Soto STM Tag Reference

Table of Contents

General

stm:abort

This tag aborts a state execution flow (internally call `abort()` on the `Result` instance).

Example

```
<stm:state id="processOrder" success="displayOk">
  <stm:if test="user.shoppingCart.items.size == 0" scopes="params">
    <stm:stateRef id="displayNoItemsInCart" />
    </stm:abort>
  </stm:if>
  ...
</stm:state>
```

stm:assert

Test if a given value is present in a scope, or given set of scopes. This tag helps avoid `NullPointerException`s. If the assertion performed by this tag fails, state execution aborts with an error.

Attributes

Name	*Value*	*Comments*	*Mandatory*
key	The name under which the “asserted” value is expected to be bound in the scope(s).	-	no
msg	The message to pass to the execution `Result` if no value could be found for the given key.	Will assign a default message if none is specified.	no
scopes	The comma- delimited list of scopes that should be searched for the given key/value.	Scopes are searched in the order in which they are specified in the list. Will search all scopes if no scope is specified.	no

Example

```
<stm:state id="processOrder" success="displayOk">
  <stm:assert key="user" scopes="session" />
  <stm:if test="user.shoppingCart.items.size == 0" scopes="params">
```

```
    <stm:stateRef id=”displayNoItemsInCart” />
    </stm:abort>
  </stm:if>
  ...
</stm:state>
```

stm:echo

Echoes a message to stdout.

Attributes

Name	*Value*	*Comments*	*Mandatory*
msg	The message to echo.	-	no

Example

```
<stm:state id=”processOrder” success=”displayOk”>
  <stm:echo msg=”processing order” />
  <stm:assert key=”user” scopes=”session” />
  <stm:if test=”user.shoppingCart.items.size == 0” scopes=”params”>
    <stm:stateRef id=”displayNoItemsInCart” />
    </stm:abort>
  </stm:if>
  ...
</stm:state>
```

stm:export

Exports an object from one scope to another (so can be considered an import also: importing an object from another...). Note that the object also remains in the scope of origin.

Attributes

Name	*Value*	*Comments*	*Mandatory*
from	The name of the scope from which the object should be copied.	-	yes
to	The name of the scope to which the object should be copied.	-	yes
key	The key/name under which the object to copy appears in the “from” scope.	-	yes
exportKey	The key/name under which the object should	If not specified, the value of the attribute “key” (see	no

Name	*Value*	*Comments*	*Mandatory*
	be copied in the destination scope.	above) is used.	

Example

```
<stm:state id="processOrder" success="displayOk">
  <stm:assert key="user" scopes="session" />
  <stm:if test="user.shoppingCart.items.size == 0" scopes="params">
    <stm:stateRef id="displayNoItemsInCart" />
    </stm:abort>
  </stm:if>
  <stm:export key="user" from="session" to="view" />
  ...
</stm:state>
```

stm:push

Push an object from a given scope to the execution stack. The object therefore becomes the “current” object on the stack, and can from then on be acquired from that stack using the `currentObject()` method on of the `Context` instance.

Attributes

Name	*Value*	*Comments*	*Mandatory*
from	The name of the scope from which the object should be copied.	-	yes
key	The key/name under which the object to copy appears in the “from” scope.	-	yes

Example

```
<stm:state id="processOrder" success="displayOk">
  <stm:assert key="user" scopes="session" />
  <stm:if test="user.shoppingCart.items.size == 0" scopes="params">
    <stm:stateRef id="displayNoItemsInCart" />
    </stm:abort>
  </stm:if>
  <stm:push key="user" from="session" />
  ...
</stm:state>
      <freemarker:template src="pages/adminMenu.ftl" />
      <freemarker:template src="pages/adminMenu.ftl" />
```

stm:stateRef

This tag is a step that redirects execution to a given state.

Attributes

Name	*Value*	*Comments*	*Mandatory*
id	The idenfifier of the state to which execution should be redirected.	-	yes
module	The module to which the state to redirect to belong.	If this attributes is omitted, then a state whose identifier matches the one specified is searched in the current .	no

stm:var

Creates a variable in a given scope.

