Présentation de la Java Standard Tag Library (JSTL)

par F. Martini (adiGuba)

Date de publication : 09/02/2005

Dernière mise à jour : 18/02/2007

La JSTL est une implémentation de Sun qui décrit plusieurs actions basiques pour les applications web J2EE. Elle propose ainsi un ensemble de librairies de tags pour le développement de pages JSP. Ce tutoriel décrit les différentes librairies de la JSTL.

Remerciement

Je tiens à remercier Ukyuu pour avoir pris le temps de relire ce tutoriel, ainsi que les multiples retours que j'ai eu de la part des lecteurs (**nicolas c**, **mimil77210**, **Kimael**, et d'autres que j'oublie peut-être : merci à tous).

Présentation

*Ce tutoriel est également disponible en version **PDF** :*

ftp://ftp-developpez.com/adiguba/tutoriels/j2ee/jsp/jstl/jstl.pdf

Mirroir : http://adiguba.ftp-developpez.com/tutoriels/j2ee/jsp/jstl/jstl.pdf

De nombreux frameworks facilitent le développement d'application J2EE (**Struts**, **Spring**, etc...). La plupart de ces frameworks proposent également des librairies de tags JSP facilitant la création de pages JSP. Il en résulte une multitude de librairies différentes pour des fonctionnalités similaires. La **JSTL** propose une librairie standard pour la plupart des fonctionnalités de base d'une application J2EE.

1.1 - Objectifs de la JSTL

Le but de la **JSTL** est de simplifier le travail des auteurs de page JSP, c'est à dire la personne responsable de la couche présentation d'une application web J2EE.

En effet, un web designer peut avoir des problèmes pour la conception de pages JSP du fait qu'il est confronté à un langage de script complexe qu'il ne maîtrise pas forcément.

La **JSTL** permet de développer des pages JSP en utilisant des balises XML, donc avec une syntaxe proche des langages utilisés par les web designers, et leur permet donc de concevoir des pages dynamiques complexes sans connaissances du langage Java.

Sun a donc proposé une spécification pour une librairie de tags standard : la **Java Standard Tag Library** (**JSTL**). C'est à dire qu'il spécifie les bases de cette librairie, mais qu'il laisse l'implémentation libre (de la même manière que pour les serveurs J2EE qui sont des implémentations de la spécification J2EE).

1.2 - Documentation

Pour plus d'information sur la **JSTL** vous pouvez consulter la page officielle chez Sun :

http://java.sun.com/products/jsp/jstl/

Ou la page officielle de la spécification de la **JSTL** :

https://jstl-spec-public.dev.java.net/

Ce tutoriel se base sur l'implémentation du projet **Jakarta** de la **JSTL 1.1**, qui est considéré comme l'implémentation de référence de la **JSTL**. Elle est disponible à l'adresse suivante :

http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html

Enfin, l'API des interfaces et classes de bases de la JSTL est disponible à l'adresse suivante :

http://java.sun.com/products/jsp/jstl/1.1/docs/api/index.html

Et la documentation des différents tags :

http://java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/

1.3 - Les versions

Actuellement, deux versions de la **JSTL** sont disponibles, avec les restrictions suivantes :

- La **JSTL 1.0** nécessite au minimum un conteneur **JSP 1.2** (**J2EE 1.3**).
- La **JSTL 1.1** nécessite au minimum un conteneur **JSP 2.0** (**J2EE 1.4**).

La **JSTL** se base sur l'utilisation des **Expressions Languages** en remplacement des **scriptlets** Java. Toutefois, ce mécanisme n'est disponible qu'avec le conteneur **JSP 2.0**. Ainsi, la **JSTL 1.0** propose deux implémentations :

L'implémentation de base intègre donc un interpréteur d'**Expressions Languages** afin de pouvoir utiliser toutes les possibilités des **Expressions Languages** dans un conteneur **JSP 1.1** ou **1.2** :

Librairie	URI	Préfixe
core	http://java.sun.com/jstl/core	c
Format	http://java.sun.com/jstl/fmt	fmt
XML	http://java.sun.com/jstl/xml	x
SQL	http://java.sun.com/jstl/sql	sql

> *Ces URI ne doivent pas être utilisées dans une application **J2EE 1.4** afin de ne pas rentrer en conflit avec l'interpréteur d'**EL** intégré dans les **JSP 2.0**. De plus cela interdit l'utilisation de **scriptlets** en tant que valeur des attributs des tags (seules les chaines de caractères sont autorisées) :*

La seconde version de la **JSTL 1.0** n'intègre pas d'interpréteur d'**Expressions Languages**. Cette implémentation se distingue par l'ajout des caractères "**_rt**" à la fin de l'URI et du préfixe, qui indique que la gestion des **EL** est éventuellement laissée au moteur JSP ("runtime") :

Librairie	URI	Préfixe
core	http://java.sun.com/jstl/core_rt	c_rt
Format	http://java.sun.com/jstl/fmt_rt	fmt_rt
XML	http://java.sun.com/jstl/xml_rt	x_rt
SQL	http://java.sun.com/jstl/sql_rt	sql_rt

> *Attention, si ces **URI**s sont utilisées dans des **JSP 1.1** ou **1.2**, les **Expressions Languages** ne seront pas interprétés.*

La **JSTL 1.1** n'apporte pas de changement majeur dans les librairies de tags, mis à part l'ajout d'une nouvelle librairie de fonctions **EL**.

De plus, comme elle se base sur les **JSP 2.0** qui intègre un moteur d'**Expressions Languages**, elle ne définit donc qu'une seule implémentation avec les URIs suivantes :

Librairie	URI	Préfixe
core	http://java.sun.com/jsp/jstl/core	c
Format	http://java.sun.com/jsp/jstl/fmt	fmt
XML	http://java.sun.com/jsp/jstl/xml	x

Librairie	URI	Préfixe
SQL	http://java.sun.com/jsp/jstl/sql	sql
Fonctions	http://java.sun.com/jsp/jstl/functions	fn

*Afin de les distinguer des URIs de la **JSTL 1.0**, la chaîne "**/jsp**" a été ajoutée afin d'indiquer que cette version se base sur le conteneur JSP pour interpréter les **Expressions Languages**.*

Ce tutoriel est basé sur la **JSTL 1.1**. Toutefois, mis à part la gestion des **Expressions Languages** et la librairie de fonctions, les différences entre les deux versions sont minimes...

*L'utilisation des **Expressions Languages** est nécessaire pour une utilisation optimale de la **JSTL**, consultez le tutoriel dédié aux **Expressions Languages** pour plus de détails :*

http://adiguba.developpez.com/tutoriels/j2ee/jsp/el/

1.4 - Configuration de la JSTL

Certaines librairies de la **JSTL** peuvent nécessiter une configuration propre à une application via le fichier **web.xml** en utilisant les **init-param**. Ainsi, pour exemple pour définir la source de donnée à utiliser par défaut, il faut renseigner le paramètre **javax.servlet.jsp.jstl.sql.dataSource** :

Configuration du web.xml

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

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
        version="2.4">

        <context-param>
                <param-name>javax.servlet.jsp.jstl.sql.dataSource</param-name>
                <param-value>jdbc/data</param-value>
        </context-param>

        ...

</web-app>
```

La classe **javax.servlet.jsp.jstl.core.Config** comporte la liste des différents paramètres possibles. Chacun de ces paramètres sont détaillés dans la section **Configuration** de chaque librairie de la librairie.

La valeur de ces paramètres peut être modifiée dynamiquement via la classe **Config** ou via des tags spécifiques décrits dans la librairie correspondante. La nouvelle valeur est alors stockées dans un des scopes de l'application, avec les influences suivantes :

Scope	Description
page	La nouvelle valeur n'affecte que la page JSP courante.
request	La nouvelle valeur affecte toute la requête courante (pages JSP forwardées/incluses compris).
session	La nouvelle valeur affecte toute la session de l'utilisateur.
application	La nouvelle valeur affecte tous les utilisateurs.

Lorsqu'une valeur de configuration est nécessaire, elle est d'abord recherchée dans les différents scopes (dans l'ordre naturel : **page**, **request**, **session** puis **application**). Si la valeur n'existe dans aucun des scopes, la valeur définit dans le **web.xml** est utilisée, ou une valeur par défaut le cas échéant.

2 - <c:/> : Librairie de base

Cette section et ses sous sections définissent les différentes actions de la librairie "**code**" de la JSTL. C'est à dire la librairie qui contient les actions de base d'une application web.

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

2.1 - Gestion des variables de scope

Cette section comporte les actions de base pour la gestion des variables de scope d'une application web :

- L'affichage de variable
- La création/modification/suppression de variable de scope
- La gestion des exceptions

<c:out/> : Afficher une expression

Evalue une expression et l'affiche dans la page JSP.

Attribut	Req.	Type	Description
value	**oui**	Object	L'expression qui sera évaluée et affichée. Si le type réel implémente **java.io.Reader**, alors c'est son contenu qui sera affiché.
default		Object	Valeur à afficher si l'expression **value** est **null** (défaut : **""**).
escapeXml		booleen	Détermine si les caractères **<**, **>**, **&**, **'**, **"** doivent être remplacés par leurs codes respectifs : **<**, **>**, **&**, **'**, **"** (défaut : **true**).

Le **Corps du tag** peut être utilisé à la place de l'attribut **default**.

Exemple
```
<!-- Afficher l'user-agent du navigateur ou "Inconnu" si il est absent : -->
<c:out value="${header['user-agent']}" default="Inconnu"/>

<!-- Même chose en utilisant le corps du tag : -->
<c:out value="${header['user-agent']}">
        Inconnu
</c:out>
```

*Le conteneur JSP 2.0 gère lui-même les **EL**, ainsi le code **<c:out value="${expression}" escapeXml="false"/>** est équivalent à **${expression}**.*

<c:set/> : Définir une variable de scope ou une propriété

Permet de définir une nouvelle variable de scope, ou de changer la valeur d'une propriété d'un **beans**.

Attribut	Req.	Type	Description
value		Object	L'expression à évaluer.

Attribut	Req.	Type	Description
var		String	Nom de l'attribut qui contiendra l'expression dans le scope.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).
target		Object	L'objet dont la propriété définit par **property** sera modifiée. Il doit correspondre soit à un bean avec la méthode mutateur correspondante (**setProperty**)), soit à un objet de type **java.util.Map**.
property		String	Nom de la propriété qui sera modifiée.

Le **Corps du tag** peut être utilisé à la place de l'attribut **value**.

Exception :

Une exception est propagé lorsque l'attribut **target** ne correspond ni à une **Map**, ni à un **bean** possédant une propriété "**property**".

Exemple

```
<!-- Mettre ${expression} dans l'attribut "varName" de la session : -->
<c:set scope="session" var="varName" value="${expression}" />

<!-- Changer la propriété "name" de l'attribut "varName" de la session : -->
<c:set target="${session['varName']}" property="name" value="new value"/>

<!-- Changer la propriété "name" de l'attribut "varName" de la session en utilisant le corps : -->
<c:set target="${session['varName']}" property="name">
        Nouvelle valeur de la propriété "name"
</c:set>
```

*Si l'attribut **value** est **null**, cela correspond à supprimer la variable ou la propriété d'une **Map**, ou à passer à **null** la propriété du **bean**.*

<c:remove/> : Supprimer une variable de scope

Supprime la variable de scope indiqué.

Attribut	Req.	Type	Description
var	**oui**	String	Nom de la variable de scope à supprimer.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Corps du tag : Aucun.

Exemple

```
<!-- Supprime l'attribut "varName" de la session -->
<c:remove var="varName" scope="session"/>
```

<c:catch/> : Intercepter les exceptions

Intercepte les exceptions qui peuvent être lancées par son corps

Attribut	Req.	Type	Description
var		String	Nom de la variable dans le scope **page** qui contiendra l'exception interceptée.

Corps du tag : Le code JSP dont les exceptions seront interceptées...

Exemple
```
<!-- Ignorer toutes les exceptions d'une partie de la page : -->
<c:catch>
        <c:set target="beans" property="prop" value="1"/>
</c:catch>

