JBossESB 4.2 Milestone Release 3

Message Action Guide

Legal Notices

The information contained in this documentation is subject to change without notice.

JBoss Inc. makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. JBoss Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Java™ and J2EE is a U.S. trademark of Sun Microsystems, Inc. Microsoft® and Windows NT® are registered trademarks of Microsoft Corporation. Oracle® is a registered U.S. trademark and Oracle9™, Oracle9 Server™ Oracle9 Enterprise Edition™ are trademarks of Oracle Corporation. Unix is used here as a generic term covering all versions of the UNIX® operating system. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Copyright

Software Version

JBossESB 4.2 Milestone Release 3

Restricted Rights Legend

Contents

About This Guide

What This Guide Contains

The goal of this document is to:

1. Provide a catalog of all Message Action implementations provided with JBoss ESB (out-of-the-box).

2. Provide a guide for developing custom Action implementations.

Audience

This guide is targeted at developers.

Prerequisites

None.

Organization

See document index.

Documentation Conventions

The following conventions are used in this guide:

Convention	Description
Italic	In paragraph text, italic identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.
Bold	Emphasizes items of particular importance.
`Code`	Text that represents programming code.
Function | Function	A path to a function or dialog box within an interface. For example, "Select File | Open." indicates that you should select the Open function from the File menu.
( ) and |	Parentheses enclose optional items in command syntax. The vertical bar separates syntax items in a list of choices. For example, any of the following three items can be entered in this syntax: `persistPolicy (Never | OnTimer | OnUpdate | NoMoreOftenThan)`
Note: **Caution**:	A note highlights important supplemental information. A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBossESB 4.2 Milestone Release 3 documentation set:

1. **JBossESB 4.2 Milestone Release 3** *Getting Started Guide***:** Quick guide to getting started with JBoss ESB..

2. **JBossESB 4.2 Milestone Release 3** *Programmers Guide*: How to use JBossESB.

3. **JBossESB 4.2 Milestone Release 3** *Administration Guide*: How to manage the ESB.

4. **JBossESB 4.2 Milestone Release 3** *Services Guides*: Various documents related to the services available with the ESB.

5. **JBossESB 4.2 Milestone Release 3** *Trailblazer Guide*: Provides guidance for using the trailblazer example.

6. **JBossESB 4.2 Milestone Release 3** *Release Notes*: Information on the differences between this release and previous releases.

Contacting Us

Questions or comments about JBossESB 4.2 Milestone Release 3 should be directed to our support team.

Pre-Packed Actions

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

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	byte[]
Input Location	● Body.contents ● Body."*org.jboss.soa.esb.actions.current.after*"
Output Type	java.lang.String
Output Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Class	org.jboss.soa.esb.actions.converters.ByteArrayToString
Properties	● "***encoding***": The binary data encoding on the message byte array. Defaults to "UTF-8" when not specified .

Sample Config	```<action name="transform"```
	``` class="org.jboss.soa.esb.actions.converters.ByteArrayToString">```
	``` <property name="encoding" value="UTF-8" />```
	```</action>```

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	java.lang.Long/long
Input Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Output Type	java.util.Date
Output Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Class	org.jboss.soa.esb.actions.converters.LongToDateConverter
Properties	None
Sample Config	`<action name="transform` `        class="org.jboss.soa.esb.actions.converters.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
Input Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Output Type	User Object
Output Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Class	org.jboss.soa.esb.actions.converters.ObjectInvoke
Properties	● "***class-processor***": The runtime class name of the processor class used to process the message payload. ● "***class-method***": The name of the method on the processor class used to process the method.

Sample Config	`<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>`

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
Input Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Output Type	java.lang.String
Output Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Class	org.jboss.soa.esb.actions.converters.ObjectToCSVString
Properties	● "***bean-properties***": 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. ● "***fail-on-missing-property***": Flag indicating whether or not the 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 Config	<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>

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
processor.

Input Type	User Object
Input Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Output Type	java.lang.String
Output Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Class	org.jboss.soa.esb.actions.converters.ObjectToXStream
Properties	● "***class-alias***": Class alias used in call to XStream.alias(String, Class) prior to serialisation. Defaults to the input Object's class name. ● "***exclude-package***": Exclude the package name from the generated XML. Default is "true". Not applicable if a "class-alias" is specified.

| Sample Config | ```
<action name="transform"
    class="org.jboss.soa.esb.actions.converters.ObjectToXStream">

    <property name="class-alias" value="MyAlias" />
    <property name="exclude-package" value="true" />
</action>
``` |
| --- | --- |

