JBossESB 4.3 GA

Message Action Guide

JBESB-MAG-5/20/08

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

JBoss, Home of Professional Open Source Copyright 2006, JBoss Inc., and individual contributors as indicated by the @authors tag. All rights reserved.

See the copyright.txt in the distribution for a full listing of individual contributors. This copyrighted material is made available to anyone wishing to use, modify, copy, or redistribute it subject to the terms and conditions of the GNU General Public License, v. 2.0. This program is distributed in the hope that it will be useful, but WITHOUT A WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

See the GNU General Public License for more details. You should have received a copy of the GNU General Public License, v. 2.0 along with this distribution; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Software Version

JBossESB 4.3 GA

Restricted Rights Legend

Use, duplication, or disclosure is subject to restrictions as set forth in contract subdivision (c)(1)(ii) of the Rights in Technical Data and Computer Software clause 52.227-FAR14.

© Copyright 2008 JBoss Inc.

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:	A note highlights important supplemental information.
Caution:	A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBossESB 4.3 GA documentation set:

1. **JBossESB 4.3 GA** *Getting Started Guide***:** Quick guide to getting started with JBoss ESB..
2. **JBossESB 4.3 GA** *Programmers Guide*: How to use JBossESB.
3. **JBossESB 4.3 GA** *Administration Guide*: How to manage the ESB.
4. **JBossESB 4.3 GA** *Services Guides*: Various documents related to the services available with the ESB.
5. **JBossESB 4.3 GA** *Trailblazer Guide*: Provides guidance for using the trailblazer example.
6. **JBossESB 4.3 GA** *Release Notes*: Information on the differences between this release and previous releases.

Contacting Us

Questions or comments about JBossESB 4.3 GA should be directed to our support team.

Chapter 1

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”).

Transformers & Converters

Converters/Transformers are a classification of Action Processor responsible for transforming a message payload from to another.

Note that, unless stated otherwise, all of these Actions use the **MessagePayloadProxy** for getting and setting the message payload (see the Programmers Guide).

ByteArrayToString

Takes a ***byte[]*** based message payload and converts it into a *java.lang.String* object instance.

Input Type	byte[]
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 Configuration	`<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.

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

ObjectInvoke

Takes the Object bound as the message payload and supplies it to a configured "processor" for processing. The processing result is bound back into the message as the new payload.

Input Type	User Object
Output Type	User Object
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 Configuration	`<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 as the message payload and converts it into a Comma Separated Value (CSV) String based on the supplied message object and a comma-separated "bean-properties" list property. (Also see the SmooksAction).

Input Type	User Object
Output Type	java.lang.String
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 does not support a getter method for the property. Default value is "false".
Sample Configuration	`<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>`

ObjectToXStream

Takes the Object bound as the Message payload and converts it into XML using the XStream processor. (Also see the SmooksAction).

Input Type	User Object
Output Type	java.lang.String
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. • "***aliases***": Optional. Specify additional aliases to help XStream to convert the xml elements to Objects • "***namespaces***": Optional. Specify namespaces that should be added to the xml generated by XStream. Each namespace-uri is associated with a local-part which is the element that this namespace should appear on.

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" />`
	`    <property name="aliases">`
	`         <alias name=”alias1” value=”com.acme.MyXXXClass1/>`
	`         <alias name=”alias2” value=”com.acme.MyXXXClass2/>`
	`         <alias name=”xyz” value=”com.acme.XyzValueObject”/>`
	`         <alias name=”x” value=”com.acme.XValueObject”/>`
	`         ...`
	`    </property>`
	`    <property name="namespaces">`
	`         <namespace namespace-uri=”http://www.xyz.com” local-part=”xyz”/>`
	`         <namespace namespace-uri=”http://www.xyz.com/x” local-part=”x”/>`
	`         ...`
	`    </property>`
	`</action>`

Input Type	User Object
Output Type	java.lang.String
Class	org.jboss.soa.esb.actions.converters.ObjectToXStream
Properties	• "***class-alias***": Class alias used in call to XStream.alias(String, Class) prior to serialization. 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 Configuration	`<action name="transform" class="org.jboss.soa.esb.actions.converters.ObjectToXStream"> <property name="class-alias" value="MyAlias" /> <property name="exclude-package" value="true" /> </action>`

XStreamToObject

Takes the XML bound as the Message payload and converts it into an Object using the XStream processor. (Also see the SmooksAction).