<!-- Stocker dans le scope page l'exception intercepté : -->
<c:catch var="varName">
        <c:set target="beans" property="prop" value="1"/>
</c:cath>
```

*Si **var** n'est pas spécifié, les exceptions interceptées ne seront pas sauvegardées.*

*Si **var** est spécifié et qu'aucune exception n'est lancée, alors la variable de page "**var**" sera supprimée.*

2.2 - Actions conditionnels

Cette section comporte les actions permettant d'effectuer les tests conditionnels de la même manière que les mots-clef **if** ou **switch** du langage Java.

<c:if/> : Traitement conditionnel

Permet d'effectuer un traitement conditionnel de la même manière que le mot-clef **if** du langage Java.

Attribut	Req.	Type	Description
test	**oui**	booleen	La condition de test qui déterminera si le corps devra être evalué ou non.
var		String	Le nom d'une variable de scope de type **Boolean** qui contiendra le résultat du test.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Corps du tag : Le code qui sera interprété selon le résultat de la condition.

Exemple
```
<!-- Afficher un message si le paramètre "page" de la requête HTTP est absent -->
<c:if test="${empty param['page']}">
        Le paramètre page est absent !
</c:if>
```

<c:choose/> : Traitement conditionnel exclusif

Permet d'effectuer un traitement conditionnel de la même manière que le mot-clef **switch** du langage Java, ou qu'une série de **if**/**else**. C'est à dire que sur plusieurs possibilités, une seule sera évaluée.

L'action **<c:choose/>** n'accepte aucun attribut, et le **corps du tag** ne peut comporter qu'un ou plusieurs tags **<c:when/>** et zéro ou un tag **<c:otherwise/>**.

L'action **<c:choose/>** exécutera le corps du premier tag **<c:when/>** dont la condition de test est évaluée à **true**. Si aucune de ces conditions n'est vérifiée, il exécutera le corps de la balise **<c:otherwise/>** si elle est présente.

Consulter les informations sur les balise **<c:when/>** et **<c:otherwise/>** pour plus de détail.

<c:when/> : Un cas du traitement conditionnel

Définit une des options de l'action **<c:choose/>**.

La balise parent doit obligatoirement être **<c:choose/>**.

Le premier tag de la balise **<c:choose/>** dont la condition est vérifiée sera le seul à évaluer son corps.

*Le tag **<c:when/>** a le même fonctionnement que le mot-clef **case** d'un bloc **switch** en Java.*

Attribut	Req.	Type	Description
test	**oui**	boolean	La condition de test qui déterminera si le corps devra être évalué ou non (on utilise généralement une **Expressions Languages**).

Corps du tag : Le code qui sera interprété selon le résultat de la condition.

Exemple

```
<!-- Afficher un message différent selon la valeur du bean 'value' : -->
<c:choose>
        <c:when test="${value==1}"> value vaut 1 (Un) </c:when>
        <c:when test="${value==2}"> value vaut 2 (Deux) </c:when>
        <c:when test="${value==3}"> value vaut 3 (Trois) </c:when>
</c:choose>
```

<c:otherwise/> : Traitement par défaut

Le traitement à effectuer si aucun tag **<c:when/>** n'a été évalué.

La balise parent doit obligatoirement être **<c:choose/>** et après la dernière balise **<c:when/>**.

Elle n'accepte aucun attribut et n'évaluera son corps que si aucune des balises **<c:when/>** n'est vérifiée.

*Le tag **<c:otherwise/>** a le même fonctionnement que le mot-clef **default** d'un bloc **switch** en Java.*

Exemple

```
<c:choose>
        <c:when test="${value==1}"> value vaut 1 (Un) </c:when>
        <c:when test="${value==2}"> value vaut 2 (Deux) </c:when>
        <c:when test="${value==3}"> value vaut 3 (Trois) </c:when>
        <c:otherwise>
                value vaut ${value} (?)
        </c:otherwise>
</c:choose>
```

2.3 - Itérations

Cette section comporte les actions permettant d'effectuer des boucles de la même manière que les mots-clef **for** ou **while** du langage Java.

2.3.1 - Attributs standards des boucles

Les tags d'itérations de la **JSTL** sont basés sur la classe **javax.servlet.jsp.jstl.core.LoopTagSupport**. Ainsi, les tags de cette section possèdent en commun les attributs suivants :

Attribut	Req.	Type	Description
var		String	Nom d'une variable de scope qui comportera l'élément courant de l'itération (visible à l'intérieur du corps du tag seulement).
varStatus		String	Nom d'une variable de scope qui comportera des informations sur le status de l'itération (visible à l'intérieur du corps du tag seulement).
begin		int	Spécifie l'index de départ de l'itération.
end		int	Spécifie l'index de fin de l'itération.
step		int	L'itération s'effectuera sur les **N** éléments de la collection. **N** correspondant à la valeur de **step**.

L'attribut **varStatus** permet d'utiliser un objet de type **LoopTagStatus** qui possèdent les propriétés suivantes :

Nom	Type	Description
begin	Integer	Valeur de l'attribut **begin** du tag (**null** si absent).
end	Integer	Valeur de l'attribut **end** du tag (**null** si absent).
step	Integer	Valeur de l'attribut **step** du tag (**null** si absent).
count	int	Comptabilise le nombre de tour de l'itération.
current	Object	L'élément courant de l'itération.
index	int	L'index de l'élément courant dans la collection.
first	booleen	Indique que la boucle courante est la première de l'itération.
last	booleen	Indique que la boucle courante est la dernière de l'itération.

*Attention : **count** et **index** ne sont pas forcément identique...*

*Toutes ces fonctionnalités sont implémentées par la classe abstraite **javax.servlet.jsp.jstl.core.LoopTagSupport**. Elle peut donc être utilisée afin de créer ses propres tags itératifs...*

<c:forEach/> : Itérer sur une collection

Permet d'effectuer simplement des itérations sur plusieurs types de collection de données.

Attribut	Req.	Type	Description
items		Object	La collection d'éléments qui contient les éléments de l'itération (Voir la "Liste des types supportés").
Ainsi que les attributs standards des boucles (Voir la liste des attributs standards).			

Corps du tag : Le code qui sera évalué à chaque itération sur la collection.

L'attribut **items** accepte les éléments suivant comme collection :

- Les tableaux d'objets ou de types primaires (ils seront alors englobés dans la classe **wrapper** correspondante).
- Une implémentation de **java.util.Collection** en utilisant la méthode **iterator()**.

- Une implémentation de **java.util.Iterator**.
- Une implémentation de **java.util.Enumeration**.
- Une implémentation de **java.util.Map**, en utilisant les méthodes **entrySet().iterator()**.
- ***[Deprecated]*** Une **String** dont les différents éléments sont séparés par des virgules (mais il est préférable d'utiliser <c:forTokens/> à la place).
- Une valeur **null** sera considérée comme une collection vide (pas d'itération).
- Si l'attribut **items** est absent, les attributs **begin** et **end** permettent d'effectuer une itération entre deux nombres entiers.

*L'utilisation d'une **String** dans le tag **<c:forEach/>** est dépréciée et ne devrait plus être utilisée. Cette fonctionnalité reste présente pour des raisons de compatibilité avec la **JSTL 1.0**, mais il est fortement conseillé d'utiliser le tag **<c:forTokens/>** à la place...*

Exemple
```
<!-- Afficher tous les éléments d'une collection dans le request-->
<c:forEach var="entry" items="${requestScope['myCollection']}" >
        ${entry}<br/>
</c:forEach>

<!-- Afficher seulement les 10 premiers éléments -->
<c:forEach var="entry" items="${requestScope['myCollection']}"
        begin="0" end="9">
        ${entry}<br/>
</c:forEach>

<!-- Afficher les nombres de 1 à 10 -->
<c:forEach var="entry" begin="1" end="10">
        ${entry},
</c:forEach>
```

*Lors de l'itération sur une **Map**, l'élément courant de chaque itération est du type **java.util.Map.Entry**, et possède donc les propriétés suivantes :*

Nom	Type	Description
key	Object	La clef utilisée pour stocker l'élément dans la **Map**.
value	Object	La valeur correspondante à la clef.

Exemple
```
<!-- Afficher tous les paramètres de la requête HTTP (param est une Map)-->
<c:forEach var="entry" items="${param}" >
        Le paramètre "${entry.key}" vaut "${entry.value}".<br/>
</c:forEach>
```

<c:forTokens/> : Itérer sur des éléments d'une String

Permet de découper des chaînes de caractères selon un ou plusieurs délimiteurs. Chaque marqueur ainsi obtenu sera traité dans une boucle de l'itération.

Attribut	Req.	Type	Description
items	**oui**	String	La chaîne de caractère qui sera découpé.
delims	**oui**	String	La liste des caractères qui serviront de délimiteurs.
Ainsi que les attributs standard des boucles (Voir la liste des attributs standards).			

Corps du tag : Le code qui sera évalué pour chaque marqueur de la chaîne.

Exemple
```
<!-- Afficher séparément des mots séparés par un point-virgule -->
```

Exemple
```
<c:forTokens var="p" items="mot1;mot2;mot3;mot4" delims=";">
        ${p}<br/>
</c:forTokens>
```

2.4 - Les URLs

Cette section décrit quelques tags utiles pour la gestion des URLs :

- Création d'URLs complexes
- Redirection vers une URLs
- Import de ressources locales ou distantes

<c:param/> : Ajouter un paramètre à une URL

Permet d'ajouter simplement un paramètre à une URL représentée par le tag parent.

Cette balise doit avoir comme balise parent une balise **<c:url/>**, **<c:import/>** ou **<c:redirect/>** (mais pas forcement comme parent direct).

Attribut	Req.	Type	Description
name	**oui**	String	Le nom du paramètre de l'URL.
value		String	La valeur du paramètre de l'URL.

Le **Corps du tag** peut être utilisé à la place de l'attribut **value** afin de définir la valeur du paramètre.

> *Le nom et la valeur du paramètre de l'URL est automatiquement encodé afin de respecter le format des URLs (les 'espaces' sont remplacés par des '+',...).*

Exemple
```
<!-- La forme suivante : -->
<c:url value="/mapage.jsp?paramName=paramValue"/>

<!-- est equivalente à : -->
<c:url value="/mapage.jsp">
        <c:param name="paramName" value="paramValue"/>
</c:url><br/>
```

<c:url/> : Créer une URL

Permet de créer des URLs absolues, relatives au contexte, ou relatives à un autre contexte.

Attribut	Req.	Type	Description
value	**oui**	String	L'URL à traiter (absolue, relative à l'application ou à la page courante).
context		String	Spécifie le chemin du contexte de l'application locale à utiliser (débute obligatoirement par le caractère '/'). Par défaut, il prend la valeur du contexte de l'application courante tel qu'il est renvoyé par **request.getContextPath()**.
var		String	Le nom de la variable de scope qui contiendra la **String** représentant l'URL.

Attribut	Req.	Type	Description
			Si absent, l'URL sera écrite dans la page JSP.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Le **corps du tag** peut contenir n'importe quel code JSP. Tous les tags **<c:param/>** pourront modifier l'URL.

Il s'agit du même mécanisme que **le corps du tag <c:url/>**.

*Contrairement au tag <c:choose/> qui n'accepte que les tags <c:when/> et <c:otherwise/>, le corps du tag **<c:url/>** accepte tout type de code JSP. Toutefois, le corps du tag est bufférisé et tout ce qui est écrit à l'intérieur n'est pas reporté sur la page JSP mais ignoré.*

Cela permet d'utiliser des balises itérations ou conditionnels pour créer la liste des paramètres :

Exemple

