

Présentation des JSP Tag Libraries (Taglibs).

par [F. Martini \(adiGuba\)](#)

Date de publication : 03/01/2005

Dernière mise à jour : 18/02/2007

Les librairies de tags JSP (Taglibs) permettent de définir des tags JSP afin d'effectuer des actions précises. Les pages JSP n'en deviennent que plus claires car cela limite l'utilisation de scriptlets Java... Ce tutoriel décrit le fonctionnement des librairies de tags (taglibs).

Mise à jours

Remerciement

Présentation

Qu'est-ce qu'un tag JSP

Comment utiliser une librairie de tag ?

Les différentes versions des taglibs

1 - Les JSP Taglibs 1.1

1.1 - Le descripteur de Taglib 1.1

1.1.1 - Le Doctype

1.1.2 - Description de la librairie

1.1.3 - Déclaration de tag

1.1.4 - Définir un attribut du tag

1.2 - L'interface Tag

1.2.1 - Exemple de tag : Hello World

1.2.2 - Gestion des attributs du tag

1.2.3 - Traitement conditionnel du corps

1.2.4 - Création de tags collaboratifs

1.3 - L'interface BodyTag

1.3.1 - Itérer sur le corps du tag

1.3.2 - Modifier le contenu du corps

1.3.3 - Interpréter d'autres langages de script

1.4 - La classe TagExtraInfo

1.4.1 - Création de variable de script

1.4.2 - Vérification des attributs

2 - Les JSP Taglibs 1.2

2.1 - Le descripteur de Taglib 1.2

2.1.1 - Le Doctype

2.1.2 - Description de la librairie

2.1.3 - Le validateur de taglib

2.1.4 - Les listeners

2.1.5 - Déclaration de tag

2.1.6 - Définir un attribut du tag

2.1.7 - Déclaration des variables de script

2.2 - L'interface IterationTag

2.2.1 - Exemple d'itération

2.2.2 - Exemple d'itération avec des variables de script

2.3 - L'interface TryCatchfinally

3 - Les JSP Taglibs 2.0

3.1 - Le descripteur de Taglib 2.0

3.1.1 - Le Doctype

3.1.2 - Description de la librairie

3.1.3 - Déclaration de tag

3.1.4 - Déclaration de tag-file

3.1.5 - Déclaration de fonction EL

3.1.6 - Définir un attribut du tag

3.2 - L'interface SimpleTag

3.2.1 - Itérer sur le corps du tag

3.3 - L'interface DynamicAttributes

3.3.1 - Exemple d'attributs dynamiques

3.4 - Les fichiers *.tag

3.4.1 - Déclaration des attributs

3.4.2 - Déclaration de variables

3.4.3 - Affichage du corps

3.5 - Les fonctions EL

- 3.6 - Utilisation de JspFragment
- 4 - Déploiement de taglib
 - 4.1 - Spécification des taglibs archivées
 - 4.2 - Et la documentation
- Conclusion

Mise à jours

Liste des modifications apportées après la première diffusion de ce tutoriel :

- Correction de la numérotation des titres et de renvois.
- Correction de la taille des colonnes des tableaux (version PDF).
- Une erreur s'était glissée dans la section "Les fichiers *.tag" : **uri** était indiqué à la place de **tagdir**.
- Ajout de la section "Déploiement de taglib".

Remerciement

Je tiens à remercier tout particulièrement [Vedaer](#) pour l'aide qu'il m'a apportée dans ce tutoriel, ainsi que [Ukyuu](#) pour avoir pris le temps de relire ce tutoriel...

Présentation

Les JSP Tag Libraries permettent la création et l'utilisation de bibliothèques de tags au sein des pages JSP.

Une JSP Taglib est une collection d'actions prédéfinies destinée à être utilisée dans une page JSP sous forme de tags (balises XML). Elle se compose d'un descripteur de taglib (Tag Librarie Descriptor) et d'un ensemble de classes Java implémentant l'interface JspTag. Le descripteur de taglib (*.tld) est un document XML qui décrit les associations entre les balises et la classe Java. Ces actions sont représentées dans le source JSP comme une balise XML. Lors de la compilation de la JSP, ces balises sont remplacées par des appels vers la classe Java correspondante.

On peut citer comme exemple les balises standard préfixées avec jsp :

```
<jsp:useBean id="monBean" scope="session" class="package.MonObject" >
  <jsp:setProperty name="monBean" property="monAttribut" value="1.0" />
</jsp:useBean>
<jsp:include page="maPage.jsp"/>
<jsp:redirect page="maPage.jsp"/>
etc.
```

L'utilisation de taglib permet de limiter l'utilisation de code Java dans une page JSP.

Une Taglib est composée de trois éléments :

- Le Tag Librarie Descriptor (fichier *.tld) qui effectue le mapping entre les tags et les classes Java.
- Les classes Java implémentant les différents Tag (implémentant l'interface Tag, ou une de ses interfaces filles IterationTag et BodyTag).
- Les classes Java implémentant TagExtraInfo afin d'apporter des informations supplémentaires sur les tags (optionnel).

Qu'est-ce qu'un tag JSP

Un tag JSP est en réalité une simple balise XML à laquelle est associée une classe Java. À la compilation d'une page JSP, ces tags sont remplacés par l'utilisation de ces classes Java qui implémentent une interface particulière.

La structure des tags est la suivante :

```
<prefix:nomDuTag attribut="valeur">
  Corps du tag
</prefix:nomDuTag>
```

On y retrouve les éléments suivants :

- Un **préfixe**, qui permet de distinguer les différentes taglibs utilisées.
- Le **nom du tag** de la bibliothèque.
- D'un certain nombre de couple d'attribut/valeur.
- D'un corps.

Ces deux derniers éléments sont optionnels et varient selon le tag lui-même.

Les tags JSP sont des balises XML, elles doivent donc être correctement fermées :

Exemples de tags JSP

```

<prefix:nomDuTag attribut="valeur" attribut2="valeur2" attribut3="valeur3"
  attribut4="valeur4" attribut5="valeur5" />

<prefix:nomDuTag attribut="valeur"></prefix:nomDuTag>

<prefix:nomDuTag attribut="valeur">
  <prefix:nomDuTag>corps</prefix:nomDuTag>
</prefix:nomDuTag>

<prefix:nomDuTag/>

```

Comment utiliser une librairie de tag ?

Une librairie de tags (taglib) nécessite l'utilisation d'un descripteur de taglib. Il s'agit d'un fichier XML qui décrit les différents tags de la librairie.

- Le descripteur de fichier est séparé des classes Java ...
- Le descripteur de fichier est inclut dans le Jar avec les classes (il possède alors le nom suivant : "META-INF/taglib.tld").

Afin de pouvoir utiliser une taglib dans un fichier JSP, il faut donc la déclarer avec la directive **taglib**. Respectivement avec le code suivant :

```

<%@ taglib uri="/WEB-INF/taglib.tld" prefix="tag-prefix" %>
// ou si le fichier tld est dans le Jar :
<%@ taglib uri="/WEB-INF/lib/taglib.jar" prefix="tag-prefix" %>

```

La première ligne permet d'utiliser un descripteur de taglib indépendant, tandis que la seconde ligne utilise le descripteur de taglib **/META-INF/taglib.tld** du fichier jar.

L'attribut **prefix** indique le préfixe qui sera utilisé dans la page JSP pour les tags de cette taglib.

Toutefois, il est préférable de définir la taglib dans le fichier **web.xml** de l'application web, avec le code suivant :

```

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

```

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

```

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

```

L'attribut **uri** permet d'identifier la taglib à utiliser et doit correspondre à la valeur de la balise **taglib-uri** du fichier **web.xml**.

Les différentes versions des taglibs

Il existe actuellement trois versions des taglibs.

Chacune d'entre elle correspond à une version de J2EE, comme indiqué sur le tableau ci dessous :

Taglib	JSP	J2EE
1.1	1.1	1.2.1
1.2	1.2	1.3
2.0	2.0	1.4

1 - Les JSP Taglibs 1.1

Les JSP Taglibs 1.1 apportent un certain nombre de classes et d'interface Java permettant de réaliser des tags personnalisés.

Ces derniers étant référencés dans un descripteur de Taglib.

Les principales classes/interfaces des taglibs 1.1 sont :

- **Tag** qui est l'interface de base pour écrire un tag, et **TagSupport** qui correspond à son implémentation par défaut.
- **BodyTag**, une interface qui étend l'interface **Tag** en apportant une meilleure gestion du corps des tags (itérations, écriture bufférisée), et **BodyTagSupport** qui correspond à son implémentation par défaut.
- Enfin, la classe **TagExtraInfo** permet d'apporter des informations complémentaires sur les tags lors de la compilation des JSP.

1.1 - Le descripteur de Taglib 1.1

Le descripteur de taglib (fichier avec l'extension *.tld) renseigne le serveur d'application sur la librairie de tag.

Cette section décrit son format pour les Taglibs 1.1.

1.1.1 - Le Doctype

Le descripteur de taglib est un fichier XML décrit par un fichier DTD (Document Type Definition) fournit par Sun.

Il prend donc la forme suivante :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  ...
</taglib>
```

1.1.2 - Description de la librairie

La balise **taglib** de base du descripteur de taglibs accepte les balises suivantes :

Nom	Description	Type
tlibversion	Le numéro de la version de la librairie de tag.	obligatoire Format : <code>[0-9]*{ "[0-9] }0..3</code>
jspversion	Le numéro de version des JSP requis pour la librairie.	optionnel (défaut: 1.1)

Nom	Description	Type
shortname	Définit un préfixe pour les tags de la librairie. Attention : Le prefix utilisé sera toujours celui de la directive taglib de la page JSP. Celui définit ici permet aux éditeurs JSP d'insérer la directive taglib avec le préfixe indiqué.	obligatoire
uri	Définit une URI (Uniform Resource Identifier) qui identifie la taglibrairie. Il s'agit de l'URI à utiliser pour identifier la taglib dans le web.xml et les pages JSP.	optionnel
info	Un texte de description de la librairie.	optionnel
tag	Description d'un tag de la librairie. (Voir la section Déclaration de tag)	Au moins une.

1.1.3 - Déclaration de tag

La balise **tag** permet de faire un mapping entre un nom de balise et une classe Java (à l'instar de web.xml qui permet la même chose entre une URL et une Servlet).

Elle accepte les balises suivantes :

Nom	Description	Type
name	Le nom du tag tel qu'il devra être utilisé dans les JSP.	obligatoire
tagclass	Le nom complet (package compris) de la classe Java qui représente ce tag (sous-classe de javax.servlet.jsp.tagext.Tag).	obligatoire
teiclass	Le nom complet (package compris) de la classe Java qui apporte des informations supplémentaires sur le tag (sous-classe de javax.servlet.jsp.tagext.TagExtraInfo).	optionnel
bodycontent	Indique le type de contenu du corps du tag : empty : Le tag n'accepte aucun corps (une exception sera lancée si il est utilisé avec un corps quelconque). JSP : Le contenu du tag est interprétés comme du JSP. tagdependent : Le corps du tag ne sera pas interprété. Si il contient du code il sera affiché comme du simple texte.	optionnel (défaut : JSP)
info	Un texte de description du tag.	optionnel
attribute	Définition des différents attributs du tag	optionnel une balise par attribut. Cf : Définir un attribut du tag.

1.1.4 - Définir un attribut du tag

La balise **attribute** permet de définir les attributs du tag JSP. Seul les attributs décrits dans le TLD seront acceptés. L'utilisation de nom d'attribut inexistant dans le TLD provoquera une exception à la compilation.

La balise **attribute** accepte les balises suivantes :

Nom	Description	Type
name	Le nom de l'attribut tel qu'il devra être utilisé dans les JSP.	obligatoire
required	Indique si l'attribut est requis (valeur possible : true/false ou yes/no).	optionnel (défaut : false)
rtexprvalue	Indique si l'attribut peut être le résultat d'une scriptlet (valeur possible : true/false ou yes/no).	optionnel (défaut : false)

Si un attribut **required** est absent, une exception sera lancée.

L'utilisation de **rtexprvalue** permet d'utiliser le résultat d'une expression de scriptlet comme valeur d'attribut (par exemple `<%=monObjet%>`).

*Tous les attributs utilisés dans le tag de la page JSP seront passés à la classe qui représente le tag via leurs mutateurs respectifs (méthode **setNomDeLAttribut()**).*

*Pour chaque attribut, il faut donc que le mutateur correspondant soit présent dans la classe Java. Par exemple, si on définit un attribut **valeur**, son mutateur pourrait être :*

