

# **JBoss Enterprise SOA Platform 5.0 Programmers Guide**

**A guide for developers using the JBoss Enterprise SOA Platform**



# **JBoss Enterprise SOA Platform 5.0 Programmers Guide**

## **A guide for developers using the JBoss Enterprise SOA Platform Edition 3.0**

Copyright © 2009 Red Hat, Inc.. This material may only be distributed subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version of the OPL is presently available at <http://www.opencontent.org/openpub/>).

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

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

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

The guide contains information for programmers developing with the JBoss Enterprise SOA Platform.

---

---

<b>Preface</b>	<b>vii</b>
1. Document Conventions .....	vii
1.1. Typographic Conventions .....	vii
1.2. Pull-quote Conventions .....	viii
1.3. Notes and Warnings .....	ix
2. We Need Feedback! .....	x
<b>1. The Enterprise Service Bus</b>	<b>1</b>
1.1. What is an Enterprise Service Bus? .....	1
1.2. When Would an ESB be Used? .....	1
<b>2. The JBoss ESB</b>	<b>5</b>
2.1. Rosetta .....	5
2.2. The JBoss ESB Core Summarized .....	6
<b>3. Updated. Services and Messages</b>	<b>9</b>
3.1. Updated. The Service .....	9
3.2. The Message .....	11
3.3. Obtaining and Setting Data .....	18
3.4. Extensions to the Body .....	18
3.5. The Message Header .....	19
3.6. LogicalEPR .....	21
3.7. The Message Payload .....	22
3.8. The MessageFactory .....	24
3.9. Message Formats .....	25
3.9.1. MessageType.JAVA_SERIALIZED .....	25
3.9.2. MessageType.JBOSS_XML .....	25
<b>4. Building and Using Services</b>	<b>27</b>
4.1. updated Listeners, Routers/Notifiers and Actions .....	27
4.1.1. Listeners .....	27
4.1.2. updated Routers .....	27
4.1.3. Notifiers .....	27
4.1.4. Actions and Messages .....	31
4.1.5. Handling Responses .....	32
4.1.6. Error Handling When Actions are Being Processed .....	33
4.2. Meta-Data and Filters .....	33
4.3. Updated What is a Service? .....	35
4.3.1. The "ServiceInvoker" .....	36
4.3.2. Updated. Transactions .....	37
4.3.3. Services and the ServiceInvoker .....	38
4.3.4. Updated. The InVM Transport .....	38
4.4. Updated. Service Contract Definition .....	42
<b>5. Other Components</b>	<b>45</b>
5.1. Updated. The Message Store .....	45
5.2. Data Transformation .....	45
5.3. Content-Based Routing .....	45
5.4. The Registry .....	45
<b>6. An Example</b>	<b>47</b>
6.1. How to Use the Message .....	47
6.1.1. The Message Structure .....	47
6.1.2. The Service .....	48

6.1.3. Unpacking the Payload .....	49
6.1.4. The Client .....	50
6.1.5. Updated. Configuring for a Remote Service Invoker .....	51
6.1.6. Sample Client .....	52
6.1.7. Hints and Tips .....	53
<b>7. Advanced Topics</b> .....	<b>55</b>
7.1. Fail-Over and Load-Balancing Support .....	55
7.1.1. Services, EPRs, Listeners and Actions .....	55
7.1.2. Replicated Services .....	56
7.1.3. Protocol Clustering .....	60
7.1.4. Clustering .....	63
7.1.5. Channel Fail-over and Load Balancing .....	63
7.1.6. Updated Message Redelivery .....	65
7.2. Scheduling of Services .....	67
7.2.1. Simple Schedule .....	67
7.2.2. Cron Schedule .....	68
7.2.3. Scheduled Listener .....	68
7.2.4. Example Configurations .....	69
7.2.5. Updated. Quartz Scheduler Property Configuration .....	70
<b>8. Fault-Tolerance and Reliability</b> .....	<b>71</b>
8.1. Failure classification .....	71
8.1.1. JBossESB and the Fault Models .....	72
8.1.2. Failure Detectors and Failure Suspectors .....	73
8.2. Reliability Guarantees .....	74
8.2.1. Message Loss .....	75
8.2.2. Suspecting Endpoint Failures .....	76
8.2.3. Supported Crash Failure Modes .....	76
8.2.4. Component Specifics .....	76
8.2.5. Gateways .....	76
8.2.6. ServiceInvoker .....	76
8.2.7. JMS Broker .....	76
8.2.8. Action Pipelining .....	76
8.3. Recommendations .....	77
<b>9. Defining Service Configurations</b> .....	<b>79</b>
9.1. Overview .....	79
9.2. Providers .....	80
9.3. Services .....	82
9.4. Transport Specific Type Implementations .....	85
9.5. FTP Provider Configuration .....	87
9.6. FTP Listener Configuration .....	89
9.6.1. Read-Only FTP Listener .....	89
9.7. Updated. UDP Gateway .....	91
9.8. Updated JBoss Remoting (JBR) Configuration .....	93
9.9. Transitioning from the Old Configuration Model .....	93
9.10. Configuration .....	94
<b>10. Web Services Support</b> .....	<b>97</b>
10.1. JBossWS .....	97
<b>11. Out-of-the-box Actions</b> .....	<b>99</b>
11.1. Updated Out-of-the-Box Actions .....	99

---

11.1.1. Updated. Transformers & Converters .....	99
11.2. Updated. Business Process Management .....	108
11.3. Updated. Scripting .....	110
11.4. Services .....	111
11.4.1. EJBProcessor .....	111
11.5. Updated. Routing .....	113
11.6. Notifier .....	120
11.6.1. Webservices/SOA-P .....	122
11.6.2. Updated. HttpClient Configuration .....	130
11.6.3. Updated. SOAPProxy .....	131
11.6.4. Miscellaneous .....	132
<b>12. Developing Custom Actions</b> .....	<b>135</b>
12.1. Configuring Actions Using Properties .....	135
<b>13. Connectors and Adapters</b> .....	<b>139</b>
13.1. Introduction .....	139
13.2. The Gateway .....	139
13.2.1. This section has changed. Gateway Data Mappings .....	140
13.2.2. How to change the Gateway Data Mappings .....	141
13.3. Connecting via JCA .....	141
13.3.1. Configuration .....	142
13.3.2. Updated. Mapping Standard Activation Properties .....	144
<b>A. Writing JAXB Annotation Introduction Configurations</b> .....	<b>147</b>
<b>B. Service Orientated Architecture Overview</b> .....	<b>149</b>
B.1. Why SOA? .....	150
B.2. Basics of SOA .....	152
B.3. Advantages of SOA .....	152
B.3.1. Interoperability .....	152
B.3.2. Efficiency .....	153
B.3.3. Standardization .....	153
B.3.4. Stateful and Stateless Services .....	153
B.4. JBossESB and its Relationship with SOA .....	155
<b>C. Revision History</b> .....	<b>157</b>

---

---

# Preface

## 1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*<sup>1</sup> set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

### 1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

#### **Mono-spaced Bold**

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

To see the contents of the file **my\_next\_bestselling\_novel** in your current working directory, enter the **cat my\_next\_bestselling\_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F1** to switch to the first virtual terminal. Press **Ctrl+Alt+F7** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **Mono-spaced Bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

#### **Proportional Bold**

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

---

<sup>1</sup> <https://fedorahosted.org/liberation-fonts/>

Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

### ***Mono-spaced Bold Italic*** or ***Proportional Bold Italic***

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh *john@example.com***.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount */home***.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — *username*, *domain.name*, *file-system*, *package*, *version* and *release*. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules (MPMs)*. Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