```
<!-- Création d'un lien dont les paramètres viennent d'une MAP -->
<c:url value="/index.jsp" var="variableURL">
        <c:forEach items="${parameterMap}" var="entry">
                <c:param name="${entry.key}" value="${entry.value}"/>
        </c:forEach>
</c:url>
<a href="${variableURL}">Mon Lien</a>
```

Les URLs sont réécrite de la manière suivante :

- Le chemin du contexte est ajouté aux URLs relatives à une application locale (URLs qui commencent par **'/'**).
- Les URLs relatives à l'application courante sont encodées afin de rajouter le **jsessionid** si nécessaire (cookies absent,...).
- Les paramètres ajoutés avec les balises **<c:param/>** sont ajoutés à l'URL.

Exemple

```
<!-- Ainsi le code suivant : -->
<c:url value="/page.jsp?param=value">

<!-- Affichera pour le contexte "contextPath" :
         /contextPath/page.jsp?param=value
         ou si les cookies sont désactivé :
         /contextPath/page.jsp;jsessionid=XXXXXXXXXX?param=value
-->

<!-- Et le code suivant créera une variable "url" dans le scope page -->
<c:url var="url" scope="page" value="/page.jsp?param=value">
        <c:param name="id" value="1"/>
</c:url>
<!-- La variable de scope "url" contiendra donc :
        /contextPath/page.jsp?param=value&id=1
         ou si les cookies sont désactivé :
         /contextPath/page.jsp;jsessionid=XXXXXXXXXX?param=value&id=1
-->
```

*Afin de pouvoir être utilisées dans d'autres tags (notamment **<jsp:include/>**), les URLs ne sont pas encodées. Afin de les utiliser dans une page XHTML stricte ou XML, il faudra d'abord protéger certains caractères (notamment : **'&'** en **'&'**).*

*Il est possible pour cela d'utiliser le tag **<c:out/>** ou la fonction **{fn:escapeXml()}**.*

<c:redirect/> : Redirection

Envoi une commande de redirection HTTP au client.

Attribut	Req.	Type	Description
url	**oui**	String	L'URL de redirection.
context		String	Spécifie le chemin du contexte de l'application locale à utiliser (débute obligatoirement par le caractère '/'). Par défaut, il prend la valeur du contexte de l'application courante tels qu'il est renvoyé par **request.getContextPath()**.

Le **corps du tag** peut contenir n'importe quel code JSP. Tous les tags **<c:param/>** pourront modifier l'URL.

Exemple

```
<!-- Redirection vers le portail de developpez.com : -->
<c:redirect url="http://www.developpez.com"/>

<!-- Redirection vers une page d'erreur avec des paramètres: -->
<c:redirect url="/error.jsp">
        <c:param name="from" value="${pageContext.request.requestURI}"/>
</c:redirect>
```

***<c:redirect/>** utilise les mêmes règles pour réécrire les URL que la balise **<c:url/>**.*

<c:import/> : Importer des ressources

Permet d'importer une ressource selon son URL. Contrairement à **<jsp:include/>**, la ressource peut appartenir à un autre contexte ou être hébergée sur un autre serveur...

Attribut	Req.	Type	Description
url	**oui**	String	L'URL de la ressource à importer.
context		String	Spécifie le chemin du contexte de l'application locale à utiliser (débute obligatoirement par le caractère '/'). Par défaut, il prend la valeur du contexte de l'application courante tels qu'il est renvoyé par **request.getContextPath()**.
var		String	Nom de la variable de scope qui comportera le contenu de la ressource en tant que **String**.
scope		String	Nom du scope qui contiendra l'attribut **var** ou **varReader** (**page**, **request**, **session** ou **application**) (défaut : **page**).
charEncoding		String	Spécifie l'encodage de caractère à utiliser.
varReader		String	Nom de la variable de scope qui comportera le contenu de la ressource en tant que **Reader** (visible à l'intérieur du corps seulement).

Ce tag a deux comportement différent :

Le **corps du tag** peut contenir n'importe quel code JSP, et tous les tags **<c:param/>** pourront modifier l'URL selon les mêmes règles que pour le corps du tag <c:url/>.

Toutefois, si l'attribut **varReader** est utilisé, le **corps du tag** doit être utilisé afin d'accéder au **Reader** dont la portée

est limitée à l'intérieur du tag. Cela s'explique par le fait que le tag **<c:import/>** a la responsabilité de fermer le flux du **Reader**.

Dans ce cas les tags **<c:param/>** ne doivent pas être utilisés car l'URL doit être complète dès le début du tag.

Exception :

Si le serveur d'application ne peut pas accéder à la ressource, une **JspException** sera lancée...

Si les attributs **var** et **varReader** sont absents, le contenu de la ressource sera directement affiché sur la page JSP.

Si le type d'encodage n'est pas spécifié, c'est celui de la ressource qui sera utilisé, ou "ISO-8859-1" en dernier recours.

Exemple

```
<!-- Importer un fichier de l'application (similaire à <jsp:include/>) -->
<c:import url="/file.jsp">
        <c:param name="page" value="1"/>
</c:import>

<!-- Importer une ressource distante FTP dans une variable -->
<c:import url="ftp://server.com/path/file.ext"
        var="file" scope="page"/>

<!-- Importe une ressource distante dans un Reader -->
<c:url value="http://www.server.com/file.jsp" var="fileUrl">
        <c:param name="file" value="filename"/>
        <c:param name="page" value="1"/>
</c:url>

<!-- Ouverte d'un flux avec un Reader -->
<c:import url="${fileUrl}" varReader="reader">
        <!-- Utilisation du reader par d'autres tags... -->
        ...
        <x:parse doc="${reader}"/>
        ...
</c:import>
```

***<c:import/>** utilise les mêmes règles pour réécrire les URL que la balise **<c:url/>**.*

3 - <fmt:/> : Librairie de Formatage

Cette librairie simplifie la localisation et le formatage des données.

Déclaration de la librairie 'fmt' :
```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
```

3.1 - Internationalisation (i18n)

La **JSTL** utilise les classes standard de Java pour la gestion de l'internationalisation. Ainsi la classe **java.util.Locale** permet de représenter les spécificités régionales, ainsi que la classe **java.util.ResourceBundle** pour accéder aux données des fichiers de localisation.

Un **ResourceBundle** permet de gérer un ensemble de fichier ***.properties** contenant les ressources localisées. Par exemple pour gérer les langues françaises, anglaises et italiennes, on pourrait avoir les fichiers suivants :

- **Message_fr.properties**
- **Message_en.properties**
- **Message_it.properties**

Il est également possible d'utiliser un code de pays afin de gérer des différences au sein même d'une langue. Par exemple si on veut pouvoir différencier le français selon que le client soit Français ou Canadien, on pourra utiliser en plus les fichiers suivants :

- Message_fr_**FR**.properties
- Message_fr_**CA**.properties

> *Il est également possible d'utiliser une variante plus spécifique, afin d'apporter une différenciation sur des critères spécifiques (système d'exploitation, variante d'une langue, ...). Par exemple, Message_fr_FR_**WIN**.properties peut prendre en compte des spécificités du système d'exploitation Windows...*

Lorsque un nouvel utilisateur se connecte, la valeur de l'header HTTP "**Accept-Language**" est utilisée pour rechercher la meilleure **Locale** à utiliser.

Par exemple, si un utilisateur utilise la **Locale** "**fr_FR**", les messages seront recherchés dans les fichiers suivants dans cet ordre :

1. **Message_fr_FR.properties**
2. **Message_fr.properties**

Si le fichier "**Message_fr_FR.properties**" n'existe pas ou qu'il ne possède pas la clef recherchée, on passe au suivant ("**Message_fr.properties**")...

Si la clef ne peut pas être trouvée un message du style "**???clef???**" est renvoyé.

Ainsi, le fichier "**Message_fr.properties**" est à privilégier afin qu'il puisse être utilisé par d'autres francophones (fr_CA, fr_BE,...), alors que le fichier **Message_fr_FR.properties** servira pour des spécifications franco-françaises...

Les fichiers ***.properties** comportent un ensemble de couple **clef/valeur**. On accède aux données localisées grâce aux différentes clefs. Par exemple, le fichier "**Message_fr.properties**" pourrait contenir :

Message_fr.properties
```
# Ligne en commentaire
message.hello = Bienvenue
message.title = Titre de la page
```

Et le code suivant affichera alors la chaîne "Bienvenue" aux utilisateurs francophones :

index.jsp
```
<fmt:param value="message.hello"/>
```

3.1.1 - Configuration

Il est possible de modifier dynamiquement la configuration de la JSTL. On peut ainsi :

- Modifier la **Locale** à utiliser à la place du header HTTP.
- Modifier la **Locale** à utiliser par défaut (si aucune des locales du header HTTP ne correspond)
- Modifier le **ResourceBundle** à utiliser.

3.1.1.1 - Locale

	Locale
Description	Définit les propriétés régionales qui devront être utilisées à la place de celles définies dans header HTTP "**Accept-Language**".
Variable	javax.servlet.jsp.jstl.fmt.locale
Constante Java	Config.FMT_LOCALE
Type	**java.util.Locale** ou une **String** représentant une Locale (Consulter l'API de la classe **java.util.Locale**).
Définit par	**<fmt:setLocale>**
Utilisé par	<fmt:bundle>, <fmt:setBundle>, <fmt:message>, <fmt:formatNumber>, <fmt:parseNumber>, <fmt:formatDate>, <fmt:parseDate>

Cette variable peut être utilisée afin de modifier la langue d'un utilisateur ou de forcer l'utilisation d'une Locale pour une page particulière. Cette variable ne devrait pas être utilisée dans le fichier **web.xml**.

Changer la locale de l'utilisateur
```
<fmt:setLocale value="fr_FR" scope="session"/>
```

3.1.1.2 - Locale par défaut

	Locale par défaut
Description	Définit les propriétés régionales par défaut de l'application. C'est à dire celles qui seront utilisées lorsque l'application ne gère aucun des langages du header HTTP "**Accept-Language**", et que la **Locale** n'est pas définit...
Variable	javax.servlet.jsp.jstl.fmt.fallbackLocale
Constante Java	Config.FMT_FALLBACK_LOCALE
Type	**java.util.Locale** ou une **String** représentant une Locale (Consulter l'API de la classe **java.util.Locale**).

	Locale par défaut
Définit par	**web.xml**
Utilisé par	<fmt:bundle>, <fmt:setBundle>, <fmt:message>, <fmt:formatNumber>, <fmt:parseNumber>, <fmt:formatDate>, <fmt:parseDate>

Cette variable devrait être configurée dans le fichier **web.xml** de l'application. Il est inutile de la modifier par la suite.

Exemple (web.xml)
```
<!-- Utilisation de l'anglais comme Locale par défaut -->
<context-param>
        <param-name>javax.servlet.jsp.jstl.fmt.fallbackLocale</param-name>
        <param-value>en</param-value>
</context-param>
```

3.1.1.3 - Contexte de localization

	Contexte de localization (ResourceBundle)
Description	Définit le **ResourceBundle** qui sera utilisé pour l'internationalisation des chaînes.
Variable	javax.servlet.jsp.jstl.fmt.localizationContext
Constante Java	Config.FMT_LOCALIZATION_CONTEXT
Type	**LocalizationContext** ou une **String** comportant le nom de base du **Resourcebundle**.
Définit par	**web.xml**, **<fmt:setBundle>**
Utilisé par	<fmt:message>, <fmt:formatNumber>, <fmt:parseNumber>, <fmt:formatDate>, <fmt:parseDate>

Cette variable indique le **Resourcebundle** par défaut de l'application. Il est conseillé de définir sa valeur dans le fichier **web.xml** pour les fichiers *.properties principaux, puis d'utiliser **<fmt:bundle>** ou **<fmt:setBundle>** pour accéder à d'autres fichiers *.properties (contenant par exemple des messages d'erreurs...)

> *Le **LocalizationContext** est une simple classe contenant les informations de localization (la **Locale** et le **ResourceBundle**).*

<fmt:setLocale/> : Définir la Locale

Permet de changer la **Locale** à utiliser dans les tags de la librairie.

Attribut	Req.	Type	Description
value	**oui**	String ou Locale	La Locale à utiliser.
variant		String	Spécifie une variante spécifique à un système ou un navigateur. Consulter la classe **Locale** pour plus de détails.
scope		String	Nom du scope qui contiendra l'attribut la **Locale** (**page**, **request**, **session** ou **application**) (défaut : **page**). Consultez la section "Configuration de la JSTL".

Corps du tag : Aucun.

Exemple
```
<!-- Force l'affichage de la page en anglais : -->
```

Exemple
```
<fmt:setLocale value="en" scope="page"/>