```
public void setValeur (String valeur) {
    this.valeur = valeur;
}
```

*On peut également utiliser des types primaires (**int**, **long**, **double**, **boolean**, etc.) en paramètre du mutateur de l'attribut.*

*Dans ce cas, une exception peut être lancée si la valeur est incorrecte (par exemple, **NumberFormatException**).*

*Si **rtexprvalue** vaut **true** (ou **yes**), l'attribut n'est pas forcément du type **String**, mais peut être n'importe quel objet Java ...*

*Attention aux **ClassCastException** que cela peut engendrer ...*

1.2 - L'interface Tag

L'interface **Tag** permet la création de Tag de base. Elle définit six méthodes à implémenter et quatre constante.

L'évaluation d'un Tag JSP dans une page JSP aboutit aux appels suivants :

- Les méthodes **setParent(Tag)** et **setPageContext(PageContext)** sont renseignées, ainsi que d'éventuels attributs présents dans le tag.
- La méthode **doStartTag()** est appelée. Son code de retour détermine l'affichage du contenu de la balise. Si le

retour vaut **Tag.EVAL_BODY_INCLUDE**, le corps est évalué et écrit dans le **JspWriter** de la page, mais il est ignoré si il vaut **Tag.SKIP_BODY**. Si **Tag.EVAL_BODY_INCLUDE** est retourné alors que la balise n'a pas de corps, il est ignoré.

- La méthode **doEndTag()** est appelée. Son code de retour détermine si le reste de la page doit être évalué ou pas. Si le retour vaut **Tag.EVAL_PAGE**, le reste de la page est évalué, mais il est ignoré si le retour vaut **Tag.SKIP_PAGE**.

Enfin, la méthode **release()** est appelée avant que l'objet ne soit rendu au garbage collector. Attention toutefois, afin d'éviter trop d'allocation, les tags sont conservés en cache et réutilisés (tout comme les Servlet/JSP) ...

La classe **javax.servlet.jsp.tagext.TagSupport** propose une implémentation par défaut de l'interface **Tag**.

1.2.1 - Exemple de tag : Hello World

Comme bien souvent, le premier exemple est un simple "Hello world" ...

Notre tag va donc se contenter d'afficher le texte **"Hello World"** dans la page JSP.

Il ne possèdera donc ni contenu, ni attribut.

La classe HelloTag

```
public class HelloTag extends TagSupport {
    public int doStartTag() throws JspException {
        try {
            pageContext.getOut().println ("Hello World !");
        } catch (IOException e) {
            throw new JspException ("I/O Error", e);
        }
        return SKIP_BODY;
    }
}
```

Explication :

- On étend **TagSupport** afin de bénéficier des implémentations par défaut des méthodes de **Tag**.
- On surcharge **doStartTag()**, dans lequel on se contente d'écrire la chaîne "Hello World" dans la sortie de la page courante (**pageContext** est initialisé par l'implémentation par défaut de **setPageContext()**).
- On retourne **Tag.SKIP_BODY** car on ne veut pas traiter le corps de la balise.

Le mapping de notre tag dans le fichier de descripteur correspond à ceci :

```
<tag>
  <name>hello</name>
  <tagclass>tutoriel.taglib.HelloTag</tagclass>
  <bodycontent>empty</bodycontent>
</tag>
```

Ainsi, dans une page JSP, le code suivant :

```
<taglib:hello/>
```

affichera dans le navigateur :

```
Hello World !
```

1.2.2 - Gestion des attributs du tag

Nous allons améliorer notre tag précédent en lui ajoutant un attribut **name**. Si l'attribut **name** est présent, on devra afficher "**Hello**" suivi de la valeur de l'attribut name, sinon on affiche "**Hello World**".

Nous devrions pour cela modifier le mapping du tag afin de spécifier qu'il peut avoir un attribut **name** optionnel (voir la section 1.1.4 pour plus de détail) :

```
<tag>
  <name>hello</name>
  <tagclass>tutoriel.taglib.HelloTag</tagclass>
  <bodycontent>empty</bodycontent>
  <attribute>
    <name>name</name>
  </attribute>
</tag>
```

La balise **<attribute>** accepte également deux autres balises :

<required> permet d'indiquer que l'attribut est requis (son absence provoquera une erreur de compilation).

<rtexprvalue> indique que la valeur de l'attribut peut être le résultat d'une expression (**<%=expression%>**).

Si l'attribut **name** est présent, sa valeur sera passée à la classe **HelloTag** grâce à son mutateur **setName()**. Il est donc obligatoire d'implémenter cette méthode.

Au final, notre classe **HelloTag** pourrait ressembler à ceci :

```
public class HelloTag extends TagSupport {
    private String name = null;

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

    public int doStartTag() throws JspException {
        if (this.name==null)
            this.name = "World";
        try {
            pageContext.getOut().println ("Hello " + this.name + " !");
        } catch (IOException e) {
            throw new JspException ("I/O Error", e);
        }
        return SKIP_BODY;
    }
}
```

Ainsi, dans la page JSP :

```
<taglib:hello/><br/>
<taglib:hello name="Fred"/>
```

donnera l'affichage suivant :

```
Hello World !
Hello Fred !
```

*Si le nom de l'attribut n'est pas spécifié dans le TLD ou qu'aucun mutateur (**setXXX()**) n'est implémenté, une exception sera lancée lors de la compilation de la page JSP ...*

1.2.3 - Traitement conditionnel du corps

Un tag peut également posséder un corps. Étant donné que l'évaluation du corps dépend du retour de la méthode **doStartTag()**, on peut facilement évaluer ou non le corps du tag selon une condition quelconque ...

Nous allons écrire un tag qui affichera son contenu seulement si le paramètre désigné par l'attribut **name** est présent dans le **request** de la page courante.

Le code de ce tag pourrait être :

ParamPresentTag

```
public class ParamPresentTag extends TagSupport {
    private String name = null;

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

    public int doStartTag() throws JspException {
        String value = pageContext.getRequest().getParameter(this.name);
        if (value!=null)
            return EVAL_BODY_INCLUDE;
        return SKIP_BODY;
    }
}
```

Nous utiliserons le mapping suivant dans notre fichier de description :

taglib.tld

```
<tag>
  <name>paramPresent</name>
  <tagclass>tutorial.taglib.ParamPresentTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <attribute>
    <name>name</name>
    <required>>true</required>
  </attribute>
</tag>
```

L'attribut **name** est requis, ainsi un oubli de l'attribut et une exception sera levée à la compilation de la page JSP. Enfin, le corps du tag possède un contenu **JSP**, c'est à dire qu'il sera interprété comme du code JSP (il s'agit du fonctionnement par défaut).

Le code suivant :

```
<taglib:paramPresent name="page">
  Le paramètre page vaut : <%=request.getParameter("page")%>.
</taglib:paramPresent>
```

N'affichera rien si la page JSP est appelée sans paramètre "page". Mais si elle est appelée avec **page=valeur**, on obtiendra le résultat suivant :

```
Le paramètre page vaut : valeur.
```

1.2.4 - Création de tags collaboratifs

L'interface **Tag** définit les méthodes **setParent()** et **getParent()** afin de renseigner le tag sur son tag parent (si il existe).

Cela permet une collaboration entre différents tags. Nous allons voir comment utiliser la méthode **getParent()** afin de simuler un switch/case grâce à des simples tags.

Ainsi, la classe qui représente le tag 'switch' pourrait correspondre à :

SwitchTag

```
public class SwitchTag extends TagSupport {
    private Object test = null;

    public void setTest (Object obj) {
        this.test = obj;
    }

    public int doStartTag() throws JspException {
        return EVAL_BODY_INCLUDE;
    }

    public boolean isValid (Object caseValue) {
        if (this.test==caseValue) return true;
        if (this.test!=null && this.test.equals(caseValue)) return true;
        return false;
    }
}
```

Explication :

La classe **SwitchTag** se contente d'évaluer son corps. Par contre, elle possède une méthode **isValid()** qui permettra de savoir si un objet est égal à son attribut **test**. Cette méthode sera utilisée par les tags 'case' :

CaseTag

```
public class CaseTag extends TagSupport {
    public Object value = null;

    public void setValue (Object object) {
        this.value = object;
    }

    public int doStartTag() throws JspException {
        if ( getParent() instanceof SwitchTag ) {
            SwitchTag parent = (SwitchTag) getParent();
            if (parent.isValid(this.value))
                return EVAL_BODY_INCLUDE;
            return SKIP_BODY;
        }
        throw new JspException ("Le tag case doit être à l'intérieur du tag switch.");
    }
}
```

Explication :

La classe **CaseTag** récupère son tag parent et vérifie qu'il est bien du type de **SwitchTag** (sinon, cela signifie que le tag est mal utilisé, donc on peut lancer une exception).

Il utilise ensuite la méthode **isValid()** de son parent afin de déterminer si il doit afficher son contenu ou pas.

Ainsi, le code suivant :

