

ESB Message Action Guide

4.2

JBoss Enterprise SOA Platform



ISBN:

Publication date: February, 2008

The SOA Platform edition of the JBoss ESB Message Action Guide

ESB Message Action Guide: JBoss Enterprise SOA Platform

Copyright © 2008 Red Hat, Inc

Copyright © 2008 Red Hat, Inc. This material may only be distributed subject to the terms and conditions set forth in the Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License (which is presently available at <http://creativecommons.org/licenses/by-nc-sa/3.0/>).

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

1801 Varsity Drive
Raleigh, NC 27606-2072
USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park, NC 27709
USA

Preface	vii
1. Document Conventions	vii
2. We Need Feedback	viii
1. Out of the Box Actions	1
1. Transformers & Converters	1
2. Business Process Management	7
3. Scripting	9
4. Routing	10
5. Webservices/SOAP	19
6. Miscellaneous	24
2. Developing Custom Actions	27
1. Configuring Actions Using Properties	27
A. JAXB Annotations	31
1. Configuring JAXB Annotation Introductions in JBossWS 2.0.0	31
2. Writing JAXB Annotation Introduction Configurations	32

Preface

1. Document Conventions

Certain words in this manual are represented in different fonts, styles, and weights. This highlighting indicates that the word is part of a specific category. The categories include the following:

Courier font

Courier font represents `commands, file names and paths, and prompts`.

When shown as below, it indicates computer output:

```
Desktop      about.html    logs          paulwesterberg.png
Mail         backupfiles   mail          reports
```

Courier font

Bold Courier font represents text that you are to type, such as: `service jonas start`

If you have to run a command as root, the root prompt (`#`) precedes the command:

```
# gconftool-2
```

italic Courier font

Italic Courier font represents a variable, such as an installation directory:

```
install_dir/bin/
```

bold font

Bold font represents **application programs** and **text found on a graphical interface**.

When shown like this: **OK**, it indicates a button on a graphical application interface.

Additionally, the manual uses different strategies to draw your attention to pieces of information. In order of how critical the information is to you, these items are marked as follows:



Note

A note is typically information that you need to understand the behavior of the system.



Tip

A tip is typically an alternative way of performing a task.



Important

Important information is necessary, but possibly unexpected, such as a configuration change that will not persist after a reboot.



Caution

A caution indicates an act that would violate your support agreement, such as recompiling the kernel.



Warning

A warning indicates potential data loss, as may happen when tuning hardware for maximum performance.

2. We Need Feedback

If you find a typographical error in the *ESB Message Action Guide*, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in JIRA: <http://jira.jboss.com/jira/> [http://jira.jboss.com/jira] against the Documentation component of the *SOA Platform* project.

When submitting a bug report, be sure to mention the manual's identifier:

ESB_MAG

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Out of the Box Actions

This section provides a catalog of all Actions that are supplied out-of-the-box with JBoss ESB ("pre-packed").

1. Transformers & Converters

Converters/Transformers are a classification of Action Processor responsible for transforming a message (payload, headers, attachments etc.) from a format produced by one message exchange participant into a format that is consumable by another message exchange participant.

ByteArrayToString

Takes a `byte[]` based message payload and converts it into a `java.lang.String` object instance, bound to the message under the name `"org.jboss.soa.esb.actions.current.after"`.

Input Type	<code>byte[]</code>
Class	<code>org.jboss.soa.esb.actions.converters.ByteArrayToString</code>
Properties	encoding: The binary data encoding on the message byte array. Defaults to "UTF-8" when not specified.
Sample Configuration	<pre><action name="transform" class="org.jboss.soa.esb.actions.converters.ByteArrayToString"> <property name="encoding" value="UTF-8" /> </action></pre>

Table 1.1. ByteArrayToString

LongToDateConverter

Takes a `long` based message payload and converts it into a `java.util.Date` object instance, bound to the message under the name `"org.jboss.soa.esb.actions.current.after"`.

Input Type	<code>java.lang.Long/long</code>
Output Type	<code>java.util.Date</code>
Class	<code>org.jboss.soa.esb.actions.converters.LongToDateConverter</code>
Properties	None
Sample Configuration	<pre><action name="transform" class="org.jboss.soa.esb.actions.converters.LongToDateConverter"/></pre>

Table 1.2. LongToDateConverter

ObjectInvoke

Takes the Object bound to a message under the name

`org.jboss.soa.esb.actions.current.after` and supplies it to a configured “processor” for processing. The processing result is bound to the message under the name

`org.jboss.soa.esb.actions.current.after` (overwriting the input parameter).

Input Type	User Object
Output Type	User Object
Class	<code>org.jboss.soa.esb.actions.converters.ObjectInvoke</code>
Properties	<p><code>class-processor</code>: The runtime class name of the processor class used to process the message payload.</p> <p><code>class-method</code>: The name of the method on the processor class used to process the method.</p>
Sample Configuration	<pre><action name="invoke" class="org.jboss.soa.esb.actions.converters.ObjectInvoke"> <property name="class-processor" value="org.jboss.MyXXXProcessor" /> <property name="class-method" value="processXXX" /> </action></pre>

Table 1.3. ObjectInvoke

ObjectToCSVString

Takes the Object bound to a message under the name

`org.jboss.soa.esb.actions.current.after` and converts it into a Comma Separated Value (CSV) String based on the supplied message object and a comma-separated `bean-properties` list property.

Input Type	User Object
Output Type	<code>java.lang.String</code>
Class	<code>org.jboss.soa.esb.actions.converters.ObjectToCSVString</code>
Properties	<p><code>bean-properties</code>: List of Object bean property names used to get CSV values for the output CSV String. The Object should support a getter method for each of listed properties.</p> <p><code>fail-on-missing-property</code>: Flag indicating whether or not the</p>

	action should fail if a property is missing from the Object i.e. if the Object doesn't support a getter method for the property. Default value is "false".
Sample Configuration	<pre> <action name="transform" class="org.jboss.soa.esb.actions.converters.ObjectToCSVString"> <property name="bean-properties" value="name,address,phoneNumber" /> <property name="fail-on-missing-property" value="true" /> </action> </pre>

Table 1.4. ObjectToCSVString**ObjectToXStream**

Takes the Object bound to a message under the name

`org.jboss.soa.esb.actions.current.after` and converts it into XML using the [XStream](http://xstream.codehaus.org/) [http://xstream.codehaus.org/] processor.