<!-- Force l'affichage en français pour un utilisateur : -->
<fmt:setLocale value="fr" scope="session"/>
```

Cette balise permet notamment d'ignorer l'header HTTP "**Accept-Language**" afin de forcer l'utilisation d'une autre langue.

*Attention à ne pas changer la langue du scope **application** car cela affecterait tous les utilisateurs...*

<fmt:setBundle/> : Définir le ResourceBundle

Permet de changer la **Locale** à utiliser dans les tags de la librairie, ou de créer une variable **LocalizationContext** afin de la réutiliser dans d'autres tags de la librairie.

Attribut	Req.	Type	Description
basename	**oui**	String	Nom de base du **ResourceBundle** à utiliser.
var		String	Le nom de la variable qui contiendra le **LocalizationContext**. Si absent, la valeur du contexte de localisation sera modifiée selon les règles du scope.
scope		String	Nom du scope qui contiendra l'attribut le **LocalizationContext** (**page**, **request**, **session** ou **application**) (défaut : **page**). Consultez la section "Configuration de la JSTL".

Corps du tag : Aucun.

Exemple
```
<!-- Change le ResourceBundle par défaut de la page : -->
<fmt:setBundle basename="package.MyRessource"/>

<!-- Création d'un autre Bundle : -->
<fmt:setBundle basename="package.MyOtherRessource" var="bundleName"/>

<!-- Affichage d'un message du ResourceBundle par défaut : -->
<fmt:message key="keyName"/>

<!-- Affichage d'un message du second Bundle : -->
<fmt:message key="keyName" bundle="${bundleName}"/>
```

<fmt:bundle/> : Définir un ResourceBundle partiel

Permet d'utiliser un **ResourceBundle** limité à une partie de la page JSP (le corps du tag).

Attribut	Req.	Type	Description
basename	**oui**	String	Nom de base du **ResourceBundle** par défaut à l'intérieur du tag.
prefix		String	Permet de définir un préfixe qui sera utilisé par tous les tags **<fmt:message/>** du corps.

Corps du tag : Le code JSP qui utilisera le **ResourceBundle**.

Exemple
```
<!-- Utilisation d'un autre Bundle pour afficher des messages : -->
<fmt:bundle basename="package.MessageResource">
        <fmt:message key="msg.error.key1"/><br/>
        <fmt:message key="msg.error.key2"/><br/>
        <fmt:message key="msg.error.key3"/><br/>
</fmt:bundle>

<!-- Même chose en utilisant un préfixe : -->
<fmt:bundle basename="package.MessageResource" prefix="msg.error."">
        <fmt:message key="key1"/><br/>
        <fmt:message key="key2"/><br/>
        <fmt:message key="key3"/><br/>
</fmt:bundle>
```

<fmt:message/> : Afficher des messages

Permet l'affichage d'un message depuis un **ResourceBundle**.

Attribut	Req.	Type	Description
key	**oui**	String	La clef du message qui doit être recherché dans le **ResourceBundle**.
bundle		LocalizationContext	Spécifie l'utilisation d'un autre **LocalizationContext** pour rechercher le message. (Créé avec <fmt:setBundle/>)
var		String	Nom de la variable de scope qui contiendra le message.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Corps du tag : Analyse les balises **<fmt:param/>** seulement.

Exemple
```
<!-- Affichage d'un message directement dans le JSP : -->
<fmt:message key="message.key"/>

<!-- Copie d'un message dans une variable pour l'afficher plus tard : -->
<fmt:message key="message.key" var="msg"/>

Le message est : <b>${msg}</b> !
```

<fmt:param/> : Ajouter un paramètre au message

Permet de paramétrer l'affichage d'un message obtenu avec **<fmt:message/>**.

Attribut	Req.	Type	Description
value		Object	L'objet qui sera utilisé pour paramétrer le message.

Le **Corps du tag** peut être utilisé à la place de l'attribut **value** afin de définir la valeur du paramètre.

Exemple
```
<!-- Affichage d'un message avec deux paramètres -->
<fmt:message key="message.key">
        <fmt:param value="${mailbox.userName}"/>
        <fmt:param value="${mailbox.messageCount}"/>
</fmt:message>
```

Ainsi, si "**message.key**" correspond à la chaîne de caractères suivante :

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

donnera les messages suivants selon la valeur de **${mailbox.messageCount}** :

- "*Bienvenue Fred, votre boîte de réception ne comporte aucun message...*"
- "*Bienvenue Fred, votre boîte de réception comporte un message...*"
- "*Bienvenue Fred, votre boîte de réception comporte 2 messages...*"
- etc...

*Pour plus de détail sur les possibilitées de formatage du texte, veuillez consulter la documentation des classes **MessageFormat**, **DecimalFormat** et **ChoiceFormat** du package **java.text**, de l'API Java...*

<fmt:requestEncoding/> : Encodage du client

Permet de définir l'encodage de caractère utilisé par le navigateur du client.

Attribut	Req.	Type	Description
value		String	Nom du format d'encodage à utiliser pour décoder les paramètres de la requête (défaut : "**ISO-8859-1**").

Corps du tag : Aucun.

Exemple
```
<fmt:requestEncoding/>
```

*Ce tag permet de fixer l'encodage de la requête HTTP du client. En effet, de nombreux navigateurs ne respectent pas les spécifications HTTP et ne spécifient pas l'header **Content-Type** dans leur requête.*

*Ce tag fait un appel à la méthode **setCharacterEncoding()** de la **Servlet**. Il doit donc être utilisé avant tout accès aux paramètres de la requête HTTP...*

3.2 - Formatage

La **JSTL** propose des tags facilitant le formatage des données numériques et des dates/heures. Ces tags prennent en compte la **Locale** de l'utilisateur pour paramétrer l'affichage...

3.2.1 - Configuration

Le traitement des dates et des heures prend en compte

3.2.1.1 - Fuseau horaire (TimeZone)

	Fuseau horaire (TimeZone)
Description	Définit le fuseau horaire qui doit être utilisé pour la gestion des dates.
Variable	javax.servlet.jsp.jstl.fmt.timeZone
Constante Java	Config.FMT_TIMEZONE

	Fuseau horaire (TimeZone)
Type	**java.util.TimeZone** ou une **String** représentant une TimeZone (Consulter l'API de la classe **java.util.TimeZone**).
Définit par	**<fmt:setTimeZone>**
Utilisé par	<fmt:formatDate>, <fmt:parseDate>

Définit le fuseau horaire à utiliser dans l'application.

Si aucune valeur n'est définie, le fuseau horaire du serveur sera utilisé.

<fmt:setTimeZone/> : Définir le fuseau horaire

Permet de changer le fuseau horaire à utiliser dans les tags de la librairie, ou de créer une variable **TimeZone** afin de la réutiliser dans d'autres tags de la librairie.

Attribut	Req.	Type	Description
value	**oui**	String ou **TimeZone**	Le nom du **TimeZone** à utiliser.
var		String	Le nom de la variable qui contiendra le **TimeZone**. Si absent, la valeur du fuseau horaire sera modifiée selon les règles du scope.
scope		String	Nom du scope qui contiendra l'attribut le **TimeZone** (**page**, **request**, **session** ou **application**) (défaut : **page**). Consultez la section "Configuration de la JSTL".

Corps du tag : Aucun.

Exemple

```
<!-- Change le TimeZone par défaut de la page : -->
<fmt:setTimeZone value="GMT-8"/>

<!-- Création d'un autre TimeZone dans une variable : -->
<fmt:setTimeZone value="GMT-1" var="timezone"/>
```

<fmt:timeZone/> : Utiliser un Timezone

Permet d'utiliser un **timeZone** limité à une partie de la page JSP (le corps du tag).

Attribut	Req.	Type	Description
value	**oui**	String ou **TimeZone**	Nom du fuseau horaire.

Corps du tag : Le code JSP qui utilisera le **TimeZone**.

Exemple

```
<!-- Affichage d'une date GMT : -->
<fmt:timeZone value="GMT">
        <fmt:formatDate value="${myDate}"/>
</fmt:timeZone>
```

<fmt:parseDate/> : Analyser une date

Permet de créer des dates en analysant une chaîne de caractère. Le **TimeZone** et la **Locale** peuvent modifier le comportement de ce tag.

Attribut	Req.	Type	Description
value		String	La chaîne représentant la date à analyser.
type		String	"**date**", "**time**" ou "**both**" indiquant respectivement que l'on traite une date, une heure ou les deux (défaut : "**date**").
dateStyle		String	"**default**", "**short**", "**medium**", "**long**" ou "**full**", qui correspondent à un style de formatage de la date, et donc à un pattern d'analyse de la date. Ces différents styles varient selon la locale.
timeStyle		String	"**default**", "**short**", "**medium**", "**long**" ou "**full**", qui correspondent à un style de formatage de l'heure, et donc à un pattern d'analyse de l'heure Ces différents styles varient selon la locale.
pattern		String	Le pattern a utilisé pour analyser la date/heure. Cet attribut est prioritaire sur **dateStyle** et **timeStyle**. Sa syntaxe est la même que celle utilisé avec la classe **java.text.SimpleDateFormat**.
timeZone		String ou **TimeZone**	Le fuseau horaire à utiliser pour l'analyser de la date/heure.
parseLocale		String ou **Locale**	La **Locale** à utiliser pour l'analyser de la date/heure.
var		String	Nom de la variable de scope qui contiendra la date/heure
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Le **Corps du tag** peut être utilisé à la place de l'attribut **value** afin de définir la valeur du paramètre.

Exemple

```
<!-- Créer une date initialisé au premier janvier 2005 : -->
<fmt:parseDate value="01/01/2005" pattern="dd/MM/yyyy" var="date"/><br/>
```

Exception :

Une exception est renvoyée si la chaîne "**value**" ne correspond pas au pattern ou au style indiqué.

> *Il est fortement recommandé d'utiliser un pattern. En effet les styles (**dateStyle** et **timeStyle**) varient selon la **Locale** et peuvent engendrer des erreurs...*

<fmt:formatDate/> : Formater une date

Permet de formater une date afin de l'afficher à l'utilisateur. Le **TimeZone** et la **Locale** peuvent modifier le résultat de ce tag.

Attribut	Req.	Type	Description
value		**java.util.Date**	La date à formater.
type		String	"**date**", "**time**" ou "**both**" indiquant respectivement que l'on traite

Attribut	Req.	Type	Description
			une date, une heure ou les deux (défaut : "**date**").
dateStyle		String	"**default**", "**short**", "**medium**", "**long**" ou "**full**", qui correspondent à un style de formatage de la date, et donc à un pattern d'analyse de la date. Ces différents styles varient selon la locale.
timeStyle		String	"**default**", "**short**", "**medium**", "**long**" ou "**full**", qui correspondent à un style de formatage de l'heure, et donc à un pattern d'analyse de l'heure Ces différents styles varient selon la locale.
pattern		String	Le pattern a utilisé pour analyser la date/heure. Cet attribut est prioritaire sur **dateStyle** et **timeStyle**. Sa syntaxe est la même que celle utilisé avec la classe **java.text.SimpleDateFormat**.
timeZone		String ou **TimeZone**	Le fuseau horaire à utiliser pour l'analyser de la date/heure.
var		String	Nom de la variable de scope qui contiendra la chaîne représentant la date/heure. Si absent, elle sera affichée directement dans la JSP.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Corps du tag : Aucun.

Exemple