## 1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.



Output sent to a terminal is set in Mono-spaced Roman and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in Mono-spaced Roman but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome        home   = (EchoHome) ref;
        Echo            echo   = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}
```

### 1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



#### Note

A note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



#### Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.



## Warning

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

## 2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: <http://bugzilla.redhat.com/bugzilla/> against the product **JBoss Enterprise SOA Platform**.

When submitting a bug report, be sure to mention the manual's identifier:  
*SOA\_ESB\_Programmers'\_Guide*

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

# The Enterprise Service Bus

## 1.1. What is an Enterprise Service Bus?

An *Enterprise Service Bus* (ESB) is regarded by many as the next generation of *Enterprise Application Integration* (EAI) technology. A good Enterprise Service Bus will offer capabilities that mirror those of existing EAI solutions but will not lock you into the offerings of one vendor.

A traditional EAI stack consists of the following:

- Business Process Monitoring
- Integrated Development Environment
- Human Work-flow User Interface
- Business Process Management
- Connectors
- Transaction Manager
- Security
- Application Container
- Messaging Service
- Meta-data Repository
- Naming and Directory Service
- Distributed Computing Architecture

As was the case with the older EAI systems, the Enterprise Service Bus does not deal with business logic; that is left to higher level programs. Rather, it deals with infrastructure logic. Although there are many different definitions of what constitutes an ESB, everyone agrees that they are a fundamental part of any *Service-Oriented Architecture* (SOA) Platform. However, a SOA is not simply a technology or a product: rather, it is a style of design, many aspects of which (such as the architecture, methodology and organisation) are unrelated to the actual technology. However, obviously at some point in time, it becomes necessary to map the abstract SOA concepts onto a concrete implementation and that is where the ESB "comes into play."

One can learn more about both the principles underlying a SOA and the architecture of an ESB architectures in [Appendix B, Service Orientated Architecture Overview](#) .

## 1.2. When Would an ESB be Used?

The figures below depict some examples of situations in which the **JBoss Enterprise Service Bus** would be of use. Although these examples are specific to interactions between participants using non-inter-operable *Java Message Service* (JMS) implementations, the principles in themselves are general and can be applied to other transports, such as *File Transfer Protocol* (FTP) and *Hypertext Transfer Protocol* (HTTP.)

This first diagram shows a simple movement of files between two systems in a situation where there is no messaging queuing:

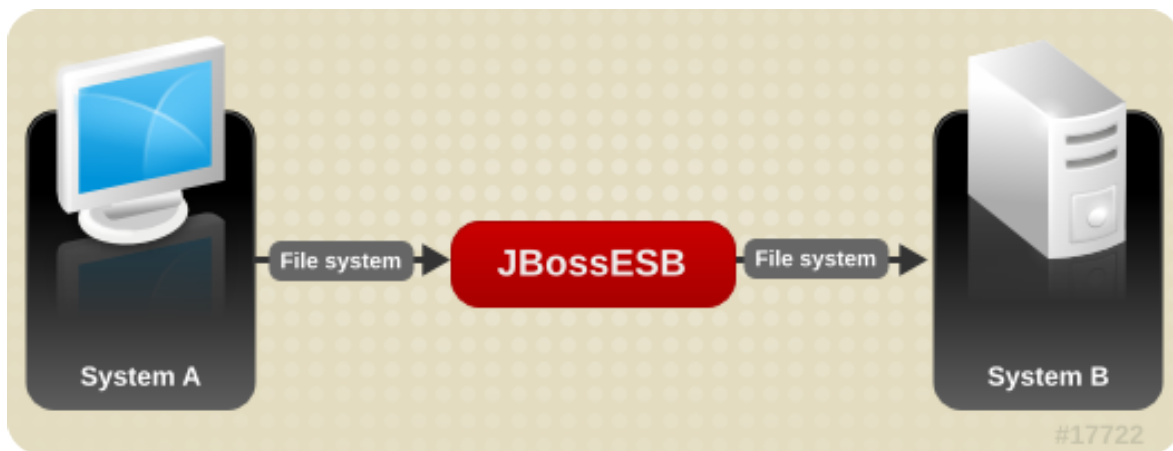


Figure 1.1. Simple File Movement Between Two Systems Without Messaging Queuing

The next diagram illustrates how a *transformation* process can be inserted into the same scenario via use of the JBoss ESB:

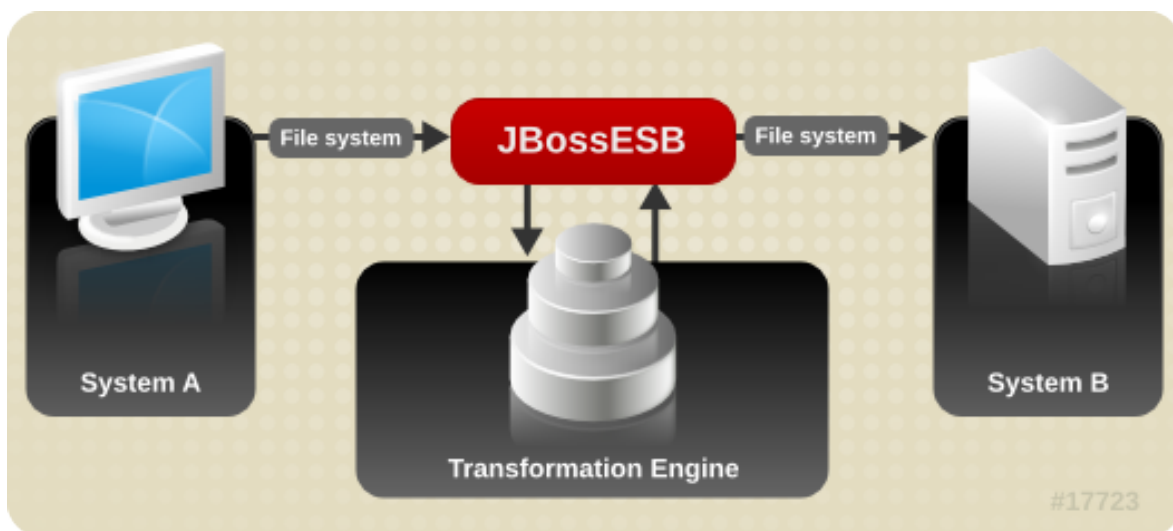


Figure 1.2. Simple File Movement with Transformation Between Two Systems Without Messaging Queuing:

In the next series of examples, a queuing system (such as a Java Message Service) is used.

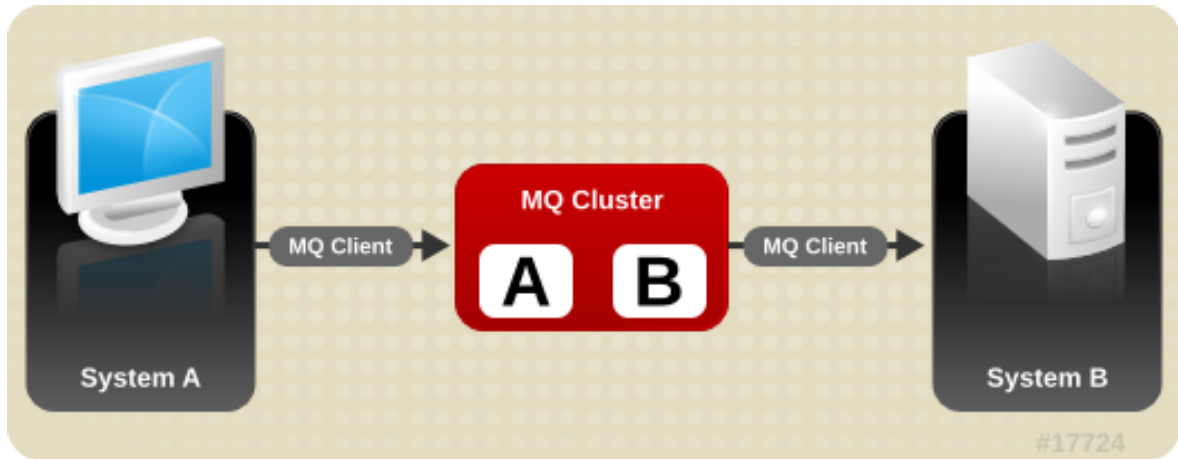


Figure 1.3. Using Messaging Queuing:

The diagram below shows transformation and queuing both being used within the same situation:

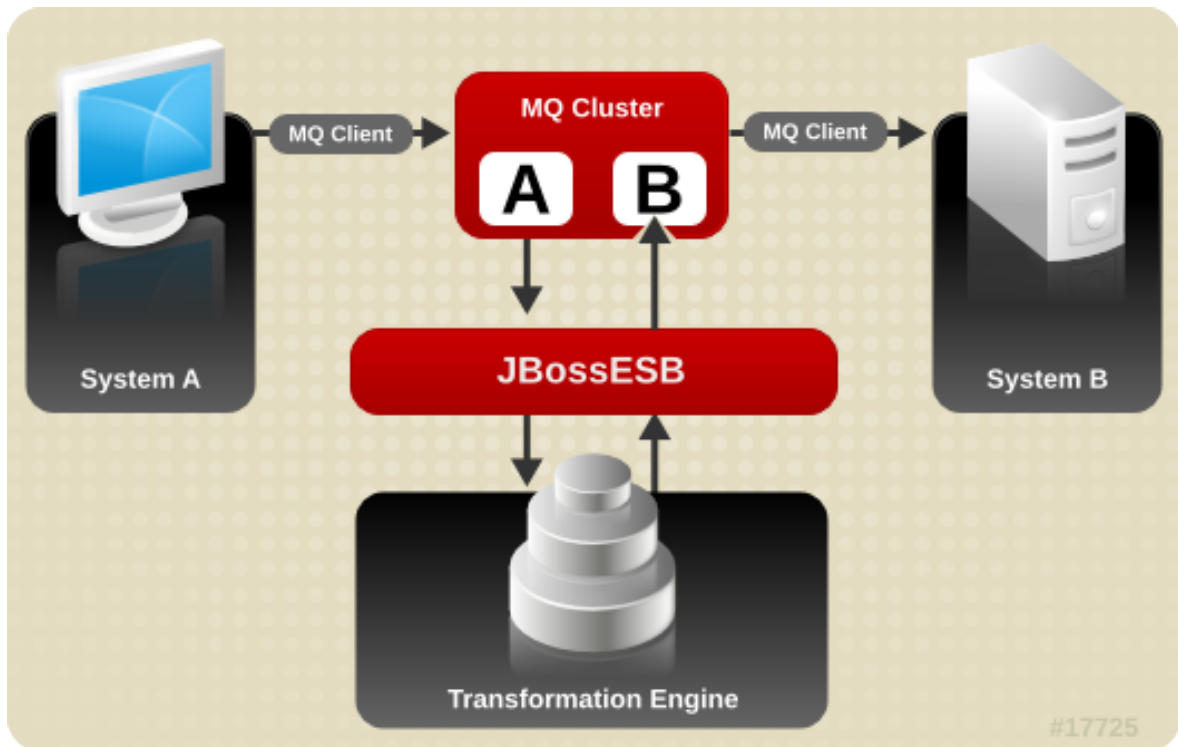


Figure 1.4. Using Messaging Queuing with Transformation

The JBoss Enterprise Service Bus can be used in more scenarios than just those involving multiple parties. For example, the diagram below shows basic data transformation undertaken via the ESB using the file system.

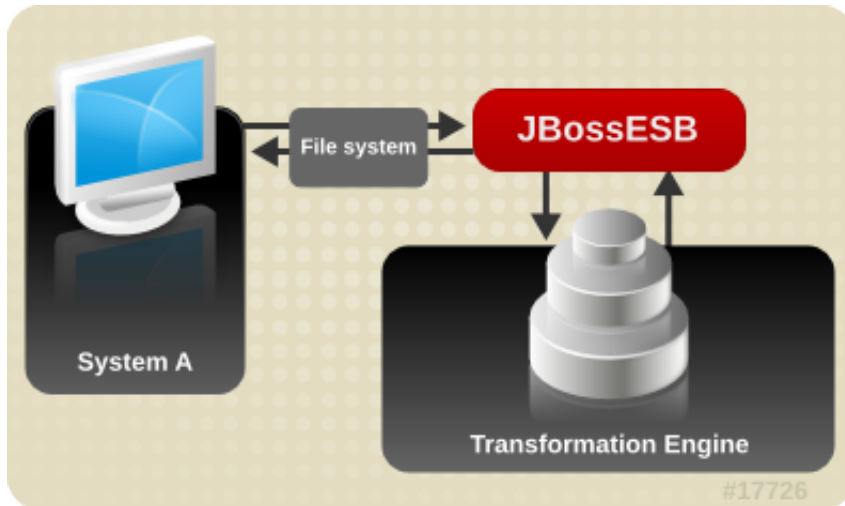


Figure 1.5. Basic Data Transformation via the ESB Using the File System

The final example is, again, a single party scenario, featuring both transformation and a queuing system.

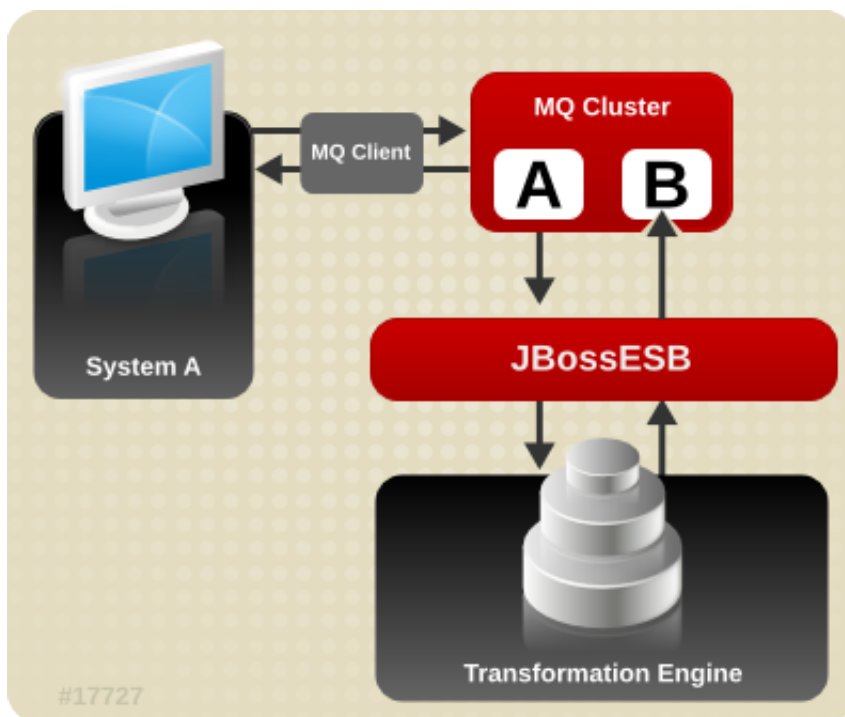


Figure 1.6. Single Party Example Using Transformation and a Queuing System:

In the following chapters, one will learn more about the core concepts behind the JBoss Enterprise Service Bus and come to an understanding of how they can be used to develop SOA-based applications.

# The JBoss ESB

## 2.1. Rosetta

At the core of the JBoss Enterprise SOA Platform is *Rosetta*, an Enterprise Service Bus that has been in commercial deployment at mission critical sites for over four years. These deployments have included highly heterogeneous environments. One such site included an IBM mainframe running z/OS, DB2 and Oracle databases, Windows and Linux servers and a variety of third-party applications, as well as other services that were outside of the corporation's information technology infrastructure.

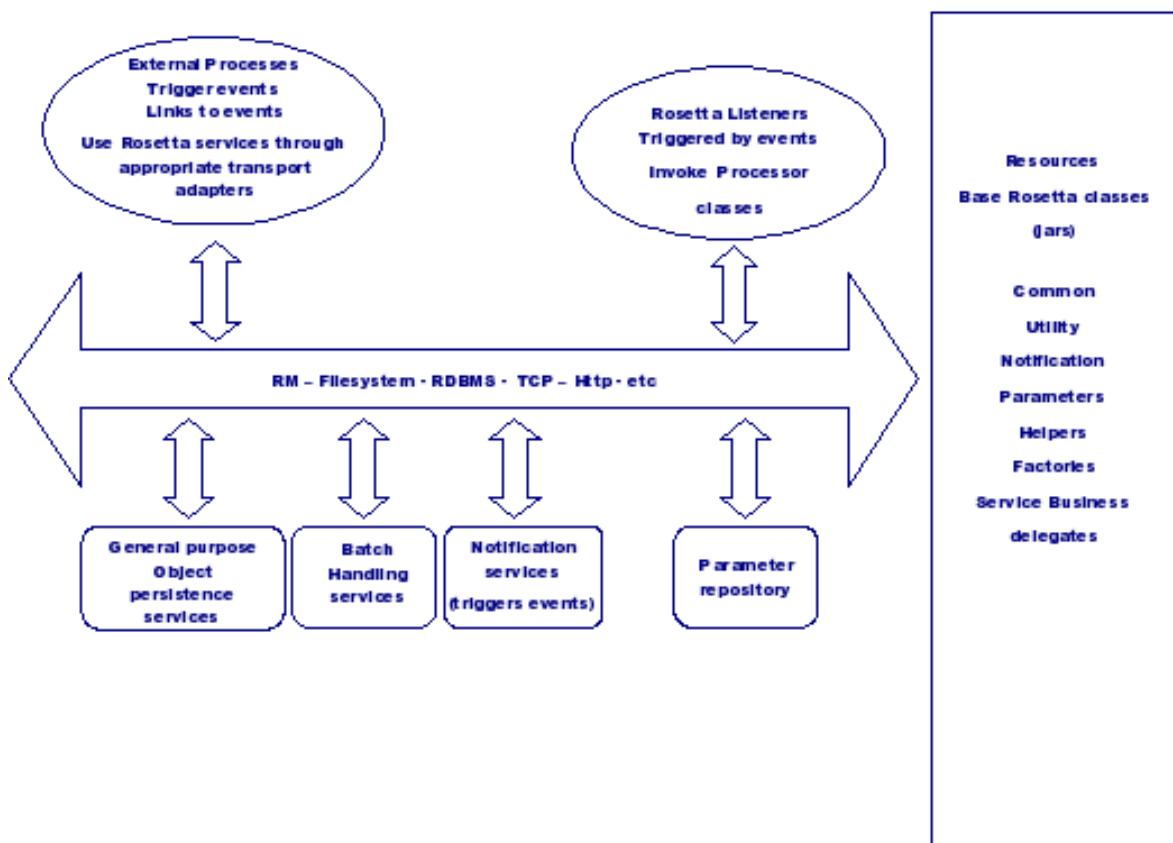


Figure 2.1. The Rosetta Architecture

In the above diagram, the term "processor classes" refers to the *action classes* within the core of Rosetta that are responsible for processing when events are triggered.

There are many reasons why one may desire to have one's disparate applications, services and components inter-operate. The most common reason is in order to leverage legacy systems in new deployments. Such interactions between these entities may occur either *synchronously* or *asynchronously*.

Rosetta was developed to not only facilitate such deployments but also to provide an infrastructure and set of tools that met the following objectives:

- To be configured easily to work with a wide variety of transport mechanisms such as e.-mail and Java Message Service.
- To offer a general-purpose object repository.

- To provide interchangeable data transformation mechanisms.
- To support the logging of those interactions, (including both business and processing events,) that flow through the framework.
- To make it simple to isolate the business logic from the transport and triggering mechanisms.
- To provide flexible plug-ins for business logic and data transformations.
- To provide a simple way for future users to replace and extend the framework's standard base classes.
- To provide for triggering of customised 'action classes' that may be unaware of the transport and triggering mechanisms.



### Important

There are two trees within the JBoss ESB source: `org.jboss.internal.soa.esb` and `org.jboss.soa.esb`. One should limit one's use of anything within the `org.jboss.internal.soa.esb` package, because its contents are subject to change without notice. However, `org.jboss.soa.esb` is covered by Red Hat's deprecation policy.

## 2.2. The JBoss ESB Core Summarized

Rosetta is built upon four core architectural components:

- *Message Listener* and *Message Filter* code

Message Listeners act as routers that listen for "inbound" messages (on either a Java Message Service Queue or Topic or on the file system.) They present the messages to a processing pipeline. This pipeline then filters each message and then uses the "outbound" router to send them to another end-point.

- Data transformation using the **SmooksAction** *action processor*.<sup>1</sup>
- A content-based routing service.<sup>2</sup>
- A Message Repository for saving messages and events that have exchanged within the Enterprise Service Bus.<sup>3</sup>

These capabilities are offered through a set of business classes, adapters and processors, which are described in detail later in this book. A range of different approaches are used to provide interaction between clients and services. These approaches include the Java Message Service, flat-files and e.-mail.

An example of a JBoss SOA-P deployment is depicted below. This diagram shall be discussed further in subsequent sections of the book.



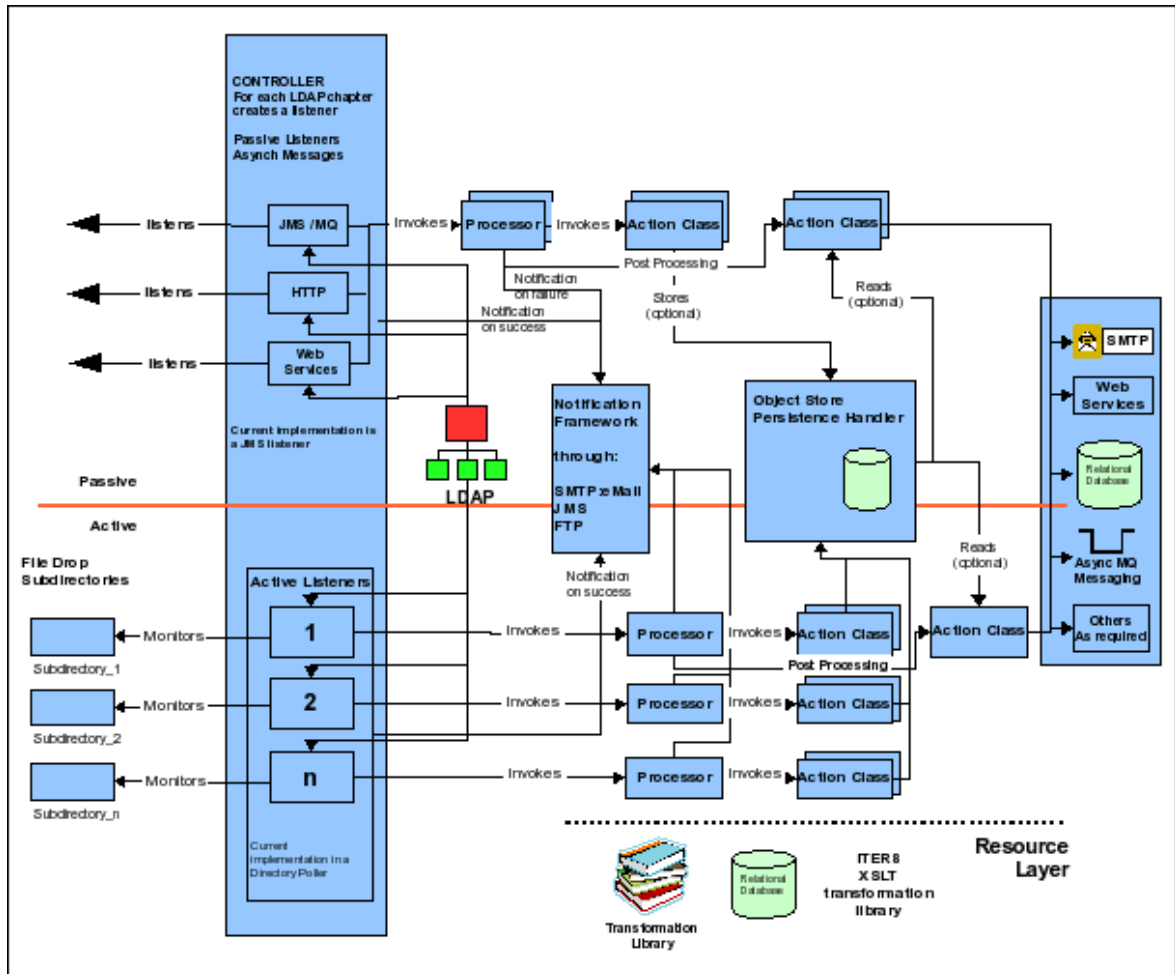


Figure 2.2. Example of a JBoss SOA-P deployment

**Important**

Some of the components in *Figure 2.2, "Example of a JBoss SOA-P deployment"* such as the LDAP server are optional and may, therefore, not be provided "out-of-the-box." Furthermore, the distinction between a Processor and an Action displayed in the above diagram is merely an illustrative convenience intended to show the concepts involved when an incoming event (that is, a message) triggers the underlying Enterprise Service Bus to invoke higher-level services.

In the following chapters, one will learn about the various components of which JBoss SOA-P consists, and how these same components interact and can be exploited to develop service-orientated applications.



## Updated. Services and Messages

In keeping with Service-Oriented Architecture principles, one is to consider everything within the JBoss ESB to be either a *service* or a *message*.

Services encapsulate either the business logic or the points of integration with legacy systems.

Messages provide the way in which clients and services communicate with each other.

In the following sections, one will learn how services and messages are supported.

### 3.1. Updated. The Service

In the JBoss Enterprise Service Bus, a *service* is defined as "a list of *action classes* that process a Message in a sequential manner."

This list of action classes to which the definition refers is known as an *action pipeline*.

A Service can also define a list of *listeners*. Listeners act like inbound routers for the Service, in that they route messages to the Action Pipeline.

The following is a very simple configuration that defines a single service which simply prints the contents of the message to the console:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product
/etc/schemas/xml/jbossesb-1.0.1.xsd" invmScope="GLOBAL">

<services>
  <service category="Retail" name="ShoeStore"
    description="Acme Shoe Store Service">
    <actions>
      <action name="println"
        class="org.jboss.soa.esb.actions.SystemPrintln" />
    </actions>
  </service>
</services>
</jbossesb>
```