Attributes

Name	*Value*	*Comments*	*Mandatory*
scope	The name of the scope to which the variable will be bound.	-	yes
key	The key/name of the variable.	-	yes
value	The value of the variable.		yes

Example

```
<stm:state id="processOrder" success="displayOk">
  <stm:assert key="user" scopes="session" />
  <stm:if test="user.shoppingCart.items.size == 0" scopes="params">
    <stm:var key="Your shopping cart is empty" scope="view" />
    <stm:stateRef id="displayNoItemsInCart" />
    </stm:abort>
  </stm:if>
  ...
</stm:state>
```

Note that any object reference can conveniently be passed as a value (as supported by Soto's configuration format):

```
<stm:state id="processOrder" success="displayOk">
  <stm:assert key="user" scopes="session" />
  <stm:if test="user.shoppingCart.items.size == 0" scopes="params">
    <stm:var key="Message"
             scope="view">
```

```
      <value>
        <stm:serviceRef id=”acme/myorg/services/internationalization” />
      </value>
    <stm:var>
    <stm:stateRef id=”displayNoItemsInCart” />
    </stm:abort>
  </stm:if>
  ...
</stm:state>
```

Conditional Branching

stm:if

This tag is a step that executes its nested tags if a given condition is verified. This tag uses EL (http://jakarta.apache.org/commons/el/) as an expression evaluation mechanism. Expressions are evaluated agagainst specified scopes (and values therein).

Attributes

Name	*Value*	*Comments*	*Mandatory*
test	An EL expression to evaluate.	-	yes
scopes	A comma delimited list of scopes in which to search for the variables in the expression.	If no scope is specified, all scopes will be searched (this is both less performant and more error prone).	no

Elements

This tag takes one to many nested steps that are executed

Example

```
<stm:state id=”checkLogin”>
  <stm:if test=”user == null” scopes=”session”>
    <stm:stateRef id=”displayLoginPage” />
    <stm:abort />
  </stm:if>
</stm:state>
```

stm:choose

This tag is a step that executes its nested tags if a given condition is verified. Many conditions to check for can be provided (as nested `when` elements).This tag uses EL (http://jakarta.apache.org/commons/el/) as an expression evaluation mechanism. Expressions are evaluated agagainst specified scopes (and values therein).

Elements

when

Attributes:

Name	*Value*	*Comments*	*Mandatory*
test	An EL expression to evaluate.	-	yes
scopes	A comma delimited list of scopes in which to search for the variables in the expression.	If no scope is specified, all scopes will be searched (this is both less performant and more error prone).	no

Elements: this element takes any nested step (that is executed if the tested condition is verified).

otherwise

Elements: this element takes any nested step (that is executed if the tested condition is verified).

Example

```
<stm:state id=”displayAdminMenu”>
  <stm:choose>
    <when test=”user.role == 'superadmin'” scopes=”session”>
      <freemarker:template src=”pages/adminMenu.ftl” />
    </when>
    <otherwise>
      <freemarker:template src=”pages/unauthorized.ftl” />
    </otherwise>
  </stm:choose>
</stm:state>
```

Form Handling

stm:form

Initializes an object conforming to the JavaBeans spec with values from given scopes.

Attributes

Name	***Value***	***Comments***	***Mandatory***
class	The JavaBean class to use.	If no class is specified, the current object on the execution stack is initialized.	no

Elements

param

Attributes:

Name	***Value***	***Comments***	***Mandatory***
scopes	The comma-delimited list of scopes that should be searched for the given key/value.	If no scope is specified, all are searched.	no
from	The key/name of the object to lookup in the specified scopes	-	yes
to	The name of the JavaBean attribute to which the “from” object will be assigned.	If no value is specified, the value of the “from” attribute (above) will be used.	no

