Advanced Control Flow (2.1 legacy document)

Table of contents

Table of contents

Warning:
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

1. JXTemplate Generator

The JXTemplate Generator is a page template processor that allows you to inject data from Java and JavaScript objects passed by a Cocoon Flowscript into a Cocoon pipeline. It provides a set of tags (similar to the JSTL core tags) that allow you to iterate over Java collections (and Java or JavaScript arrays) and to test for the presence of optional or alternate bean properties, as well as embedded expressions to specify conditions and to access the properties of objects. The *JX*Template Generator gets its name from the embedded expression languages it supports, namely Apache JXPath and Apache JeXl.

To use the JXTemplate Generator, add a generator entry to your sitemap with the src attribute set to org.apache.cocoon.generation.JXTemplateGenerator, for example like this:

```
<map:generators> <map:generator label="content,data" logger="sitemap.generator.jx"
name="jx" src="org.apache.cocoon.generation.JXTemplateGenerator"/> </map:generators>
```

2. Expression Languages

The JXTemplate Generator supports two embedded expression languages: Jexl and JXPath. Apache Jexl provides an extended version of the expression language of the JSTL. Apache JXPath provides an interpreter of the XPath expression language that can apply XPath expressions to graphs of Java objects of all kinds: JavaBeans, Maps, Servlet contexts, DOM etc, including mixtures thereof.

Having an embedded expression language allows a page author to access an object using a simple syntax such as

```
<site signOn="${accountForm.signOn}">
```

Embedded Jexl expressions are contained in ${}.

Embedded JXPath expressions are contained in #{}.

The referenced objects may be Java Beans, DOM, or JavaScript objects from a Flowscript. In addition, a special cocoon object providing access to the Cocoon FOM is available as both a JXPath and Jexl variable in a template.

The cocoon object contains the following properties:

request
The current Cocoon request
session
The user session associated with the current request
context
The Cocoon context associated with the current request
parameters
A map containing the parameters passed to the generator in the pipeline
continuation
The current Web Continuation from your Flowscript

Jexl Example:

```
The content type of the current request is ${cocoon.request.contentType}
```

JXPath Example:

The content type of the current request is #{$cocoon/request/contentType}

You would typically access the id of the Web Continuation:

<form action="${cocoon.continuation.id}">

You can also reach previous continuations via its parent property:

<form action="${cocoon.continuation.parent.id}" >

or using an XPath expression:

<form action="#{$cocoon/continuation/parent/id}" >

Deprecated Variables:

The following variables are deprecated but still supported:

org.apache.cocoon.environment.Request request
The current Cocoon request (deprecated: use cocoon.request instead)
org.apache.cocoon.environment.Session session
The current user session (deprecated: use cocoon.session instead)
org.apache.cocoon.environment.Context context
The current context (deprecated: use cocoon.context instead)
org.apache.cocoon.components.flow.WebContinuation continuation
The current Web Continuation (deprecated: use cocoon.continuation instead)

3. Parameters

3.1. lenient-xpath

By default XPath evaluation throws an exception if the supplied XPath does not map to an existing property. This constraint can be relaxed by setting the parameter lenient-xpath to true. In the lenient mode evaluation simply returns null if the path maps to nothing.

Example:

<map:match pattern="*.jx"> <map:generate type="jx" src="documents/{1}.jx"> <map:parameter name="lenient-xpath" value="true"/> </map:generate> <map:serialize type="xhtml"/> </map:match>

4. Tags

The JXTemplate Generator tags are defined in the namespace

http://apache.org/cocoon/templates/jx/1.0

4.1. template

The template tag defines a new template:

<jx:template xmlns:jx="http://apache.org/cocoon/templates/jx/1.0"> body </jx:template>

4.2. import

The import tag allows you to include another template within the current template. The content of the imported template is compiled and will be executed in place of the import tag:

<jx:import uri="URI" [context="Expression"]/>

The Cocoon source resolver is used to resolve uri. If context is present, then its value is used as the context for evaluating the imported template, otherwise the current context is used.

4.3. set

The set tag creates a local alias of an object. The var attribute specifies the name of a variable to assign the object to. The value attribute specifies the object (defaults to body if not present):

<jx:set var="Name" [value="Value"]> [body] </jx:set>

If used within a macro definition (see below) variables created by set are only visible within the body of the macro.