Input Type	User Object
Output Type	<code>java.lang.String</code>
Class	<code>org.jboss.soa.esb.actions.converters.ObjectToXStream</code>
Properties	<p><code>class-alias</code>: Class alias used in call to XStream.alias(String, Class) [http://xstream.codehaus.org/javadoc/com/thoughtworks/xstream/XStream.html] prior to serialization. Defaults to the input Object's class name.</p> <p><code>exclude-package</code>: Exclude the package name from the generated XML. Default is "true". Not applicable if a "class-alias" is specified.</p>
Sample Configuration	<pre> <action name="transform" class="org.jboss.soa.esb.actions.converters.ObjectToXStream"> <property name="class-alias" value="MyAlias" /> <property name="exclude-package" value="true" /> </action> </pre>

Table 1.5. ObjectToXStream**SmooksTransformer**

Message Transformation on JBossESB is supported by the `SmooksTransformer` component. This is an ESB Action component that allows the Smooks Data

Transformation/Processing Framework to be plugged into an ESB Action Processing Pipeline.

A wide range of source (XML, CSV, EDI etc.) and target (XML, Java, CSV, EDI etc.) data formats are supported by the SmooksTransformer component. A wide range of Transformation Technologies are also supported, all within a single framework. See [Smooks](http://milyn.codehaus.org/Smooks) [http://milyn.codehaus.org/Smooks] for more details.

Class	org.jboss.soa.esb.actions.converters.SmooksTransformer
Properties	<p>Smooks Resource Configuration. resource-config: The Smooks resource configuration file.</p> <p>Message Profile Properties (Optional). from: Message Exchange Participant name. Message Producer. from-type: Message type/format produced by the “from” message exchange participant. to: Message Exchange Participant name. Message Consumer. to-type: Message type/format consumed by the “to” message exchange participant.</p> <p>Note: All the above properties can be overridden by supplying them as properties to the message (Message.Properties).</p>
Sample Configuration	<p>Default Input/Output: .</p> <pre><action name="transform" class="org.jboss.soa.esb.actions.converters.SmooksTransformer"> <property name="resource-config" value="/smooks/config-01.xml" /> </action></pre> <p>Named Input/Output: .</p> <pre><action name="transform" class="org.jboss.soa.esb.actions.converters.SmooksTransformer"> <property name="resource-config" value="/smooks/config-01.xml" /> <property name="get-payload-location" value="get-order-params" /> <property name="set-payload-location" value="get-order-response" /> </action></pre>

Using Message Profiles: .

```

<action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
  <property name="resource-config"
value="/smooks/config-01.xml" />

  <property name="from"
value="DVDStore:OrderDispatchService" />
  <property name="from-type"
value="text/xml:fullFillOrder" />
  <property name="to"
value="DVDWarehouse_1:OrderHandlingService" />
  <property name="to-type"
value="text/xml:shipOrder" />
</action>

```

Table 1.6. SmooksTransformer

Java Objects are bound to the Message.Body under their “[beanId](http://milyn.codehaus.org/javadoc/v1.0/smooks-cartridges/javabean/org/milyn/javabean/BeanPopulator.html)”.

[<http://milyn.codehaus.org/javadoc/v1.0/smooks-cartridges/javabean/org/milyn/javabean/BeanPopulator.html>].

For more on this, please refer to the MessageTransformation document, or the [Wiki](http://wiki.jboss.org/wiki/Wiki.jsp?page=MessageTransformation)

[<http://wiki.jboss.org/wiki/Wiki.jsp?page=MessageTransformation>].

PersistAction

This is used to interact with the MessageStore, where necessary

Input Type	Message
Output Type	The input Message
Class	org.jboss.soa.esb.actions.MessagePersister
Properties	<p>classification: used to classify where the Message will be stored. If the Message Property org.jboss.soa.esb.messagestore.classification is defined on the Message then that will be used instead. Otherwise a default may be provided at instantiation time.</p> <p>message-store-class: the implementation of the MessageStore.</p>
Sample Configuration	<pre> <action name="PersistAction" class="org.jboss.soa.esb.actions.MessagePersister" > <property name="classification" value="test"/> <property name="message-store-class" value="org.jboss.internal.soa.esb.persistence.format.db.DBMessageStor </action> </pre>

Table 1.7. PersistAction**XStreamToObject**

Takes the XML bound to a message under the name

"org.jboss.soa.esb.actions.current.after" and converts it into an Object using the [XStream](http://xstream.codehaus.org/) [http://xstream.codehaus.org/] processor.

Input Type	java.lang.String
Output Type	User Object (specified by "incoming-type" property)
Class	org.jboss.soa.esb.actions.converters.XStreamToObject
Properties	<p>class-alias: Class alias used during serialisation. Defaults to the input Object's class name.</p> <p>exclude-package: Flag indicating whether or not the XML includes a package name.</p> <p>incoming-type: incoming-type</p> <p>root-node: Optional. Specify a different root node then the actual root node in the XML. Takes an XPath expression.</p> <p>aliases: Optional. Specify additional aliases to help XStream to convert the xml elements to Objects</p>
Sample Configuration	<pre><action name="transform" class="org.jboss.soa.esb.actions.converters.XStreamToObject"> <property name="class-alias" value="MyAlias" /> <property name="exclude-package" value="true" /> <property name="incoming-type" value="com.acme.MyXXXClass" /> <property name="root-node" value="/rootNode/MyAlias" /> <property name="aliases"> <alias name="alias1" value="com.acme.MyXXXClass1"/> <alias name="alias2" value="com.acme.MyXXXClass2"/> ... </property> </action></pre>