```
<taglib:switch test="2">
  <taglib:case value="0">Zéro</taglib:case>
  <taglib:case value="1">Un</taglib:case>
  <taglib:case value="2">Deux</taglib:case>
  <taglib:case value="3">Trois</taglib:case>
</taglib:switch>
```

affichera le résultat suivant :

Deux

*La classe **TagSupport** propose la méthode statique **findAncestorWithClass()** qui permet de retrouver un tag parent de plus haut niveau selon son type.*

1.3 - L'interface BodyTag

L'interface **BodyTag** étend l'interface **Tag**. Elle hérite donc de toutes ses propriétés, mais permet un traitement bufférisé du corps de la balise ainsi que des itérations sur le corps du tag.

L'interface **BodyTag** définit trois nouvelles méthodes :

- **setBodyContent()** afin d'affecter un objet de type **BodyContent** qui correspond au buffer.
- **doInitBody()** qui est appelée avant la première évaluation du corps du tag.
- **doAfterBody()** qui est appelée après chaque évaluation du corps.

La méthode **doAfterBody()** permet d'effectuer des itérations sur le corps du tag, en fonction de son retour :

- **BodyTag.EVAL_BODY_TAG** permet de réévaluer le corps du tag une nouvelle fois.
- **Tag.SKIP_BODY** permet de passer à la fin du tag (méthode **doEndTag()**).

Les méthodes **setBodyContent()** et **doInitBody()** sont appelées une seule fois si et seulement si la méthode **doStartTag()** renvoie **BodyTag.EVAL_BODY_TAG**.

Le contenu du corps du tag sera alors écrit dans le buffer représenté par l'objet **BodyContent** passé à **setBodyContent()**.

il faudra recopier manuellement ce buffer dans la sortie de la page JSP (avec **getEnclosingWriter()** de l'objet **BodyContent**, ou **getPreviousOut()** de la classe **BodyTagSupport**).

*Dans la version 1.2 des Taglibs, la valeur **BodyTag.EVAL_BODY_TAG** est dépréciée (voir Les JSP Taglibs 1.2).*

*Elle est remplacée par **BodyTag.EVAL_BODY_BUFFERED** et **IterationTag.EVAL_BODY_AGAIN**, respectivement lorsque **BodyTag.EVAL_BODY_TAG** devait être retourné par les méthodes **doStartTag()** et **doAfterBody** ...*

La classe **javax.servlet.jsp.tagext.BodyTagSupport** propose une implémentation par défaut de l'interface **BodyTag**.

1.3.1 - Itérer sur le corps du tag

Afin de comprendre le fonctionnement d'un tag itératif, nous allons écrire un tag qui effectuera un certain nombre de boucles selon un paramètre **count**.

Son code pourrait ressembler à cela :

```
IterateTag
public class IterateTag extends BodyTagSupport {
    private int count = 0;
    private int current;

    public void setCount(int i) {
        count = i;
    }

    public int doStartTag() throws JspException {
        current = 0;
        if (current < count)
            return Tag.EVAL_BODY_TAG;
        return Tag.SKIP_BODY;
    }

    public int doAfterBody() throws JspException {
        current++;
        if (current < count)
            return IterationTag.EVAL_BODY_TAG;
        return Tag.SKIP_BODY;
    }
}
```

Explication :

L'attribut **count** contiendra la valeur de l'attribut de la balise. Il n'y a pas de conversion String/int à effectuer car elle est automatique puisque la méthode **setCount()** prend un **int** en paramètre. En cas de valeur incorrecte, une exception sera lancée ...

Dans **doStartTag()**, on initialise **current** qui contiendra le nombre de lignes déjà affichées. Il est important de l'initialiser dans **doStartTag()** car la même instance de Tag peut être utilisée plusieurs fois.

Enfin, à chaque passage dans **doAfterBody()**, on évalue la condition de fin afin de retourner le code correspondant ...

Ainsi, le code suivant :

```
<taglib:iterate count="3">
    Cette ligne sera affichée trois fois<br/>
</taglib:iterate>
```

Donnera bien le résultat suivant :

```
Cette ligne sera affichée trois fois

Cette ligne sera affichée trois fois
```

```
Cette ligne sera affichée trois fois
```

Le fonctionnement des itérations avec la classe **BodyTag** est similaire à celui de la classe **IterationTag** des taglibs 1.2.

Je vous invite donc à consulter le passage sur l'interface **IterationTag**.

1.3.2 - Modifier le contenu du corps

En utilisant **BodyTag.EVAL_BODY_TAG**, le corps des tags n'est pas écrit directement dans la sortie de la page JSP mais dans un buffer. Le premier intérêt de cette méthode est de pouvoir modifier dynamiquement ce corps avant de l'envoyer vers la sortie de la page.

Par exemple, le tag suivant permet de mettre le texte du corps du tag en majuscule :

UpperCaseTag

```
public class UpperCaseTag extends BodyTagSupport {
    public int doStartTag() throws JspException {
        return EVAL_BODY_TAG;
    }
    public int doAfterBody() throws JspException {
        try {
            if ( getBodyContent()!=null ) {
                String bodyString = getBodyContent().getString();
                bodyString = bodyString.toUpperCase();
                getBodyContent().getEnclosingWriter().println( bodyString );
            }
        } catch (IOException e) {
            throw new JspException (e);
        }
        return EVAL_PAGE;
    }
}
```

Explication :

On se contente dans **doAfterbody()** de récupérer l'objet **BodyContent** sous forme de **String** que l'on passe en majuscule.

getBodyContent().getEnclosingWriter() permettant d'accéder au **JspWriter** de la page JSP.

Ainsi, le code suivant :

```
<taglib:upperCase>Ce texte sera mis en majuscule, ainsi que le résultat de ce
<%= "scriptlet" %>.</taglib:upperCase>
```

affichera le texte en majuscule :

```
CE TEXTE SERA MIS EN MAJUSCULE, AINSI QUE LE RÉSULTAT DE CE SCRIPTLET.
```

1.3.3 - Interpréter d'autres langages de script

L'interface **BodyTag** nous permet également d'utiliser d'autres langages de script au sein d'une page JSP. En effet, le corps du tag n'étant plus directement écrit dans la page JSP, on peut très bien l'envoyer à un interpréteur et n'afficher que le résultat ...

Ainsi, ce tag envoie le contenu de son corps à l'interpréteur PERL :

PerlTag

```
public class PerlTag extends BodyTagSupport {

    private Process process = null;

    protected void startProcess () throws JspException {
        try {
            process = Runtime.getRuntime().exec("perl");
        } catch (IOException e) {
            throw new JspException ("Erreur de creation du process 'perl'.", e);
        }
    }

    protected void writeBody () throws JspException {
        // On recupere les flux STDIN du process 'perl'
        OutputStream stdin = process.getOutputStream();
        OutputStreamWriter stdinWriter = new OutputStreamWriter (stdin);

        try {
            // On écrit dans le STDIN du process le contenu du tag :
            getBodyContent().writeOut( stdinWriter );
        } catch (IOException e) {
            throw new JspException ("Erreur d'écriture du body.", e);
        } finally {
            // Fermeture des flux ...
            try { stdinWriter.close(); } catch (IOException e) {}
            try { stdin.close(); } catch (IOException e) {}
        }
    }

    protected void readErrors () throws JspException {
        // On recupere les flux STDERR du process 'perl'
        InputStream stdout = process.getErrorStream();
        InputStreamReader stdoutReader = new InputStreamReader (stdout);
        BufferedReader stdoutBuffer = new BufferedReader (stdoutReader);
        StringBuffer errorBuffer = null;

        try {
            // On récupère chaque ligne du STDERR dans un buffer :
            String line = null;
            while ( ( line = stdoutBuffer.readLine() ) != null ) {
                if (errorBuffer==null)
                    errorBuffer = new StringBuffer ();
                errorBuffer.append(line);
            }
        } catch (IOException e) {
            throw new JspException ("Erreur de lecture du résultat.", e);
        } finally {
            // Fermeture des flux ...
            try { stdoutBuffer.close(); } catch (IOException e) {}
            try { stdoutReader.close(); } catch (IOException e) {}
            try { stdout.close(); } catch (IOException e) {}
        }

        // Si le buffer n'est pas vide, on envoi une exception
        if (errorBuffer!=null)
            throw new JspException ("Perl Script Error : " + errorBuffer);
    }

    protected void readResult () throws JspException {
        // On recupere les flux STDOUT du process 'perl'
    }
}
```

PerlTag

```

InputStream stdout = process.getInputStream();
InputStreamReader stdoutReader = new InputStreamReader (stdout);
BufferedReader stdoutBuffer = new BufferedReader (stdoutReader);

try {
    // On récupère chaque ligne du STDOUT et on l'affiche :
    String line = null;
    while ( ( line = stdoutBuffer.readLine() ) != null ) {
        getPreviousOut().print(line);
    }
} catch (IOException e) {
    throw new JspException ("Erreur de lecture du résultat.", e);
} finally {
    // Fermeture des flux ...
    try { stdoutBuffer.close(); } catch (IOException e) {}
    try { stdoutReader.close(); } catch (IOException e) {}
    try { stdout.close(); } catch (IOException e) {}
}

public int doStartTag() throws JspException {
    return EVAL_BODY_TAG;
}

public int doAfterBody() throws JspException {
    startProcess();
    writeBody ();
    readErrors ();
    readResult ();

    return SKIP_BODY;
}
}

```

Explication :

doStartTag() renvoie **EVAL_BODY_TAG** afin que le contenu du corps soit écrit dans un buffer.

Une fois dans **doAfterBody()** :

- **startProcess()** lance le processus "perl" (l'interpréteur perl installé sur le système).
- **writeBody()** écrit le contenu du corps dans le flux **stdin** du processus "perl".
- **readErrors()** récupère le flux **stderr** du processus "perl" afin de lancer une exception en cas d'erreur dans le script.
- **readResult()** récupère le flux **stdout** du processus "perl" et le copie dans la page JSP.

Et le code suivant :

```

<taglib:perl>
$lang = "perl";
printf ( "Utilisation du langage %s dans une page JSP", $lang );
</taglib:perl>

```

Affiche correctement le résultat du script PERL :

```
Utilisation du langage perl dans une page JSP
```

*Pour cet exemple, il est préférable d'utiliser dans le descripteur de taglib la valeur **tagdependent** pour le **<bodycontent>**.*

Ceci afin d'éviter que des éléments du script PERL soient interprétés par le JSP ...

1.4 - La classe TagExtraInfo

La classe `javax.servlet.jsp.tagext.TagExtraInfo` permet de fournir des informations supplémentaires sur le tag au moment de la compilation de la page JSP.

Cette classe définit les méthodes suivantes :

- **setTagInfo()** et **getTagInfo()** qui permettent d'accéder aux informations sur le tag contenu dans le descripteur de taglib (TLD). **setTagInfo()** est renseigné automatiquement par le serveur d'application.
- **getVariableInfo()** qui permet de mapper des éléments des scopes vers des variables de script dans la page JSP.
- **isValid()** qui permet de valider le tag (et ses attributs) avant même que la classe du tag soit exécutée ...

En plus de l'objet **TagInfo** renseigné par **setTagInfo()** et qui permet d'accéder aux informations du fichier TLD, les méthodes **getVariableInfo()** et **isValid()** possèdent en paramètre un objet de type **TagData** permettant d'accéder aux différents attributs du tag et à leurs valeurs.

Attention, si la valeur d'un attribut est le résultat d'une scriptlet (<%=object%> avec `rtexprvalue=true`), on ne peut évidemment pas accéder à cette valeur ...

1.4.1 - Création de variable de script

Il est possible de créer une variable de script dans la page JSP depuis un tag. Nous allons donc modifier le tag **paramPresent** (voir la section Traitement conditionnel du corps) afin qu'il déclare une variable de script nommée "value" dans la page JSP. Cette variable devra contenir la valeur du paramètre.

Nous allons commencer par modifier le code du tag afin qu'il place la valeur du paramètre dans le scope "page" de la JSP :