```
<!-- Afficher une date selon un format spécifique : -->
<fmt:formatDate value="${dateBeans}" pattern="dd/MM/yyyy"/>

<!-- Afficher une date selon un format standard : -->
<fmt:formatDate value="${dateBeans}" style="full"/>
```

<fmt:parseNumber/> : Analyser un nombre

Permet d'analyser des chaînes de caractères afin de créer des objets représentant un nombre (sous-class de **java.lang.Number**).

La **Locale** peut modifier le résultat de ce tag.

Attribut	Req.	Type	Description
value		String	La chaîne de caractères à analyser.
type		String	Spécifie le type de la chaîne de caractère, qui peut être un simple nombre ("**number**"), une valeur monétaire ("**currency**"), ou un pourcentage ("**precent**"). (défaut : **number**).
pattern		String	Le pattern a utilisé pour analyser le nombre. Sa syntaxe est la même que celle utilisé avec la classe **java.text.DecimalFormat**.
parseLocale		String ou **Locale**	La **Locale** à utiliser pour l'analyser de la date/heure.
integerOnly		booleen	Spécifie que seule la partie entière du nombre sera analyser. (défaut : **false**).
var		String	Nom de la variable de scope qui contiendra le nombre.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**,

Attribut	Req.	Type	Description
			session ou **application**) (défaut : **page**).

Le **Corps du tag** peut être utilisé à la place de l'attribut **value** afin de définir la valeur du paramètre.

Exemple
```
<!-- Créer un objet nombre : -->
<fmt:parseNumber value="1250,00" var="num"/>
```

Exception :

Une exception est renvoyée si la chaîne "**value**" ne correspond pas à un nombre valide.

<fmt:formatNumber/> : Formater un nombre

Permet de formater un nombre afin de l'afficher à l'utilisateur final...

La **Locale** peut modifier le résultat de ce tag.

Attribut	Req.	Type	Description
value		String ou Number	La valeur numérique a affiché.
type		String	Spécifie le type de la chaîne de caractère, qui peut être un simple nombre ("**number**"), une valeur monétaire ("**currency**"), ou un pourcentage ("**precent**"). (défaut : **number**).
pattern		String	Le pattern a utilisé pour analyser le nombre. Sa syntaxe est la même que celle utilisé avec la classe **java.text.DecimalFormat**.
currencyCode		String	Le code ISO 4217 de la monnaie (applicable seulement pour le type "**currency**").
currencySymbol		String	Le code monnaitaire à afficher (applicable seulement pour le type "**currency**").
groupingUsed		boolean	Spécifie si on doit utiliser des séparateurs pour afficher les grands nombres (défaut : **true**).
maxIntegerDigits		int	Spécifie le nombre maximun de caractères à utiliser pour représenter la valeur entière. (défaut : pas de limite).
minIntegerDigits		int	Spécifie le nombre minimun de caractères à utiliser pour représenter la valeur entière. (défaut : **1**). Des '0' seront ajouté au début de la chaîne si le nombre est trop petit.
maxFractionDigits		int	Spécifie le nombre maximun de caractères à utiliser pour représenter la partie décimale (défaut : pas de limite).
minFractionDigits		int	Spécifie le nombre minimun de caractères à utiliser pour représenter la partie décimale. (défaut : **0**).
var		String	Nom de la variable de scope qui contiendra la chaîne
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Le **Corps du tag** peut être utilisé à la place de l'attribut **value** afin de définir la valeur du paramètre.

Exemple

```
<!-- Afficher un pourcentage -->
<fmt:formatNumber value="0.25" type="percent"/>

<!-- Afficher une somme en Euros -->
<ftm:formatNumber value="15" type="currency" currencySymbol="&euro;"/>
```

Exception :

Une exception est renvoyée si la chaîne "**value**" ne correspond pas à un nombre valide.

4 - <sql:/> : Librairie SQL

Cette librairie facilite l'accès aux bases de données via le langage SQL au sein d'une page JSP.

Déclaration de la librairie 'SQL' :
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
```

Je vous invite à consulter les tutoriels SQL de developpez.com :

http://sql.developpez.com/

Ainsi que la FAQ JDBC :

http://java.developpez.com/faq/jdbc/

4.1 - Configuration

La librairie SQL permet de configurer le DataSource et le nombre de ligne maximum d'une requête. On peut bien sûr utiliser des valeurs différentes grâce aux attributs du même nom des différents tags...

4.1.1 - DataSource

	DataSource
Description	Définit la source de données qui sera attaquée par les actions SQL de cette librairie.
Variable	javax.servlet.jsp.jstl.sql.dataSource
Constante Java	Config.SQL_DATA_SOURCE
Type	**DataSource** ou une **String** représentant un chemin JNDI ou une URL JDBC.
Définit par	**web.xml**, **<fmt:setDataSource>**
Utilisé par	<sql:query>, <sql:update>, <sql:transaction>

Cette variable défini la source de données qui sera utilisé dans les tags de la librairie. Cette variable accepte les éléments suivants :

- Une instance de **DataSource**.
- Une chaîne représentant le nom JNDI du DataSource (si le conteneur supporte JNDI).
- Une chaîne paramétrée contenant les informations sur la connection (**url,driver,user,password**)

Configuration du web.xml
```
<!-- Utilisation de JNDI :         -->
<context-param>
        <param-name>javax.servlet.jsp.jstl.sql.dataSource</param-name>
        <param-value>jdbc/myDataBase</param-value>
</context-param>

<!-- Utilisation d'une chaîne paramétrée sur une base mySQL :         -->
<context-param>
        <param-name>javax.servlet.jsp.jstl.sql.dataSource</param-name>
        <param-value>jdbc:mysql://localhost/base,org.gjt.mm.mysql.Driver,login,password</param-value>
</context-param>
```

*Les chaînes paramétrées utilisent la classe **DriverManager** pour accéder à la base de données et ne dispose donc pas des dispositifs de gestion de connections d'un vrai **DataSource**. Elles ne devraient donc pas être*

utilisées dans un environnement de production.

4.1.2 - Nombre de ligne maximum

	Maxrows
Description	Permet de limiter le nombre maximum de ligne de résultat qu'une requête peut retourner.
Variable	javax.servlet.jsp.jstl.sql.maxRow
Constante Java	Config.SQL_MAX_ROWS
Type	**Integer**
Définit par	**web.xml**
Utilisé par	<sql:query>

Si le nombre maximum de ligne vaut **-1** où qu'il n'est pas spécifié, cela signifie qu'aucune limite ne sera appliquée aux requêtes SQL.

Configuration du web.xml

```
<!-- Limiter les requêtes à 100 lignes : -->
<context-param>
        <param-name>javax.servlet.jsp.jstl.sql.maxRows</param-name>
        <param-value>100</param-value>
</context-param>
```

<sql:setDataSource/> : Définir le Datasource

Permet de définir le **Datasource** a utiliser pour les connections à la base de données, ou de créer un objet **DataSource**.

Attribut	Req.	Type	Description
dataSource		String ou **Datasource**	Un **DataSource** tels qu'il est défini dans la section "DataSource".
driver		String	Paramètre JDBC : Nom du driver JDBC à utiliser.
url		String	Paramètre JDBC : URL de la base de données.
user		String	Paramètre JDBC : Nom de l'utilisateur.
password		String	Paramètre JDBC : Mot de passe de l'utilisateur.
var		String	Le nom de la variable qui contiendra le **Datasource**. Si absent, la valeur du Datasource sera modifiée selon les règles du scope.
scope		String	Nom du scope qui contiendra l'attribut le **Datasource** (**page**, **request**, **session** ou **application**) (défaut : **page**). Consultez la section "Configuration de la JSTL".

Corps du tag : Aucun.

Exemple

```
<!-- Changer le DataSource a utiliser sur la page (avec JNDI) -->
<sql:setDataSource dataSource="jdbc/myDataBase" scope="page"/>

<!-- Changer le DataSource pour l'utilisateur courant (avec JNDI) -->
<sql:setDataSource dataSource="jdbc/myDataBase" scope="session"/>
```

Exemple
```
<!-- Créer un bean dateSource (avec JNDI) -->
<sql:setDataSource dataSource="jdbc/myDataBase"
        var="monBeanDataSource" scope="page"/>

<!-- Créer un DataSource via un DriverManager (non recommandé) : -->
<sql:setDataSource var="monBeanDataSource" scope="page"
        dataSource="jdbc:mysql://localhost/base,org.gjt.mm.mysql.Driver,login,password"/>

<!-- Même solution décomposé : -->
<sql:setDataSource var="monBeanDataSource" scope="page"
        url="jdbc:mysql://localhost/base"
        driver="org.gjt.mm.mysql.Driver"
        user="login"
        password="password"/>
```

<sql:query/> : Exécuter une requête

Permet d'exécuter des requêtes (**SELECT**) sur la base de données.

Attribut	Req.	Type	Description
sql		String	La requête SQL.
dataSource		String ou **DataSource**	Le **DataSource** à utiliser.
maxRows		int	Le nombre maximum de lignes qui seront retournées. Si absent, la valeur globale de la configuration sera utilisée (Voir "Nombre de ligne maximum").
startRow		int	Index de la première ligne a insérer dans le résultat. La première ligne possède l'index **0**. Si la valeur de **startRow** est plus grande que le nombre de ligne rien n'est retourné.
var	**oui**	String	Le nom de la variable de scope qui contiendra le résultat sous forme de **javax.servlet.jsp.jstl.sql.Result**.
scope		String	Nom du scope qui contiendra l'attribut le **Datasource** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Le **corps du tag** peut contenir la requête SQL à la place de l'attribut **sql**. De plus elle peut contenir des tags afin de paramétrer la requête SQL (Voir <sql:param/>).

Exemple
```
<!-- Rechercher tous les éléments d'une table -->
<sql:query sql="select * from tableName" var="result"/>

<!-- Même chose en utilisant le corps du tag -->
<sql:query var="result">
        select * from tableName
</sql:query>

<!-- Rechercher seulement 10 éléments de la table.
        Le premier élément est défini par la paramètre HTTP start : -->
<sql:query var="result"  maxRows="10" startRow="${param['start']}">
        select * from tableName
</sql:query>
```

Exception :

Les éventuelles exceptions SQL sont encapsulées dans une **JspException**...

L'action **<sql:query/>** stocke le résultat de la requête dans un objet qui implémente l'interface

javax.servlet.jsp.jstl.sql.Result qui comporte les propriétés suivants :

Nom	Type	Description
columnNames	**String[]**	Tableau de String contenant le nom des différentes colonnes de la requête.
rowCount	**int**	Le Nombre de ligne retourné par la requête.
rows	**SortedMap[]**	Un tableau de **SortedMap** représentant chacun une ligne du résultat. La **SortedMap** comporte les différentes valeurs d'une ligne en utilisant le nom du champ comme clef. Elle est triée selon le nom des champs.
rowsByIndex	**Object[][]**	Un tableau de tableau contenant toutes les valeurs renvoyées par la requête. Le premier tableau représente une ligne spécifique tandis que le second représente les différentes valeurs retournées.
limitedByMaxRows	**boolean**	Indique si le résultat de la requête est limité par un nombre maximum de ligne.

Il est ainsi possible de paramétrer l'affichage des données à sa guise. **rowsByIndex** permettant d'accéder aux éléments selon leurs index tandis que **rows** permet d'utiliser le nom du champs SQL.