Table 1.8. XStreamToObject

2. Business Process Management

jBPM - BpmProcessor

JBossESB can make calls into jBPM using the BpmProcessor action. Please also read the jBPIntegrationGuide to learn how to call JBossESB from jBPM. The BpmProcessor action uses the jBPM command API to make calls into jBPM. The following jBPM commands have been implemented:

- `NewProcessInstanceCommand`
- `StartProcessCommand`
- `SignalCommand`
- `CancelProcessInstanceCommand`
- `setProcessInstanceVariables`
- `getTokenVariables`

Input Type	<code>org.jboss.soa.esb.message.Message</code> generated by <code>AbstractCommandVehicle.toCommandMessage()</code>
Output Type	Message – same as the input message
Class	<code>org.jboss.soa.esb.services.jbpm.actions.BpmProcessor</code>
Properties	<ul style="list-style-type: none"> • <code>command</code> - required property. Needs to be one of: <code>NewProcessInstance-Command</code>, <code>StartProcessInstanceCommand</code>, <code>SignalCommand</code> or <code>Cancel-ProcessInstanceCommand</code> • <code>processdefinition</code> - required property for the <code>New-</code> and <code>Start-ProcessInstanceCommands</code> if the <code>process-definition-id</code> property is not used. The value of this property should reference a process definition that is already deployed to jBPM and of which you want to create a new instance. This property does not apply to the <code>Signal-</code> and <code>CancelProcessInstance-Commands</code>. • <code>process-definition-id</code> - required property for the <code>New-</code> and <code>Start-ProcessInstanceCommands</code> if the <code>processdefinition</code> property is not used. The value of this property should reference a <code>processdefinition</code> id in jBPM of which you want to create a new instance. This property does not apply to the <code>Signal-</code> and <code>CancelProcessInstanceCommands</code>. • <code>actor</code> - optional property to specify the jBPM actor id, which applies to the <code>New-</code> and <code>StartProcessInstanceCommands</code> only.

	<ul style="list-style-type: none"> • key - optional property to specify the value of the jBPM key. For example one can pass a unique invoice id as the value for this key. On the jBPM side this key is as the “business” key id field. The key is a string based business key property on the process instance. The combination of business key + process definition must be unique if a business key is supplied. The key value can hold an MVEL expression to extract the desired value from the <code>EsbMessage</code>. For example if you have a named parameter called “businessKey” in the body of your message you would use “body.businessKey”. Note that this property is used for the <code>New-</code> and <code>StartProcessInstanceCommands</code> only. • transition-name - optional property. This property only applies to the <code>StartProcessInstance-</code> and <code>Signal</code> Commands, and is of use only if there are more then one transition out of the current node. If this property is not specified the default transition out of the node is taken. The default transition is the first transition in the list of transition defined for that node in the jBPM <code>processdefinition.xml</code>. • esbToBpmVars - optional property for the <code>New-</code> and <code>StartProcessInstanceCommands</code> and the <code>SignalCommand</code>. This property defines a list of variables that need to be extracted from the <code>EsbMessage</code> and set into jBPM context for the particular process instance. The list consists of mapping elements. Each mapping element can have the following attributes: <ul style="list-style-type: none"> • esb - required attribute which can contain an MVEL expression to extract a value anywhere from the <code>EsbMessage</code>. • bpm - optional attribute containing the name which be used on the jBPM side. If omitted the esb name is used. • default - optional attribute which can hold a default value if the esb MVEL expression does not find a value set in the <code>EsbMessage</code>.
Message variables	<p>Finally some variables can be set on the body of the <code>EsbMessage</code>:</p> <ul style="list-style-type: none"> • jbpmProcessInstId - required parameter which applies to the <code>Cancel-ProcessInstanceCommand</code> only. It is up to the user make sure this value is set as a named parameter on the <code>EsbMessage</code> body. • jbpmTokenId or jbpmProcessInstId – either one is a required parameter and applies to the <code>SignalCommand</code> only. The <code>SignalCommand</code> first looks for the value of the token id to which

	it will send a signal. If this is not set it will try to obtain the process instance id and get the root token It is up to the user make sure either the <code>jbpmTokenId</code> or the <code>jbpmProcessInstId</code> is set on the <code>EsbMessage</code> body.
Sample Configuration	<pre> <action name="create_new_process_instance" class="org.jboss.soa.esb.services.jbpm.actions.BpmProcessor"> <property name="command" value="StartProcessInstanceCommand" /> <property name="process-definition-name" value="processDefinition2"/> <property name="actor" value="FrankSinatra"/> <property name="esbToBpmVars"> <!-- esb-name maps to getBody().get("eVar1") --> <mapping esb="eVar1" bpm="counter" default="45" /> <mapping esb="BODY_CONTENT" bpm="theBody" /> </property> </action> </pre>

Table 1.9. jBPM - CommandInterpreter

3. Scripting

Scripting Action Processors support definition of action processing logic via Scripting languages.

GroovyActionProcessor

Executes a [Groovy](http://groovy.codehaus.org/) [http://groovy.codehaus.org/] action processing script, receiving the message and action configuration as input.

Script Bindings	<ul style="list-style-type: none"> • <code>message</code>: The message. • <code>config</code>: The action configuration (<code>ConfigTree</code>).
Class	<code>org.jboss.soa.esb.actions.scripting.GroovyActionProcessor</code>
Properties	<code>script</code> : Path (classpath) to Groovy script.
Sample Configuration	<pre> <action name="process" class="org.jboss.soa.esb.scripting.GroovyActionProcessor"> <property name="script" value="/scripts/ActionXProcessor.groovy"/> </action> </pre>

Table 1.10. GroovyActionProcessor

4. Routing

Routing Actions support conditional routing of messages between two or more message exchange participants.

Aggregator

Message aggregation action. An implementation of the [Aggregator Enterprise Integration Pattern](http://www.enterpriseintegrationpatterns.com/Aggregator.html) [<http://www.enterpriseintegrationpatterns.com/Aggregator.html>].