```
public class ParamPresentTag extends TagSupport {
    private String name = null;

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

    public int doStartTag() throws JspException {
        String value = pageContext.getRequest().getParameter(this.name);
        if (value!=null) {
            pageContext.setAttribute("value", value);
            return EVAL_BODY_INCLUDE;
        }
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
        pageContext.removeAttribute("value");
        return EVAL_PAGE;
    }
}
```

L'attribut "value" est donc placé dans le scope "page" au début du tag et il est effacé à la fin du tag. On peut désormais utiliser le code suivant pour afficher la valeur du paramètre :

```
<taglib:paramPresent name="page">
    Le paramètre page vaut : <%=pageContext.getAttribute("value")%>.
</taglib:paramPresent>
```

Pour le moment, cela apporte peu d'intérêt, mais c'est nécessaire car la variable de script sera automatiquement mise à jour avec le contenu de l'attribut du même nom dans le scope "page"

Nous allons donc déclarer une variable de script. Pour cela, nous avons besoin d'écrire une classe qui hérite de la classe `javax.servlet.jsp.tagext.TagExtraInfo` en surchargeant la méthode `getVariableInfo()` :

```
ParamPresentTEI
public class ParamPresentTEI extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData tagData) {
        VariableInfo[] vi = new VariableInfo[1];
        vi[0] = new VariableInfo("value", "java.lang.String", true, VariableInfo.NESTED);
        return vi;
    }
}
```

La méthode `getVariableInfo()` retourne un tableau de `VariableInfo`. Chaque élément du tableau définit une variable de script. Le constructeur de `VariableInfo` prend les quatre paramètres suivant :

- 1 Le nom de la variable de script
- 2 Le nom du type de la variable
- 3 Un booléen qui indique si la variable doit être déclarée (**true**) ou si on doit réutiliser une variable déjà déclarée (**false**).
- 4 Enfin, on définit la zone de portée de la variable, qui peut prendre les valeurs suivantes :
 - **VariableInfo.AT_BEGIN** : La variable est déclarée au début du tag et reste valable jusqu'à la fin du bloc contenant le tag.
 - **VariableInfo.AT_END** : La variable est déclarée à la fin du tag et reste valable jusqu'à la fin du bloc contenant le tag.
 - **VariableInfo.NESTED** : La variable est déclarée au début du tag et n'est valable qu'à l'intérieur du tag.

Les différentes déclarations correspondent au code suivant :

```
<%
//
// Les variables AT_BEGIN sont déclarées ici
//
{ // Début du bloc du tag
    //
    // Les variables NESTED sont déclarées ici
    //
    ...
    tag.doStartTag();
    ...
    tag.doInitBody();
    ...
    tag.doAfterBody();
    ...
    tag.doEndTag();
    ...
} // Fin du bloc du tag
//
// Les variables AT_END sont déclarées ici
//
%>
```

Il faut ensuite déclarer cette classe dans le descripteur de taglib grâce à la balise `teiclass` :

```
taglib.tld
<tag>
    <name>paramPresent</name>
    <tagclass>tutoriel.taglib.ParamPresentTag</tagclass>
```

taglib.tld

```

<teiclass>tutoriel.taglib.ParamPresentTEI</teiclass>
<bodycontent>JSP</bodycontent>
<attribute>
  <name>name</name>
  <required>true</required>
</attribute>
</tag>

```

Désormais, la classe **ParamPresentTEI** sera utilisée pour définir les variables de script du tag **paramPresent**. On peut donc désormais utiliser le code suivant :

```

<taglib:paramPresent name="page">
  Le paramètre page vaut : <%=value%>.<br/>
  En Majuscule : <%=value.toUpperCase()%>.
</taglib:paramPresent>

```

Explication :

Une variable de script nommé "value" est créée dans la page à l'intérieur du corps du tag. Son contenu est mis à jours selon le contenu du scope "page" lorsque c'est nécessaire.

Cette variable de script est bien du type **java.lang.String**, comme indiqué par le retour de la méthode **getVariableInfo()** (comme le montre l'appel de la méthode **toUpperCase()**).

*Le nom de la variable de script peut être également 'dynamique'. On peut par exemple utiliser le paramètre **TagData** de **getVariableInfo()** afin d'accéder aux attributs du tag.*

*Toutefois, étant donné que la classe **TagExtraInfo** est utilisée à la compilation, on ne peut pas accéder aux attributs dont la valeur est le résultat d'une scriptlet (<%=object%> avec **rtexprvalue=true**).*

En réalité, la variable de script est mise à jours avec l'attribut du même nom existant dans le premier scope dans l'ordre suivant :

"page", "request", "session", et "application" Il est ainsi possible d'utiliser un autre scope que "page" mais cela peut poser des conflits si un attribut du même nom existe dans une scope précédent ...

1.4.2 - Vérification des attributs

La méthode **isValid()** permet de valider dynamiquement les attributs du tag avant qu'il ne soit exécuté.

Toutefois, la méthode **isValid()** étant appelée à la compilation, elle ne permet pas de vérifier la valeur des attributs dont la valeur est le résultat d'une scriptlet (<%=object%> avec **rtexprvalue=true**).

Nous allons ainsi surcharger la méthode **isValid()** de la classe **ParamPresentTEI** afin de vérifier que son attribut **name** ne soit pas vide :

```

public boolean isValid(TagData tagData) {
    if (tagData.getAttributeString("name").trim().equals(""))
        return false;
    return true;
}

```

Désormais, si on utilise le code suivant :

```
<tag11:paramPresent name="">corps</tag11:paramPresent>
```

Nous aurons bien une exception à la compilation ...

Cette validation par la classe **TagExtraInfo** a deux intérêts :

- La validation est effectuée pour tous les tags à la compilation. Les tags non évalués lors de l'affichage de la JSP (par exemple : si ils se trouvent dans un autre tag qui n'évalue pas son corps) seront quand même validés ...
- La vérification peut très bien être longue, elle ne sera effectuée qu'à la compilation ...

2 - Les JSP Taglibs 1.2

La version 1.2 des Taglibs apporte deux nouvelles interfaces :

- **IterationTag** (qui étend de **Tag**) et permet d'effectuer des itérations.
- **TryCatchFinally** permet une meilleure gestion des exceptions.

De plus, l'interface **BodyTag** n'hérite plus directement de l'interface **Tag** mais d'**IterationTag**.

Enfin, la variable statique **BodyTag.EVAL_BODY_TAG** est dépréciée au profit de **IterationTag.EVAL_BODY_BUFFERED** et **BodyTag.EVAL_BODY_AGAIN** (respectivement pour la valeur de retour de **doStartTag()** et de **doAfterBody()**).

2.1 - Le descripteur de Taglib 1.2

La version 1.2 des taglibs possède son propre descripteur de taglib.

Cette section décrit ce nouveau descripteur par rapport à la version 1.1.

2.1.1 - Le Doctype

Le descripteur de taglib est un fichier XML décrit par un fichier DTD (Document Type Definition) fournit par Sun.

Le doctype d'une taglib 1.2 doit avoir le doctype suivant :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
...
</taglib>
```

2.1.2 - Description de la librairie

Avec la version 1.2, les noms de balise ont été normalisés : les termes des noms de balises sont séparés par des tirets (exemple: **jspversion** devient **jsp-version**. Les anciens noms de balises sont indiqués entre parenthèses.

La balise **taglib** de base du descripteur de taglibs 1.2 accepte les balises suivantes :

Nom (nom 1.1)	Description	Type
tlib-version (<i>tlibversion</i>)	Le numéro de la version de la librairie de tag.	obligatoire Format : <code>[0-9]*{ "[0-9] }0..3</code>
jsp-version (<i>jspversion</i>)	Le numéro de version des JSP requis pour la librairie.	obligatoire (minimum : 1.2)

Nom (nom 1.1)	Description	Type
short-name (shortname)	Définit un préfixe pour les tags de la librairie. Attention : Le prefix utilisé sera toujours celui de la directive taglib de la page JSP. Celui définit ici permet aux éditeurs JSP d'insérer la directive taglib avec le préfixe indiqué ici.	obligatoire
uri (uri)	Définit une URI (Uniform Resource Identifier) qui identifie la taglibrairie. Il s'agit de l'URI à utiliser pour identifier la taglib dans le web.xml et les pages JSP.	optionnel
display-name	Nom de la librairie qui représentera la taglib dans les outils de développement graphiques compatibles.	optionnel
small-icon	Chemin relatif d'une image GIF ou JPEG (format 16x16) qui représentera la taglib dans les outils de développement graphiques compatibles.	optionnel
large-icon	Chemin relatif d'une image GIF ou JPEG (format 32x32) qui représentera la taglib dans les outils de développement graphiques compatibles.	optionnel
description (info)	Un texte de description de la librairie.	optionnel
validator	Définit une classe qui valide la page JSP qui utilise cette librairie de tag. (Voir la section Le validateur de taglib)	optionnel
listener	Définit un listener qui sera instancié et qui surveillera les événements de l'application. (Voir la section Les listeners)	optionnel plusieurs listeners possible
tag	Description d'un tag de la librairie. (Voir la section Déclaration de tag)	Au moins une.

2.1.3 - Le validateur de taglib

Les Taglibs 1.2 permettent de valider le format de la page JSP lors de sa compilation. Cette validation n'est effectuée que pendant la compilation de la JSP.

La déclaration d'un validator accepte les éléments suivants :

Nom	Description	Type
validator-class	Le nom de la classe qui étend de javax.servlet.jsp.tagext.TagLibraryValidator .	obligatoire
init-param	Un (ou plusieurs) paramètre d'initialisation du validateur. Ils seront passés à l'instance du validateur via la méthode setInitParameters(Map) .	optionnel
description	Une description du validateur.	optionnel

Si un validateur est spécifié dans le descripteur de taglib, l'attribut **validator-class** doit correspondre au nom d'une classe qui étend la classe **javax.servlet.jsp.tagext.TagLibraryValidator**.