Example

```
<stm:state id=”fillUserInfo” success=”displayOk”>
  <stm:assert key=”firstName”   scopes=”params” />
  <stm:assert key=”lastName”    scopes=”params” />
  <stm:assert key=”phoneNumber” scopes=”params” />
  <stm:assert key=”userAccount” scopes=”session” />
  <stm:push key=”userAccount” from=”session” />
  <stm:form>
    <param scopes=”params” from=”firstName” />
    <param scopes=”params” from=”lastName” />
    <param scopes=”params” from=”phoneNumber” />
  </stm:form>
  ...
</stm:state>
```

Scripting

STM integrates various scripting languages that allow greatly reducing development time. All language binding implementations make a `result` variable available to the executing script – this variable is in fact the `org.sapia.soto.state.Result` instance that is passed to the script's corresponding tag implementation upon execution.

stm:groovy

This tag executes code written in the Groovy (http://groovy.codehaus.org)language. The code is passed as character data as part of the tag's XML content.

The Groovy code compiles to Java byte code. Therefore, a signifcant overhead can be observed at load time.

Elements

Name	*Value*	*Comments*	*Mandatory*
import	An “import” string (corresponds to Java's `import` statement).	-	no

Example

```
<stm:state id=”createUserAccount” success=”displayOk”>
  <stm:assert key=”firstName”   scopes=”params” />
  <stm:assert key=”lastName”    scopes=”params” />
  <stm:assert key=”phoneNumber” scopes=”params” />

  <stm:form class=”net.acme.myorg.database.UserAccount”>
    <param scopes=”params” from=”firstName” />
    <param scopes=”params” from=”lastName” />
    <param scopes=”params” from=”phoneNumber” />
  </stm:form>

  <stm:groovy>
    <import>net.acme.myorg.database.UserAccountManager</import>
    account = result.getContext().getCurrentObject();
    UserAccountManager.getInstance().save(account);
  </stm:groovy>
  ...
</stm:state>
```

stm:bsh

This tag executes code in the Beanshell (http://www.beanshell.org)language. The code is passed as character data as part of the tag's XML content.

Example

```
<stm:state id=”createUserAccount” success=”displayOk”>
  <stm:assert key=”firstName”   scopes=”params” />
  <stm:assert key=”lastName”    scopes=”params” />
  <stm:assert key=”phoneNumber” scopes=”params” />

  <stm:form class=”net.acme.myorg.database.UserAccount”>
    <param scopes=”params” from=”firstName” />
    <param scopes=”params” from=”lastName” />
    <param scopes=”params” from=”phoneNumber” />
  </stm:form>

  <stm:bsh>
    import net.acme.myorg.database.UserAccountManager;
    account = result.getContext().getCurrentObject();
    UserAccountManager.getInstance().save(account);
  </stm:bsh>
  ...
</stm:state>
```

stm:jython

This tag executes code in the Python (http://www.jython.org)language. The code is passed as character data as part of the tag's XML content.

Note that using this tag first requires configuration a `org.sapia.soto.jython.JythonService` as part of your application. The service implementation initializes Jython's runtime. Configuring the Jython service is done like so:

```
<jython:configuration>
  <!-- sets the python.path property; this path indicates were your python
       modules are located. -->
  <pythonPath>/some/path1;/some/path2</pythonPath>
  <!-- indicates the home of your jython installation -->
  <pythonHome>/path/to/jython/home</pyhtonHome>
  <!-- allows specifying additional properties that should be
        passed to the Jython  runtime -->
  <property name=”prop1” value=”propValue1” />
  <property name=”prop2” value=”propValue2” />
</jython:configuration>
```

Example

```
<stm:state id=”createUserAccount” success=”displayOk”>
  <stm:assert key=”firstName”   scopes=”params” />
  <stm:assert key=”lastName”    scopes=”params” />
  <stm:assert key=”phoneNumber” scopes=”params” />

  <stm:form class=”net.acme.myorg.database.UserAccount”>
```

```
    <param scopes="params" from="firstName" />
    <param scopes="params" from="lastName" />
    <param scopes="params" from="phoneNumber" />
  </stm:form>

  <stm:jython>
    from net.acme.myorg.database import UserAccountManager;
    account = result.getContext().getCurrentObject();
    UserAccountManager.getInstance().save(account);
  </stm:jython>
  ...
</stm:jython>
```

stm:pnuts

This tag executes code in the Pnuts (http://www.jython.org)language. The code is passed as character data as part of the tag's XML content.