Input Type	java.lang.String
Output Type	User Object (specified by "incoming-type" property)
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 • "***attribute-aliases***": Optional. Specify additional attribute aliases to help Xstream to convert the xml attributes to Objects • "***converters***": Optional. Specify converters to help Xstream to convert the xml elements and attributes to Objects. For more information about converters see http://xstream.codehaus.org/converters.html
Sample Config	see below

```
<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>
    <property name="attribute-aliases">
        <attribute-alias name="alias1" value="com.acme.MyXXXClass1"/>
        <attribute-alias name="alias2" value="com.acme.MyXXXClass2"/>
        ...
    </property>
    <property name="converters">
        <converter class="com.acme.MyXXXConverter1"/>
        <converter class="com.acme.MyXXXConverter2"/>
        ...
    </property>
</action>
```

SmooksTransformer

NOTE: *Check out the SmooksAction for a more general purpose (and more flexible) Smooks action class. The SmooksTransformer action will be deprecated in a future release.*

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 for more details.

<table>
<tr><td>Class</td><td>org.jboss.soa.esb.actions.converters.SmooksTransformer</td></tr>
<tr><td>Properties</td><td>Smooks Resource Configuration:
"resource-config": The Smooks resource configuration file.
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.
Note: All the above properties can be overridden by supplying them as properties to the message (Message.Properties).</td></tr>
<tr><td>Sample Configuration</td><td><pre>
Default Input/Output:
<action name="transform" class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
    <property name="resource-config" value="/smooks/config-01.xml" />
</action>

Named Input/Output:
<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>

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>
</pre></td></tr>
</table>

Java Objects are bound to the Message.Body under their “beanId”. For more on this, please refer to the MessageTransformation document, or the WIKI.

SmooksAction

The SmooksAction class (*org.jboss.soa.esb.actions.smooks.SmooksAction*) is the second generation ESB action class for executing Smooks "processes" (it can do more than just transform messages – splitting etc). The SmooksTransformer action will be deprecated (and eventaully removed) in a future release of the ESB.

The SmooksAction class can process (using Smooks PayloadProcessor) a wider range of ESB Message payloads e.g. Strings, byte arrays, InputStreams, Readers, POJOs and more (see the PayloadProcessor docs). As such, it can perform a wide range of transformations including Java to Java transforms. It can also perform other types of operations on a Source messages stream, including content based payload Splitting and Routing (not ESB Message routing). The SmooksAction enables the full range of Smooks capabilities from within JBoss ESB.

The Smooks User Guide (and other documentation) is available on the Smooks website. Also, check out the Smooks Tutorials.

SmooksAction Configuration

The following illustrates the basic SmooksAction configuration:

```
<action name="transform" class="org.jboss.soa.esb.actions.smooks.SmooksAction">
    <property name="smooksConfig" value="/smooks/order-to-java.xml" />
</action>
```

The optional configuration properties are:

Name	Description	Default
get-payload-location	Message Body location containing the message payload.	Default Payload Location
set-payload-location	Message Body location where result payload is to be placed.	Default Payload Location
excludeNonSerializables	Exclude non Serializable Objects when mapping the contents of the Smooks ExecutionContext back onto the ESB Message.	true
resultType	The type of Result to be set as the result Message payload. See Specifying the Result Type for more details.	STRING
javaResultBeanId	**Note**: *Only relevant when resultType=JAVA* The Smooks bean context beanId to be mapped as the result when the resultType is "JAVA". If not specified, the whole bean context bean Map is mapped as the JAVA result.	
reportPath	The path and file name for generating a Smooks Execution Report. This is a development aid i.e. not to be used in production.	