Cette classe possède la méthode suivante :

```
public ValidationMessage[] validate(String prefix, String uri, PageData page);
```

Lors de la compilation d'une page JSP qui utilise la **taglib**, la méthode **validate()** ci dessus est appelée avec les paramètres suivant :

- **prefix** : La valeur de l'attribut **prefix** de la directive **taglib**.
- **uri** : La valeur de l'attribut **uri** de la directive **taglib**.
- **page** : Un objet **PageData** qui permet d'accéder à un **InputStream** sur la vue XML de la JSP.

Cette méthode **validate()** permet donc de valider le format XML de la JSP à la compilation.

Si la méthode retourne un tableau non-null et non-vide (length>0), une erreur de compilation sera générée avec les messages contenus dans les différents **ValidationMessage** du tableau retourné ...

Le validateur permet donc de vérifier le format XML de la JSP.

En particulier, on pourra vérifier que les tags collaboratifs sont correctement utilisés ...

L'instance du validateur peut être réutilisée plusieurs fois par le conteneur de JSP. Toutefois, la méthode **release()** sera appelée avant de détruire l'instance du validateur ...

2.1.4 - Les listeners

Les listeners permettent de surveiller certains aspects de l'application web. Il s'agit des mêmes listeners que ceux qui peuvent être définis dans le fichier **web.xml** de votre application.

Afin de déclarer un listener dans le descripteur de taglib, on utilise le même code que dans le fichier **web.xml**, c'est à dire les balises suivantes :

```
<listener>
  <listener-class>package.ListenerClassName</listener-class>
</listener>
```

où **package.ListenerClassName** correspond à une classe qui implémente l'une des interfaces suivantes :

- **javax.servlet.ServletContextListener** afin de surveiller la création/destruction du **ServletContext**.
- **javax.servlet.ServletContextAttributesListener** afin de surveiller l'ajout/modification/suppression des attributs du **ServletContext**.
- **javax.servlet.http.HttpSessionListener** afin de surveiller la création/destruction des **HttpSession**.
- **javax.servlet.http.HttpSessionAttributeListener** afin de surveiller l'ajout/modification/suppression des attributs des **HttpSession**.

*J2EE 1.4 ajoute deux nouveaux listeners : **javax.servlet.ServletRequestListener** et **javax.servlet.ServletRequestAttributesListener** qui permettent de surveiller toutes les requests de l'application web...*

Le fonctionnement est identique aux listeners déclarés dans le **web.xml**. Cela permet simplement de regrouper des listeners propres à une taglib dans son propre descripteur afin de faciliter son déploiement.

On peut naturellement définir plusieurs listeners ...

Les listeners sont initialisés au démarrage de l'application : le serveur d'application recherche donc tous les

descripteurs de taglib lors de la séquence de démarrage. C'est pourquoi le descripteur de taglib doit se situer dans le répertoire **WEB-INF** ou l'un de ses sous-répertoires, ou dans un fichier jar du répertoire **WEB-INF/lib**.

Si le descripteur de taglib se situe dans un autre répertoire de l'application web, les listeners ne pourront pas être chargés ...

2.1.5 - Déclaration de tag

La balise **tag** du descripteur de taglibs 1.2 accepte les balises suivante (les noms entre parenthèses correspondent aux noms dans le TLD de la version 1.1, si il existe) :

Nom (<i>nom 1.1</i>)	Description	Type
name (<i>name</i>)	Le nom du tag tel qu'il devra être utilisé dans les JSP.	obligatoire
tag-class (<i>tagclass</i>)	Le nom complet (package compris) de la classe Java qui représente ce tag (sous-classe de javax.servlet.jsp.tagext.Tag).	obligatoire
tei-class (<i>teiclass</i>)	Le nom complet (package compris) de la classe Java qui apporte des informations supplémentaires sur le tag (sous-classe de javax.servlet.jsp.tagext.TagExtraInfo).	optionnel
body-content (<i>bodycontent</i>)	Indique le type de contenu du corps du tag : empty : Le tag n'accepte aucun corps (une exception sera lancée si il est utilisé avec un corps quelconque). JSP : Le contenu du tag est interprété comme du JSP. tagdependent : Le corps du tag ne sera pas interprété. Si il contient du code il sera affiché comme du simple texte.	optionnel (défaut : JSP)
display-name	Nom du tag utilisé dans les outils de développement graphiques compatibles.	optionnel
small-icon	Chemin relatif d'une image GIF ou JPEG (format 16x16) qui représentera le tag dans les outils de développement graphiques compatibles.	optionnel
large-icon	Chemin relatif d'une image GIF ou JPEG (format 32x32) qui représentera le tag dans les outils de développement graphiques compatibles.	optionnel
description (<i>info</i>)	Un texte de description du tag.	optionnel
variable	Déclaration des variables de script utilisées par le tag. (Voir [JSP 1.2] Déclaration des variables de script	optionnel Une balise par variable
attribute (<i>attribute</i>)	Définition des différents attributs du tag	optionnel une balise par attribut. Cf : Définir un attribut du tag.
exemple	Description d'un exemple d'utilisation du tag.	optionnel

2.1.6 - Définir un attribut du tag

La balise **attribute** permet de définir les attributs du tag JSP. Seul les attributs décrits dans le TLD seront acceptés. L'utilisation d'attribut non-indiqué dans le TLD provoquera une exception à la compilation.

La balise **attribute** du descripteur de taglibs 1.2 accepte les balises suivantes (les noms entre parenthèses correspondent aux noms dans le TLD de la version 1.1, si il existe) :

Nom	Description	Type
name	Le nom de l'attribut tel qu'il devra être utilisé dans les JSP.	obligatoire
required	Indique si l'attribut est requis (valeur possible : true/false ou yes/no).	optionnel (défaut : false)
rtexprvalue	Indique si l'attribut peut être le résultat d'une scriptlet (valeur possible : true/false ou yes/no).	optionnel (défaut : false)
type	Indique le type Java de l'attribut (si rtexprvalue==true).	optionnel (défaut : "java.lang.String")
description	Un texte de description de l'attribut.	optionnel

Si un attribut **required** est absent, une exception sera lancée.

L'utilisation de **rtexprvalue** permet d'utiliser le résultat d'une expression de scriptlet comme valeur d'attribut (par exemple `<%=monObjet%>`).

2.1.7 - Déclaration des variables de script

La balise **variable** permet de déclarer des variables de script depuis le descripteur de taglib, tout comme la méthode **getVariableInfo()** de la classe **TagExtraInfo** associé au tag (voir la section Création de variable de script).

*On ne peut toutefois pas cumuler les deux méthodes. Si une variable de script est déclarée dans le descripteur de taglib et qu'une classe **TagExtraInfo** est associée au tag, sa méthode **getVariableInfo()** devra obligatoirement retourner **null**.*

Les variables de script du tag sont donc déclarées dans le descripteur de taglib. La balise **variable** accepte les balises suivantes :

Nom	Description	Type
name-given	Le nom de la variable de script.	Un de name-given ou name-from-attribute est obligatoire.
name-from-attribute	Le nom de l'attribut du tag qui contiendra le nom de la variable de script. Dans ce cas, la valeur de cet attribut doit avoir une valeur statique (rtexprvalue=false).	Un de name-given ou name-from-attribute est obligatoire.
variable-class	Nom de la classe utilisé pour la déclaration de la variable.	optionnel

Nom	Description	Type
		(défaut: "java.lang.String")
declare	Indique si la variable doit être déclarée ou pas.	optionnel (défaut: true)
scope	Détermine la zone de portée de la variable. Valeurs possibles : NESTED , AT_BEGIN ou AT_END . (Voir Création de variable de script avec TagExtraInfo)	optionnel (défaut: NESTED)
description	Un texte de description de la variable.	optionnel

Ainsi, pour le tag suivant, qui se contente de placer un objet dans le scope "page" :

```

BeanTag
public class BeanTag extends TagSupport {
    public String name = null;
    public Object value= null;

    public void setName(String string) {
        name = string;
    }
    public void setValue(Object object) {
        value = object;
    }

    public int doStartTag() throws JspException {
        pageContext.setAttribute(name,value);
        return SKIP_BODY;
    }
}

```

Le mapping suivant permet de déclarer des variables de script dans la page JSP :

```

<tag>
  <name>bean</name>
  <tag-class>tutoriel.taglib_1_2.BeanTag</tag-class>
  <bodycontent>empty</bodycontent>
  <attribute>
    <name>name</name>
    <required>>true</required>
    <rtexprvalue>>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>value</name>
    <required>>true</required>
    <rtexprvalue>>true</rtexprvalue>
  </attribute>
  <variable>
    <name-from-attribute>name</name-from-attribute>
    <variable-class>java.lang.Object</variable-class>
    <declare>true</declare>
    <scope>AT_END</scope>
  </variable>
</tag>

```

Ainsi, le code suivant :

```

<taglib:bean name="date" value="<%new java.util.Date()%>" />
<taglib:bean name="text" value="Un simple message" />
<%=date + ": " + text%>

```

affichera le résultat suivant :

```
Sun Dec 26 20:18:32 CET 2004: Un simple message
```

2.2 - L'interface IterationTag

L'interface **IterationTag** étend l'interface **Tag**. Elle hérite donc de toutes ses propriétés et permet en plus d'effectuer des itérations sur le corps de la balise.

Elle définit pour cela une nouvelle méthode **doAfterBody()** qui est appelée après chaque évaluation du corps du tag.

Son code de retour détermine si le corps doit être réévalué ou pas, de la manière suivante :

- Si le retour vaut `IterationTag.EVAL_BODY_AGAIN`, le corps sera réévalué (et donc **doAfterBody()** sera **exécuté à nouveau**).
- Si le retour vaut `Tag.SKIP_BODY`, on passe à la fin du tag (méthode **doEndTag()**).

La classe `javax.servlet.jsp.tagext.TagSupport` des Taglibs 1.2 propose une implémentation par défaut de l'interface **IterationTag** (et donc également de **Tag**).

2.2.1 - Exemple d'itération

L'interface **IterationTag** a le même fonctionnement que l'interface **BodyTag** lorsque **doStartTag()** retourne **EVAL_BODY_INCLUDE**.

Les seules différences viennent du fait que la méthode **doAfterBody()** doit retourner **EVAL_BODY_AGAIN** afin de réévaluer le corps du tag.

Le code du tag **iterate** avec **BodyTag** avec les taglibs 1.1 (voir Itérer sur le corps du tag devient donc en utilisant l'interface **IterationTag** :

NewIterateTag

```
public class NewIterateTag extends TagSupport {

    private int count = 0;
    private int current;

    public void setCount(int i) {
        count = i;
    }

    public int doStartTag() throws JspException {
        current = 0;
        if (current < count)
            return Tag.EVAL_BODY_INCLUDE;
        return Tag.SKIP_BODY;
    }

    public int doAfterBody() throws JspException {
        current++;
        if (current < count)
            return IterationTag.EVAL_BODY_AGAIN;
        return Tag.SKIP_BODY;
    }
}
```

NewIterateTag

```

}
}

```

2.2.2 - Exemple d'itération avec des variables de script

En utilisant des variables de script avec un tag itératif, on peut manipuler des Collections ou des tableaux très simplement.