**Example 3.1.** Simple Example Service that Prints Contents of Message to Console.

A service has "category" and "name" attributes. When it is deployed by Enterprise Service Bus, it uses these attributes to register its listeners as *endpoints* in the *Service Registry*. Clients can invoke a service using the class called **ServiceInvoker**.

```
ServiceInvoker invoker = new ServiceInvoker("Retail", "ShoeStore");
Message message = MessageFactory.getInstance().getMessage();

message.getBody().add("Hi there!");
invoker.deliverAsync(message);
```

**Example 3.2.** Invoking the Service from the Client

In this example, the **ServiceInvoker** uses the Services Registry <sup>1</sup> to look up the available endpoint addresses for the service entitled "Retail:ShoeStore." It takes care of all the details of sending the message from the client to one of the available service endpoints. To the client, the message transport process is completely transparent.

Which endpoint addresses are made available to the ServiceInvoker will depend on the list of listeners configured on the service. These could, for example, be JMS, FTP or HTTP. In the above example, no listeners are configured, but the service's *InVM listener* has been enabled by the use of `invmScope="GLOBAL"`.

The *InVM transport* is an Enterprise Service Bus feature that provides communication between services running on the same *Java Virtual Machine* (JVM.) [Section 4.3.4, "Updated. The InVM Transport"](#) contains more information about this functionality.

One must explicitly add listener configurations to a service in order to enable additional endpoints.

The JBoss Enterprise Service Bus supports two forms of listener configuration. These are:

- *Gateway Listeners*

These listener configurations provide *gateway endpoints*. This type of endpoint provides a point of entry for messages coming from outside the ESB deployment. They also have the responsibility for "normalizing" the message payload by "wrapping" it in an *ESB Message* before shipping it to the service's action pipeline.

- *ESB-Aware Listeners*

These listener configurations provide *ESB-Aware Endpoints*. This type of endpoint types is used to exchange ESB Messages between ESB-Aware components. They can, for example, be used to exchange messages on the Bus.



### Note

An "ESB Message" is an implementation of the `org.jboss.soa.esb.message.Message`. A component is considered "ESB-Aware" if it can deal with ESB Messages.

The service's endpoints are set in the same configuration file as its other details. The *transport level* details are defined by adding a `<providers>` section to the file. A reference to the provider is then added as a `<listener>`.

In the following example, a `<jms-provider>` section has been added. It defines a single `<jms-bus>` for the "Shoe Store JMS Queue." This is then referenced in the `<jms-listener>` defined on the "Shoe Store Service."

---

<sup>1</sup> This information is in the *JBoss Enterprise SOA Platform Services Guide*, which is provided as the file `Services_Guide.pdf` or can be viewed online at [http://www.redhat.com/docs/en-US/JBoss\\_SOA\\_Platform/](http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/)

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product
/etc/schemas/xml/jbossesb-1.0.1.xsd" invmScope="GLOBAL">

  <providers>
    <jms-provider name="JBossMQ" connection-factory="ConnectionFactory">
      <jms-bus busid="shoeStoreJMSGateway">
        <jms-message-filter dest-type="QUEUE"
          dest-name="queue/shoeStoreJMSGateway"/>
      </jms-bus>
    </jms-provider>
  </providers>

  <services>

    <service category="Retail" name="ShoeStore" invmScope="GLOBAL"
      description="Acme Shoe Store Service">

      <listeners>
        <jms-listener name="shoeStoreJMSGateway"
          busidref="shoeStoreJMSGateway" is-gateway="true"/>
      </listeners>

      <actions>
        <action name="println"
          class="org.jboss.soa.esb.actions.SystemPrintln" />
      </actions>

    </service>

  </services>

</jbossesb>
```

**Example 3.3. A JMS Gateway Listener is Added to the Shoe Store Service Example from Above**

The Shoe Store Service can now be accessed by using either one of two endpoints, namely the InVM Endpoint or the new JMS Gateway Endpoint. For performance reasons, the **ServiceInvoker** will always try to use a service's local InVM endpoint, in preference to other types, provided that it is available.

## 3.2. The Message

All of the interactions between clients and services within the Enterprise Service Bus occur via the exchange of messages. Therefore, development using a *message-exchange pattern* is recommended, as this will encourage loose coupling. Requests and responses should be independent messages, correlated where necessary by the infrastructure or the application. Programs constructed in this way will be more tolerant of failure and give developers more flexibility to select their deployment and message delivery requirements.

One is recommended to follow these guidelines in order to ensure that services are loosely coupled and that robust SOA-P applications result:

1. Use one-way message exchanges rather than a request-response architecture.
2. Keep the contract definition within the exchanged messages. Avoid defining a service interface that exposes one's back-end implementation choices, as this will make it very difficult to change the implementation at a later date.
3. Use an extensible message structure for the message payload so that changes to it can be versioned over time for the purpose of backward-compatibility.
4. Do not develop excessively fine-grained services as these often lead to extremely complex applications that cannot be easily adapted to environmental changes. SOA-P's paradigm is one of services, not distributed objects.

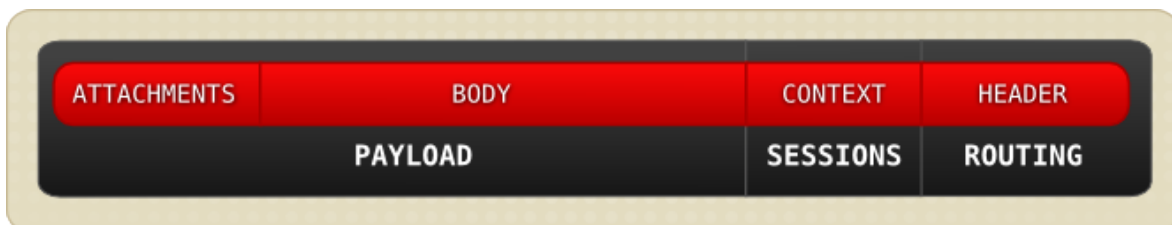
It is a requirement that information be encoded within the message itself as to where responses should be sent when one is using a one-way message delivery pattern with requests and responses. This information may be present in the message body (the payload) and dealt with by the application. Alternatively, it may part of the initial request message and, therefore, be dealt with by the ESB infrastructure.

Central to the Enterprise Service Bus is the notion of a message whose structure is similar to that found in SOA-P:

```
<xs:complexType name="Envelope">
  <xs:attribute ref="Header" use="required"/>
  <xs:attribute ref="Context" use="required"/>
  <xs:attribute ref="Body" use="required"/>
  <xs:attribute ref="Attachment" use="optional"/>
  <xs:attribute ref="Properties" use="optional"/>
  <xs:attribute ref="Fault" use="optional"/>
</xs:complexType>
```

**Example 3.4. Sample ESB Message Schema**

Pictorially, the basic structure of the message can be represented in the form shown below. (In the rest of this section, each of the components shown in this illustration shall be examined in more detail.)



**Figure 3.1. Basic Structure of a Message**

The message structure can also be represented in Unified Modeling Language (UML):

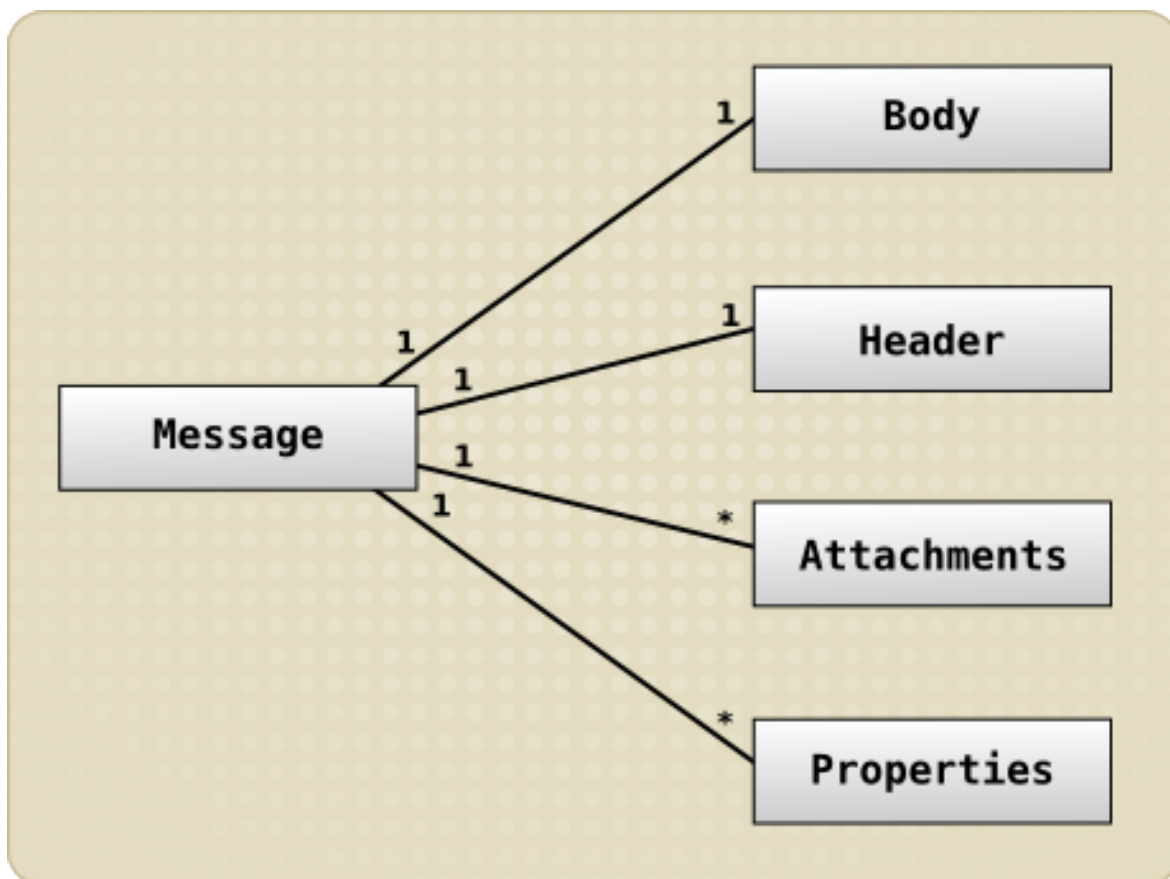


Figure 3.2. The Message Structure Represented as UML

Each message is an implementation of the `org.jboss.soa.esb.message.Message` interface. (This package contains the interfaces for the various fields within the Message.)

```
public interface Message
{
    public Header getHeader ();
    public Context getContext ();
    public Body getBody ();
    public Fault getFault ();
    public Attachment getAttachment ();
    public URI getType ();
    public Properties getProperties ();
}
```

Example 3.5. The `org.jboss.soa.esb.message.Message` Interface



### Warning

At the current time, developers should not use either Properties or Attachments.

The general concepts that they embody are under re-evaluation and may change significantly in future releases.

Red Hat recommend that you include Property and Attachment data in the Message Body instead.

The *Header* contains routing and addressing information for the message in the form of *End Point References* (EPRs). The Header also hold information that is used to uniquely identify the message. The JBoss Enterprise Service Bus uses an addressing scheme based on the WS-Addressing standard created by the W3C. The `org.jboss.soa.esb.addressing.Call` class will be discussed in the next section.

```
public interface Header
{
    public Call getCall ();
    public void setCall (Call call);
}
```

### Example 3.6. The Public Interface Header

The *Context* contains session-related information. This might include transaction or security contexts. Version 5.0 of the JBoss Enterprise Service Bus is the first to support *user-enhanced contexts*.

The *Body* will typically contain the so-called "payload" of the message. One can use the Body to send an arbitrary number of differing data types. One is not restricted within the Body to just sending and receiving single data items. How these objects are serialized when they go "to" and "from" the message body is decided by the specific object-type.



### Note

One should be extremely careful about sending serialized objects within the Body. This is because not everything that can be Serialized will necessarily be meaningful at the receiving end, an example of this being database connections.



```
public interface Body
{
    public static final String DEFAULT_LOCATION
        = "org.jboss.soa.esb.message.defaultEntry";

    public void add (String name, Object value);
    public Object get (String name);
    public void add (Object value);
    public Object get ();
    public Object remove (String name);
    public void replace (Body b);
    public void merge (Body b);
    public String[] getNames ();
}
```

**Example 3.7. The `org.jboss.soa.esb.message.Body` Interface**

The `Body` can be used to convey arbitrary information numbers of each type. In other words, it is not necessary for one to restrict oneself to just sending and receiving single data items within a `Body`.



### Important

The *byte array* component of the `Body` was deprecated in JBossESB 4.2.1. If you wish to continue using a byte array in conjunction with other data stored in the `Body`, then simply use `add` with a unique name. If your clients and services want to agree to a location for a byte array, then you can use the one that the JBoss ESB itself uses: `ByteBody.BYTES_LOCATION`.



### Note

The default, named Object (`DEFAULT_LOCATION`) should be used with care, so that multiple services or Actions do not inadvertently overwrite each other's data.

The *Fault* is used to convey error information. This information is represented within the `Body`.

```
public interface Fault
{
    public URI getCode ();
    public void setCode (URI code);

    public String getReason ();
    public void setReason (String reason);

    public Throwable getCause ();
    public void setCause (Throwable ex);
}
```

Example 3.8. The **org.jboss.soa.esb.message.Fault** interface



### Warning

At the current time, developers should not use either Properties or Attachments.

The general concepts that they embody are under re-evaluation and may change significantly in future releases.

Red Hat recommend that you include Property and Attachment data in the Message Body instead.

Message properties are used to define additional meta-data. This meta-data is for the message. Here is a set of such properties:

```
public interface Properties
{
    public Object getProperty(String name);
    public Object getProperty(String name, Object defaultVal);

    public Object setProperty(String name, Object value);
    public Object remove(String name);

    public int size();
    public String[] getNames();
}
```

Example 3.9. The **org.jboss.soa.esb.message.Properties** Interface



### Note

Those Properties that use the *java.util.Properties* have not been implemented within the JBoss Enterprise Service Bus. That is because they would place restrictions on the types of clients and services that could be used. Web Services stacks also do not implement them for the same reason. If one does need to send *java.util.Properties*, then they can be embedded within the current abstraction.

Messages can contain attachments that do not appear in the main body. Such attachments may include images, drawings, binary document formats and compressed files. The Attachment interface supports both named and unnamed attachments. However, note that, in the current release of the JBoss ESB, only Java Serialized objects may be attachments. This restriction will be removed in a subsequent release.

```
public interface Attachment
{
    Object get(String name);
    Object put(String name, Object value);

    Object remove(String name);

    String[] getNames();

    Object itemAt (int index) throws IndexOutOfBoundsException;
    Object removeItemAt (int index) throws IndexOutOfBoundsException
    Object replaceItemAt(int index, Object value)
    throws IndexOutOfBoundsException;

    void addItem (Object value);
    void addItemAt (int index, Object value)
    throws IndexOutOfBoundsException;

    public int getNamedCount();
}
```

**Example 3.10. The `org.jboss.soa.esb.message.Attachment` Interface**

Attachments may be used for a number of reasons. Generally, they are employed to provide the message with a more logical structure. The performance of large messages can also be improved by allowing the attachments to be streamed between end-points.



### Note

The JBoss ESB does not allow one to specify other encoding mechanisms for either the message or the streaming of attachments. This feature will be added in a later release and, where appropriate, it will be tied to the *SOAP-with-attachments* delivery mechanism. Currently, attachments are treated in the same way as named objects within the Body.

Users may find themselves overwhelmed when they find that they have to choose between attachments, properties and named objects when deciding where to put the payload. However, the choice is really quite straightforward:

- A service developer defines the contract that the clients will use in order to interact with his or her service. As part of that contract, one will specify both functional and non-functional aspects of the service; for example, that it is an airline reservation service (functional) and that it is transactional in nature (non-functional.)

The developer will also define the operations (messages) that the service can understand. The format (such as Java Serialized Message or XML) is also defined as part of the message definition

(In the example case, they will be the transaction context, seat number, customer name and so forth.) When one defines the content, one can specify whereabouts in the Message the service will expect to find the payload. (This can be in the form of attachments or specific, named objects or even the default named object, if one so wishes.) It is entirely up to the service developer to determine. The only restriction is that objects and attachments must have a globally-unique name (otherwise one service (or action) may inadvertently pick up a partial payload meant for another, if it is the case that the same Message Body is forwarded along on multiple "hops.")

- As a service user, one can obtain the contract definition about the service (for example, through either UDDI or an out-of-band communication) and this will define whereabouts in the message the payload has to be placed. Information put in other locations is likely to be ignored and, therefore, result in the incorrect operation of the service.

### 3.3. Obtaining and Setting Data

The default behavior of all Enterprise Service Bus components (whether they be Actions, Listeners, Gateways, Routers, Notifiers or so on) is to "get" and "set" data on the message using that message's *Default Payload Location*.

All ESB components use the **MessagePayloadProxy** to manage the retrieval of the payload and to set it on the message. It handles the default case that was outlined above, but it also allows it to be overridden in a uniform manner across all components. It also allows for one to override the "get" and "set" location for the message payload in a uniform way using the following component properties:

- `get-payload-location`: The location from which to get the message payload.
- `set-payload-location`: The location on which to set the message payload.



#### Note

Prior to JBossESB 4.2.1GA, there was no default message payload exchange pattern in place. Subsequent releases can be configured to be backwards compatible by setting the `use.legacy.message.payload.exchange.patterns` property to `true` in the core section of the `jbossesb-properties.xml` file in the `jbossesb.sar`.

### 3.4. Extensions to the Body

In addition to letting one manipulate the contents of a Message Body directly in terms of bytes or name/value pairs, there are also a number of interfaces available that simplify this process. They do so by providing predefined message structures and methods with which to manipulate them.

These interfaces are extensions to the basic Body interface and can be used in conjunction with existing clients and services. Message consumers do not need to be aware of these new types because the underlying data structure of the message remains unchanged. However, it is important to realise that these extensions do not store their data in the default location; rather, data should be retrieved by using the corresponding methods for that extension class.

#### Extensions Types

##### **org.jboss.soa.esb.message.body.content.TextBody**

The content of the Body is an arbitrary String and can be manipulated using the `getText` and `setText` methods.

**org.jboss.soa.esb.message.body.content.ObjectBody**

The content of the Body is a Serialized Object and can be manipulated using the `getObject` and `setObject` methods.

**org.jboss.soa.esb.message.body.content.MapBody**

The content of the Body is a `Map(String, Serialized)` and can be manipulated using the `setMap` and other methods.

**org.jboss.soa.esb.message.body.content.TextBody**

The content of the Body is an arbitrary String and can be manipulated using the `getText` and `setText` methods.

**org.jboss.soa.esb.message.body.content.ObjectBody**

The content of the Body is a Serialized Object and can be manipulated using the `getObject` and `setObject` methods.

**org.jboss.soa.esb.message.body.content.MapBody**

The content of the Body is a `Map(String, Serialized)` and can be manipulated using the `setMap` and other methods.

**org.jboss.soa.esb.message.body.content.BytesBody**

The content of the Body is a byte stream that contains an arbitrary Java data-type. It can be manipulated using the methods for the data-type being . Once created, the `BytesMessage` should be placed into either a read-only or write-only mode, depending upon how it needs to be manipulated. You can change between these modes by using the `readMode()` and `writeMode()` methods but each time the mode is changed the buffer pointer will be reset. It is necessary to call the `flush()` method to ensure that all of your updates have been applied to the Body.

One can create messages that have Body implementations based on one of these specific interfaces by using either the **XMLMessageFactory** or the **SerializedMessageFactory** classes. (The **XMLMessageFactory** and **SerializedMessageFactory** classes are more convenient to use when working with messages than **MessageFactory** and its associated classes.)

A "create" method will be found associated with each of the various Body types. An example is `createTextBody`, which allows one to create and initialize a message of that specific type. Once created, the message can be manipulated directly through editing the raw Body or by using the methods of its interface. The Body structure is maintained even after transmission so that it can be manipulated by the message recipient using the methods of the interface that created it.

The **XMLMessageFactory** and **SerializedMessageFactory** are more convenient ways with which to work with messages than the **MessageFactory** and its associated classes. The latter are described in the following sections.

## 3.5. The Message Header

The content of the Message Header is contained in an instance of the **org.jboss.soa.esb.addressing.Call** class.