Message Input Payload

The SmooksAction uses the ESB *MessagePayloadProxy* class for getting and setting the message payload on the ESB Message. Therefore, unless otherwise configured via the "get-payload-location" and "set-payload-location" action properties, the SmooksAction gets and sets the Message payload on the default message location (i.e. using *Message.getBody().get()* and *Message.getBody().set(Object)*).

As stated above, the SmooksAction automatically supports a wide range of Message payload types (see the PayloadProcessor). This means that the SmooksAction itself can handle most payload types without requiring "fixup" actions before it in the action chain.

XML, EDI, CSV etc Input Payloads

To process these message types using the SmooksAction, simply supply the Source message as a:

1. String,
2. InputStream,
3. Reader, or
4. byte array

Apart from that, you just need to perform the standard Smooks configurations (in the Smooks config, not the ESB config) for processing the message type in question e.g. configure a parser if it's not an XML Source (e.g. EDI, CSV etc). See the Smooks User Guide.

Java Input Payload

If the supplied Message payload is not one of type String, InputStream, Reader or byte[], the SmooksAction processes the payload as a JavaSource, allowing you to perform Java to XML, Java to Java etc transforms.

Specifying the Result Type

Because the Smooks Action can produce a number of different Result types, you need to be able to specify which type of Result you want. This effects the result that's bound back into the ESB Message payload location.

By default the ResultType is "STRING", but can also be "BYTES", "JAVA" or "NORESULT" by setting the "resultType" configuration property.

Specifying a *resultType* of "JAVA" allows you to select one or more Java Objects from the Smooks ExecutionContext (specifically, the bean context). The *javaResultBeanId* configuration property complements the *resultType* property by allowing you to specify a specific bean to be bound from the bean context to the ESB Message payload location. The following is an example that binds the “order” bean from the Smooks bean context onto the ESB Message as the Message payload.

```
<action name="transform" class="org.jboss.soa.esb.actions.smooks.SmooksAction">
    <property name="smooksConfig" value="/smooks/order-to-java.xml" />
    <property name="resultType" value="JAVA" />
    <property name="javaResultBeanId" value="order" />
</action>
```

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	• 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. • message-store-class: the implementation of the MessageStore.

Sample Configuration	`<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.DBMessageStoreImpl"/>`
	`</action>`

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

Input Type	org.jboss.soa.esb.message.Message generated by AbstractCommandVehicle.toCommandMessage()
Output Type	Message – same as the input message
Class	org.jboss.soa.esb.services.jbpm.actions.BpmProcessor
Properties	• ***command*** - required property. Needs to be one of: NewProcessInstance-Command, StartProcessInstanceCommand, SignalCommand or Cancel-ProcessInstanceCommand • ***processdefinition*** – required property for the New- and Start-ProcessInstanceCommands *if the process-definition-id 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 Signal- and CancelProcessInstance-Commands. • ***process-definition-id*** – required property for the New- and Start-ProcessInstanceCommands *if the processdefinition property is not used*. The value of this property should reference a processdefintion id in jBPM of which you want to create a new instance. This property does not apply to the Signal- and CancelProcessInstanceCommands. • ***actor*** – optional property to specify the jBPM actor id, which applies to the New- and StartProcessInstanceCommands only.

Properties	• ***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 EsbMessage. 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 New- and StartProcessInstanceCommands only.
	• ***transition-name –*** optional property. This property only applies to the StartProcessInstance- and Signal 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 processdefinition.xml.
	• ***esbToBpmVars*** - optional property for the New- and StartProcessInstanceCommands and the SignalCommand. This property defines a list of variables that need to be extracted from the EsbMessage and set into jBPM context for the particular process instance. The list consists of mapping elements. Each mapping element can have the following attributes:
	  ◦ ***esb*** – required attribute which can contain an MVEL expression to extract a value anywhere from the EsbMessage.
	  ◦ ***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 EsbMessage.