Class	org.jboss.soa.esb.actions.Aggregator
Properties	timeoutInMillis: <i>Optional</i> . Timeout time in milliseconds before the aggregation process times out.
Sample Configuration	<pre><action class="org.jboss.soa.esb.actions.Aggregator" name="Aggregator"> <property name="timeoutInMillis" value="60000" /> </action></pre>

Table 1.11. Aggregator

This action relies on all messages having the correct correlation data. This data is set on the message as a property called `aggregatorTag` (`Message.Properties`). See the [ContentBasedRouter](#) and [StaticRouter](#) actions.

The data has the following format:

```
[UUID] ":" [message-number] ":" [message-count]
```

If all the messages have been received by the aggregator, it returns a new `Message` containing all the messages as part of the `Message.Attachment` list (unnamed), otherwise the action returns null.

ContentBasedRouter

Content (plus rules) based message routing action.

Class	org.jboss.soa.esb.actions.ContentBasedRouter
Properties	ruleSet: JBoss Rules ruleset.

	<p>ruleLanguage: CBR evaluation Domain Specific Language (DSL) file.</p> <p>ruleReload: Flag indicating whether or not the rules file should be reloaded each time. Default is "false".</p> <p>destinations: Container property for the <code><route-to></code> configurations.</p> <pre><route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService" /></pre>
"process" methods	<p>process: Don't append aggregation data to message.</p> <p>split: Append aggregation data to message.</p> <p>See Aggregator</p>
Sample Configuration	<pre><action process="split" name="ContentBasedRouter" class="org.jboss.soa.esb.actions.ContentBasedRouter"> <property name="ruleSet" value="MyESBRules-XPath.drl" /> <property name="ruleLanguage" value="XPathLanguage.dsl" /> <property name="ruleReload" value="true" /> <property name="destinations"> <route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService" /> <route-to destination-name="normal" service-category="NormalShipping" service-name="NormalShippingService" /> </property> </action></pre>

Table 1.12. ContentBasedRouter

Refer to the Content Based Routing Guide for more details.

StaticRouter

Static message routing action. This is basically a simplified version of the Content Based Router, except it doesn't support content based routing rules.

Class	org.jboss.soa.esb.actions.StaticRouter
Properties	

	<p>destinations: Container property for the <route-to> configurations.</p> <pre><route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService" /></pre>
"process" methods	<p>process: Don't append aggregation data to message.</p> <p>split: Append aggregation data to message.</p> <p>See Aggregator.</p>
Sample Configuration	<pre><action name="routeAction" class="org.jboss.soa.esb.actions.StaticRouter"> <property name="destinations"> <route-to service-category="ExpressShipping" service-name="ExpressShippingService" /> <route-to service-category="NormalShipping" service-name="NormalShippingService" /> </property> </action></pre>

Table 1.13. StaticRouter**StaticWireTap**

Static message wiretapping action. The StaticWiretap differs from the StaticRouter in that the StaticWiretap "listens in" on the action chain and allows the message to continue in the chain to subsequent actions, while the StaticRouter action only pushes the message to destinations that are defined in its route-to chain.

Class	org.jboss.soa.esb.actions.StaticWiretap
Properties	<p>destinations: Container property for the <route-to> configurations.</p> <pre><route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService" /></pre>
"process" methods	<ul style="list-style-type: none"> process: Don't append aggregation data to message.

	See Aggregator .
Sample Configuration	<pre> <action name="routeAction" class="org.jboss.soa.esb.actions.StaticWiretap"> <property name="destinations"> <route-to service-category="ExpressShipping" service-name="ExpressShippingService"/> <route-to service-category="NormalShipping" service-name="NormalShippingService"/> </property> </action> </pre>

Table 1.14. StaticWireTap**Notifier**

Sends a notification to a list of notification targets specified in configuration, based on the result of action pipeline processing.

The action pipeline works in two stages, normal processing followed by outcome processing. In the first stage, the pipeline calls the process method(s) on each action (by default it is called process) in sequence until the end of the pipeline has been reached or an error occurs. At this point the pipeline reverses (the second stage) and calls the outcome method on each preceding action (by default it is `processException` or `processSuccess`). It starts with the current action (the final one on success or the one which raised the exception) and travels backwards until it has reached the start of the pipeline. The `Notifier` is an action which does no processing of the message during the first stage (it is a no-op) but sends the specified notifications during the second stage.

The `Notifier` class configuration is used to define `NotificationList` elements, which can be used to specify a list of `NotificationTargets`. A `NotificationList` of type “ok” specifies targets which should receive notification upon successful action pipeline processing; a `NotificationList` of type “err” specifies targets to receive notifications upon exceptional action pipeline processing, according to the action pipeline processing semantics mentioned earlier. Both “err” and “ok” are case insensitive.

The notification sent to the `NotificationTarget` is target-specific, but essentially consists of a copy of the ESB message undergoing action pipeline processing. A list of notification target types and their parameters appears at the end of this section.

If you wish the ability to notify of success or failure at each step of the action processing pipeline, use the “okMethod” and “exceptionMethod” attributes in each `<action>` element instead of having an `<action>` that uses the `Notifier` class.

Class	<code>org.jboss.soa.esb.actions.Notifier</code>
-------	---

Properties	NotificationList subtree indicating targets.
Sample Configuration	<pre><action class="org.jboss.soa.esb.actions.Notifier" okMethod="notifyOK"> <property name="destinations"> <NotificationList type="OK"> <target class="NotifyConsole" /> <target class="NotifyFiles" > <file name="@results.dir@/goodresult.log" /> </target> </NotificationList> <NotificationList type="err"> <target class="NotifyConsole" /> <target class="NotifyFiles" > <file name="@results.dir@/badresult.log" /> </target> </NotificationList> </property> </action></pre>

Table 1.15. Notifier

Notifications can be sent to targets of various types. The table below provides a list of the NotificationTarget types and their parameters.