```

public class Call
{
    public Call ();
    public Call (EPR epr);

    public void setTo (EPR epr);
    public EPR getTo () throws URISyntaxException;

    public void setFrom (EPR from);
    public EPR getFrom () throws URISyntaxException;

    public void setReplyTo (EPR replyTo);
    public EPR getReplyTo () throws URISyntaxException;

    public void setFaultTo (EPR uri);
    public EPR getFaultTo () throws URISyntaxException;

    public void setRelatesTo (URI uri);
    public URI getRelatesTo () throws URISyntaxException;

    public void setAction (URI uri);
    public URI getAction () throws URISyntaxException;

    public void setMessageID (URI uri);
    public URI getMessageID () throws URISyntaxException;

    public void copy (Call from);
}

```

**Example 3.11. org.jboss.soa.esb.addressing.Call**

**org.jboss.soa.esb.addressing.Call** supports both one-way and request-reply interaction patterns.

Property	Type	Required	Description
To	EPR	Yes	The address of the intended receiver of this message.
From	EPR	No	Reference of the endpoint where the message originated.
ReplyTo	EPR	No	An EPR that identifies the intended receiver for replies to this message.
FaultTo	EPR	No	An endpoint reference that identifies the intended receiver for faults related to this message.
Action	URI	Yes	An identifier that uniquely and opaquely identifies the semantics implied by this message.
MessageID	URI	Depends	A URI that uniquely identifies this message.

**Table 3.1. org.jboss.soa.esb.addressing.Call Properties**

The relationship between the Header and the various end-point references can be depicted in UML as:

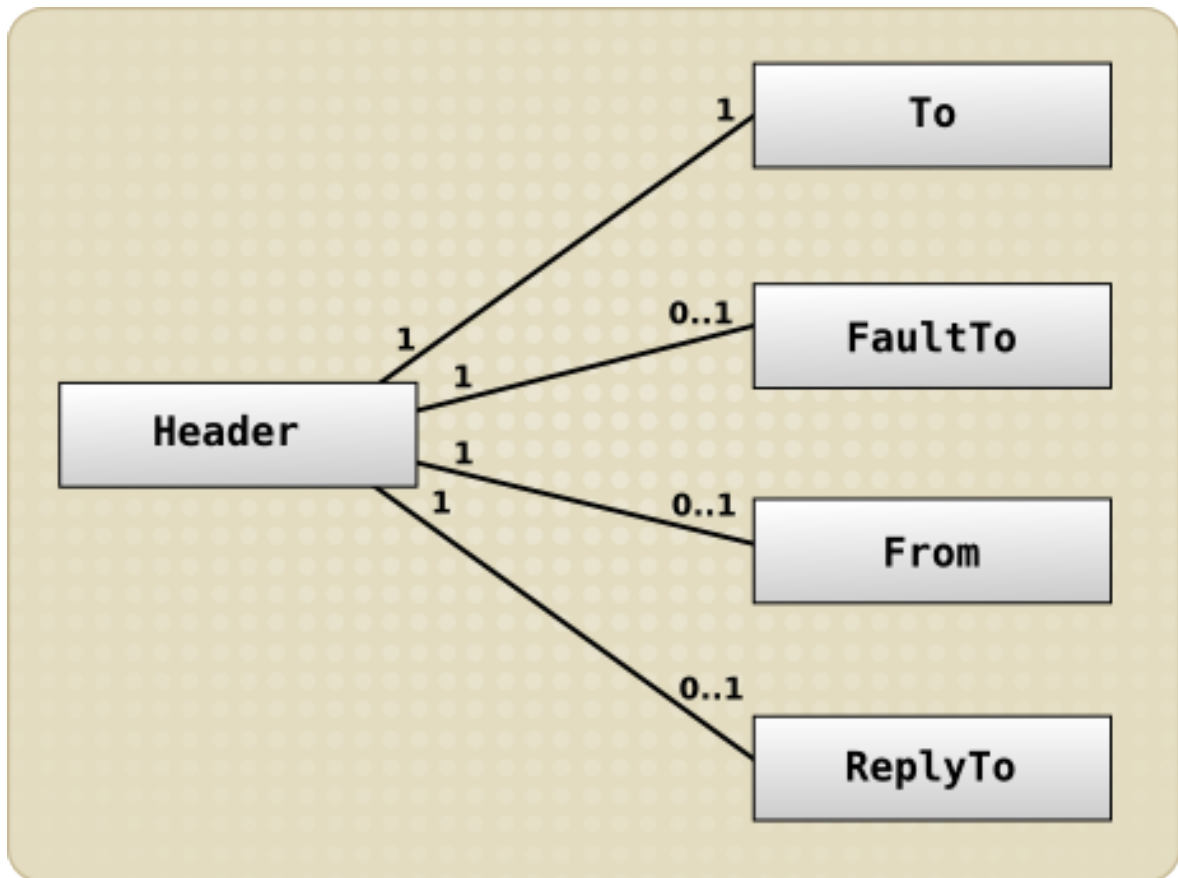


Figure 3.3. Relationship Between the Header and EPRs in UML

The role of the Header must be considered when one is developing and using one's own services. For example, if a synchronous interaction pattern based upon request and response is required, one will be expected to set the **ReplyTo** field or a default EPR will be used. Even with "request/response," the response need not go back to the original sender, if one so chooses. Likewise, when sending one-way (no response) messages, do not set the **ReplyTo** field because it will be ignored. Please see the following section on the LogicalEPR for more details.



### Note

The Message Header is formed in conjunction with the Message by its creator. It is immutable once it has been transmitted between end-points. Although the interfaces allow the recipient to modify the individual values, the JBoss ESB will ignore such modifications. In future releases it is likely that such modifications will also be disallowed by the API, in order to improve clarity. These rules can be found in the WS-Addressing standards.

## 3.6. LogicalEPR

One's ReplyTo or FaultTo end-point references should always use the *LogicalEPR*, as opposed to one of the "physical" EPRs (such as JMS-EPR.) A LogicalEPR is an end-point reference that simply specifies the name and category of an ESB service or end point. It contains no physical addressing information.

The LogicalEPR is the preferred option because it makes no assumptions about the capabilities of the user of the end-point reference (which is usually, but not necessarily always, the Enterprise Service Bus itself.) The client of the LogicalEPR can use the service name and category details that have been supplied within it to "look up" the details of the physical end-point for that service. It will do so at the point in time when it intends to make the invocation (that is, receive the relevant addressing information.) The client will also be able to select a physical end-point type that suits it.

### Default FaultTo

The FaultTo is an end-point reference that identifies the intended receiver of Faults related to the message.

The JBoss Enterprise Service Bus will route any Fault to the end-point reference in the FaultTo property of the incoming message. If FaultTo is not set, the JBoss ESB will check the ReplyTo and From properties in turn. If no valid end-point reference is obtained from checking all of these fields, the error will be output to the console.

This property can be absent if the sender cannot receive fault messages or if one does not want any response at all. However, it is recommended in such scenarios that one use the *DeadLetter Queue Service* end-point reference as one's FaultTo or, otherwise, any faults that do occur will be saved for later processing.

### Default ReplyTo

The ReplyTo property is an end-point reference that identifies the intended receiver of replies to the message. The message header must contain a ReplyTo property if a reply is to be expected.


The JBoss ESB supports default ReplyTo values for each type of transport. These are used in situations in which a response is required but the ReplyTo property has not been supplied. Some of these defaults require system administrators to configure the JBoss ESB correctly.

Transport	ReplyTo
JMS	A queue with the same name as the one used to deliver the original request with the suffix of <i>_reply</i> .
JDBC	A table in the same database with the same name as the one used to deliver the original request with the suffix of <i>_reply_table</i> . The reply table needs the same column definitions as the request table.
files	For both local and remote files, no administration changes are required. Responses are written into the same directory as the request but with a unique suffix to ensure that only the original sender will pick up the response.

Table 3.2. Default ReplyTo by transport

## 3.7. The Message Payload

From the perspective of an application or service, the message payload is viewed as a combination of the Body, the Attachments and the Properties.



**Warning**  
At the current time, developers should not use either Properties or Attachments.



The general concepts that they embody are under re-evaluation and may change significantly in future releases.

Red Hat recommend that you include Property and Attachment data in the Message Body instead.

A representation of the payload in UML is shown below:

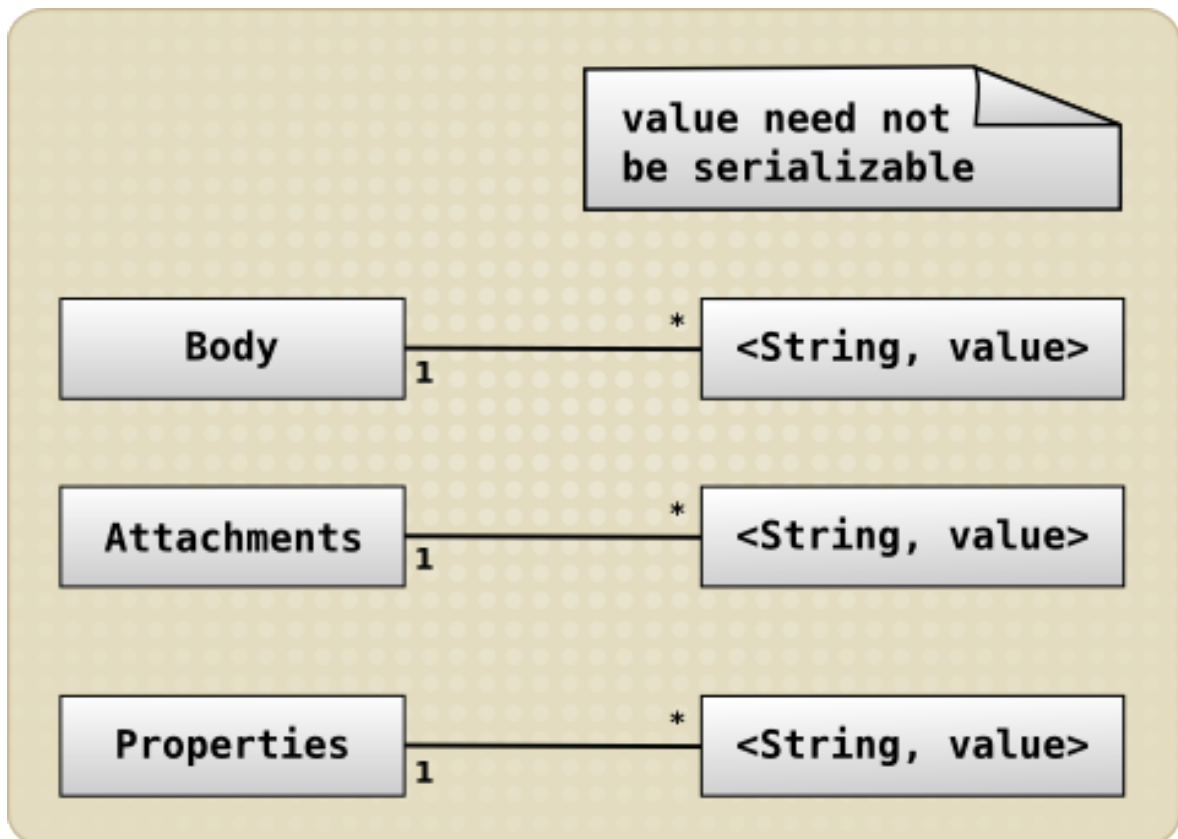


Figure 3.4. UML Representation of the Message Payload

One can apply more complex content by using the `add` method, which supports named Objects. A finer "granularity" of data access can be achieved by using named Object-pairs. Any type of Object can be added to the Body. If one adds objects that are not Java-serializable, one must provide the JBoss Enterprise Service Bus with the ability to *marshal* and *un-marshal* the message. See the section on "Message Formats" for more details on this process.

If no name has been supplied for "setting" or "getting," then that which was defined by `DEFAULT_LOCATION` will be used.



### Note

Be careful when using serialized Java objects in messages, because they constrain the service implementations.

It is easiest to work with the Message Body through the named Object approach. One can add, remove and inspect individual data items within the Message payload without having to decode the entire Body. Furthermore, one can combine named Objects with the byte array within the payload.



### Note

In the current release of the JBoss ESB, only Java-serialized objects may be attachments. This restriction will be removed in a subsequent release.

## 3.8. The MessageFactory

Although each Enterprise Service Bus component deals with an ESB Message as a collection of Java objects, it is often necessary to serialize these messages. Situations where this might be undertaken include when one is saving to a data-store, sending the message between different JBoss ESB processes or debugging.

The JBoss ESB does not impose a single, specific "normalized" format for message serialization because the requirements of the format will be influenced by the unique characteristics of each ESB deployment. All implementations of the `org.jboss.soa.esb.message.Message` interface are obtained from the `org.jboss.soa.esb.message.format.MessageFactory` class:

```
public abstract class MessageFactory
{
    public abstract Message getMessage ();
    public abstract Message getMessage (URI type);

    public static MessageFactory getInstance ();
}
```

### Example 3.12. `org.jboss.soa.esb.message.format.MessageFactory`

Message serialization implementations are uniquely identified by uniform resource indicators. One can either specify the implementation when creating a new instance, or use the pre-configured default.

Currently, the JBoss ESB provides two implementations, `JBOSS_XML` and `JBOSS_SERIALIZED`. These implementations are defined in the `org.jboss.soa.esb.message.format.MessageType` class.



### Important

One should be wary about using the `JBOSS_SERIALIZED` version of the message format because it can easily tie one's applications to specific service implementations.

Additional message implementations may be provided at run-time through the `org.jboss.soa.esb.message.format.MessagePlugin`.