SmooksTransformer

Performs a message transformation based on the specified message exchange.

Input Type	java.lang.String
Input Location	- Body.contents - Body."*org.jboss.soa.esb.actions.current.after*"
Output Type	java.lang.String (User Objects, where beans populated by Smooks Javabean based transforms)
Output Location	- Body."*org.jboss.soa.esb.actions.current.after*" - Body."EXTRACTED_BEANS_HASH" (for beans populated by Smooks Javabean based transforms).
Class	org.jboss.soa.esb.actions.converters.SmooksTransformer
Properties	"***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. Note: All the above properties can be overridden by supplying them as properties to the message (Message.Properties).
| Sample Config | ```
<action name="transform"
    class="org.jboss.soa.esb.actions.converters.SmooksTransformer">

    <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>
``` |

See the MessageTransformation.pdf for more details on the SmooksTransformer.

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 processor.

Input Type	java.lang.String
Input Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Output Type	User Object (specified by "incoming-type" property)
Output Location	● Body."*org.jboss.soa.esb.actions.current.after*"
Class	org.jboss.soa.esb.actions.converters.XStreamToObject
Properties	● "***class-alias***": Class alias used during serialisation. Defaults to the input Object's class name. ● "***exclude-package***": Flag indicating whether or not the XML includes a package name. ● "***incoming-type***": Class type. ● "***root-node***": Optional. Specify a different root node then the actual root node in the XML. Takes an XPath expression. ● "***aliases***": Optional. Specify additional aliases to help Xstream to convert the xml elements to Objects

Sample Config	<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>

jBPM - CommandInterpreter

Expects the argument to be a command message and tries to execute the corresponding jBPM api invocation. If Call in message header contains a replyToEpr, will send response to it.

jBPM configuration files (jbpm.cfg.xml and hibernate.cfg.xml) must be present where the Jbpm.Configuration.getInstance() expects them to be found.

At present time, the following operations are implemented :