Class	NotifyConsole
Purpose	Performs a notification by printing out the contents of the ESB message on the console.
Attributes	none
Child	none
Child Attributes	none.
Sample Configuration	<pre><target class="NotifyConsole" /></pre>

Table 1.16. NotifyConsole

Class	NotifyFiles
-------	-------------

Purpose	Performs a notification by writing the contents of the ESB message to a specified set of files.
Attributes	none
Child	file
Child Attributes	<ul style="list-style-type: none"> • <code>append</code> – if value is true, append the notification to an existing file • <code>URI</code> – any valid URI specifying a file
Sample Configuration	<pre><target class="NotifyFiles" > <file append="true" URI="anyValidURI"/> <file URI="anotherValidURI"/> </target></pre>

Table 1.17. NotifyFiles

Class	NotifySQLTable
Purpose	Performs a notification by inserting a record into an existing database table. The database record contains the ESB message contents and, optionally, other values specified using nested <code><column></code> elements.
Attributes	<ul style="list-style-type: none"> • <code>driver-class</code> • <code>connection-url</code> • <code>user-name</code> • <code>password</code> • <code>table</code> - table in which notification record is stored • <code>dataColumn</code> - name of table column in which ESB message contents are stored
Child	column
Child Attributes	<ul style="list-style-type: none"> • <code>name</code> – name of table column in which to store additional value • <code>value</code> – value to be stored
Sample Configuration	<pre><target class="NotifySQLTable" driver-class="com.mysql.jdbc.Driver" connection-url="jdbc:mysql://localhost/db" user-name="user"</pre>

	<pre> password="password" table="table" dataColumn="messageData"> <column name="aColumnlName" value="aColumnValue" /> </target> </pre>
--	---

Table 1.18. NotifySQLTable

Class	NotifyQueues
Purpose	Performs a notification by translating the ESB message (including its attached properties) into a JMS message and sending the JMS message to a list of Queues. Additional properties may be attached using the <messageProp> element.
Attributes	none
Child	queue
Child Attributes	<ul style="list-style-type: none"> • jndiName – the JNDI name of the Queue • jndi-URL – the JNDI provider URL (optional) • jndi-context-factory - the JNDI initial context factory (optional) • jndi-pkg-prefix – the JNDI package prefixes (optional) • connection-factory - the JNDI name of the JMS connection factory (by default, "ConnectionFactory")
Child	messageProp
Child Attributes	<ul style="list-style-type: none"> • name - name of the new property to be added • value - value of the new property
Sample Configuration	<pre> <target class="NotifyQueues" > <messageProp name="aNewProperty" value="theValue" /> <queue jndiName="queue/quickstarts_notifications_queue" /> </target> </pre>

Table 1.19. NotifyQueues

Class	NotifyTopics
Purpose	Performs a notification by translating the ESB message (including its attached properties) into a JMS message and publishing the JMS message to a list of Topics. Additional properties may be attached using the <messageProp> element.
Attributes	none
Child	topic
Child Attributes	<ul style="list-style-type: none"> • <code>jndiName</code> – the JNDI name of the Queue • <code>jndi-URL</code> – the JNDI provider URL (optional) • <code>jndi-context-factory</code> - the JNDI initial context factory (optional) • <code>jndi-pkg-prefix</code> – the JNDI package prefixes (optional) • <code>connection-factory</code> - the JNDI name of the JMS connection factory (by default, "ConnectionFactory")
Child	messageProp
Child Attributes	<ul style="list-style-type: none"> • <code>name</code> - name of the new property to be added • <code>value</code> - value of the new property
Sample Configuration	<pre> <target class="NotifyTopics" > <messageProp name="aNewProperty" value="theValue" /> <queue jndiName="queue/quickstarts_notifications_topic" /> </target> </pre>

Table 1.20. NotifyTopics

Class	NotifyEmail
Purpose	Performs a notification by sending an email containing the ESB message content and, optionally, any file attachments.
Attributes	none
Child	topic
Child Attributes	<ul style="list-style-type: none"> • <code>from</code> – email address (<code>javax.email.InternetAddress</code>) • <code>sendTo</code> – comma-separated list of email addresses

	<ul style="list-style-type: none"> • <code>ccTo</code> - comma-separated list of email addresses (optional) • <code>subject</code> – email subject • <code>message</code> - a string to be prepended to the ESB message contents which make up the e-mail message (optional)
Child	Attachment (optional)
Child Text	the name of the file to be attached
Sample Configuration	<pre> <target class="NotifyEmail" from="person@somewhere.com" sendTo="person@elsewhere.com" subject="theSubject"> <attachment>attachThisFile.txt</attachment> </target> </pre>

Table 1.21. NotifyEmail

Class	NotifyFTP
Purpose	Performs a notification by creating a file containing the ESB message content and transferring it via FTP to a remote file system.
Attributes	none
Child	ftp
Child Attributes	<ul style="list-style-type: none"> • <code>URL</code> – a valid FTP URL • <code>filename</code> – the name of the file to contain the ESB message content on the remote system • <code>passive</code> - true false. Specify whether to use active or passive FTP in connecting to the server
Child	Attachment (optional)
Child Text	the name of the file to be attached
Sample Configuration	<pre> <target class="NotifyFTP" > <ftp URL="ftp://username:pwd@server.com/remote/dir" filename="someFile.txt" passive="true"/> </target> </pre>

Table 1.22. NotifyFTP

5. Webservices/SOAP

SOAPProcessor

JBoss Webservices SOAP Processor.

This action supports invocation of a JBossWS hosted webservice endpoint through any JBossESB hosted listener. This means the ESB can be used to expose Webservice endpoints for Services that don't already expose a Webservice endpoint. You can do this by writing a thin Service Wrapper Webservice (e.g. a JSR 181 implementation) that wraps calls to the target Service (that doesn't have a Webservice endpoint), exposing that Service via endpoints (listeners) running on the ESB. This also means that these Services are invocable over any transport channel supported by the ESB (http, ftp, jms etc.).

Dependencies .

1. JBoss Application Server 4.2.0GA or higher.
2. JBossWS 2.0.x or higher
3. The `soap.esb` Service. This is available in the `lib` folder of the distribution.

"ESB Message Aware" Webservice Endpoints.

Note that Webservice endpoints exposed via this action have direct access to the current JBossESB `Message` instance used to invoke this action's `process(Message)` method. It can access the current `Message` instance via the `SOAPProcessor.getMessage()` method and can change the `Message` instance via the `SOAPProcessor.setMessage(Message)` method. This means that Webservice endpoints exposed via this action are "ESB Message Aware".