```

public interface MessagePlugin
{
    public static final String MESSAGE_PLUGIN =
        "org.jboss.soa.esb.message.format.plugin";

    public Message getMessage ();
    public URI getType ();
}

```

### Example 3.13. `org.jboss.soa.esb.message.format.MessagePlugin`

Each plug-in must uniquely identify the type of message implementation it provides by using the `getType()` method. Plug-in implementations must be identified to the system in the `jbossesb-properties.xml` file by using property names with the `org.jboss.soa.esb.message.format.plugin` extension.



#### Note

The default Message type is `JBOSS_XML`. However, this can be changed by setting the property `org.jboss.soa.esb.message.default.uri` to the desired uniform resource indicator.

## 3.9. Message Formats

The two serialized message formats, `JBOSS_XML` and `JBOSS_SERIALIZED`, will now be studied in more detail.

### 3.9.1. Message Type . JAVA\_SERIALIZED

This implementation requires that all of the components of a message are serializable. It requires that the recipients of this type of message are able to de-serialize it. In other words, this implementation must be able to instantiate the Java classes contained within the message.

It also requires that all contents be Java-serializable. Any attempt to add a non-serializable object to the Message will result in an `IllegalArgumentException` being thrown.

The URI for it is `urn:jboss/esb/message/type/JAVA_SERIALIZED`.

### 3.9.2. Message Type . JBOSS\_XML

This uses an XML representation of the message. The schema for the message is defined in `message.xsd`, which is within the `schemas` subdirectory.

The URI for it is `urn:jboss/esb/message/type/JBOSS_XML`.

If one adds non-Java serializable objects to the message, one will have to provide a mechanism for marshaling those objects to and from XML. This can be achieved by creating a plug-in using the `org.jboss.soa.esb.message.format.xml.marshal.MarshalUnmarshalPlugin` interface.

```
public interface MarshalUnmarshalPlugin
{
    public static final String MARSHAL_UNMARSHAL_PLUGIN =
        "org.jboss.soa.esb.message.format.xml.plugin";

    public boolean marshal (Element doc, Object param)
        throws MarshalException;

    public Object unmarshal (Element doc) throws UnmarshalException;

    public URI type ();
}
```

**Example 3.14.** public interface MarshalUnmarshalPlugin

Marshaling plug-ins must be registered with the system through the **jbossesb-properties.xml** configuration file. They must have attribute names that start with the prefix **MARSHAL\_UNMARSHAL\_PLUGIN**.

When it is packing objects in XML, The JBoss ESB runs through the list of registered plug-ins until it finds one that can deal with the object type in question. If it does not find a suitable plug-in, it returns a Fault message. The name of the plug-in that packed the object is also attached to the message. This all facilitates unpacking at the Message receiver.

Now that one has studied the concepts behind services and messages, one can learn how to construct services using the framework provided by **Rosetta**.

# Building and Using Services

## 4.1. **updated** Listeners, Routers/Notifiers and Actions

### 4.1.1. Listeners

*Listeners* encapsulate the end-points for ESB-aware message reception. Upon receipt of a message, a Listener feeds that message into a “pipeline” of message processors. These process the message before routing the result to the “replyTo” end-point. The action processing that takes place in the pipeline may consist of steps in which the message is transformed by one processor, then some business logic is applied by the next and so on, before the result is routed to either the next step in the pipeline, or to another end-point.



#### Note

One can configure various parameters for listeners. An example is that of the number of active worker threads. See *Section 9.1, “Overview”* for a full range of these options.

### 4.1.2. **updated** Routers

*Routers* are used to direct either the ESB Message itself or its payload to an end-point destination. Some routers support the `unwrap` property. If this property is set to `true`, then the ESB Message payload will be extracted and sent to the ESB-unaware end-point by itself. Setting the `unwrap` option to `false` will pass the ESB Message in its entirety, without extracting the payload. In this case, the receiving end-point must be ESB-aware so that it can handle the message.

No further processing of the action pipeline will occur after the router operation, even if there are further actions waiting in the configuration. If one requires this type of splitting, one should use the *StaticWiretap* action.

Tools like **StaticWiretap** and *StaticRouter* can only be used for routing to other services. There are also programs available that can route dynamically, based on the content of the message. Please see the section entitled “What is Content Based Routing?” in the *Services Guide* for more information on content-based routers.

For information about the usage of different routers please see the “Routers” section of Chapter 11, (“Out-of-the-Box Actions.”)

### 4.1.3. Notifiers

*Notifiers* are used to pass on success or error information to ESB-unaware endpoints. One should not use notifiers for communicating with ESB-aware endpoints. This may mean that one cannot have ESB-aware and -unaware endpoints listening to the same channel. Instead, consider using either *Couriers* or the **ServiceInvoker** within actions to communicate with ESB-aware end-points.

Not all ESB-aware transports are supported by notifiers, and, by the same token, not all transports support notifiers. Notifiers are deliberately kept simple in the sense of what they allow to be transported: either a `byte[]` or a `String` (which is obtained by calling `toString()` on the payload.)



### Note

The `JMSNotifier` previously sent different types of JMS message (`TextMessage` or `ObjectMessage`) based on the type of ESB Message (whether it was XML or Serializable, respectively.) This was wrong, as the type of ESB Message should not affect the way in which the Notifier sends responses. Therefore, as of **JBossESB 4.2.1CP02**, the message type to be used by the Notifier can be set as a property (`org.jboss.soa.esb.message.transport.jms.nativeMessageType`) on the ESB message. Possible values are `NotifyJMS.NativeMessage.text` or `NotifyJMS.NativeMessage.object`. For compatibility with previous releases, the default value depends upon the ESB Message type: "object" for Serializable and "text" for XML. However, Red Hat discourage users from relying upon defaults.

As outlined above, it is the responsibility of a listener to act as a message delivery endpoint and to deliver messages to an *action processing pipeline*. Each listener configuration needs to supply information for:

- The Registry (see the **service-category**, **service-name**, **service-description** and **EPR-description** tag names.) If one sets the optional **remove-old-service** tag name to `true`, then the Enterprise Service Bus will remove any pre-existing service entry from the Registry and then add this new instance. However, use this functionality with care, because the entire service will be removed, including all end point references.
- Instantiation of the listener class (see the **listenerClass** tag name.)
- This the end point reference that the listener will service. It is transport-specific. The following example corresponds to a Java Message Service end point reference (see the **connection-factory**, **destination-type**, **destination-name**, **jndi-type**, **jndi-URL** and **message-selector** tag names).
- The "action processing pipeline". One or more `<action>` elements, each of which must contain, at the very least, the "class" tag name. These will determine which action class will be instantiated for that link in the processing chain.

```

<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/
etc/schemas/xml/jbossesb-1.0.1.xsd" parameterReloadSecs="5">

<providers>
  <jms-provider name="JBossMQ"
    connection-factory="ConnectionFactory"
    jndi-URL="jnp://127.0.0.1:1099"
    jndi-context-factory="org.jnp.interfaces.NamingContextFactory"
    jndi-pkg-prefix="org.jboss.naming:org.jnp.interfaces">
    <jms-bus busid="quickstartGwChannel">
      <jms-message-filter dest-type="QUEUE"
        dest-name="queue/quickstart_helloworld_Request_gw"/>
    </jms-bus>
    <jms-bus busid="quickstartEsbChannel">
      <jms-message-filter dest-type="QUEUE"
        dest-name="queue/quickstart_helloworld_Request_esb"/>
    </jms-bus>
  </jms-provider>
</providers>

<services>
  <service category="FirstServiceESB"
    name="SimpleListener" description="Hello World">
    <listeners>
      <jms-listener name="JMS-Gateway"
        busidref="quickstartGwChannel" maxThreads="1"
        is-gateway="true"/>
      <jms-listener name="helloWorld"
        busidref="quickstartEsbChannel" maxThreads="1"/>
    </listeners>

    <actions>
      <action name="action1" class="org.jboss.soa.esb.samples.
quickstart.helloworld.MyJMSListenerAction"
        process="displayMessage" />
      <action name="notificationAction"
        class="org.jboss.soa.esb.actions.Notifier">
        <property name="okMethod" value="notifyOK" />
        <property name="notification-details">
          <NotificationList type="ok">
            <target class="NotifyConsole"/>
          </NotificationList>
          <NotificationList type="err">
            <target class="NotifyConsole"/>
          </NotificationList>
        </property>
      </action>
    </actions>
  </service>
</services>
</jbossesb>

```

Example 4.1. HelloWorld Quick Start Service Configuration

This example configuration will instantiate a listener object (`jms-listener` attribute), which will wait for those incoming ESB Messages that are serialized within a `javax.jms.ObjectMessage`. It will deliver each incoming message to an action processing pipeline consisting of two steps (`<action>` elements):

action1.

**MyJMSListenerAction** (a trivial example follows)

notificationAction.

An `org.jboss.soa.esb.actions.SystemPrintLn`

The following action class will prove useful when one is debugging one's XML action configuration:

```
public class MyJMSListenerAction
{
    ConfigTree _config;