Pnuts is a very fast scripting language.

Example

```
  <stm:pnuts>
    System::out.println(result.getContext().currentObject());
  </stm:pnuts>
```

XML / XSL

XML/XSL processing is fully supported when embedding STM in a servlet or in Cocoon. (see Sapia's web site for details on how this is done). The same goes for Freemarker usage.

xml:jelly

This tag implements the `org.sapia.soto.state.State` and `org.sapia.soto.state.Step` interfaces. Therefore, it can be configured as a state, or as a step in another state (if that state accepts nested steps).

This tag isprovided with the URI of a Jelly script (http://jakarta.apache.org/commons/jelly). The tag exports the current object on the context stack to the script, under the `Model` key. In addition, it exports all view parameters that have been specified to the script (therefore, the passed in `Context` instance is expected to implement the `org.sapia.soto.state.MVC` interface).

This tag emits SAX events (resulting from the script's execution) that are piped to the context's `ContentHandler`. Therefore, the passed in context is also expected to implement he `org.sapia.soto.state.xml.XMLContext` interface.

In addition, this tag supports nested steps. This is mainly of use in conjunction with `xml:xsl` tags: XSL transformations can thus be applied to the result of a Jelly script (multiple successive transformations can be applied, creating an XSL transformation pipeline).

Attributes

Name	***Value***	***Comments***	***Mandatory***
src	The URI of a Jelly script.	-	yes

Elements

This tag takes any element that corresponds to a step. Furthermore, it supports adding parameters prior to the script's execution (see the examples further below).

Name	***Value***	***Comments***	***Mandatory***
param	-	-	no

Examples

The following illustrates how an XSL transformation is applied to the output of a Jelly script.

Here, some `UserAccount` object would be pushed on the context stack before triggering the `displayUserAccount` state. The object is exported to the script, which generates the XML corresponding to the object. Then, that XML is processed by an XSL stylesheet to produce XHTML that displays the user account data.

```
<xml:jelly id="displayUserAccount" src="jelly/domain/userAccount.xml">
  <xml:xsl src="style/html/domain/userAccount.xsl">
</xml:jelly>
```

Adding conditional branching to the mix provides a powerful combination:

```
<xml:jelly id="displayUserAccount" src="jelly/userAccount.xml">
  <stm:choose>
    <when test="outputType == 'WAP'" scopes="request">
      <xml:xsl src="style/wap/domain/userAccount.xsl">
    </when>
    <otherwise>
      <xml:xsl src="style/html/domain/userAccount.xsl">
    </otherwise>
  </stm:choose>
</xml:jelly>
```

Furthermore, arbitratry parameters can be bound to the script using `param` elements, like so:

```
<xml:jelly id="displayUserAccount" src="jelly/userAccount.xml">
  <param name="UserService">
    <value>
      <soto:serviceRef id="domain/users">
    </value>
  </param>
  <param name="SomeNumber">
    <value>
      <soto:int value="10" />
    </value>
  </param>
  <stm:choose>
    <when test="outputType == 'WAP'" scopes="request">
      <xml:xsl src="style/wap/domain/userAccount.xsl">
    </when>
    <otherwise>
      <xml:xsl src="style/html/domain/userAccount.xsl">
    </otherwise>
  </stm:choose>
</xml:jelly>
```

In the above, the given parameters would be exported to the Jelly script, and recuperable through Jelly's variable interpolation notation (for example: `${UserService}`).

xml:domify

This tag implements the `org.sapia.soto.state.State` and `org.sapia.soto.state.Step` interfaces. Therefore, it can be configured as a state, or as a step in another state (if that state accepts nested steps).