Jexl Example:

<jx:set var="greeting" value="Hello ${user}"/> The value of greeting is ${greeting}

JXPath Example:

<jx:set var="greeting" value="Hello #{user}"/> The value of greeting is #{$greeting}

4.4. if

The if tag allows the conditional execution of its body according to value of its test attribute:

<jx:if test="Expression"> body </jx:if>

Jexl Example:

<jx:if test="${cart.numberOfItems == 0}"> Your cart is empty </jx:if>

JXPath Example:

<jx:if test="#{cart/numberOfItems = 0}"> Your cart is empty </jx:if>

4.5. choose

The choose tag performs conditional block execution by its embedded when sub tags. It renders the body of the first when tag whose test condition evaluates to true. If none of the test conditions of its nested when tags evaluate to true, then the body of its otherwise tag is evaluated, if present:

<jx:choose> <jx:when test="Expression"> body </jx:when>+ <jx:otherwise> body </jx:otherwise>? </jx:choose>

Jexl Example:

<jx:choose> <jx:when test="${!user.loggedIn}"> <jx:set var="label" value="Log in"> </jx:when> <jx:otherwise> <jx:set var="label" value="Log out"> </jx:otherwise> </jx:choose>

JXPath Example:

<jx:choose> <jx:when test="#{not(user/loggedIn)}"> <jx:set var="label" value="Log in"> </jx:when> <jx:otherwise> <jx:set var="label" value="Log out"> </jx:otherwise> </jx:choose>

4.6. out

The out tag evaluates an expression and outputs the result of the evaluation:

<jx:out value="Expression"/>

Jexl Example:

<jx:out value="${cart.numberOfItems}">

JXPath Example:

<jx:out value="#{cart/numberOfItems}">

4.7. forEach

The forEach tag allows you to iterate over a collection of objects:

<jx:forEach [var="Name"] [varStatus="Name"] [items="Expression"] [begin="NumExpr"] [end="NumExpr"] [step="NumExpr"]> body </jx:forEach>

The items attribute specifies the list of items to iterate over. The var attribute specifies the name of a variable to hold the current item. The begin attribute specifies the element to start with (0 = first item, 1 = second item, ...). If unspecified it defaults to 0. The end attribute specifies the item to end with (0 = first item, 1 = second item, ...). If unspecified it defaults to the last item in the list. Every step items are processed (defaults to 1 if step is absent). Either items or both begin and end must be present.

An alternate form of forEach is supported for convenience when using XPath (since you can specify the selection criteria for the collection using XPath itself):

<jx:forEach select="XPathExpression"> body </jx:forEach>

When using XPath expressions within forEach the current element is the context node and can be referenced with: #{.}

Jexl Example:

<jx:forEach var="item" items="${cart.cartItems}" begin="${start}" end="${count-start}" step="1"> <td>${item.productId}</td> </jx:forEach>

JXPath Example:

<jx:forEach select="#{cart/cartItems[position() <= $count]}}> <td>#{./productId}</td> </jx:forEach>

If the varStatus attribute is present a variable will be created to hold information about the current loop status. The variable named by the varStatus attribute will hold a reference to an object with the following properties:

Property:	Description:
current	The item from the collection for the current round of iteration
index	The zero-based index for the current round of iteration
count	The one-based count for the current round of iteration
first	True if this is the first round of iteration
last	True if this is the last round of iteration
begin	The value of the begin attribute
end	The value of the end attribute
step	The value of the step attribute

Jexl Example:

<jx:forEach items="${items}" varStatus="status"> index=${status.index}, count=${status.count}, current=${status.current}, first=${status.first}, last=${status.last}, begin=${status.begin}, end=${status.end} </jx:forEach>

JXPath Example:

<jx:forEach select="#{items}" varStatus="status"> index=#{$status/index}, count=#{$status/count}, current=#{$status/current}, first=#{$status/first}, last=#{$status/last}, begin=#{$status/begin}, end=#{$status/end} </jx:forEach>

4.8. formatNumber

The formatNumber tag is used to display numeric data, including currencies and percentages, in a locale-specific manner. It determines from the locale, for example, whether to use a period or a comma for delimiting the integer and decimal portions of a number. Here is its syntax:

<jx:formatNumber value="Expression" [type="Type"] [pattern="Expression"] [currencyCode="Expression"] [currencySymbol="Expression"] [maxIntegerDigits="Expression"] [minIntegerDigits="Expression"] [maxFractionDigits="Expression"] [minFractionDigits="Expression"] [groupingUsed="Expression"] [var="Name"] [locale="Expression"]>

Only the value attribute is required. It is used to specify the numeric value that is to be formatted.

The value of the type attribute should be either "number", "currency", or "percent", and indicates what type of numeric value is being formatted. The default value for this attribute is "number". The pattern attribute takes precedence over the type attribute and allows more precise formatting of numeric values following the pattern conventions of the java.text.DecimalFormat class.

When the type attribute has a value of "currency", the currencyCode attribute can be used to explicitly specify the currency for the numerical value being displayed. As with language and country codes, currency codes are governed by an ISO standard. This code is used to determine the currency symbol to display as part of the formatted value.

Alternatively, you can use the currencySymbol attribute to explicitly specify the currency symbol. Note that as of JDK 1.4 and the associated introduction of the java.util.Currency class, the currencyCode attribute of formatNumber takes precedence over the currencySymbol attribute. For earlier versions of the JDK, however, the currencySymbol attribute takes precedence.

The maxIntegerDigits, minIntegerDigits, maxFractionDigits, and minFractionDigits attributes are used to control the number of significant digits displayed before and after the decimal point. These attributes require integer values.

The groupingUsed attribute takes a Boolean value and controls whether digits before the decimal point are grouped. For example, in English-language locales, large numbers have their digits grouped by threes, with each set of three delimited by a comma. Other locales delimit such groupings with a period or a space. The default value for this attribute is true.