    public MyJMSListenerAction(ConfigTree config) { _config = config; }

    public Message process (Message message) throws Exception
    {
        System.out.println(message.getBody().get());
        return message;
    }
}
```

**Example 4.2. A Java Message Service Gateway listener added to the above Shoe Store Service example**

Action classes are the main way in which ESB users can tailor the framework to their specific needs. The **ActionProcessingPipeline** class will expect any other action class to provide, at a minimum, the following information:

- A public constructor that takes a single argument of the type **ConfigTree**
- One or more public methods that take a **Message** argument and return a **Message** result.

Optional public callback methods each take a **Message** argument. This will be used to notify of the result of the specific step in the processing pipeline (see items five and six below.)

The `org.jboss.soa.esb.listeners.message.ActionProcessingPipeline` class will perform the following steps for all of those steps configured via `<action>` elements:

1. Instantiate an object of the class specified in the 'class' attribute. This needs a constructor that takes a single argument of the type **ConfigTree**.
2. Analyze contents of the 'process' attribute.

The contents can take the form of a comma-separated list of public method names for the instantiated class (step one). Each of these methods must take a single argument of the type **Message** and, subsequently, return a **Message** object that will be passed to the next step in the pipeline.



If the 'process' attribute is not present, the pipeline will assume the use of a single processing method called process.

Using a list of method names in a single <action> element has some advantages over the use of successive <action> elements. This is because the action class is instantiated once and methods will be invoked on the same instance of that class. This reduces the overhead and means that "state" information can be kept in the instance objects.

This approach is useful for user-supplied (new) action classes. The alternative, (a list of <action> elements), continues to be a way of re-using pre-existing action classes.

3. Sequentially invoke each method in the list by using the Message returned in the previous step.
4. If the value returned by any step is "null," the pipeline will stop processing immediately.
5. The callback method for success in each of the <action> element is as follows: if the list of methods in the 'process' attribute was executed successfully, the pipeline will analyze the contents of the **okMethod** attribute. If no methods were specified, processing will continue with the next <action> element. If a method name is provided in the **okMethod** attribute, it will be invoked using the **Message** returned by the last method in step three. If the pipeline succeeds, then the **okMethod** notification will be called on all handlers from the last back through to the first.
6. The callback method for failure in each <action> element is as follows: if an exception occurs, then the **exceptionMethod** notification will be called on all handlers from the current (failing) handler back to the initial one. (Currently, if no **exceptionMethod** was specified, the only output will be the logged error.) If an **ActionProcessingFaultException** is thrown from any process method, then an error message will be returned as per the rules defined in the next section. The contents of the error message will be either whatever is returned from the exception's **getFaultMessage**, or a default **Fault** that contains the information found within the original exception.

Any action classes that have been supplied by users in order to customise the behavior of the Enterprise Service Bus, might need extra runtime configuration work (for example, in the XML above, the Notifier class requires the <NotificationList> child element). Each <action> element will utilize the attributes mentioned above and will ignore any others and also any optional child elements. These will, however, be passed through to the action class constructor in the require **ConfigTree** argument. Each action class will be instantiated with its corresponding <action> element and, thus, will not and, indeed, must not, see its "sibling" action elements.



#### Note

In JBoss ESB 5.0, the name of the property that is used to enclose the **NotificationList** elements in the <action> target is not validated.

### 4.1.4. Actions and Messages

Actions are triggered by the arrival of a message. (The specific Action implementation is expected to know where the data resides within a message.) Because a Service may be implemented using an arbitrary number of Actions, it is possible that a single input message could contain information on behalf of more than one Action. In this case, it is incumbent upon the Action developer to choose one

or more unique locations within the message body in which to place its data and then to communicate this location to the Service's consumers.

Furthermore, because Actions may be chained together, it is possible that an Action earlier in the chain will modify the originally-input message, or even entirely replaces it.



### Note

From a security perspective, one should be careful about using unknown Actions within one's Service chain. Red Hat recommend that one should encrypts one's information.



### Note

Red Hat recommends that one retains the original information in the situation whereby multiple Actions are sharing data within an input Message, and each one is modifying the information as it flows through the chain. This is so that Actions further down the chain still have access to it. Obviously, there will be situations in which this is either impossible or simply unwise.

Within the JBoss ESB, Actions that modify the input data can place it within the Body location named by `org.jboss.soa.esb.actions.post`. This means that if there are N Actions in the chain, Action N can find the original data in the place where it would normally look. It also means that, if Action N-1 modified the data, then N will find it within the other specified location. To further facilitate Action chaining, Action N can see if Action N-2 modified the data by looking in the `org.jboss.soa.esb.actions.pre`-named Body location.



### Note

As mentioned earlier, one should utilize the default-named Body location with care when chaining Actions, in case they use it in a conflicting manner.

### 4.1.5. Handling Responses

Two processing mechanisms are supported for the purpose of handling responses in the action pipeline. These are called the *explicit* and *implicit* processing mechanisms, the latter of which is based on the response of the actions.

If the implicit mechanism is used, then responses will be processed as follows:

- If any action in the pipeline returns a null message, then no response will be sent.
- If the final action in the pipeline returned a non-error response, then a reply will be sent to the ReplyTo EPR belonging to the request message or, if not set, to the request message's From EPR. In the event that there is no way to route responses, an error message will be logged by the system.

If the explicit mechanism is used, then the responses will be processed in the following manner:

- If the action pipeline is specified to be `OneWay`, then it will never send a response.

- If the pipeline is specified as RequestResponse, then a reply will be sent to the ReplyTo EPR of the request message. If it is not set, it will be sent to the From EPR of the request message. If no end-point reference has been specified, then no error message will be logged by the system.

Red Hat recommend that all action pipelines should use the explicit processing mechanism. This can be enabled by simply adding the mep attribute to the actions element in the `jboss-esb.xml` file. The value of this attribute should be set to either Oneway or RequestResponse.

### 4.1.6. Error Handling When Actions are Being Processed

Errors may occur when an action chain is being processed. Such errors should be thrown as exceptions from the Action pipeline and, hence, the processing of the pipeline itself. As mentioned earlier, a Fault Message may be returned within an `ActionProcessingFaultException`. If it is important that information about errors be returned to the sender (or an intermediary), then the `FaultTo` EPR should be set. If it is not set, then the JBoss Enterprise Service Bus will attempt to deliver error messages based on the `ReplyTo` EPR and, if that too, is not set, then based on the `From` EPR. If none of these end-point references has been set, then the error information will be logged locally.

Error messages of various types can be returned from the Action implementations. However, the JBoss Enterprise Service Bus supports the following “system” error messages. In the case that an exception is thrown and no application-specific Fault Message is present, all of these may be identified by the uniform resource indicator that is mentioned in the message fault:

`urn:action/error/actionprocessingerror`

This means that an action in the chain threw an `ActionProcessingFaultException` but that it did not include a Fault Message to return. The exception details will be contained within the **Fault**'s 'reason String.

`urn:action/error/unexpectederror`

This means that an unexpected exception was caught during the processing. Details about the exception can be found in the **Fault**'s reason String.

`urn:action/error/disabled`

This means that action processing is disabled.

If an exception is thrown within the **Action** chain, then it will be passed as a `FaultMessageException` back to the client. It is then thrown again from the **Courier** or **ServiceInvoker** classes. This exception, which is also thrown whenever a **Fault** message is received, will contain the **Fault** code and reason, as well as any exception that has been passed on.

## 4.2. Meta-Data and Filters

As a message flows through the Enterprise Service Bus, it may be useful to attach meta-data to it. This could include such information as the time it entered the ESB and the time it left. Furthermore, it may be necessary to dynamically augment the message by, for example, adding transaction or security information. Both of these capabilities are supported in the Enterprise Service Bus, (for both gateway and ESB nodes), via the filter mechanism.



### Note

Please be aware that the name of the filter property, the package for the `InputOutputFilter` and its signature all changed in **JBossESB 4.2 MR3** from those that were present in earlier milestone releases.

The `org.jboss.soa.esb.filter.InputOutputFilter` class has two methods:

- `public Message onOutput (Message msg, Map<String, Object> params)` throws a `CourierException`. This is called as a message flows to the transport. An implementation may modify the message and return a new version. Additional information may be provided by the caller in the form of extra parameters.
- `public Message onInput (Message msg, Map<String, Object> params)` throws a `CourierException`. This is called as a message flows from the transport. An implementation may modify the message and return a new version. Additional information may be provided by the caller in the form of extra parameters.

Filters are defined in the `filters` section of the `jbossesb-properties.xml` file (which is normally located in the `jbossesb.sar` archive) by using the property `org.jboss.soa.esb.filter.<number>`, where `<number>` can be any value. This value is used to indicate the order in which multiple filters are to be called (from lowest to highest.)

The JBoss Enterprise Service Bus ships with `org.jboss.internal.soa.esb.message.filter.MetadataFilter` and `org.jboss.internal.soa.esb.message.filter.GatewayFilter`, which add the following meta-data to the `Message` as `Properties` with the indicated property names and the returned `String` values:

#### Gateway-related Message Properties

##### `org.jboss.soa.esb.message.transport.type`

File, FTP, JMS, SQL, or Hibernate.

##### `org.jboss.soa.esb.message.source`

The name of the file from which the message was read.

##### `org.jboss.soa.esb.message.time.dob`

The time the message entered the ESB. This could be the time it was sent or the time it arrived at a gateway.

##### `org.jboss.soa.esb.message.time.dod`

The time the message left the Enterprise Service Bus (in other words, the time it was received.)

##### `org.jboss.soa.esb.gateway.original.file.name`

If the message was received via a file-related gateway node, then this element will contain the name of the original file from which the message was sourced.

##### `org.jboss.soa.esb.gatway.original.queue.name`

If the message was received via a Java Message Service gateway node, then this element will contain the name of the queue from which it was received.

**org.jboss.soa.esb.gateway.original.url**

If the message was received via an SQL gateway node, then this element will contain the original database Uniform Resource Locator.

**Note**

Although it is safe to deploy the **GatewayFilter** on all of the Enterprise Service Bus' nodes, it will only add information to a **Message** if it is deployed on a *gateway* node.

One can add more meta-data to the message by creating and registering suitable filters. Such a filter can determine whether or not it is running within a gateway node through the presence (or absence) of the following named-entries within additional parameters:

**Gateway-Generated Message Parameters****org.jboss.soa.esb.gateway.file**

The file from which the Message was sourced. This will only be present if this gateway is file-based.

**org.jboss.soa.esb.gateway.config**

The ConfigTree that was used to initialize the gateway instance.

**Note**

The **GatewayFilter** is only supported by file-based Java Message Service and SQL gateways in JBoss ESB 5.0.

**4.3. Updated What is a Service?**

The JBoss Enterprise Service Bus does not impose restrictions with regard to what should constitute a service. As discussed earlier in this document, the ideal SOA infrastructure encourages a loosely-coupled interaction pattern between clients and services, whereby the message is of critical importance and implementation specific-details are hidden behind an abstract interface. This means that the implementations can be changed without the need to alter clients/users at the same time. Only changes to the message definitions will necessitate updates to the clients.

For this reason, the Enterprise Service Bus uses a message-driven pattern for service definitions and structures: clients send Messages to services and the basic service interface is essentially a single process-method that subsequently operates on the Message as it is received. Internally, a service is structured from one or more Actions, that can be chained together to process the incoming Message. All of the tasks undertaken by an Action are implementation-dependent. Examples of this are the processes to update a database table entry or call an Enterprise Java Bean.

When developing customised services, one must first determine the type of conceptual interface/contract that it will exposes to users/consumers. This contract should be defined in terms of Messages, (for example, what the payload looks like, what type of Response Message, if any, will be generated and so forth.)



### Note

Once defined, the contract information should be published in the registry. At present, the JBoss Enterprise Service Bus does not have a way of doing this automatically.

Clients can then use the service, as long as they do so in accordance with the published contract. How the service processes the Message and performs the work necessary is an implementation choice. It could be done within a single Action or within multiple Actions. One will have the usual trade-offs to bear in mind when making this decision (those of manageability versus re-usability.)



### Note

In subsequent releases, Red Hat will be improving tool support in order to facilitate the development of services.

### 4.3.1. The "ServiceInvoker"

From a client's perspective, the Courier interface and its various implementations can be used to interact with services. However, this is still a relatively low-level approach, as it requires the developer code to contact the registry and deal with failures. Furthermore, since the JBoss Enterprise Service Bus has fail-over capabilities for stateless services, this would, again, have to be managed by the application. See the "Advanced" chapter for more details on fail-over functionality.

The **ServiceInvoker** was introduced in **JBossESB 4.2** to help simplify the development effort. It assists by hiding many of the lower level details and works opaquely with the stateless service fail-over mechanisms. Because of this, the **ServiceInvoker** is the recommended client-side interface for using services within the ESB.

```
public class ServiceInvoker
{
    public ServiceInvoker(Service service) throws MessageDeliverException;
    public ServiceInvoker(String serviceCategory, String serviceName) throws
    MessageDeliverException;

    public Message deliverSync(Message message, long timeoutMillis) throws
    MessageDeliverException, RegistryException, FaultMessageException;
    public void deliverAsync(Message message) throws MessageDeliverException;
}
```

#### Example 4.3. Enabling **GLOBAL** inVM scope for a service

An instance of the **ServiceInvoker** can be created for each service with which the client needs to interact. Once created, the instance contacts the registry where appropriate, in order to determine the primary end-point reference (and, in the case of fail-overs, any alternative end point references.)

Then, the client can determine how to send Messages to the service: synchronously (via **deliverSync**) or asynchronously (via **deliverAsync**.) When it is synchronous, a time-out value must be specified. This time-out represents for how long the client will await a response. If no response is received during this period, an `MessageDeliverException` is thrown.

**Note**

From **JBossESB 4.5** onwards, the `ResponseTimeoutException` is thrown. This is derived from `MessageDeliverException`.

Failures to either contact the Registry or to successfully look up the service are indicated by throwing a `RegistryException` from **deliverSync**. Time-out values may indicate one of three possibilities: 1.) A service failure, 2.) An indication that it is simply overloaded and cannot respond in time, or 3.) The work requested is taking longer than the allowed duration of time-out. In some cases the problem will be transient and, therefore, it may work on the next try.

**Note**

Any other kind of failure during communication with the service will cause a `FaultMessageException` to be thrown.

As mentioned earlier in this document, when sending a Message it is possible for one to specify values for **To**, **ReplyTo**, **FaultTo** and so forth, within the **Message** header. When using the **ServiceInvoker**, the **To** field is unnecessary. This is because it has already contacted the registry at construction time. In fact, when sending a Message through `ServiceInvoker`, the **To** field will be ignored in both the synchronous and asynchronous delivery modes. In a future release of the JBoss ESB, it may be possible to use any supplied **To** field as an alternate delivery destination, should the end-point references returned by the registry fail to resolve to an active service.

The JBoss ESB's fail-over properties will be discussed in the "Advanced" section of this book. In that section, there is also a discussion of how the **ServiceInvoker** can opaquely mask failures of individual service instances if multiple copies appear in the Registry. However, one may find that, in some cases, you desire to prevent automatic fail-over and, instead, inform the application immediately that a failure occurs. This can be set at the global level by configuring the `org.jboss.soa.esb.exceptionOnDeliverFailure` property to `true` in the JBoss ESB **property** file. Alternatively, this can be configured on a per-message basis. One can do this by setting that same property in the specific Message property to `true`. In both cases, the default value is `false`.

**4.3.2. Updated. Transactions**

Some couriers, such as InVM and the Java Message Service, support *transactional delivery semantics*. The current delivery semantics for such a courier are based on JMS *transacted sessions*. In other words, the message is placed in a queue within the scope of a transaction. However, it is not actually delivered until the enclosing transaction has been committed. It is then pulled by the receiver within the scope of a separate transaction.

In the case of synchronous request/response interactions, this can result in a time-out whilst waiting for the response. This is because the sender blocks while it is waiting for it and terminates the delivery transaction. **JBossESB 4.5** onwards attempts to detect these blocking situations and will throw an `IncompatibleTransactionScopeException`. The application should "catch" this and act accordingly.

### 4.3.3. Services and the ServiceInvoker

In a client-service environment, the terms "client" and "service" are used to represent roles. A single entity can be both a client and a service simultaneously. As such, one should not consider the **ServiceInvoker** to be the domain of "pure" clients; it can be used within your Services and, specifically, within Actions. For example, rather than using the built-in Content-Based Routing, an Action may wish to re-route an incoming Message to a different service based on evaluation of certain business logic. Alternatively, an Action could decide to route specific categories of fault Messages to the **Dead Letter Queue** for later administration.

The advantage of using the **ServiceInvoker** in this way, is that your Services will be able to benefit from the opaque fail-over mechanism (which is described in the "Advanced" chapter.) This means that one-way requests to other Services, faults and so on can be routed in a more robust manner without imposing more complexity upon the developer.

### 4.3.4. **Updated.** The InVM Transport

The *InVM Transport* is a new Enterprise Service Bus feature that provides communication between services running on the same Java Virtual Machine (InVM stands for *In Virtual Machine*.) This means that instances of the **ServiceInvoker** can, as their name implies, invoke a service from within the same JVM without any networking or message serialization overhead.



#### Note

It is important to realize that InVM achieves its performance benefits by optimizing those internal data structures that are used to facilitate inter-service communication. For example, the queue used to store messages is not persistent (durable.) This means that Messages may be lost in the event of a failure. Furthermore, if a service is shutdown before the queue is emptied, those Messages will not be delivered. Other limitations are mentioned throughout this section (within corresponding notes.)

Because the JBoss ESB allows services to be invoked concurrently across multiple different transports, one should be able to design ones services in such a way that they can achieve high performance and reliability. Do this by making a suitable choice of transport for each specific Message type.

The JBoss ESB currently supports two scopes:

#### **NONE**

The Service cannot be invoked over the InVM Transport. The JBoss Enterprise Service Bus is configured with this as the default value.

#### **GLOBAL**

The Service can be invoked over the InVM Transport from within the same Classloader scope.

A **LOCAL** scope will be added in a future release. It will restrict invocation to within the same deployed **.esb** archive.

Each service can specify its own InVM scope in the appropriately named *invmScope* attribute on the `<service>` element of its services configuration.