Par exemple, pour afficher le résultat d'une requête sous forme de tableau HTML, on peut utiliser le code suivant :

Exemple

```
<!-- Affichage d'une requête :
        ${result} étant la variable de scope crée par <sql:query/>
-->

<table>
        <!-- Affichage de l'entête avec le nom des colonnes -->
        <tr>
        <c:forEach var="name" items="${result.columnNames}">
                <th>${name}</th>
        </c:forEach>
        </tr>

        <!-- Affichage des données avec 'rowsByIndex' -->
        <c:forEach var="ligne" items="${result.rowsByIndex}">
        <tr>
                <c:forEach var="valeur" items="${ligne}">
                        <td>${valeur}</td>
                </c:forEach>
        </tr>
        </c:forEach>
</table>
```

*Chaque éléments de la requête est un objet de l'instance du type Java associé au type SQL tel que définit par la spécification JDBC et obtenu par la méthode **getObject()** du **ResultSet**...*

<sql:update/> : Exécuter une commande SQL

Permet d'exécuter des commandes SQL tel que **INSERT**, **UPDATE** ou **DELETE** ainsi que les commandes SQL qui ne retournent pas de résultats...

Attribut	Req.	Type	Description
sql		String	La requête SQL.
dataSource		String ou **DataSource**	Le **DataSource** à utiliser.

Attribut	Req.	Type	Description
var	**oui**	String	Le nom de la variable de scope qui contiendra le résultat de la requête (un **Integer** contenant le nombre de ligne ajouté/modifié/supprimé ou **0** si la requête ne retourne rien).
scope		String	Nom du scope qui contiendra l'attribut le **Datasource** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Le **corps du tag** peut contenir la requête SQL à la place de l'attribut **sql**. De plus elle peut contenir des tags afin de paramétrer la requête SQL (Voir <sql:param/>).

Exemple
```
<!-- Supprimer tous les éléments d'une table -->
<sql:query sql="delete from tableName" var="nbRowDeleted"/>
```

Exception :

Les éventuelles exceptions SQL sont encapsulées dans une **JspException**...

<sql:transaction/> : Exécuter une commande SQL

Permet d'exécuter des commandes SQL tel que **INSERT**, **UPDATE** ou **DELETE** ainsi que les commandes SQL qui ne retournent pas de résultats...

Attribut	Req.	Type	Description
dataSource		String ou **DataSource**	Le **DataSource** à utiliser.
isolation		String	Définit le niveau d'isolation de la transaction (valeur possible : "**read_uncommitted**", "**read_committed**", "**repeatable_read**" ou "**serializable**" qui correspondent respectivement aux constantes Java **TRANSACTION_READ_UNCOMMITTED**, **TRANSACTION_READ_COMMITTED**, **TRANSACTION_REPEATABLE_READ**, **TRANSACTION_SERIALIZABLE**). Si il n'est pas spécifié, le niveau d'isolation du **DataSource** sera utilisé.

Pour plus d'information sur les transactions, consultez la FAQ JDBC de developpez.com :

http://java.developpez.com/faq/jdbc/?page=transactions#isolationTransactions

Le **corps du tag** contient un ensemble de tags **<sql:query/>** et/ou **<sql:update/>**. Toutes modifications de la base ne sera pas effectuées si une exception survient pendant la transaction.

Exemple
```
<!--
<sql:transaction>
        <sql:query sql="requete SQL 1 ... " />
        <sql:query sql="requete SQL 1 ... " />
        <sql:query sql="requete SQL 1 ... " />
</sql:transaction>
```

*Lorsqu'ils sont utilisés à l'intérieur de la balise **<sql:transaction/>**, **<sql:query/>** et **<sql:update/>** utilise le dataSource du tag **<sql:transaction/>** et ne doivent donc pas être utilisés avec l'attribut **dataSource**...*

<sql:param/> : Définir un paramètre de la requête

Permet de définir la valeur d'un paramètre d'une requête lorsque le marqueur "**?**" est utilisé. Il doit obligatoirement être un sous tag de **<sql:query/>** ou de **<sql:update/>**.

Attribut	Req.	Type	Description
value		Object	La valeur du paramètre de la requête.

Le **Corps du tag** peut être utilisé à la place de l'attribut **value** afin de définir la valeur du paramètre.

Exemple

```
<!-- Exemple de requête paramétrée : -->
<sql:query var="result">
        SELECT * FROM tableName
        WHERE row1 = ?
        WHERE row1 = ?

        <sql:param value="valeur1"/>
        <sql:param>
                valeur2
        </sql:param>
</sql:query>
```

Les différents marqueurs **?** de la requête seront remplacés par la valeur des tags **<sql:param/>** en respectant leurs ordres d'apparitions...

Le type SQL a utilisé dépend du type Java de l'objet passé et dépend donc du mapping JDBC des types Java/SQL.

<sql:dateParam/> : Définir une date en paramètre de la requête

Ce tag est similaire au tag **<sql:param/>** mais permet de définir le format exacte de la date. Il doit obligatoirement être un sous tag de **<sql:query/>** ou de **<sql:update/>**.

Attribut	Req.	Type	Description
value		java.util.Date	La valeur du paramètre de la requête.
type		String	Le type de date qui sera utilisé ("**timestamp**", "**date**" ou "**time**" qui correspondent respectivement aux types SQL **TIMESTAMP**, **DATE** et **TIME**) (défaut : "**timestamp**).

Corps du tag : Aucun.

Exemple

```
<!-- Exemple de requête paramétrée avec deux dates : -->
<sql:query var="result">
        SELECT * FROM tableName
        WHERE date BETWEEN ? AND ?

        <sql:dateParam value="${startDate}" type="date"/>
        <sql:dateParam value="${endDate}" type="date"/>

</sql:query>
```

*Le tag **<sql:dateParam/>** est identique à l'utilisation du tag **<sql:param/>** avec comme valeur un objet du type **java.sql.Time**, **java.sql.Date**, ou **java.sql.Timestamp**.*

*Le tag **<sql:dateParam/>** permet d'effectuer simplement les conversions entre ces différents types et le type **java.util.Date**.*

5 - <x:/> : Librairie XML

Cette librairie permet de traiter des fichiers XML au sein d'une page JSP.

Déclaration de la librairie 'XML' :
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>
```

*L'implémentation de la **JSTL** du projet **Jakarta** nécessite la présence de la librairie **Xalan** pour la transformation des documents XML. Cette dernière peut être téléchargé à l'adresse suivante :*

http://xml.apache.org/xalan-j/

*Note: Depuis le **J2SE 5.0**, **Xalan** est inclut de base avec Java. Toutefois la version actuelle de la **JSTL** de **Jakarta** (c'est à dire la version **1.1.2**) ne le supporte pas encore (le nom du package a été modifié).*

*La version 1.0 de la JSTL utilisait plusieurs attributs dont le nom commençait par "**xml**". Or ce préfixe ne doit pas être utilisé dans un document **XML**. Afin de permettre une compatibilité avec le langage **XML**, ces attributs ont été renommé et sont donc signalé comme **[Deprecated]** (**déprécié**).*

5.1 - XPath

Afin d'accéder aux données des documents XML, le langage **XPath** (**XML Path Language**) est utilisé. Ce dernier est une recommandation du W3C dont la documentation est accessible à l'adresse suivante :

http://www.w3.org/TR/xpath

*Un tutoriel en français est également disponible sur sur **developpez.com**. Je vous invite fortement à la consulter :*

http://jerome.developpez.com/xmlxsl/xpath/

Afin d'interagir avec les données de l'application web, les expressions **XPath** peuvent être complété par des variables correspondant aux différents objets implicites. On peut ainsi utiliser les expressions suivantes à l'intérieur d'une expression **XPath** :

Expression	Correspondance
$foo	*pageContext.findAttribute("foo")*
$param:foo	*request.getParameter("foo")*
$header:foo	*request.getHeader("foo")*
$cookie:foo	*(retourne la valeur du cookie correspondant)*
$initParam:foo	*application.getInitParameter("foo")*
$pageScope:foo	*pageContext.getAttribute("foo", PageContext.PAGE_SCOPE)*
$requestScope:foo	*pageContext.getAttribute("foo", PageContext.REQUEST_SCOPE)*
$sessionScope:foo	*pageContext.getAttribute("foo", PageContext.SESSION_SCOPE)*
$applicationScope:foo	*pageContext.getAttribute("foo", PageContext.APPLICATION_SCOPE)*

Exemple XPath
```
<!-- Recherche des balises "balise" avec un attribut "attribut"
        dont la valeur est égal au paramètre HTTP name : -->
```

Exemple XPath
```
/root/balise[@attribut=$param:name]
```

De plus, lorsqu'elles sont utilisées dans un des tags de cette librairie, les expressions **XPath** doivent être précédé de l'objet représentant le document ou le noeud XML :

Exemple
```
<!-- Pour afficher le résultat de l'expression, on pourrait faire : -->
<x:out select="$myXmlDoc/root/balise[@attribut=$param:name]"/>

<!-- "myXmlDoc" étant une variable de scope représentant le document XML -->
```

5.2 - Actions XML de base

Cette section décrit les actions de bases sur les fichiers XML, c'est à dire l'analyse de fichier XML et l'accès à ses valeurs via des expressions **XPath**.

<x:parse/> : Analyser un XML

Analyse un document XML.

Attribut	Req.	Type	Description
doc		String ou Reader	La source XML du document à analyser.
xml		String ou Reader	[Deprecated] : remplacé par l'attribut **doc**.
systemId		String	L'identifiant système (URI) de l'emplacement physique du fichier.
filter		**XMLFilter**	Le filtre à appliquer sur le document (**org.xml.sax.XMLFilter**).
var		String	Le nom de la variable de scope qui contiendra le XML. Le type de cette variable est dépendant de l'implémentation.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).
varDom		String	Le nom de la variable de scope qui contiendra le XML. Le type de cette variable est toujours **org.w3c.dom.Document**.
scopeDom		String	Nom du scope qui contiendra l'attribut **varDom** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Le **Corps du tag** peut contenir le code XML à analyser. Dans ce cas l'attribut **doc** ne doit pas être utilisé.

Exemple
```
<!-- Chargement du XML dans un Reader : -->
<c:import url="/docs/myFile.xml" varReader="myFileReader"/>

<!-- Analyse du fichier XML chargé avec c:import -->
<x:parse doc="${myFileReader}" var="parsedDoc"/>


<!-- Analyse d'un XML depuis le corps du tag : -->
<x:parse var="parsedDoc">
        <stock>
                <produit id="01">
                        <nom>PC</nom>
                        <quantite>3</quantite>
                        <min>10</min>
                        <commande>10</commande>
                </produit>
```

Exemple
```
                    <produit id="02">
                            <nom>Mac</nom>
                            <quantite>5</quantite>
                            <min>10</min>
                            <commande>5</commande>
                    </produit>
            </stock>
    </x:parse>
```

Exception :

Si le document XML est vide, une erreur est levée.

<x:out/> : Evaluer une expression

Permet d'évaluer une expression XPath afin de l'afficher dans le flux de la page JSP.

Attribut	Req.	Type	Description
select	**oui**	String	L'expression XPath à évaluer.
escapeXml		boolean	Détermines si les caractères **<**, **>**, **&**, **'**, **"** doivent être remplacés par leurs codes respectifs : **<**, **>**, **&**, **'**, **"** (défaut : **true**).

Corps du tag : Aucun.

Exemple
```
    <!-- Afficher la quantité de PC en stock : -->
    <x:out select="$parsedDoc/stock/produit[nom='PC']/quantite"/>
```

<x:set/> : Créer une variable de scope

Permet d'évaluer une expression XPath afin de créer une variable de scope.

Attribut	Req.	Type	Description
select	**oui**	String	L'expression XPath à évaluer.
var	**oui**	String	Nom de l'attribut qui contiendra l'expression dans le scope.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Corps du tag : Aucun.

Exemple
```
    <!-- Stocker le noeud du PC dans un bean : -->
    <x:set select="$parsedDoc/stock/produit[name='PC']" var="pc" />

