglib>
        <taglib-uri>taglib-URI</taglib-uri>
        <taglib-location>/WEB-INF/lib/taglib.jar</taglib-location>
</taglib>
```

Ainsi, dans les pages JSP, la directive **taglib** devient :

```
<%@ taglib uri="taglib-URI" prefix="tag-prefix" %>
```


Interfaces Tag et BodyTag

. javax.servlet.jsp.tagext.Tag

- *doStartTag() appelée à la rencontre du tag ouvrant*
- *doEndTag() appelée à la rencontre du tag fermant*

• **javax.servlet.jsp.tagext.BodyTag** qui spécialise Tag

- setBodyContent()
- *doInitBody()* appelée avant le traitement éventuel du contenu de la balise
- *doAfterBody()* appelée après le traitement éventuel du contenu de la balise

Exemple de tag handler

```
public class HelloTag extends TagSupport {
    private String name=null;
    public void setName(String value){name = value;}
    public String getName(){return(name);}

    public int doStartTag() {
      try {
         JspWriter out = pageContext.getOut();
          if (name != null) out.println("Hello " + name");
         else out.println("Hello World");
      } catch (Exception ex) {
         throw new Error("All is not well in the world.");}
      // Must return SKIP_BODY because we are not supporting
a body for this tag.
      return SKIP_BODY;
    }
```


Exemple de descripteur

```
 <tag>
      <name>hi</name>
      <tag-class>com.isnel.HelloTag </tag-
class>
      <body-content>empty</body-content>
      <info>
            This is a simple hello tag.
      </info>
      <attribute>
            <name>name</name>
            <required>false</required>
      </attribute>
</tag>
```

Exemple d'utilisation

```
<%@ taglib uri="/WEB-INF/tld/mytaglib.tld" prefix="sample" %>
    <html>
        <body>
            La balise vous dit :
            <font color=red>
              <sample:hi nom=''monNom''/>
            </font color>
        </body>
     </html>
```


Utilisation de l'api commons-logging dans les JSP

Commons-logging permet de générer des traces d'exécution à différents niveaux, y compris dans les JSP.

Pour mettre en œuvre commons-logging dans une JSP :

- Rendre les fichiers log4J.properties et commons-logging.properties accessibles dans le classpath de l'application (il suffit donc de les mettre dans le répertoire classes sous WEB-INF)
- Placer les fichiers taglibs-log.jar, log4j-1.2.9.jar et commons-logging.jar dans le répertoire lib
- Déclarer taglibs-log.jar dans la JSP

```
<%@ taglib uri="/WEB-INF/lib/taglibs-log.jar" prefix="logger" %>
```

Fichiers .tag

A l'instar des fichiers *.jsp pour les Servlets, les fichiers *.tag permettent de créer des tags sans avoir à générer du code HTML à l'intérieur d'un classe

La déclaration des attributs d'un tag dans un fichier *.tag est décrite directement dans ce dernier avec la directive **<%@ attribute %>**

<%@ attribute name="attribute-name"
required="true | false"
fragment="true | false"
rtexprvalue="true | false"
type="java.lang.String | a non-primitive type"
description="text" %>

Utilisation d'attributs JspFragment

Les JspFragment permettent de définir des attributs dans les tags qui ne sont rien d'autres que des fragments de code JSP (scriptless)

```
<jsp:invoke fragment="nomFragment"/>
```

Utilisation du tag (même s'il n'accepte pas de corps) :

```
<prefix:nomDuTag>
        <jsp:attribute name="nomFragment">
                code JSP du fragment
        </jsp:attribute>
</prefix:nomDuTag>
```