Nous allons écrire un tag pour parcourir toutes les propriétés du système (avec la méthode **System.getProperties()**) afin de les afficher à sa guise dans le corps du tag.

Par exemple :

SystemPropertiesTag

```

public class SystemPropertiesTag extends TagSupport {

    private Enumeration propertyName = null;

    public boolean doNext () {
        if ( propertyName.hasMoreElements() ) {
            String name = (String) propertyName.nextElement();
            String value = System.getProperty(name);
            pageContext.setAttribute("name", name);
            pageContext.setAttribute("value", value);
            return true;
        }
        return false;
    }

    public int doStartTag() throws JspException {

        // Iterateur sur les propriétés
        propertyName = System.getProperties().propertyNames();

        if (doNext())
            return EVAL_BODY_INCLUDE;
        return SKIP_BODY;
    }

    public int doAfterBody() throws JspException {
        if (doNext())
            return EVAL_BODY_AGAIN;
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
        pageContext.removeAttribute("name");
        pageContext.removeAttribute("value");
        return EVAL_PAGE;
    }
}

```

Nous utilisons une **Enumeration** pour parcourir la liste des noms des propriétés. Notre méthode **doNext()** nous permet de savoir si il y a encore des éléments à afficher.

Elle s'occupera également de stocker le couple **name/value** dans le scope **page** (nécessaire pour mettre à jour la variable de script).

Bien entendu, il faut déclarer les variables de script en étendant la classe **TagExtraInfo** (voir la section Création de variable de script) :

SystemPropertiesTEI

```
public class SystemPropertiesTEI extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData tagData) {
        return new VariableInfo[] {
            new VariableInfo("name", "java.lang.String", true, VariableInfo.NESTED),
            new VariableInfo("value", "java.lang.String", true, VariableInfo.NESTED)
        };
    }
}
```

Ainsi, le code suivant affichera toutes les propriétés du système dans un tableau HTML :

```
<table>
  <tr><td>Nom</td><td>Valeur</td></tr>
  <tag12:systemProperties>
    <tr><td><%=name%></td><td><%=value%></td></tr>
  </tag12:systemProperties>
</table>
```

2.3 - L'interface TryCatchfinally

L'API des taglibs propose une interface **TryCatchFinally** afin de gérer les exceptions générées par un Tag.

Si un tag implémente l'interface **TryCatchFinally**, il doit implémenter deux nouvelles méthodes :

- **doCatch()** qui correspondra au bloc catch.
- **doFinally()** qui correspondra au bloc finally.

Concrètement, cela signifie que si un tag implémente **TryCatchFinally**, les appels aux méthodes **doXXXX()** des interfaces **Tag**, **IterationTag** et **BodyTag** seront exécutés à l'intérieur d'un bloc try/catch de la forme suivante :

```
try {
    ...
    tag.doStartTag();
    ...
    tag.doInitBody();
    ...
    tag.doAfterBody();
    ...
    tag.doEndTag();
    ...
} catch (Throwable t) {
    tag.doCatch(t);
} finally {
    tag.doFinally();
}
```

3 - Les JSP Taglibs 2.0

3.1 - Le descripteur de Taglib 2.0

3.1.1 - Le Doctype

Les Taglibs 2.0 n'utilisent plus de fichier **DTD** (Document Type Definition) pour définir le descripteur de taglib.

Désormais le format des descripteurs de taglib est défini par un fichier **XSD** (XML Schema Description). Le 'Doctype' pour les Taglibs 2.0 devient donc :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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/web-jsptaglibrary_2_0.xsd"
        version="2.0">
    ...
</taglib>
```

3.1.2 - Description de la librairie

Nom	Description	Type
description (<i>info</i>)	Un texte de description de la librairie.	optionnel
display-name	Nom de la librairie qui représentera la taglib dans les outils de développement graphiques compatibles.	optionnel
icon	Chemin relatif d'une image GIF ou JPEG qui représentera la taglib dans les outils de développement graphiques compatibles (remplace small-icon et large-icon).	optionnel
tlib-version (<i>tlibversion</i>)	Le numéro de la version de la librairie de tag.	obligatoire Format : <code>[0-9]*{". "[0-9]}0..3</code>
short-name (<i>shortname</i>)	Définit un préfixe pour les tags de la librairie. Attention : Le préfixe utilisé sera toujours celui de la directive taglib de la page JSP. Celui défini ici permet aux éditeurs JSP d'insérer la directive taglib avec le préfixe indiqué ici.	obligatoire
uri (<i>uri</i>)	Définit une URI (Uniform Resource Identifier) qui identifie la taglibrairie. Il s'agit de l'URI à utiliser pour identifier la taglib dans le web.xml et les pages JSP.	optionnel
validator	Définit une classe qui valide la page JSP qui utilise cette librairie de tag. (Voir la section [JSP1.2] Le validateur de taglib)	optionnel
listener	Définit un listener qui sera instancié et qui surveillera les événements de l'application. (Voir la section [JSP1.2] Les listeners)	optionnel plusieurs listeners possible
tag	Description d'un tag de la librairie depuis une classe Java.	optionnel, une

Nom	Description	Type
	(Voir la section Déclaration de tag)	balise par classe.
tag-file	Description d'un tag de la librairie depuis un fichier *.tag. (Voir la section Déclaration de tag-file)	optionnel, une balise par fichier.
function	Description d'une fonction EL. (Voir la section Déclaration de fonction EL)	optionnel, une balise par fonction
taglib-extension	Permet aux outils de développement de définir des extensions pour la taglib.	optionnel

3.1.3 - Déclaration de tag

Nom	Description	Type
description	Un texte de description du tag.	optionnel
display-name	Nom du tag utilisé dans les outils de développement graphiques compatibles.	optionnel
icon	Chemin relative d'une image GIF ou JPEG qui représentera le tag dans les outils de développement graphiques compatibles (remplace small-icon et large-icon).	optionnel
name	Le nom du tag tel qu'il devra être utilisé dans les JSP.	obligatoire
tag-class	Le nom complet (package compris) de la classe Java qui représente ce tag (sous-classe de javax.servlet.jsp.tagext.Tag).	obligatoire
tei-class	Le nom complet (package compris) de la classe Java qui apporte des informations supplémentaires sur le tag (sous-classe de javax.servlet.jsp.tagext.TagExtraInfo).	optionnel
body-content	Indique le type de contenu du corps du tag : empty : Le tag n'accepte aucun corps (une exception sera lancée si il est utilisé avec un corps quelconque). JSP : Le contenu du tag est interprété comme du JSP. tagdependent : Le corps du tag ne sera pas interprété. Si il contient du code il sera affiché comme du simple texte. scriptless : Le corps du tag ne peut alors contenir que du texte, des expressions EL et d'autres tags JSP, mais aucun scripts Java (<% ... %>).	obligatoire
variable	Déclaration des variables de script utilisées par le tag. (Voir [JSP1.2] Déclaration des variables de script)	optionnel Une balise par variable
attribute	Définition des différents attributs du tag. (Voir Définir un attribut du tag)	optionnel une balise par attribut.
dynamic-attributes	Autorise l'utilisation d'attributs dynamiques, la classe Java	optionnel

Nom	Description	Type
	représentant le tag doit alors implémenter l'interface DynamicAttributes . (Voir L'interface DynamicAttributes)	(défaut : false)
exemple	Description d'un exemple d'utilisation du tag.	optionnel
tag-extension	Permet aux outils de développement de définir des extensions pour le tag.	optionnel

3.1.4 - Déclaration de tag-file

La nouvelle balise **tag-file** permet de déclarer un tag à partir d'un fichier *.tag (Voir la section Les fichiers *.tag).

Nom	Description	Type
description	Un texte de description du tag.	optionnel
display-name	Nom du tag utilisé dans les outils de développement graphiques compatibles.	optionnel
icon	Chemin relatif d'une image GIF ou JPEG qui représentera le tag dans les outils de développement graphiques compatibles (remplace small-icon et large-icon).	optionnel
name	Le nom du tag tel qu'il devra être utilisé dans les JSP.	obligatoire
path	Le chemin d'accès relatif vers le fichier *.tag. (Voir Les fichiers *.tag)	obligatoire
exemple	Description d'un exemple d'utilisation du tag.	optionnel
tag-extension	Permet aux outils de développement de définir des extensions pour le tag.	optionnel

3.1.5 - Déclaration de fonction EL

Nom	Description	Type
description	Un texte de description du de la fonction.	optionnel
display-name	Nom de la fonction utilisé dans les outils de développement graphiques compatibles.	optionnel
icon	Chemin relatif d'une image GIF ou JPEG qui représentera la fonction dans les outils de développement graphiques compatibles (remplace small-icon et large-icon).	optionnel
name	Le nom du fonction tel qu'il devra être utilisé dans les JSP.	obligatoire
function-class	Le nom de la classe Java qui implémente le code de la fonction.	obligatoire
function-signature	La signature exacte de la fonction, selon la spécification de Java. <u>Exemple</u> : java.lang.String functionName (java.lang.String, int)	obligatoire
exemple	Description d'un exemple d'utilisation de la fonction.	optionnel
function-extension	Permet aux outils de développement de définir des extensions pour la fonction.	optionnel

3.1.6 - Définir un attribut du tag

La balise **attribute** permet de définir les attributs du tag JSP. Seul les attributs décrits dans le TLD seront acceptés. L'utilisation d'attribut non indiqué dans le TLD provoquera une exception à la compilation.

La balise **attribute** du descripteur de taglibs 1.2 accepte les balises suivante (les noms entre parenthèses correspondent aux noms dans le TLD de la version 1.1, si il existe) :

Nom	Description	Type
description	Un texte de description de l'attribut.	optionnel
name	Le nom de l'attribut tel qu'il devra être utilisé dans les JSP.	obligatoire
required	Indique si l'attribut est requis (valeur possible : true/false ou yes/no).	optionnel (défaut : false)
rtexprvalue	Indique si l'attribut peut être le résultat d'une scriptlet ou d'une Expression Language (EL) (valeur possible : true/false ou yes/no).	optionnel (défaut : false)
type	Indique le type Java de l'attribut (si rtexprvalue==true).	optionnel (défaut : "java.lang.String")
fragment	Indique si l'attribut est un fragment. (Voir Utilisation de JspFragment	optionnel (défaut : false)

Si un attribut **required** est absent, une exception sera lancée.

L'utilisation de **rtexprvalue** permet d'utiliser le résultat d'une expression de scriptlet comme valeur d'attribut (par exemple <%=monObjet%>).

3.2 - L'interface SimpleTag

L'interface **SimpleTag** permet une implémentation différente de tag JSP. Contrairement à **IterationTag** et **BodyTag**, elle n'hérite pas de l'interface **Tag** car son fonctionnement est totalement différent.

Afin de conserver une cohérence dans l'organisation des interfaces, les interfaces **Tag** et **SimpleTag** héritent de l'interface **JspTag**. Cette interface est juste un interface 'marqueur' pour pouvoir distinguer les différents type de tag JSP, c'est à dire qu'elle ne comporte aucune méthode ...

L'interface **SimpleTag** définit les méthodes suivantes :

- **setParent(JspTag)** et **getParent()** qui permettent de définir et d'accéder au tag parent.
- **setJspContext(JspContext)** qui remplace le **PageContext** de l'interface **Tag**.
- **setJspBody(JspFragment)** qui définit le corps du tag.
- **doTag()** qui est l'unique méthode de traitement du tag.

Lors de l'exécution d'un tag implémentant **SimpleTag**, on obtient les appels suivant :

- Les méthodes **setParent(JspTag)** et **setJspContext(JspContext)** sont renseignées, ainsi que d'éventuels attributs présents dans le tag.
- Si le tag possède un corps non vide, **setJspBody(JspFragment)** est renseignée.
- La méthode **doTag()** est appelée.

*Contrairement à l'interface **Tag**, les objets de type **SimpleTag** ne sont pas mis en cache. A chaque exécution*

de la page un nouvel objet est crée ...

La méthode **doTag()** traite alors son corps via l'objet **JspFragment** renseignée par le serveur d'application. Cet objet représente le code du corps du tag et peut être évalué autant de fois que nécessaire grâce à la méthode **invoke(Writer)** qui écrit le résultat dans le **Writer** spécifié.

invoke(null) écrira le résultat de l'évaluation du corps du tag directement dans la page JSP ...

L'interface **SimpleTag** permet une gestion plus simple des tags JSP. En effet, au lieu de gérer plusieurs méthodes pour chaque étape du tag (**doStartTag()**, **doInitBody()**, **doAfterBody()**, **doEndTag()**), l'unique méthode **doTag()** permet autant de possibilité en utilisant un **JspFragment** représentant le corps du tag.

*Pour une raison de compatibilité avec les anciennes versions des taglibs, les méthodes **getParent()** et **setParent()** de l'interface **Tag** et ses sous interfaces utilisent des objets de type **Tag**. Afin de pouvoir gérer la collaboration avec des tags de type **SimpleTag**, la classe **TagAdapter** est utilisée par le serveur d'application. Elle permet d'accéder à l'instance de **SimpleTag** via la méthode **getAdaptee()** ...*

La classe **javax.servlet.jsp.tagext.SimpleTagSupport** propose une implémentation par défaut de l'interface **SimpleTag**.

3.2.1 - Itérer sur le corps du tag

L'itération sur le contenu du tag devient une simple boucle. L'utilisation du corps comme un objet permet d'éviter la multiplication de variables d'instances utilisées dans les différentes méthodes :

SimpleIterateTag