Message variables	Finally some variables can be set on the **body** of the EsbMessage: • ***jbpmProcessInstId*** – required parameter which applies to the Cancel-ProcessInstanceCommand only. It is up to the user make sure this value is set as a named parameter on the EsbMessage body. • ***jbpmTokenId* or jbpmProcessInstId –** either one is a required parameter and applies to the SignalCommand only. The SignalCommand 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 jbpmTokenId or the jbpmProcessInstId is set on the EsbMessage body.

Sample Configuration	`<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>`

Scripting

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. • "***cacheScript***": Should the script be cached. Default "true".
Sample Configuration	`<action name="process" class="org.jboss.soa.esb.scripting.GroovyActionProcessor"> <property name="script" value="/scripts/ActionXProcessor.groovy"/> </action>`

Actions defined within the ESB Services.

EJBProcessor

Takes an input Message and uses the contents to invoke a Stateless Session Bean.

Input Type	EJB method name and parameters.
Output Type	EJB specific Object.
Class	org.jboss.soa.esb.actions.EJBProcessor
Properties	• "***ejb-name***": The identity of the EJB. • "***jndi-name***": Relevant JNDI lookup. • "***initial-context-factory***": JNDI lookup mechanism. • "***provider-url***": Relevant provider. • "***method***": EJB method name to call. • "***ejb-params***": list of parameters to use when calling the method and where in the input Message they reside. • "***esb-out-var***": the location of the output (default value is DEFAULT_EJB_OUT).

Sample Configuration	`<action name="EJBTest" class="org.jboss.soa.esb.actions.EJBProcessor">`
	`      <property name="ejb-name" value="MyBean" />`
	`      <property name="jndi-name" value="ejb/MyBean" />`
	`      <property name="initial-context-factory"`
	`value="org.jnp.interfaces.NamingContextFactory" />`
	`      <property name="provider-url" value="localhost:1099" />`
	`      <property name="method" value="login" />`
	`      <!-- Optional output location, defaults to "DEFAULT_EJB_OUT"`
	`      <property name="esb-out-var" value="MY_OUT_LOCATION"/> -->`
	`      <property name="ejb-params">`
	`            <!-- arguments of the operation and where to find them in`
	`the message -->`
	`            <arg0 type="java.lang.String">username</arg0>`
	`            <arg1 type="java.lang.String">password</arg1>`
	`      </property>`
	`</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***": OPTIONAL, timeout time in milliseconds before the aggregation process times out.
Sample Configuration	`<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***”: Do not append aggregation data to the message. • “***split***”: Append aggregation data to the message. See the Aggregator action.

Sample Configuration	`<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, except it does not support content based routing rules.

Class	org.jboss.soa.esb.actions.StaticRouter
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 Configuration	`<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>`

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	• "***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. See the Aggregator action.
Sample Configuration	`<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>`

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 processsing; 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	org.jboss.soa.esb.actions.Notifier
Properties	NotificationList subtree indicating targets
Sample Configuration	<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>

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	<target class="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	• append – if value is true, append the notification to an existing file • URI – any valid URI specifying a file
Sample Configuration	<target class="NotifyFiles" > <file append="true" URI="anyValidURI"/> <file URI="anotherValidURI"/> </target>

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 <column> elements.
Attributes	• driver-class • connection-url • user-name • password • table – table in which notification record is stored • dataColumn – name of table column in which ESB message contents are stored
Child	column
Child Attributes	• name – name of table column in which to store additional value • value – value to be stored
Sample Configuration	<target class="NotifySQLTable" driver-class="com.mysql.jdbc.Driver" connection-url="jdbc:mysql://localhost/db" user-name="user" password="password" table="table" dataColumn="messageData"> <column name="aColumnlName" value="aColumnValue"/> </target>