    <!-- Affichage en utilisant le noeud créé -->
    <x:out select="$pc/nom"/>
```

5.3 - Actions de contrôle XML

Cette section décrit les actions de contrôle qui peuvent être utilisés sur les documents XML. Il fonctionnent de la même manière que les actions du même nom de la librairie <core/>, mis à part qu'il s'applique à une expression

XPath sur un document XML.

<x:if/> : Action conditionnelle

Evalue une expression XPath afin de déterminer si le corps doit être exécuté.

Attribut	Req.	Type	Description
select	**oui**	String	L'expression XPath correspondant à une condition qui déterminera si le corps du tag doit être évalué ou pas.
var	**oui**	String	Nom de l'attribut qui contiendra l'expression dans le scope.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).

Corps du tag :

Exemple
```
<!-- Afficher un message si la quantité de PC est inférieur au minimum : -->
<x:if select="$parsedDoc/stock/produit[nom='PC'][quantite<min]">
        Il faut commander des PCs !!!!!
</x:if>
```

<x:choose/> : Traitement conditionnel exclusif

Ce tag permet d'effectuer un traitement conditionnel. Il fonctionnent de la même manière que les actions du même nom de la librairie <core/>, mis à part qu'il s'applique à une expression XPath sur un document XML.

L'action **<x:choose/>** n'accepte aucun attribut, et le **corps du tag** ne peut comporter qu'un ou plusieurs tags **<x:when/>** et zéro ou un tag **<x:otherwise/>**.

L'action **<x:choose/>** exécutera le corps du premier tag **<x:when/>** dont la condition de test est évalué à **true**. Si aucune de ces conditions n'est vérifiées, il exécutera le corps de la balise **<x:otherwise/>** si elle est présente.

Consulter les informations sur les balise **<x:when/>** et **<x:otherwise/>** pour plus de détail.

<x:when/> : Un cas du traitement conditionnel

Définit une des options de l'action **<x:choose/>**.

La balise parent doit obligatoirement être **<x:choose/>**.

Le premier tag de la balise **<x:choose/>** dont la condition est vérifié sera le seul à évalué son corps.

Attribut	Req.	Type	Description
select	**oui**	String	L'expression XPath correspondant à une condition qui déterminera si le corps du tag doit être évalué ou pas.

Corps du tag : Le code qui sera interprété selon le résultat de la condition.

Exemple
```
<!-- Afficher un message différent selon la quantité en stock : -->
```

Exemple
```
<x:choose>
        <x:when select="parsedDoc/stock/produit[quantite<min]">
                Le stock est en dessous de la quantité minimum !
        </x:when>
        <x:when select="parsedDoc/stock/produit[quantite=0]">
                Il n'y a plus de stock !
        </x:when>
</x:choose>
```

<x:otherwise/> : Traitement par défaut

Le traitement à effectué si aucun tag **<x:when/>** n'a été évalué.

La balise parent doit obligatoirement être **<x:choose/>** et après la dernière balise **<x:when/>**.

Elle n'accepte aucun attribut et n'évaluera son corps que si aucune des balises **<x:when/>** n'est vérifié.

Exemple
```
<code langage="xml" titre="Exemple"><![CDATA[
<!-- Afficher un message différent selon la quantité en stock : -->
<x:choose>
        <x:when select="parsedDoc/stock/produit[quantite<min]">
                Le stock est en dessous de la quantité minimum !
        </x:when>
        <x:when select="parsedDoc/stock/produit[quantite=0]">
                Il n'y a plus de stock !
        </x:when>
        <x:otherwise>
                La quantité en stock est suffisante.
        </x:otherwise>
</x:choose>
```

<x:forEach/> : Itérer sur le fichier XML

Permet d'effectuer simplement des itérations sur des éléments du fichier XML.

Attribut	Req.	Type	Description
select	**oui**	String	L'expression XPath qui retourne la liste des éléments de l'itération.
Ainsi que les attributs standard des boucles (Voir la liste des attributs standards).			

Corps du tag : Le code qui sera évalué pour chaque élément de l'expression **XPath**.

Exemple
```
<!-- Afficher tous les produits du stock : -->
<x:forEach var="produit" select="$stockXml/stock/produit">
        <b><x:out select="$produit/nom"/></b> :
        <x:out select="$produit/quantite"/> unité(s) en stock.<br/>
</x:forEach>
```

5.4 - Transformation XSLT

Cette section décrit l'utilisation de transformation **XLS** (Stylesheet Language for XML) sur des documents XML. Une bonne connaissance d'**XSLT** est donc obligatoire.

> ***XSLT*** *est une recommandation du w3c :*

http://www.w3.org/TR/xslt

Je vous invite également à consulter les cours de developpez.com pour plus de détail :

http://xml.developpez.com/cours/

<x:transform/> : Appliquer un XSLT

Permet d'appliquer une transformation **XSLT** sur un document XML.

Attribut	Req.	Type	Description
doc		Object	Le document XML à transformer. Il peut correspondre à un des types suivants : **String**, **Reader**, **javax.xml.transform.Source**, **org.w3c.dom.Document**, ou un objet créé avec **<x:parse/>** ou **<x:set/>**.
xml		Object	[Deprecated] : remplacé par l'attribut **doc**.
xslt		Object	Le document **XLST** de transformation. Il peut correspondre à un des types suivants : **String**, **Reader** ou **javax.xml.transform.Source**.
docSystemId		String	L'identifiant système (URI) de l'emplacement physique du fichier.
xmlSystemId		String.	[Deprecated] : remplacé par l'attribut **docSystemId**.
xsltSystemId		String	L'identifiant système (URI) de l'emplacement physique du fichier.
var		String	Nom de la variable de scope qui contiendra le résultat de la transformation. Si absent, le résultat sera directement affiché sur la page JSP.
scope		String	Nom du scope qui contiendra l'attribut **var** (**page**, **request**, **session** ou **application**) (défaut : **page**).
result	**oui**	**Result**	Objet de capture du résultat du processus de transformation (**javax.xml.transform.Result**).

Le **Corps du tag** peut contenir le document XML à la place de l'attribut **doc**. Il peut également contenir des tags **<x:param/>** afin de paramétrer la transformation **XSL**.

Exemple

```
<!-- chargement des fichiers XML et XLS : -->
<c:import url="/stock.xml" var="stockXml"/>
<c:import url="/stock.xsl" var="stockXsl"/>

<!-- Transformation XLS : -->
<x:transform doc="${stockXml}" xslt="${stockXsl}"/>
```

<x:param/> : Ajouter un paramètre XSLT

Permet d'ajouter simplement un paramètre à une transformation XSLT.

Cette balise doit obligatoirement être dans une balise **<x:transform/>**.

Attribut	Req.	Type	Description
name	**oui**	String	Le nom du paramètre.

Attribut	Req.	Type	Description
value		String	La valeur du paramètre.

Le **Corps du tag** peut être utilisé à la place de l'attribut **value** afin de définir la valeur du paramètre.

Exemple

```
<!-- Transformation XLS avec un paramètre : -->
<x:transform doc="${stockXml}" xslt="${stockXsl}">
        <x:param name="param" value="01"/>
</x:transform>
```

6 - ${fn:} : Librairie de fonctions EL

La création de librairies de fonctions est une nouveauté des **JSP 2.0** qui utilise à la fois les **Expressions Languages** et les **librairies de tags**. En effet, les fonctions **EL** doivent être définit dans un descripteur de taglib. Ainsi, cette librairie n'est disponible que dans la **JSTL 1.1** puisque elle nécessite un conteneur **JSP 2.0** (contrairement à la **JSTL 1.0** qui doit pouvoir fonctionner avec les **JSP 1.2**).

Déclaration de la librairie de fonctions :
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
```

La **JSTL 1.0** est prévu pour fonctionner avec un conteneur **JSP 1.2** et ne peux donc pas utiliser des fonctions **EL**...

La plupart des fonctions de cette librairie concernent la gestion des chaînes de caractères. Toutes ces fonctions interprète la valeur **null** comme un chaîne vide (**""**).

${fn:contains()}

Vérifie si une chaîne contient une autre chaîne :

Syntaxe
```
fn:contains(string, substring) -> boolean
```

Attribut	Type	Description
string	String	La chaîne sur laquelle le test sera appliqué.
substring	String	La sous chaîne à rechercher.
Retour	booleen	**true** si la chaîne représentée par **substring** est présente dans la chaîne **string**, **false** sinon.

Exemple
```
${ fn:contains("Il était une fois", "une") }
retournera true
```

*Le résultat est équivalent à l'utilisation de la méthode **contains()** de la classe **String**.*

${fn:containsIgnoreCase()}

Vérifie si une chaîne contient une autre chaîne en ignorant la case :

Syntaxe
```
fn:containsIgnoreCase(string, substring) -> boolean
```

Attribut	Type	Description
string	String	La chaîne sur laquelle le test sera appliqué.
substring	String	La sous chaîne à rechercher.
Retour	booleen	**true** si la chaîne représentée par **substring** est présente dans la chaîne **string**, **false** sinon.

Exemple
```
${ fn:containsIgnoreCase("Il était une fois", "Une") }
retournera true
```

*Le résultat est équivalent à l'utilisation de la méthode **containsIgnoreCase()** de la classe **String**.*

${fn:endsWith()}

Vérifie si une chaîne se termine par le suffixe indiqué :

Syntaxe
```
fn:endsWith(string, suffix) -> boolean
```

Attribut	Type	Description
string	String	La chaîne sur laquelle le test sera appliqué.
suffix	String	Le suffixe à vérifier.
Retour	booleen	**true** si la chaîne **string** se termine par **suffix**, **false** sinon.

Exemple
```
${ fn:endsWith("Il était une fois", "fois") }
retournera true
```

*Le résultat est équivalent à l'utilisation de la méthode **endsWith()** de la classe **String**.*

${fn:escapeXml()}

Protège les caractères qui peuvent être interprétés comme des marqueurs XML.

Syntaxe
```
fn:escapeXml(string) -> String
```

Attribut	Type	Description
string	String	La chaîne à convertir.
Retour	String	La chaîne convertit.

Exemple
```
${ fn:escapeXml("les <balises> xml & html") }
retournera "les &lt;balises&gt; xml &amp; html"
```

*Les caractères **<**, **>**, **&**, **'**, **"** seront remplacés par leurs codes respectifs : **<**, **>**, **&**, **'**, **"**.*

${fn:indexOf()}

Retourne l'index de la sous chaîne dans la chaîne :

Syntaxe
```
fn:indexOf(string, substring) -> int
```

Attribut	Type	Description
string	String	La chaîne de référence.
substring	String	La sous chaîne à rechercher.
Retour	int	L'index de la chaîne **substring** dans la chaîne **string**, ou **-1** si **substring** n'est pas trouvé dans **string**.

Exemple
```
${ fn:indexOf("Il était une fois", "une") }
retournera 10
```

*Le résultat est équivalent à l'utilisation de la méthode **indexOf()** de la classe **String**.*

${fn:join()}

Joint tous les éléments d'un tableau de chaîne dans une unique chaîne.

Syntaxe
```
fn:join(array, separator) -> String
```

Attribut	Type	Description
array	String[]	Le tableau de chaîne à joindre.
separator	String	Le séparateur à utiliser entre chacun des éléments du tableau.
Retour	String	La chaîne contenant tous les éléments du tableau.

Exemple
```
<!-- nameArray est un tableau contenant {"Il","était","une","foie"} -->
${ fn:join( nameArray, " ") }
retournera "Il était une fois"
```

${fn:length()}

Indique le nombre d'éléments d'une collection (tableau, **List**, **Set**, **Map**...) ou le nombre de caractères d'une **String** :

Syntaxe
```
fn:length(input) -> integer
```

Attribut	Type	Description
input	String ou une collection	Un des types supporté par le tag <forEach/> dont on veut connaître la taille, ou une **String** dont on veut connaître le nombre de caractères.
Retour	int	Le nombre d'éléments de la collection, ou le nombre de caractères de la chaîne.