```
public class SimpleIterateTag extends SimpleTagSupport {
    private int count = 0;
    public void setCount (int value) {
        this.count = value;
    }
    public void doTag() throws JspException, IOException {
        for (int i=0; i<count; i++)
            getJspBody().invoke(null);
    }
}
```

Et la déclaration dans le TLD reste identique :

```
<tag>
  <name>simpleIterate</name>
  <tag-class>com.developpez.adiguba.tutorial.taglibs_20.SimpleIterateTag</tag-class>
  <body-content>scriptless</body-content>
  <attribute>
    <name>count</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

Les **SimpleTag** n'accepte pas de contenu de type JSP.

En fait, les objets **JspFragment** ne gèrent pas les scriptlets (<% ... %>), il faut donc utiliser à la place la valeur **scriptless** en remplacement...

3.3 - L'interface DynamicAttributes

L'interface **DynamicAttributes** vient combler un des principaux défauts des précédentes versions des taglibs : la gestion d'attributs dynamiques.

En effet, dans la section 1.2.2 (Gestion des attributs du tag), on a vu que tous les attributs devaient obligatoirement posséder un méthode mutateur et être déclarer dans le fichier TLD.

L'interface **DynamicAttributes** résout ce problème avec la méthode suivante :

DynamicAttributes

```
public void setDynamicAttribute(String uri, String localName, Object value);
```

Pour pouvoir utiliser les attributs dynamiques, il faut donc que le tag implémente l'interface **DynamicAttributes** en plus d'une des interfaces **JspTag**. Lors de l'initialisation du tag, si un attribut n'est pas déclaré dans le TLD, aucune exception n'est lancée et la méthode **setDynamicAttribute()** est appelée.

Bien entendu, les attributs dynamiques peuvent être utilisés conjointement avec des attributs classiques ...

*En plus d'implémenter l'interface **DynamicAttributes**, il faut utiliser `<dynamic-attributes>true</dynamic-attributes>` dans la déclaration du tag dans le TLD.*

3.3.1 - Exemple d'attributs dynamiques

L'utilisation d'attributs dynamiques est particulièrement utilisée si l'on désire remplacer des balises HTML par des tags JSP afin d'y ajouter un traitement coté serveur.

Par exemple, si l'on désire associer des champs INPUT avec des beans afin qu'ils soient automatiquement renseignés, par exemple afin que le code suivant :

```
<tag20:input type="text" name="nomDuBean" />
```

Donne le résultat suivant :

```
<input type="text" name="nomDuBean" value=valeurDuBean" />
```

Le code du tag en lui même n'est pas compliqué, mais le problème vient du fait que dans les spécifications HTML, la balise INPUT accepte 12 attributs (accept, align, alt, checked, disabled, maxlength, name, readonly, size, src, type et value).

Pour pouvoir utiliser ces attributs dans notre tag, il faudrait les définir dans le TLD et ajouter leurs variables et mutateurs dans le source Java du tag.

De même, la balise INPUT accepte également 16 attributs d'événement (tabindex, accesskey, onfocus, onblur, onselect, onchange, onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown et onkeyup).

Au final, cela reviendrait à gérer une trentaine de variables alors que seulement quelques unes sont utiles ...

En implémentant l'interface **DynamicAttributes**, on peut simplement réécrire directement les attributs inutiles :

```

InputTag
public class InputTag extends SimpleTagSupport implements DynamicAttributes {

    private Map attributes = new HashMap();
    private String type = null;
    private String name = null;
    private String value = null;

    public void setType(String string) {
        type = string;
    }
    public void setName(String string) {
        name = string;
    }
    public void setValue(String string) {
        value = string;
    }

    public void setDynamicAttribute(String uri, String localName, Object value)
        throws JspException {
        // On place chaque attribut/valeur dans la Map attributes
        attributes.put (localName, value);
    }

    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();

        out.print("<input type='" + type + "' name='" + name + "' ");
        if (value==null) {
            // On recherche un attribut avec le même nom dans les != scopes
            Object o = getJspContext().findAttribute(name);
            value = o==null ? "" : o.toString();
        }
        out.print("value='" + value + "' ");

        Iterator iterator = attributes.entrySet().iterator();
        while ( iterator.hasNext() ) {
            Map.Entry entry = (Map.Entry) iterator.next();
            out.print ( entry.getKey() + "=" + entry.getValue() + " ");
        }
        out.print(">");
    }
}

```

Explications :

Le tag gère trois attributs :

- **type** qui permet de s'assurer que le type du champs INPUT est bien présent.
- **name** qui correspond à la fois au nom du champs INPUT et du bean coté serveur.
- **value** qui permet d'associer une autre valeur que celle du bean (si **value** est absent, on utilise le bean).

Tous les autres attributs qui seront passés au tag seront envoyés à la méthode **setDynamicAttribute()** qui se contentera de les placer dans une **Map**.

Les attributs **type** et **name** sont obligatoire. Dans la méthode **doTag()**, on utilise **getJspContext().findAttribute()** pour rechercher le bean si l'attribut **value** est absent, puis on affiche tous les attributs de la **Map** remplie par les appels successif à **setDynamicAttribute()**.

Ainsi, avec le mapping suivant :

```

<tag>
  <name>input</name>
  <tag-class>com.developpez.adiguba.tutorial.taglibs_20.InputTag</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>name</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>type</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>value</name>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <dynamic-attributes>true</dynamic-attributes>
</tag>

```

Le code suivant :

```

<jsp:useBean id="input3" scope="page" class="java.util.Date"/>
    Input1 : <tag20:input type="text" name="input1" class="green" readonly="true"
value="statique"/><br/>
    Input2 : <tag20:input type="text" name="input2" class="red"
onfocus="methodeJavascript();" /><br/>
    Input3 : <tag20:input type="text" name="input3" class="blue" /><br/>

```

permet de conserver les autres attributs dans le code HTML généré :

```

    Input1 : <input type='text' name='input1' value='statique' readonly='true' class='green'
/><br/>
    Input2 : <input type='text' name='input2' value='' class='red'
onfocus='methodeJavascript();' /><br/>
    Input3 : <input type='text' name='input3' value='Thu Dec 30 16:17:38 CET 2004' class='blue'
/><br/>

```

*L'interface **DynamicAttributes** peut également être appliquée à des classes qui implémentent l'interface **Tag** ou ses interfaces filles ...*

3.4 - Les fichiers *.tag

Les Tags permettent de faire des traitements complexes. Toutefois, il n'est pas pratique de générer du code HTML à l'intérieur d'une classe (utilisation fastidieuse de **out.println()** ...).

A l'instar des fichiers *.jsp pour les Servlets, les fichiers *.tag permettent de créer des tags simplement.

Un fichier *.tag est compilé en une classe qui implémente **SimpleTag**, et permet donc la réalisation de tag de la même manière qu'une JSP ...

Pour cela, la directive **<%@page%>** est remplacé par la directive **<%@tag%>** qui accepte principalement les attributs suivants :

Nom	Description	Type
description (<i>info</i>)	Un texte de description de la librairie.	optionnel
display-name	Nom de la librairie qui représentera la taglib dans les outils de	optionnel

Nom	Description	Type
	développement graphiques compatibles.	
body-content	Indique le type de contenu du corps du tag : empty : Le tag n'accepte aucun corps (une exception sera lancée si il est utilisé avec un corps quelconque). tagdependent : Le corps du tag ne sera pas interprété. Si il contient du code il sera affiché comme du simple texte. scriptless : Le corps du tag ne peut alors contenir que du texte, des expressions EL et d'autres tags JSP, mais aucun scripts Java (<% ... %>). Attention, les fichiers *.tag n'acceptent pas de body-content JSP ...	optionnel (défaut : scriptless)
dynamic-attributes	Permet l'utilisation d'attribut dynamique. dynamic-attributes doit contenir le nom d'un bean de type Map qui sera créé et stocké dans le scope "page". Il contiendra tous les attributs qui n'ont pas été spécifiquement déclarés ...	optionnel
exemple	Description d'un exemple d'utilisation du tag.	optionnel
import	Importation d'une classe Java (identique à <%@ page import="" %>).	optionnel

Les fichiers *.tag doivent être placés soit dans le répertoire **/META-INF/tags/** (ou un sous-répertoire) de l'archive *.jar de la taglib.