Webservice Endpoint Deployment.

Any JBossWS Webservice endpoint can be exposed via ESB listeners using this action. That includes endpoints that are deployed from inside (i.e. the Webservice `.war` is bundled inside the `.esb`) and outside (e.g. standalone Webservice `.war` deployments, Webservice `.war` deployments bundled inside a `.ear`) a `.esb` deployment. This however means that this action can only be used when your `.esb` deployment is installed on the JBoss Application Server i.e. It is not supported on the JBossESB Server.

Endpoint Publishing.

See the "Contract Publishing" section of the Administration Guide.

JAXB Annotation Introductions.

The native JBossWS SOAP stack uses JAXB to bind to and from SOAP. This means that an unannotated typeset cannot be used to build a JBossWS endpoint. To overcome this we provide a JBossESB and JBossWS feature called "JAXB Annotation Introductions" which basically means you can define an XML configuration to "Introduce" the JAXB Annotations. For details on how to enable this feature in JBossWS 2.0.0, see [Appendix A, JAXB Annotations](#).

This XML configuration must be packaged in a file called `jaxb-intros.xml` in the `META-INF` directory of the endpoint deployment.

For details on how to write a JAXB Annotation Introductions configuration, see the Appendix.

Action Configuration.

The `<action ... />` configuration for this action is very straightforward. The action requires only one mandatory property value, which is the `"jbossws-endpoint"` property. This property names the JBossWS endpoint that the SOAPProcessor is exposing (invoking).

```
<action name="ShippingProcessor"
        class="org.jboss.soa.esb.actions.soap.SOAPProcessor">
  <property name="jbossws-endpoint" value="ABI_Shipping"/>
  <property name="rewrite-endpoint-url" value="true/false"/>
</action>
```

The optional `"rewrite-endpoint-url"` property is there to support load balancing on HTTP endpoints, in which case the Webservice endpoint container will have been configured to set the HTTP(S) endpoint address in the WSDL to that of the Load Balancer. The `"rewrite-endpoint-url"` property can be used to turn off HTTP endpoint address rewriting in situations such as this. It has no effect for non-HTTP protocols.

Quickstarts.

A number of quickstarts demonstrating how to use this action are available in the JBossESB distribution (`samples/quickstarts`). See the `webservice_jbossws_adapter_01` and `webservice_bpel` quickstarts.

SOAPClient

SOAP Client action processor.

Uses the [soapUI](http://www.soapui.org/) [http://www.soapui.org/] Client Service to construct and populate a message for the target service. This action then routes that message to that service.

Endpoint Operation Specification.

Specifying the endpoint operation is a straightforward task. Simply specify the `"wsdl"` and `"operation"` properties on the SOAPClient action as follows:

```
<action name="soapui-client-action"
class="org.jboss.soa.esb.actions.soap.SOAPClient">
    <property name="wsdl"
value="http://localhost:18080/acme/services/RetailerCallback?wsdl"/>
    <property name="operation" value="SendSalesOrderNotification"/>
</action>
```

SOAP Request Message Construction.

The SOAP operation parameters are supplied in one of 2 ways:

1. As a Map instance set on the *default body location* (`Message.getBody().add(Map)`)
2. As a Map instance set on in a named body location (`Message.getBody().add(String, Map)`), where the name of that body location is specified as the value of the `get-payload-location` action property.

The parameter Map itself can also be populated in one of 2 ways:

1. Option 1: With a set of Objects that are accessed (for SOAP message parameters) using the OGNL framework. More on the use of [OGNL](http://www ognl.org/) [http://www ognl.org/] below.
2. Option 2: With a set of String based key-value pairs(<String, Object>), where the key is an OGNL expression identifying the SOAP parameter to be populated with the key's value. More on the use of OGNL below.

As stated above, OGNL is the mechanism we use for selecting the SOAP parameter values to be injected into the SOAP message from the supplied parameter Map. The OGNL expression for a specific parameter within the SOAP message depends on that the position of that parameter within the SOAP body. In the following message:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:cus="http://schemas.acme.com">
    <soapenv:Header/>
    <soapenv:Body>
        <cus:customerOrder>
            <cus:header>
<cus:customerNumber>123456</cus:customerNumber>
            </cus:header>
        </cus:customerOrder>
    </soapenv:Body>
</soapenv:Envelope>
```

The OGNL expression representing the `customerNumber` parameter is `customerOrder.header.customerNumber`.

Once the OGNL expression has been calculated for a parameter, this class will check the supplied parameter map for an Object keyed off the full OGNL expression (Option 1 above). If no such parameter Object is present on the map, this class will then attempt to load the

parameter by supplying the map and OGNL expression instances to the OGNL toolkit (Option 2 above). If this doesn't yield a value, this parameter location within the SOAP message will remain blank.

Taking the sample message above and using the "Option 1" approach to populating the `customerNumber` requires an object instance (e.g. an `Order` object instance) to be set on the parameters map under the key `customerOrder`. The `customerOrder` object instance needs to contain a `header` property (e.g. a `Header` object instance). The object instance behind the `header` property (e.g. a `Header` object instance) should have a `customerNumber` property.

OGNL expressions associated with Collections are constructed in a slightly different way. This is easiest explained through an example:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:cus="http://schemas.active-endpoints.com/sample/customerorder/2006/04/CustomerOrder.xsd"
xmlns:stan="http://schemas.active-endpoints.com/sample/standardtypes/2006/04/StandardTypes.xsd">
  <soapenv:Header/>
  <soapenv:Body>
    <cus:customerOrder>
      <cus:items>
        <cus:item>
<cus:partNumber>FLT16100</cus:partNumber>
                                <cus:description>Flat 16 feet 100
count</cus:description>
                                <cus:quantity>50</cus:quantity>
                                <cus:price>490.00</cus:price>
<cus:extensionAmount>24500.00</cus:extensionAmount>
                                </cus:item>
                                <cus:item>
<cus:partNumber>RND08065</cus:partNumber>
                                <cus:description>Round 8 feet 65
count</cus:description>
                                <cus:quantity>9</cus:quantity>
                                <cus:price>178.00</cus:price>
<cus:extensionAmount>7852.00</cus:extensionAmount>
                                </cus:item>
                                </cus:items>
          </cus:customerOrder>
        </soapenv:Body>
      </soapenv:Envelope>
```