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	• URL – a valid FTP URL • filename – the name of the file to contain the ESB message content on the remote system

Sample Configuration	<target class="NotifyFTP" > <ftp URL=”ftp://username:pwd@server.com/remote/dir” filename=”someFile.txt” /> </target>

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	• 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	• name – name of the new property to be added • value – value of the new property
Sample Configuration	<target class="NotifyQueues" > <messageProp name=”aNewProperty” value=”theValue”/> <queue jndiName=”queue/quickstarts_notifications_queue” /> </target>

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	• 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

<table>
<tr><td>Child Attributes</td><td>name – name of the new property to be added
value – value of the new property</td></tr>
<tr><td>Sample Configuration</td><td><pre><target class="NotifyTopics" >
  <messageProp name=”aNewProperty” value=”theValue”/>
  <queue jndiName=”topic/quickstarts_notifications_topic” />
</target></pre></td></tr>
</table>

<table>
<tr><td>Class</td><td>NotifyEmail</td></tr>
<tr><td>Purpose</td><td>Performs a notification by sending an email containing the ESB message content and, optionally, any file attachments.</td></tr>
<tr><td>Attributes</td><td>from – email address (javax.email.InternetAddress)
sendTo – comma-separated list of email addresses
ccTo – comma-separated list of email addresses (optional)
subject – email subject
message – a string to be prepended to the ESB message contents which make up the e-mail message (optional)
msgAttachmentName - filename of an attachment containing the message payload (optional). If not specified the message payload will be included in the message body.</td></tr>
<tr><td>Child</td><td>Attachment (optional)</td></tr>
<tr><td>Child Text</td><td>the name of the file to be attached</td></tr>
<tr><td>Sample Configuration</td><td><pre><target class="NotifyEmail"
        from=”person@somewhere.com”
        sendTo=”person@elsewhere.com”
        subject=”theSubject”>
  <attachment>attachThisFile.txt</attachment>
</target></pre></td></tr>
</table>

<table>
<tr><td>Class</td><td>NotifyFTP</td></tr>
<tr><td>Purpose</td><td>Performs a notification by creating a file containing the ESB message content and transferring it via FTP to a remote file system.</td></tr>
<tr><td>Attributes</td><td>none</td></tr>
<tr><td>Child</td><td>ftp</td></tr>
<tr><td>Child Attributes</td><td>URL – a valid FTP URL
filename – the name of the file to contain the ESB message content on the remote system</td></tr>
<tr><td>Sample Configuration</td><td><pre><target class="NotifyFTP" >
  <ftp URL=”ftp://username:pwd@server.com/remote/dir”
       filename=”someFile.txt” />
</target></pre></td></tr>
</table>

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 4.2.2.GA.

2. 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 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 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 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 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 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.

To see the SOAP message template as it's being constructed and populated, add the "**dumpSOAP**" parameter to the parameter Map. This can be a very useful developer aid, but should not be left on outside of development.

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 in 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 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="get-payload-location" value="get-order-params" />
```

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

```
   <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.

We also provide, in addition to the above XStream configuration options, the ability to specify field name mappings and XStream annotated classes.

```
       <property name="responseXStreamConfig">
           <fieldAlias name="header" class="com.acme.order.Order"
               fieldName="headerFieldName" />
           <annotation class="com.acme.order.Order" />
       </property>
```

Field mappings can be used to map XML elements onto Java fields on those occasions when the local name of the element does not correspond to the field name in the Java class.

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.

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	java.lang.String
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 Configuration	`<action name="print-before" class="org.jboss.soa.esb.actions.SystemPrintln">`
	`    <property name="message" value="Message before action XXX" />`
	`</action>`

Chapter 2

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 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:

```
<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.

Chapter 3

Appendix

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>
```