Ils peuvent également être placés dans le répertoire **/WEB-INF/tags/** (ou un sous-répertoire) de l'application Web.

Si la librairie de tag doit être distribuée, il est préférable de placer les fichiers *.tag dans le jar afin de faciliter la maintenance.

Le répertoire **/WEB-INF/tags/** pourrait être utilisé pour des tags spécifique à une application...

*Lorsque les fichiers *.tag font partie de l'application web, on peut se passer de descripteur de fichier en utilisant l'attribut **tagdir** de la directive <%@taglib%> avec le chemin relatif du répertoire contenant les fichiers *.tag. Le nom du tag sera alors celui du fichier *.tag (sans l'extension).*

*Par exemple, si le répertoire **/WEB-INF/tags/mestags/** contient les fichiers **monTag.tag** et **unAutreTag.tag**, il suffit d'utiliser le code suivant dans n'importe quel JSP pour pouvoir utiliser ces tags :*

```
<%@ taglib tagdir="/WEB-INF/tags/mestags/" prefix="tag-prefix" %>
<tag-prefix:monTag/>
<tag-prefix:unAutreTag param="value">
    Corps
</tag-prefix:unautreTag>
```

3.4.1 - Déclaration des attributs

La déclaration des attributs d'un tag d'un fichier *.tag est décrite directement dans ce dernier avec la directive `<%@ attribute %>`. Elle accepte les mêmes valeurs que dans le descripteur de taglib (Voir Définir un attribut du tag).

Cette directive à la forme suivante :

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

Le tag comportera alors une variable de script locale du même nom que l'attribut et qui contiendra la valeur de celui-ci.

3.4.2 - Déclaration de variables

De même, il est possible de déclarer les variables de scripts créées par le tag directement dans le fichier *.tag avec la directive suivantes :

```
<%@ variable
{
    name-given="scripting variable"
    | (name-from-attribute="scripting variable" alias="locally-scoped attribute") }
[ variable-class="java.lang.String | name of the variable class" ]
[ declare="true | false" ]
[ scope="AT_BEGIN | AT_END | NESTED" ]
[ description="text" ]
%>
```

Le fonctionnement est le même que la balise **variable** du descripteur de taglib (Voir Déclaration de variable de script).

La seule différence intervient si l'on utilise **name-from-attribute**, on est alors obligé d'utiliser un alias pour l'attribut du même nom afin d'éviter un conflit de nom de variable, étant donné qu'une variable locale est créée pour chaque attribut du tag.

3.4.3 - Affichage du corps

Enfin, on a à notre disposition le tag `<jsp:doBody/>` qui permet d'évaluer le corps du tag. Ce tag peut avoir trois comportements :

- `<jsp:doBody/>` affichera l'évaluation du corps du tag directement dans la page.
- `<jsp:doBody var="nom" scope="page"/>` créera une **String** "nom" dans le scope "page" qui contient l'évaluation du corps.
- `<jsp:doBody varReader="nom" scope="page"/>` créera un **StringReader** "nom" dans le scope "page" qui contient l'évaluation du corps.

3.5 - Les fonctions EL

Les fonctions EL permettent de décrire des fonctions qui pourront être utilisés dans des EL (Expressions Languages).

Elle sont de la forme suivante :

```
<div>
    ${prefix:fonction(param1,param2)}
</div>
```

Cette fonction doit obligatoirement être statique et accessible par le JSP (donc publique dans la plupart des cas).

Il suffit donc de faire un mapping entre un nom et une méthode statique existante.

Par exemple, pour pouvoir utiliser la méthode **System.getProperty(String)**, on peut utiliser la déclaration suivante dans le descripteur de taglib :

```
<function>
    <name>prop</name>
    <function-class>java.lang.System</function-class>
    <function-signature>java.lang.String
    getProperty(java.lang.String)</function-signature>
</function>
```

Ainsi, désormais les deux lignes suivantes sont équivalentes :

```
${tag20:prop("os.name")}
<!-- est équivalent à : -->
<%= System.getProperty("os.name") %>
```

Cela permet une simplification du code des JSP, en particulier lorsqu'on l'utilise avec des tags JSP ...

3.6 - Utilisation de JspFragment

Enfin, les **JspFragment** permettent d'utiliser des attributs qui ne sont rien d'autres que des fragments de code JSP (scriptless). L'utilisation de **JspFragment** permet de remplacer l'utilisation de certains tags collaboratifs.

Lorsque un attribut est un **fragment** (la valeur **fragment** vaut **true** dans le TLD ou dans la directive **attributes** d'un fichier *.tag), ce dernier comportera du code complexe qui pourra être évalué plusieurs fois, à l'instar du corps des tags implémentant **SimpleTag**.

*Si un attribut d'un tag est un fragment, les valeurs de **rtextvalue** et **type** ne doivent pas être utilisées. Les valeurs **true** et **javax.servlet.jsp.tagext.JspFragment** seront utilisées...*

Les attributs de type **fragment** doivent en effet être utilisés de la manière suivante (même si le tag n'accepte pas de corps) :

```
<prefix:nomDuTag>
    <jsp:attribute name="nomAttribut">
        code JSP ici
    </jsp:attribute>
</prefix:nomDuTag>
```

Ensuite, on peut disposer à sa guise du corps de la balise **jsp:attribute** représenté par l'attribut **nomAttribut**.

Il suffit ensuite d'utiliser la méthode **invoke()** de **JspFragment** pour évaluer le corps du tag autant de fois que nécessaire ...

Dans un fichier *.tag, voir dans le contenu même du corps du tag, on peut utiliser la balise standard **<jsp:invoke/>** qui peut être utilisé de trois manières :

- **<jsp:invoke fragment="nomFragment"/>** affichera l'évaluation du corps du fragment "nomFragment" directement dans la page.
- **<jsp:invoke fragment="nomFragment" var="nom" scope="page"/>** créera une **String** "nom" dans le scope "page" qui contient l'évaluation du corps du fragment "nomFragment".
- **<jsp:invoke fragment="nomFragment" varReader="nom" scope="page"/>** créera un **StringReader** "nom" dans le scope "page" qui contient l'évaluation du corps du fragment "nomFragment".

4 - Déploiement de taglib

Cette section regroupe quelques informations utiles pour le déploiement de librairie de tags...

4.1 - Spécification des taglibs archivées

Afin de simplifier le déploiement des taglibs, il est possible de les distribuer sous forme d'archive *.jar, comme pour n'importe quelle librairie. Afin de les utiliser dans une application web, il suffit alors de placer l'archive dans le répertoire **/WEB-INF/lib/**.

L'archive peut alors comporter un fichier **/META-INF/taglib.tld** qui correspond au descripteur de taglib utilisé lorsque la taglib est référencé dans le fichier **web.xml** d'après son fichier *.jar (Voir Comment utiliser une librairie de tag ?) :

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

*Au démarrage de votre application web J2EE 1.3 (ou supérieur), tous les descripteurs de librairies de tag distribuées sous forme de Jar dans le répertoire **/WEB-INF/lib/** sont chargées par le serveur d'application (notamment afin de démarrer les listeners). Toutes ces librairies sont alors automatiquement référencées en utilisant l'**uri** indiquée dans leurs descripteurs respectifs.*

*Les taglibs sont donc directement accessibles via leurs **uris** sans avoir à toutes les déclarer dans le **web.xml**.*

De plus, les spécifications J2EE 1.3 autorisent l'utilisation de plusieurs descripteurs de taglibs dans une même archive Jar. Ainsi, au démarrage de l'application, tous les fichiers *.tld du répertoire **/META-INF/** des archives du répertoire **/WEB-INF/lib/** sont automatiquement référencés grâce à leurs **uris**...

Il est ainsi possible de déployer plusieurs taglibs avec un simple fichier Jar, la directive **<%@ taglib %>** des pages JSP utilise alors l'**uri** de la taglib (d'où l'importance de l'unicité de cette **uri**).

*Il est toutefois préférable d'également distribuer les fichiers *.tld séparément de l'archive. Notamment afin de conserver une compatibilité avec les anciens serveurs d'applications, ou de permettre au développeur de gérer lui-même les différentes taglibs de son application...*

4.2 - Et la documentation

Le site **java.net** propose un outil afin de générer automatiquement la documentation des librairies de tags. Ainsi, en renseignant correctement les balises d'informations des descripteurs de taglibs (**<description>**, **<exemple>**, etc...), on obtient une documentation dans le style de l'API Java.

En effet, cet outil, nommé **Tag Library Documentation Generator (tlddoc)** est une application Java qui analyse le contenu des fichiers *.tld (et les directives des fichiers *.tag) afin de générer une documentation HTML similaire à celles obtenues avec l'outil **javadoc** pour les classes Java.

*Le programme **tlddoc** (license BSD) peut être téléchargé sur le site de **java.net** à l'adresse suivante :*

<https://taglibrarydoc.dev.java.net/>

Le programme **tlddoc** est distribué sous forme de Jar et s'utilise de la manière suivante :

Utilisation de tlddoc.jar

```
java -jar tlddoc.jar [options] taglib1 taglib2...
```

Les principales options du programme sont :

- **-d <directory>** : Spécifie le répertoire de destination de la documentation (par défaut, un répertoire **out** est créé).
- **-doctitle <html>** : Spécifie le titre qui sera affiché sur la page d'index de la documentation.
- **-windowtitle <text>** : Spécifie le titre qui sera affiché dans la barre de titre du navigateur.

Puis il accepte une ou plusieurs taglibs qui peuvent correspondre aux éléments suivants :

- Un fichier ***.tld** : le fichier sera alors analysé.
- Un fichier ***.jar** : tous les fichiers *.tld de l'archive (dans **/META-INF/**) seront analysés.
- Un fichier ***.war** : toutes les librairies de l'application web seront analysées.
- Un répertoire comportant **/WEB-INF/tags/** : le répertoire sera analysé comme un descripteur implicite pour les fichiers *.tag.
- Un répertoire comportant **/WEB-INF/** : toutes les taglibs du répertoire seront analysées.

Exemple d'utilisation

```
java -jar tlddoc.jar -d docs -doctitle "My Taglib" -windowtitle "My Taglib" mytaglib.jar
```

*Vous pouvez consulter la documentation de la **JSTL** afin d'avoir un aperçu du résultat :*

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

Conclusion

L'utilisation conjointe de tag JSP et des EL (Expression Language) des JSP 2.0 permet une conception plus aisée des pages JSP. Elle peut permettre à des personnes ne développant pas en Java de concevoir des pages JSP complexes, en particulier s'ils ont des connaissances d'HTML et surtout d'XML...

Enfin, n'oubliez pas que Sun a décrit la spécification d'une **JSTL** (JSP Standard Tag Library) qui propose un ensemble de taglib de base pour des actions courantes dans un projet J2EE...

Je vous invite donc à lire mon tutoriel sur la **JSTL** :

<http://adiguba.developpez.com/tutoriels/j2ee/jsp/jstl/>