The above order message contains a collection of order `items`. Each entry in the collection is represented by an `item` element. The OGNL expressions for the order item `partNumber` is constructed as `customerOrder.items[0].partnumber` and `customerOrder.items[1].partnumber`. As you can see from this, the collection entry element (the `item` element) makes no explicit appearance in the OGNL expression. It is represented implicitly by the indexing notation. In terms of an Object Graph (Option 1 above), this could be represented by an `Order` object instance (keyed on the map as `customerOrder`) containing an `items` list (List or array), with the list entries being `OrderItem` instances, which in turn contains `partNumber` etc. properties.

Option 2 (above) provides a quick-and-dirty way to populate a SOAP message without having to create an Object model ala Option 1. The OGNL expressions that correspond with the SOAP operation parameters are exactly the same as for Option 1, except that there's not Object Graph Navigation involved. The OGNL expression is simply used as the key into the Map, with the corresponding key-value being the parameter.

SOAP Response Message Consumption.

The SOAP response object instance can be attached to the ESB Message instance in one of the following ways:

1. On the *default body location* (`Message.getBody().add(Map)`)
2. On a *named body location* (`Message.getBody().add(String, Map)`), where the name of that body location is specified as the value of the `set-payload-location` action property.

The response object instance can also be populated (from the SOAP response) in one of 3 ways:

1. Option 1: As an Object Graph created and populated by the [XStream](http://xstream.codehaus.org/) [http://xstream.codehaus.org/] toolkit.

Option 2: As a set of String based key-value pairs(<String, String>), where the key is an OGNL expression identifying the SOAP response element and the value is a String representing the value from the SOAP message.

Option 3: If Options 1 or 2 are not specified in the action configuration, the raw SOAP response message (String) is attached to the message.

Using XStream as a mechanism for populating an Object Graph (Option 1 above) is straightforward and works well, as long as the XML and Java object models are in line with each other.

The XStream approach (Option 1) is configured on the action as follows:

```
<action name="soapui-client-action"
class="org.jboss.soa.esb.actions.soap.SOAPClient">
  <property name="wsdl"
value="http://localhost:18080/acme/services/RetailerService?wsdl"/>
  <property name="operation" value="GetOrder"/>
  <property name="get-payload-location" value="get-order-params" />
  <property name="set-payload-location" value="get-order-response" />
  <property name="responseXStreamConfig">
    <alias name="customerOrder" class="com.acme.order.Order"
namespace="http://schemas.acme.com/services/CustomerOrder.xsd" />
    <alias name="orderheader" class="com.acme.order.Header"
namespace="http://schemas.acme.com/services/CustomerOrder.xsd" />
    <alias name="item" class="com.acme.order.OrderItem"
namespace="http://schemas.acme.com/services/CustomerOrder.xsd" />
  </property>
</action>
```

In the above example, we also include an example of how to specify non-default named locations for the request parameters Map and response object instance.

To have the SOAP response data extracted into an OGNL keyed map (Option 2 above) and attached to the ESB Message, simply replace the `responseXStreamConfig` property with the `responseAsOgnlMap` property having a value of `true` as follows:

```
<action name="soapui-client-action"
class="org.jboss.soa.esb.actions.soap.SOAPClient">
  <property name="wsdl"
value="http://localhost:18080/acme/services/RetailerService?wsdl"/>
  <property name="operation" value="GetOrder"/>
  <property name="get-payload-location" value="get-order-params" />
  <property name="set-payload-location" value="get-order-response" />
  <property name="responseAsOgnlMap" value="true" />
</action>
```

To return the raw SOAP message as a String (Option 3), simply omit both the `responseXStreamConfig` and `responseAsOgnlMap` properties.

6. Miscellaneous

Miscellaneous Action Processors.

SystemPrintln

Simple action for printing out the contents of a message (ala `System.out.println`).

Will attempt to format the message contents as XML.

Input Type	<code>java.lang.String</code>
Class	<code>org.jboss.soa.esb.actions.SystemPrintln</code>
Properties	<ul style="list-style-type: none">• <code>message</code> - A message prefix.• <code>printfull</code> - If true then the entire message is printed, otherwise just the byte array and attachments.• <code>outputstream</code> - if true then <code>System.out</code> is used, otherwise <code>System.err</code>.
Sample Configuration	<pre><action name="print-before" class="org.jboss.soa.esb.actions.SystemPrintln"> <property name="message" value="Message before action XXX" /> </action></pre>

--	--

Table 1.23. SystemPrintIn

Developing Custom Actions

To implement a custom Action Processor, simply implement the `org.jboss.soa.esb.actions.ActionPipelineProcessor` interface.

This interface supports implementation of stateless actions that have a managed lifecycle. A single instance of a class implementing this interface is instantiated on a per pipeline basis (i.e. per action configuration). This means you can cache resources needed by the action in the `initialise` method, and clean them up in the `destroy` method.

The implementing class should process the message from within the `process` method implementation.

As a convenience, you should simply extend the `org.jboss.soa.esb.actions.AbstractActionPipelineProcessor`.