4.9. formatDate

The formatDate tag provides facilities to format Date values:

<jx:formatDate value="Expression" [dateStyle="Style"] [timeStyle="Style"] [pattern="Expression"] [type="Type"] [var="Name"] [locale="Expression"]>

Only the value attribute is required. Its value should be an instance of the java.util.Date class, specifying the date and/or time data to be formatted and displayed.

The optional timeZone attribute indicates the time zone in which the date and/or time are to be displayed. If not present, then the JVM's default time zone is used (that is, the time zone setting specified for the local operating system).

The type attribute indicates which fields of the specified Date instance are to be displayed, and should be either "time", "date", or "both". The default value for this attribute is "date", so if no type attribute is present, the formatDate tag -- true to its name -- will only display the date information associated with the Date instance, specified using the tag's value attribute.

The dateStyle and timeStyle attributes indicate how the date and time information should be formatted, respectively. Valid styles are "default", "short", "medium", "long", and "full". The default value is, naturally, "default", indicating that a locale-specific style should be used. The semantics for the other four style values are as defined by the java.text.DateFormat class.

Rather than relying on the built-in styles, you can use the pattern attribute to specify a custom style. When present, the value of the pattern attribute should be a pattern string following the conventions of the java.text.SimpleDateFormat class. These patterns are based on replacing designated characters within the pattern with corresponding date and time fields. For example, the pattern MM/dd/yyyy indicates that two-digit month and date values and a four-digit year value should be displayed, separated by forward slashes.

If the var attribute is specified, then a String value containing the formatted date is assigned to the named variable. Otherwise, the formatDate tag will write out the formatting results.

4.10. macro

The macro tag allows you define a new custom tag.

<jx:macro name="Name" [targetNamespace="Namespace"]> <jx:parameter name="Name" [optional="Boolean"] [default="Value"]/>* body </jx:macro>

For example:

<jx:macro name="d"> <tr><td></td></tr> </jx:macro>

The tag being defined in this example is <d> and it can be used like any other tag:

<d/>

However, when this tag is used it will be replaced with a row containing a single empty data cell.

When such a tag is used, the attributes and content of the tag become available as variables in the body of the macro's definition, for example:

<jx:macro name="tablerows"> <jx:parameter name="list"/> <jx:parameter name="color"/> <jx:forEach var="item" items="${list}"> <tr><td bgcolor="${color}">${item}</td></tr> </jx:forEach> </jx:macro>

The parameter tags in the macro definition define formal parameters, which are replaced with the actual attribute values of the tag when it is used.

Assuming you had this code in your flowscript:

var greatlakes = ["Superior", "Michigan", "Huron", "Erie", "Ontario"]; sendPage(uri, {greatlakes: greatlakes});

and a template like this:

<table> <tablerows list="${greatlakes}" color="blue"/> </table>

When the tablerows tag is used in this situation the following output would be generated:

<table> <tr><td bgcolor="blue">Superior</td></tr> <tr><td bgcolor="blue">Michigan</td></tr> <tr><td bgcolor="blue">Huron</td></tr> <tr><td bgcolor="blue">Erie</td></tr> <tr><td bgcolor="blue">Ontario</td></tr> </table>

4.11. evalBody

Within the body of a macro the evalBody tag treats the content of the macro tag invocation as a *JX*Template and executes it. For example, the below macro uses this facility to implement the JSTL forTokens tag:

<jx:macro name="forTokens"> <jx:parameter name="var"/> <jx:parameter name="items"/> <jx:parameter name="delims"/> <jx:forEach var="${var}" items="${java.util.StringTokenizer(items, delims)}"> <jx:evalBody/> </jx:forEach> </jx:macro>

The tag produced by this macro can be used like this:

<forTokens var="letter" items="a,b,c,d,e,f,g" delims=","> letter = ${letter}
 </forTokens>

which would create the following output:

letter = a
 letter = b
 letter = c
 letter = d
 letter = e
 letter = f
 letter = g

4.12. eval

The eval tag permits dynamic evaluation of custom tags.

<jx:eval select="Expression"/>

Within the body of a macro, information about the current invocation is available via a special variable macro. This variable contains the following properties:

Property:	Description:
arguments	A map containing the all of the attributes from the tag invocation
body	A reference to the content of the tag invocation

You can store the value of body in another variable and invoke it later using jx:eval. The select attribute of jx:eval specifies an expression that must evaluate to a macro invocation. For example:

<jx:set var="tags" value="${java.util.HashMap()}"/> <jx:macro name="dynamic-tag"> <jx:parameter name="id"/> <jx:set var="ignored" value="${tags.put(id, macro.body)}"/> </jx:macro> <dynamic-tag id="example"> This tag was invoked dynamically </dynamic-tag> I'm about to invoke a dynamic tag: <jx:eval select="${tags.example}"/>

The above template produces the following output:

I'm about to invoke a dynamic tag: This tag was invoked dynamically

1. Comments

add your comments