Exemple
```
${ fn:length("Il était une fois") }
retournera 17
```

${fn:replace()}

Retourne la chaîne après avoir remplacé toutes les occurrences d'une chaîne par une autres :

Syntaxe
```
fn:replace(string, before, after) -> String
```

Attribut	Type	Description
string	String	La chaîne qui sera modifié.
before	String	La sous chaîne dont toutes les occurrences seront remplacé.
after	String	La valeur de remplacement.
Retour	String	La chaîne **string** dont toutes les occurrences de **before** ont été remplacé par **after**.

Exemple
```
${ fn:replace("Il était une fois", " ", " ") }
retournera "Il était une fois"
```

${fn:split()}

Permet de découper une chaîne de caractère en plusieurs sous chaîne :

Syntaxe
```
fn:split(string, delimiters) -> String[]
```

Attribut	Type	Description
string	String	La chaîne à découper.
délimiters	String	Les caractères qui seront utilisés comme délimiteurs.
Retour	String[]	Un tableau de String contenant les différentes sous chaînes.

Exemple
```
${ fn:split("Il était une fois", " ") }
retournera un tableau avec les éléments suivant : {"Il","était","une","foie"}
```

*Le résultat est le même que celui obtenu avec la classe **java.util.StringTokenizer**.*

${fn:startsWith()}

Vérifie si une chaîne commence par le préfixe indiqué :

Syntaxe
```
fn:startsWith(string, prefix) -> boolean
```

Attribut	Type	Description
string	String	La chaîne sur laquelle le test sera appliqué.
prefix	String	Le préfixe à vérifier.

Attribut	Type	Description
Retour	booleen	**true** si la chaîne **string** commence par **prefix**, **false** sinon.

Exemple
```
${ fn:startsWith("Il était une fois", "Il") }
retournera true
```

*Le résultat est équivalent à l'utilisation de la méthode **startsWith()** de la classe **String**.*

${fn:substring()}

Retourne une partie d'une chaîne de caractère selon deux index :

Syntaxe
```
fn:substring(string, begin, end) -> String
```

Attribut	Type	Description
string	String	La chaîne sur laquelle on découpera la sous chaîne.
begin	int	La position de départ de la sous chaîne (inclus).
end	int	La position de fin de la sous chaîne (exclus).
Retour	String	La sous chaîne comprise entre les deux index.

Exemple
```
${ fn:substring("Il était une fois", 3, 8) }
retournera "était"
```

*Le résultat est équivalent à l'utilisation de la méthode **substring()** de la classe **String**.*

${fn:substringAfter()}

Retourne la sous chaîne de caractère situé après la sous chaîne spécifiée :

Syntaxe
```
fn:substringAfter(string, substring) -> String
```

Attribut	Type	Description
string	String	La chaîne sur laquelle on découpera la sous chaîne.
substring	String	La sous chaîne qui délimitera le début de la chaîne à retourner.
Retour	String	La sous chaîne comprise entre la fin de **substring** et la fin de la chaîne.

Exemple
```
${ fn:substringAfter("Il était une fois", "était") }
retournera " une fois"
```

${fn:substringBefore()}

Retourne la sous chaîne de caractère situé avant la sous chaîne spécifié :

Syntaxe
```
fn:substringBefore(string, substring) -> String
```

Attribut	Type	Description
string	String	La chaîne sur laquelle on découpera la sous chaîne.
substring	String	La sous chaîne qui délimitera la fin de la chaîne à retourner.
Retour	String	La sous chaîne comprise entre le début de la chaîne et le début de **substring**.

Exemple
```
${ fn:substringBefore("Il était une fois", "était") }
retournera "Il "
```

${fn:toLowerCase()}

Convertit tous les caractères de la chaîne en minuscule :

Syntaxe
```
fn:toLowerCase(string) -> String
```

Attribut	Type	Description
string	String	La chaîne qui devra être mise en minuscule.
Retour	String	La chaîne en minuscule.

Exemple
```
${ fn:toLowerCase("IL ÉTAIT UNE FOIS") }
retournera "il était une fois"
```

*Le résultat est équivalent à l'utilisation de la méthode **toLowerCase()** de la classe **String**.*

${fn:toUpperCase()}

Convertit tous les caractères de la chaîne en majuscule :

Syntaxe
```
fn:toUpperCase(string) -> String
```

Attribut	Type	Description
string	String	La chaîne qui devra être mise en majuscule.
Retour	String	La chaîne en majuscule.

Exemple
```
${ fn:toUpperCase("Il était une fois") }
retournera "IL ÉTAIT UNE FOIS"
```

*Le résultat est équivalent à l'utilisation de la méthode **toUpperCase()** de la classe **String**.*

${fn:trim()}

Supprime les espaces au début et à la fin de la chaîne :

Syntaxe
```
fn:trim(string) -> String
```

Attribut	Type	Description
string	String	La chaîne dont les espaces (avant et après) seront supprimés.
Retour	String	La chaîne **string** débarrassée de ses espaces inutile en début et fin de chaîne.

Exemple
```
${ fn:trim("   Il était une fois   ") }
retournera "Il était une fois"
```

*Le résultat est équivalent à l'utilisation de la méthode **trim()** de la classe **String**.*

7 - Créer une taglib compatible avec la JSTL

L'API standard de la **JSTL** proposent un certain nombre de classe et d'interface de base son l'implémentation. Elle permet également de créer des tags compatibles avec ceux de la **JSTL** indépendamment de son implémentation.

Je n'indique ici qu'un aperçut rapide des possibilités d'interaction avec la **JSTL**. Pour plus d'information, je vous invite à consulter l'API de la **JSTL** :

http://java.sun.com/products/jsp/jstl/1.1/docs/api/index.html

Vous trouverez plus d'information sur la création de librairie de tags dans ce tutoriel :

http://adiguba.developpez.com/tutoriels/j2ee/jsp/taglib/

7.1 - Accès à la configuration

La classe **javax.servlet.jsp.jstl.core.Config** définit un certain nombre de méthode statique permettant d'accéder/modifier les différents éléments de la configuration.

- **Config.find(PageContext,String)** recherche la valeur de la configuration dans l'ordre des scopes.
- **Config.get(PageContext,String)**, **Config.get(ServletRequest,String)**, **Config.get(HttpSession,String)** et **Config.get(Servletcontext,String)** recherchent la valeur de la configuration dans le scope spécifié.
- **Config.set(PageContext,String,Object)**, **Config.set(ServletRequest,String,Object)**, **Config.get(HttpSession,String,Object)** et **Config.get(Servletcontext,String,Object)** permettent de modifier la valeur de la configuration dans le scope spécifié.

Cette classe comporte également des attributs statiques contenant le nom des différents éléments de la configuration.

Par exemple, pour changer la locale de la page courante en anglais, on peut utiliser le code suivant :

```
Config.set (pageContext, Config.FMT_LOCALE, Locale.ENGLISH );
```

7.2 - Tags conditionnels

La classe **javax.servlet.jsp.jstl.core.ConditionalTagSupport** est une classe abstraite permettant de faire un tag conditionnel simplement. Il suffit ainsi d'implémenter le code de la méthode **condition()** qui déterminera si le corps doit être affiché ou pas. Il n'y a pas à se soucier du fonctionnement réel du tag.

Par exemple, pour écrire un tag conditionnel qui n'évalue son corps que si la session comporte un attribut "**login-info**", il suffit d'utiliser le code suivant :

Exemple

```
public class LoginConditionalTag extends ConditionalTagSupport {

        protected boolean condition() {
                HttpSession session = pageContext.getSession();
                if (session!=null && session.getAttribute("login-info"))
                        return true;
                return false;
        }

}
```

7.3 - Tags itératifs

La classe **javax.servlet.jsp.jstl.core.LoopTagSupport** est une classe abstraite permettant de faire un tag itératif simplement sans se soucier du fonctionnement réel du tag.

Il suffit d'implémenter les méthodes **prepare()**, **hasNext()**, et **next()** qui permettent respectivement de :

- Préparer les éléments de la boucle.
- Indiquer si il y a encore un élément.
- Accéder à l'élément suivant.

Ainsi, le tag appellera la méthode **prepare()** à l'initialisation, puis les méthodes **hasNext()** et **next()** autant de fois que nécessaire...

Par exemple, pour parcourir les différents caractères d'une chaîne, on peut utiliser le code suivant :

Exemple

```
public class StringCharTag extends LoopTagSupport {

        private String string;
        private int position;
        private int length ;

        public void setString(String string) {
                this.string = string;
        }

        protected void prepare() {
                this.position = 0;

                if (string!=null)
                        this.length = string.length();
                else
                        this.length = 0;
        }

        protected boolean hasNext() {
                return (this.position < this.length)
        }

        protected Object next() {
                Character c = new Character ( string.charAt(position) );
                position++;
                return c;
        }

}
```

Le principal avantage de cette solution vient du fait qu'il est possible d'apporter un grand nombre de condition sur la boucle. En effet, les tags itératifs de la **JSTL** sont tous basés sur cette classe. La classe **LoopTagSupport** gère en effet tous les attributs standards des boucles. Il suffit donc d'assigner les valeurs aux attributs du même nom pour pouvoir en profiter...

> *Attention, il faut pour cela définir les méthodes **getter()** et **setter()** de ces différents attributs de **LoopTagSupport**, et les déclarer dans le descripteur de taglib...*

7.4 - Accès aux données localisées

La classe **javax.servlet.jsp.jstl.fmt.LocaleSupport** permet elle d'accéder aux données localisées de la même manière que le tag **<fmt:message/>**. Elle possèdent en effet plusieurs méthodes statiques

getLocalizedMessage() afin d'accéder aux informations des **ResourceBundle**s dans la locale courante en précisant ou non des arguments et/ou le nom du **ResourceBundle** :

- **getLocalizedMessage(PageContext pageContext, String key)**
- **getLocalizedMessage(PageContext pageContext, key, Object[] args)**
- **getLocalizedMessage(PageContext pageContext, String key, String basename)**
- **getLocalizedMessage(PageContext pageContext, key, Object[] args, String basename)**

Par exemple, pour accéder à un message du **ResourceBundle** par défaut, il suffit d'utiliser le code suivant :

```
String message = LocaleSupport.getLocalizedMessage(pageContext, "message.key");
```

Conclusion

En apportant la plupart des fonctionnalités de bases d'une application web, la **JSTL** devrait s'imposer dans le développement d'application J2EE. En effet, bien que de nombreuses librairies de tags proposent déjà les mêmes fonctionnalités, la **JSTL** ne se présente pas comme une énième librairie concurrente, mais propose une harmonisation de toutes ces librairies.

Il y a de fortes chances que les prochains frameworks J2EE s'appuie sur la **JSTL** afin de se concentrer sur leurs fonctionnalités propres. Ainsi, le framework **Struts** de **Jakarta** propose déjà une version compatible avec la **JSTL 1.0** :

http://struts.apache.org/faqs/struts-el.html

En effet, la **JSTL** propose également la possibilité d'utiliser ses ressources (**Locale**, **ResourceBundle**, **DataSource**,...) de manière portable (indépendamment de son implémentation). De ce fait un frameworks ou une librairies basés sur la **JSTL** peuvent permettre un déploiement et une intégration simple dans un projet JSP/JSTL.

Enfin, le tiercé **JSTL/EL/Taglibs** change radicalement la conception de pages JSP, ce qui peut troubler les développeurs Java. En effet, les **scriptlets** Java sont amenées à disparaître et les pages JSP s'apparentes désormais plus à des fichiers XML...