Example:

```
public class ActionXXXProcessor extends AbstractActionPipelineProcessor {

    public void initialise() throws ActionLifecycleException {
        // Initialise resources...
    }

    public Message process(final Message message) throws
    ActionProcessingException {
        // Process messages in a stateless fashion...
    }

    public void destroy() throws ActionLifecycleException {
        // Cleanup resources...
    }
}
```

1. Configuring Actions Using Properties

Actions generally act as templates that require external configuration to perform their tasks. For example, a `PrintMessage` action might take a property named `message` to indicate what to print and a property `repeatCount` to indicate the number of times to print it. The action configuration in the `jboss-esb.xml` file might look like this:

```
<action name="PrintAMessage" class="test.PrintMessage">
    <property name="information" value="Hello World!" />
    <property name="repeatCount" value="5" />
</action>
```

The default method for loading property values in an action implementation is the use of a `ConfigTree` instance. The `ConfigTree` provides a DOM-like view of the action XML. By default,

actions are expected to have a public constructor that takes a `ConfigTree` as a parameter. For example:

```
public class PrintMessage extends AbstractActionPipelineProcessor {

    private String information;

    private Integer repeatCount;

    public PrintMessage(ConfigTree config) {
        information = config.getAttribute("information");
        repeatCount = new Integer(config.getAttribute("repeatCount"));
    }

    public Message process(Message message) throws
    ActionProcessingException {
        for (int i=0; i < repeatCount; i++) {
            System.out.println(information);
        }
    }
}
```

Another approach to setting action properties is to add setters on the action that correspond to the property names and allow the framework to populate them automatically. In order to have the action bean auto-populated, the action class must implement the `org.jboss.soa.esb.actions.BeanConfiguredAction` marker interface. For example, the following class has the same behavior as the one above.

```
public class PrintMessage extends AbstractActionPipelineProcessor
    implements BeanConfiguredAction {

    private String information;

    private Integer repeatCount;

    public setInformation(String information) {
        this.information = information;
    }

    public setRepeatCount(Integer repeatCount) {
        this.repeatCount = repeatCount;
    }

    public Message process(Message message) {
        for (int i=0; i < repeatCount; i++) {
            System.out.println(information);
        }
    }
}
```

Note that the `Integer` parameter in `setRepeatCount()` is automatically converted from the `String` representation specified in the XML.

The `BeanConfiguredAction` method of loading properties is a good choice for actions that take simple arguments, while the `ConfigTree` method is better when you need to deal with the XML representation directly.

Appendix A. JAXB Annotations

1. Configuring JAXB Annotation Introductions in JBossWS 2.0.0

After installing JBossWS 2.0.x on your JBoss Application Server, you need to do the following in order to enable the JAXB Annotation Introductions feature:

1. Copy `jboss-jaxb-intros.jar` from the `extras/jaxbintros` folder (in the distribution) to the root of the `jbossws.sar` folder in your JBoss Application Server deploy folder.
2. Go to `jbossws.sar/jbossws.beans/META-INF/jboss-beans.xml` on your App Server and add the following bean config. Add it just before the `WSEndpointHandlerDeployer` bean config:

```
<bean name="WSEndpointJAXBIntrosCustomizationsDeployer"
class="org.jboss.wsf.spi.deployment.JAXBIntrosCustomizationsDeployer" />
```

3. Then add an "inject" element for the above bean config in the deployer list configured on the `WSMainDeployerManager` bean. e.g.:

```
<bean name="WSMainDeployerManager"
  class="org.jboss.wsf.spi.deployment.BasicDeployerManager">
  <property name="deployers">
    <list class="java.util.LinkedList"
elementClass="org.jboss.wsf.spi.deployment.Deployer">
      <inject bean="WSEndpointNameDeployer"/>
      <inject bean="WSEndpointJAXBIntrosCustomizationsDeployer"/>
      <inject bean="WSEndpointHandlerDeployer"/>
      <inject bean="WSPublishContractDeployer"/>
      <inject bean="WSClassLoaderInjectionDeployer"/>
      <inject bean="WSServiceEndpointInvokerDeployer"/>
      <inject bean="WSEagerInitializeDeployer"/>
      <inject bean="WSEventingDeployer"/>
      <inject bean="WSEndpointMetricsDeployer"/>
      <inject bean="WSEndpointRegistryDeployer"/>
      <inject bean="WSEndpointLifecycleDeployer"/>
    </list>
  </property>
</bean>
```



Note

Note that after performing these configurations, you must restart your Application Server instance.

2. Writing JAXB Annotation Introduction Configurations

JAXB Annotation Introduction configurations are very easy to write. If you're already familiar with the JAXB Annotations, you'll have no problem writing a JAXB Annotation Introduction configuration.

The XSD for the configuration is available online. In your IDE, register this XSD against the <http://www.jboss.org/xsd/jaxb/intros> namespace.

Only 3 annotations are currently supported:

1. [@XmlType](#)

[<https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlType.html>]: On the `Class` element.

2. [@XmlElement](#)

[<https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlElement.html>]: On the `Field` and `Method` elements.

3. [@XmlAttribute](#)

[<https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlAttribute.html>]: On the `Field` and `Method` elements.

The basic structure of the configuration file follows the basic structure of a Java class i.e. a `Class` containing `Fields` and `Methods`. The `<Class>`, `<Field>` and `<Method>` elements all require a `name` attribute for the name of the `Class`, `Field` or `Method`. The value of this `name` attribute supports regular expressions. This allows a single Annotation Introduction configuration to be targeted at more than one `Class`, `Field` or `Member` e.g. setting the namespace for a fields in a `Class`, or for all `Classes` in a package etc.

The Annotation Introduction configurations match exactly with the Annotation definitions themselves, with each annotation “element-value pair” represented by an attribute on the annotations introduction configuration. Use the XSD and your IDE to editing the configuration.

So here's an example:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jaxb-intros xmlns="http://www.jboss.org/xsd/jaxb/intros">

  <!--
    The type namespaces on the customerOrder are different from the rest of
```

```
the message...
-->
<Class name="com.activebpel.ordermanagement.CustomerOrder">
  <XmlType propOrder="orderDate,name,address,items" />
  <Field name="orderDate">
    <XmlAttribute name="date" required="true" />
  </Field>
  <Method name="getXYZ">
    <XmlElement
namespace="http://org.jboss.esb/quickstarts/bpel/ABI_OrderManager"
  nillable="true" />
  </Method>
</Class>
<!--
More general namespace config for the rest of the message...
-->
<Class name="com.activebpel.ordermanagement.*">
  <Method name="get.*">
    <XmlElement
namespace="http://ordermanagement.activebpel.com/jaws" />
  </Method>
</Class>

</jaxb-intros>
```