This tag makes use of the Domify utility (http://domify.sourceforge.net/). Domify allows wrapping an arbitratry JavaBean into a w3c dom `Node` implementation. Thus, it allows passing any JavaBean to an XSL stylesheet that generates XSL based on that been (or more precisely, based on the `Node` implementation that wraps the bean). Thus, this tag must be provided with one to many nested `xml:xsl` elements that will apply transformations on the passed in `Node`. The bean that the note will wrap is in fact the current object on the context stack, with the `Model` string acting as the name of the node .

Attributes

Name	*Value*	*Comments*	*Mandatory*
rootName	The name of the root element of the Node implementation that wraps the JavaBean serving as a model.	If this attribute is not specified, then the `Model` string is used.	no

Examples

Here, some `UserAccount` object would be pushed on the context stack before triggering the `displayUserAccount` state. The object is wrapped in a `Node` before the nested `xml:xsl` tags are executed.

```
<xml:domify id=”displayUserAccount” rootName=”UserAccount”>
  <xml:xsl src=”domify/domain/userAccount.xsl”>
  <xml:xsl src=”style/html/domain/userAccount.xsl”>
</xml:domify>
```

Adding conditional branching:

```
<xml:domify id=”displayUserAccount” rootName=”UserAccount”>
  <xml:xsl src=”domify/domain/userAccount.xsl”>
  <stm:choose>
    <when test=”outputType == 'WAP'” scopes=”request”>
      <xml:xsl src=”style/wml/domain/userAccount.xsl”>
    </when>
    <otherwise>
      <xml:xsl src=”style/html/domain/userAccount.xsl”>
    </otherwise>
  </stm:choose>
</xml:domify>
```

Here's an wha the first stylesheet (the one receiving the wrapped JavaBean) would look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sapia="http://www.sapia-oss.org/2003/styles">

  <xsl:template match="/UserAccount">
    <userAccount>
      <xsl:apply-templates select=”.” />
    </userAccount>
  </xsl:template>

  <xsl:template match=”username”>
    <username>
      <xsl:value-of select=”.” />
    </username>
  </xsl:template>

  <xsl:template match=”firstName”>
    ...
  </xsl:template>
  ...
</xsl:stylesheet>
```

xml:include

This tag is used to include the SAX events resulting of a given state's execution. Its class implements both the `State` and `Step` interfaces (in can therefore be used as a state, or as a step within a state) One of the use of this tag is to enable the post-processing of these SAX events as part of an XSL pipeline.

Attributes

Name	*Value*	*Comments*	*Mandatory*
id	The identifier of the state that this tag corresponds to (if used as a state rather then a step).	-	no (if used as a step).
target	The path of the target state to include.	Can be an absolute path (that will be resolved from the root state machine) or a relative one (that will be resolved relatively to the current state machine).	yes

Example

```
<xml:include id="viewContactList" target="/rest-api/contactList">
  <stm:choose>
    <when test="outputType == 'xml'" scopes="params, request">
      <xml:style src="xsl/domain/xml2html.xsl" />
    </when>
    <otherwise>
      <xml:style src="xsl/listContacts.xsl" />
      <xml:style src="xsl/sapia.xsl" />
    </otherwise>
  </stm:choose>
</xml:include>
```

The incuded source of SAX events (in this case, the `/rest-api/contactList` state) emits XML that can be transformed according to an expected output; the source is made independent of the type of transformations that will be applied.

xml:resource

Thi s tag can be used as a state, or as a step within a state (its class implements both the `Step` and `State` interfaces). Itparses an XML resource corresponding to a specified URI. The resource is acquired using Soto's resource resolving mechanism. This tag generates SAX events upon parsing the resource; the events are piped to the `XMLContext` that the tag expects upon execution.

Attributes

Name	*Value*	*Comments*	*Mandatory*
src	The URI of the resource to parse.	-	yes

Example

This example shows how a resource's SAX events are passed through an XSL pipeline by way of an `include`:

```
<xml:include id="getApiSpec" target="apiSpec">
  <stm:choose>
    <when test="outputType == 'WAP'" scopes="request">
      <xml:xsl src="style/wml/doc/apiToWml.xsl">
    </when>
    <otherwise>
      <xml:xsl src="style/html/doc/apiToHtml.xsl">
    </otherwise>
  </stm:choose>
</xml:include>

<xml:resource id="apiSpec" src="xml/docs/apiSpec.xml" />
```

xml:dom

This tag can be used as a state, or as a step within a state (its class implements both the `Step` and `State` interfaces). It is meant to convert SAX events into `org.w3c.dom.DOM` trees. Therefore, it can conveniently be used in conjunction with other STM tags that generate SAX events. The DOM object (in fact a `org.w3c.dom.Node`) is pushed onto the context stack. It is thus made available to other tags, for example the XPath-related ones. This usage allows processing XML in a manner analogous to BPEL, where XML nodes are handled declaratively.

Attributes

Name	***Value***	***Comments***	***Mandatory***
target	The path whose corresponding state should be triggered – and expected to produce SAX events.	This attribute can be omitted. In this case, this tag takes nested tags that are expected to produce SAX events.	no

Examples

Refering to another state

The configuration below shows that we can trigger another state whose outcome consists of SAX events

```
<xml:jelly id="getOutput" src="scripts/jelly/output.xml" />
...
<xml:dom target="getOutput" />
```

Nesting

The configuration below shows that we can trigger another state whose outcome consists of SAX events

```
<xml:dom id="getOutput">
  <xml:jelly id="getOutput" src="scripts/jelly/output.xml">
</xml:dom>
```

xml:if

An instance of this tag is used to conditionally execute nested tags based on the result of an XPath expression. The tag expects a `org.w3c.dom.Node` instance to be the current object on the execution context's stack, and the XPath expression is thus evaluated on that `Node`.

This tag can conveniently works with the `xml:dom` tag.

Attributes

Name	*Value*	*Comments*	*Mandatory*
test	An XPath expression that is evalutated on the current object on the execution stack (expected to be an `org.w3c.dom.Node`).	-	true

Example

```
<stm:state id="processBook">
  <xml:if test="order/book/publisher/@name = 'Simon and Schuster'">
    <stm:stateRef id="applyDiscount" />
  </xml:if>
  ...
</stm:state>
```

xml:choose

An instance of this tag is used to conditionally execute nested tags based on the result of an XPath expression, but in a "switch-like" manner. The tag expects a `org.w3c.dom.Node` instance to be the current object on the execution context's stack, and the XPath expression is thus evaluated on that `Node`.

This tag can conveniently works with the `xml:dom` tag.

Elements

This tag takes any element that corresponds to a step. Furthermore, it supports adding parameters prior to the script's execution (see the examples further below).

Name	***Attribute***	***Comments***	***Mandatory***
when	test	This tag can appear more than once under the `xml:choose` tag. The `test` attribute takes an XPath expression that, if evaluated, will trigger execution brancbing into the tags nested within this one.	no
otherwise	-	This tag acts as a default if none of the previous XPath expressions evaluated to `true`.	-

Example

```
<stm:state id=”processMessage”>
  <xml:choose>
    <xml:when test="message/partner/@trust = ’secure’">
      <stm:stateRef id=”encryptMessage” />
    </xml:when>
    <xml:when test="message/partner/@type = ’supplier’">
      <stm:stateRef id=”sendPurchaseOrder” />
    </xml:when>
    <xml:otherwise>
      ...
    </xml:otherwise>
  </xml:choose>
  ...
</stm:state>
```

Freemarker

---

freemarker:template

This tags uses Freermarker (http://freemarker.sourceforge.net) to generate output based on so-called templates. This tag implements the `org.sapia.soto.state.State` and `org.sapia.soto.state.Step` interfaces. Therefore, it can be configured as a state, or as a step in another state (if that state accepts nested steps).

A `freemarker:template` tag is provided with the URI of a Freemarker template.By convention, this tag passes the current object on the context stack to the template; the object is bound to the template under the `Model` key. The bound object can thereafter be recuperated from within the template, using Freemarker's variable interpolation notation (`${Model}`). Furtermore, all view parameters are also exported to the template and can be used therein in a similar manner (therefore, the passed in `Context` instance is expected to implement the `org.sapia.soto.state.MVC` interface).

Note that using this tag requires first configuring `org.sapia.soto.state.freemarker.FreemarkerService` instance. This can be done in your Soto application configuration, like so:

```
<soto:service>
  <freemarker:configuration>
    <!-- Enables/disables Freemarker's localized lookup feature -->
    <localized>true</localized>
    <
    <setting name="default_encoding" value="UTF-8" />
  </freemarker:configuration>
</soto:service>
```

Upon rendering a template, this tag passes the `Locale` instance that is encapsulated within the current `Context` to the `FreemarkerService`. so that the template can be retrieve using Freemarker's localization capabilities.

The `FreemarkerService` implementation in fact wraps a `freemarker.template.Configuration` instance. Refer to Freemarker's documentation for more details on the supported settings.

Attributes

Name	***Value***	***Comments***	***Mandatory***
src	The URI of the Freemarker template to execute.	-	yes

Example

```
<freemarker:template id=”displayUserAccount”
                     src=”freemarker/domain/userAccount.ftl” />
```

freemarker:include

This tag is used in conjunction with the `freemarker:template` tag. It is also encapsulating a Freemarker template, yet this time it is meant to “import” the content generated by it as a variable into the parent template. For example, it can be used in cases where pages obey a predefined, constant layout, and where only part of layout changes. Such a structure is typically encountered in header/content/footer type of web sites.

Attributes

Name	***Value***	***Comments***	***Mandatory***
name	The name of the variable under which the content generated by this tag will be bound (in the “`view`” scope)	-	yes
src	The URI of the Freemarker template to execute.	-	yes

Example

```
<freemarker:template src="freemarker/layout.ftl">
  <!-- the following is meant to illustrate that other STM tags
       can be mingled -->
  <stm:choose>
    <when test="ERROR != null " scopes="view">
      <freemarker:include name="CONTENT" src="freemarker/error.ftl" />
    </when>
    <otherwise>
      <freemarker:include name="CONTENT"
                          src="freemarker/user/account.ftl" />
    </otherwise>
  </stm:choose>
</freemarker:template>
```

In the example above, the `CONTENT` variable would be imported into the `layout.ftl` template. The content in this case corresponds to the rendition of the `account.ftl` template (or `error.ftl` template, depending on the context). Here is what the `layout.ftl` template would look like:

```
<html>
  <body>
    <div id=”menu”>
      <h1>Menu</h1>
        ...
    </div>
    <div id=”content”>
      ${CONTENT}
    </div>
  <body>
</html>
```

HTTP

http:rest

This tag acts as a conditional switch that branches based on the method (GET, PUT, POST, PUT) of the incoming HTTP request.This tag expects a `CocoonContext` upon execution. This tag can be used as a state, or as a step within a state (it implements the `State` and `Step` interfaces).

Attributes

Name	*Value*	*Comments*	*Mandatory*
type	The URI of the Freemarker template to execute.	-	yes

Elements

This tag takes `get`, `post`, `put`, `delete` element s that take nested steps as arguments. The nested steps will be executed if their parent element corresponds to the method of the current HTTP request.

If no match occurs (if the method of the current HTTP request is not matched by one of the child elements of this tag), an error is generated.

Example

```
<cocoon:rest id="contact">
  <get>
    <stm:stateRef id="doGetContact" />
  </get>
  <post>
    <stm:stateRef id="doUpdateContact" />
  </post>
  <put>
    <stm:stateRef id="doAddContact" />
  </put>
</cocoon:rest>
```