```
deployProcessDefinition
,newProcessInstance
,signalProcess
,signalToken
,getProcessInstanceVariables
,setProcessInstanceVariables
,getTokenVariables
,setTokenVariables
,hasInstanceEnded
```

Input Type	org.jboss.soa.esb.message.Message generated by AbstractCommandVehicle.toCommandMessage()
Input Location	● full message
Output Type	Message – output of util.jbpm.CommandVehicle.toCommandMessage() containing result of jBPM api call
Output Location	● full message
Class	org.jboss.soa.esb.actions.jbpm.CommandInterpreter
Properties	●

Sample Config	`<action name="process"`
	`    class="org.jboss.soa.esb.actions.jbpm.CommandInterpreter">`
	`</action>`

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

GroovyActionProcessor

Executes a [Groovy](#) action processing script, receiving the message and action configuration as input.

Script Bindings	"***message***": The message. "***config***": The action configuration (ConfigTree).
Class	org.jboss.soa.esb.actions.scripting.GroovyActionProcessor
Properties	"***script***": Path (classpath) to Groovy script.
Sample Config	``` <action name="process" class="org.jboss.soa.esb.scripting.GroovyActionProcessor"> <property name="script" value="/scripts/ActionXProcessor.groovy"/> </action> ```

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.

Class	org.jboss.soa.esb.actions.Aggregator
Properties	"***timeoutInMillies***": Timeout time in milliseconds before the aggregation process times out.

Sample Config	`<action class="org.jboss.soa.esb.actions.Aggregator`
	`    name="Aggregator">`
	`    <property name="timeoutInMillies" value="60000"/>`
	`</action>`

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. "**_ruleLanguage_**": CBR evaluation Domain Specific Language (DSL) file. "**_ruleReload_**": Flag indicating whether or not the rules file should be reloaded each time. Default is "false". "**_destinations_**": Container property for the <route-to> configurations. `<route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/>`
"process" methods	"**_process_**": Don't append aggregation data to message. "**_split_**": Append aggregation data to message. See the Aggregator action.

Sample Config	```
	<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>
	``

See ContentBasedRouting.pdf for more details on the Content Based Routing.

StaticRouter

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

Class	org.jboss.soa.esb.actions.ContentBasedRouter
Properties	"***destinations***": Container property for the <route-to> configurations. ➢ `<route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/>`
"process" methods	"***process***": Don't append aggregation data to message. "***split***": Append aggregation data to message. See the [Aggregator](#) action.
Sample Config	```<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>```

Notifier

Send notifications to list specified in configuration. This class has a dummy process(Message) method that simply returns the argument.

Intended as example of what's needed to have your own notifier. You would typically extend this class and override notifyOk() and notifyError() methods to produce the desired output.

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	org.jboss.soa.esb.actions.Notifier
Properties	NotificationList subtree indicating targets
Sample Config	```<action class="org.jboss.soa.esb.actions.Notifier"` `okMethod="notifyOK">` ` <property name="destinations">` ` <NotificationList type="OK">` ` <target class="NotifyConsole" />` ` </NotificationList>` ` </property>` `</action>```

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[1].

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.

---

1 As of writing this section on the SOAPProcessor, JBossWS 2.0.0 was not officially released (due for release in early July). In th meantime, the JBossWS 2.0.x codebase can be downloaded an built/deployed from source. Goto JBoss Labs.

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 the Appendix.

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 just takes one property value, which is the name of the JBossWS endpoint it's exposing (invoking).

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

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 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 "paramsLocation" 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 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:

```xml
<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 is attached to the ESB Message instance in one of the following ways:

1. On the *default body location* (Message.getBody().add(Map))

2. 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 "responseLocation" 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 toolkit[1].

2. **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.

3. **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="paramsLocation" value="get-order-params" />
   <property name="responseLocation" 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 reponse 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:

---

1 We also plan to add support for unmarshaling the response using JAXB and JAXB Annotation Introductions.

```
<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="paramsLocation" value="get-order-params" />
      <property name="responseLocation" 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.

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	java.lang.String
Input Location	- Body.contents - Body."*org.jboss.soa.esb.actions.current.after*"
Class	org.jboss.soa.esb.actions.SystemPrintln
Properties	"***message***": A message prefix. "**printfull**": If true then the entire message is printed, otherwise just the byte array and attachments. "**outputstream**": if true then System.out is used, otherwise System.err.
Sample Config	```<action name="print-before" class="org.jboss.soa.esb.actions.SystemPrintln"> <property name="message" value="Message before action XXX" /> </action>```

Developing Custom Actions

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

This interface supports implementation of <u>stateless</u> 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 simple 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...
    }
}
```

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:

```xml
<action name="PrintAMessage" class="test.PrintMessage">
    <property name="message" 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:

```java
public class PrintMessage extends AbstractActionPipelineProcessor {

        private String message;

        private Integer repeatCount;

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

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

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.

```java
public class PrintMessage extends AbstractActionPipelineProcessor
    implements BeanConfiguredAction {

  private String message;

  private Integer repeatCount;

  public setMessage(String message) {
          this.message = message;
  }

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

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

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

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

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: On the "Class" element.

2. @XmlElement: On the "Field" and "Method" elements.

3. @XmlAttribute: 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>
```