```

<service category="ServiceCat" name="ServiceName" invmScope="GLOBAL "
  description="Test Service">
  <actions mep="RequestResponse">
    <action name="action"
      class="org.jboss.soa.esb.listeners.SetPayloadAction">
      <property name="payload" value="Tom Fennelly" />
    </action>
  </actions>
</service>

```

#### Example 4.4. Enabling a **GLOBAL** inVM scope for a service

The default InVM scope for an Enterprise Service Bus deployment is specified within the **jbosbesb-properties.xml** file via the `core:jboss.esb.invm.scope.default` property. The pre-configured value is `NONE`. However, if this property were undefined, the default scope would actually be `GLOBAL`.

### InVM Transacted

The InVM Listener can execute within either a transacted or a non-transacted scope. It does so in the same manner as the other transports which support transactions. This behaviour can be controlled through either explicit or implicit configuration.

The explicit configuration of the transacted scope is controlled through the definition of the `invmTransacted` attribute on the service element and will always take precedence over the implicit configuration.

The InVM Listener will be transacted implicitly if there is another transacted transport configured on that service. At present, these additional transports can be JMS, Scheduled or SQL in type.

### Transaction Semantics

The InVM Transport in the JBoss ESB is not transactional and the message queue is held only in volatile memory. This means that the latter will be lost in the case of system failure or shutdown.



#### Note

One may not be able to achieve all of the ACID (Atomicity, Consistency, Isolation, Durability) semantics, particularly when it is used in conjunction with other transactional resources such as databases. This is because of the volatility of the InVM queue. However, the performance benefits of InVM will outweigh this downside in the majority of cases. In situations in which full ACID semantics are required, Red hat recommend that you use one of the other transactional transports, such as the Java Message Service or a database.

When InVM is used within a transaction, the message will not appear on the receiver's queue until that transaction actually commits. However, the sender will receive an immediate acknowledgment that the message has been accepted for later queuing. If a receiver attempts to pull a message within the scope of a transaction from the queue, then the message will automatically be placed back in the queue in the case that the transaction subsequently rolls back. If either a sender or a receiver of a message needs to know the transaction outcome, then they should either monitor it directly, or register a **synchronization** with the transaction.

When a message is placed back in the queue by the *Transaction Manager*, it may not go back to the same location. This was a deliberate design choice intended to maximize performance. If one's application needs specific ordering of messages then either consider using a different transport or group related messages within one single "wrapper" message.

### Threading

In order to change the number of listener threads associated with an InVM Transport, use this code:

```
<service category="HelloWorld" name="Service2" description="Service 2"
  invmScope="GLOBAL">
  <property name="maxThreads" value="100" />
  <listeners>...
  <actions>...
```

#### Example 4.5. Threading

### Lock-Step Delivery

The InVM Transport delivers low-overhead messages to an in-memory queue. This occurs very quickly and, as a result, the message queue can become overwhelmed if delivery is happening too rapidly for the Service that is "consuming" the messages. To mitigate this situation, the InVM Transport provides a *lock-step delivery* mechanism.

This lock-step delivery method attempts to ensure that messages are not delivered to a service at a rate faster than that at which the Service is able to retrieve them. It does this by blocking delivery until either the receiving Service picks up the message or a time-out period expires.

Please note that this is not a synchronous delivery method. It does not wait for a response or for the service to process the message. It only blocks until the message is removed from the queue by the service.

Lock-step delivery is disabled by default but it can be enabled for a service. Use its <property> settings on the <service>:

#### inVMLockStep

This is a Boolean value that controls whether or not lock-step delivery is enabled.

#### inVMLockStepTimeout

This is the maximum number of milliseconds for which message delivery will be blocked whilst waiting for a message to be retrieved.

```
<service category="ServiceCat" name="Service2"
  description="Test Service">
  <property name="inVMLockStep" value="true" />
  <property name="inVMLockStepTimeout" value="4000" />

  <actions mep="RequestResponse">
    <action name="action" class="org.jboss.soa.esb.mock.MockAction" />
  </actions>
</service>
```

#### Example 4.6. Enabling "Lock-Step" Delivery



#### Note

If one is using InVM within the scope of a transaction, lock-step delivery is disabled. This is because the insertion of a message into the queue is contingent on the commit of the enclosing transaction. This commit may occur at an arbitrary time before or after the expected lock-step wait period.

### Load Balancing

If a service is available, it is always invoked in preference to its InVM transport when the **ServiceInvoker** is used. Other *load balancing strategies* are only applied in the absence of an InVM end-point for the target Service.

### Pass-by-Value/Pass-by-Reference

By default, the InVM Transport passes Messages "by reference". In some situations, this can cause data integrity issues. It can also result in *class cast* issues, whereby messages are exchanged across *ClassLoader* boundaries.

In order to avoid these problems, one can set message-passing to work "by value." This can be turned on by setting the `inVMPassByValue` property on the Service in question to `true`:

```
<service category="ServiceCat" name="Service2" description="Test Service">
  <property name="inVMPassByValue" value="true" />

  <actions mep="RequestResponse">
    <action name="action" class="org.jboss.soa.esb.mock.MockAction" />
  </actions>
</service>
```

#### Example 4.7. Message Passing by Value

### Temporary InVM Queues

There are two types of InVM queue, the difference between them being that one type represents a service end-point and the other, a temporary end-point. Queues for service end-points are registered through a *registry interceptor*. All messages that have been delivered onto these

queues will remain on there until consumed by the service. Temporary end-points, by contrast, are created on demand, usually to handle responses for synchronous delivery through the **ServiceInvoker**. Messages delivered to these queues will automatically expire after a defined period. The default time for this expiration is five minutes but this can be altered by modifying the `org.jboss.soa.esb.invm.expiryTime` property within the `jbossesb-properties.xml` file.

### 4.4. **Updated.** Service Contract Definition

One can include an XML Schema Definition in order to specify a contract definition on a service. These schema definitions represent the incoming request, outgoing response and fault detail messages which are supported by the corresponding service. The schemas representing the request and response messages are used to define the format of the contents of the main body section of the message and can also enforce validation of that same content.

The schemas are declared by specifying the following attributes on the `<actions>` element associated with the service:

Name	Description	Type
<code>inXsd</code>	The resource containing the schema for the request message, representing a single element.	<code>xsd:string</code>
<code>outXsd</code>	The resource containing the schema for the response message, representing a single element.	<code>xsd:string</code>
<code>faultXsd</code>	A comma separated list of schemas, each representing one or more fault elements.	<code>xsd:string</code>
<code>requestLocation</code>	The location of the request contents within the body, if not the default location.	<code>xsd:string</code>
<code>responseLocation</code>	The location of the response contents within the body, if not the default location.	<code>xsd:string</code>

Table 4.1. Service Contract Attributes

#### Message Validation

One can automatically validate the contents of the request and response messages. This is provided that the associated schema has been declared on the `<actions>` element. This validation function can be enabled by specifying the `validate` attribute on the `<actions>` element to have a value of `true`.

Note that validation is disabled by default.

#### Exposing an ESB Service as a Web Service

The declaration of the contract schemas will automatically enable the exposure of the ESB service through a web service end-point, the contract for which can be located via the **Contract** web application. This functionality can be modified by specifying the `webservice` attribute, the values for which are as follows:

- `false`  
No web service end-point will be published.
- `true`

A web service end-point is published (default).

By default, the web service end-point does not support **WS-Addressing**. However, this feature can be enabled through use of the addressing attribute.

Value	Description
false	No support for WS-Addressing (default).
true	Require WS-Addressing support.

Table 4.2. WS-Addressing Values

When support for addressing is enabled, the WS-Addressing Message Id, Relates To URIs and Relationship Types will be added to the incoming messages as properties.

Property	Description
<b>org.jboss.soa.esb.gateway.ebws.messageID</b>	The WS-Addressing message id.
<b>org.jboss.soa.esb.gateway.ebws.relatesTo</b>	A String array containing the WS-Addressing RelatesTo URIs.
<b>org.jboss.soa.esb.gateway.ebws.relationshipType</b>	A String array containing the WS-Addressing Relationship Types corresponding to the RelatesTo URIs.

Table 4.3. WS-Addressing Properties

The following example illustrates the declaration of a service which, (without being exposed through a web service end-point), "wishes" to validate the request/response messages:

```
<service category="ServiceCat" name="ServiceName" description="Test
  Service">
  <actions mep="RequestResponse" inXsd="/request.xsd" outXsd="/
  response.xsd"
    webservice="false" validate="true">
    <!-- .... >
  </actions>
</service>
```

Example 4.8. Enabling the **GLOBAL** InVM Scope for a Service

The following, final example depicts the declaration of a Service which wishes to validate the request/response messages and be exposed through a web service end-point. In addition to that, the Service expects the request to be provided in the named body location, REQUEST. It will give its response in the named body location RESPONSE.

```
<service category="ServiceCat" name="ServiceName" description="Test
Service">
  <actions mep="RequestResponse" inXsd="/request.xsd" outXsd="/
response.xsd"
          validate="true" requestLocation="REQUEST"
          responseLocation="RESPONSE">
    <!-- .... -->
  </actions>
</service>
```

Example 4.9. Enabling **GLOBAL** InVM Scope for a Service

---

## Other Components

In this chapter, one shall learn about the other infrastructural components and services that exist within the JBoss Enterprise Service Bus. Several of these services have their own documentation which one should also read; the aim of this chapter is simply to provide an overview of other things that are available for developers.

### 5.1. **Updated.** The Message Store

The JBoss ESB's *Message Store* mechanism was designed with the purpose of audit tracking in mind. As is the case with other ESB services, it is "pluggable", which allows the developer to plug in his or her own persistence mechanism should there be the special need to do so. The Message Store implementation supplied with the JBoss ESB takes the form of a *database persistence mechanism*. If one requires something else, such as, for instance, a file persistence mechanism, then it is simply a matter of one writing one's own service, and then over-riding the default behavior with a configuration change.

One thing to note about the Message Store is that, at present, it is a base implementation only. Red Hat will be working with the community and partners in the future to drive the feature functionality set forward to support advanced auditing functionality and management requirements. What one sees at the moment is only a starting point.

### 5.2. Data Transformation

Most often clients and services will communicate by using the same vocabulary. However, there are situations where this is not the case and "on-the-fly" transformation from one data format to another will be required. It is unrealistic to assume that a single data format will be suitable for all business objects, particularly in a large-scale or long-running deployment. Therefore, it is necessary to provide a mechanism for transforming data from one format to another.

In the JBoss Enterprise Service Bus, this is the role of the *Transformation Service*. The current version of the ESB ships with an out-of-the-box Transformation Service based on **Milyn Smooks**. *Smooks* is a *Transformation, Implementation and Management* framework. It allows one to implement transformation logic in formats like XSLT and Java. It also provides a management framework through which one can centrally manage the transformation logic for one's message-set.

For more details see the "Message Transformations" chapter in the *Services Guide*.

### 5.3. Content-Based Routing

Sometimes it is necessary for the Enterprise Service Bus to dynamically route messages to their sources. Some scenarios in which this might be the case are when the original destination is no longer available, the service has moved or the application simply wants to have more control over where messages go based on either content, time-of-day or other such factors. The Content-Based Routing mechanism within the JBoss ESB can be used to route messages based on arbitrarily complex rules. These rules can be defined in **XPath** or *JBoss Rules* notation.

### 5.4. The Registry

In the context of the SOA-P, a registry provides applications and businesses with a central point in which to store information about their services. The registry is expected to provide the same

level of information and the same breadth of services to its clients as that of a conventional market place would do. Ideally, a registry should also facilitate the automated discovery and execution of electronic commerce transactions. It should enable a dynamic environment for business transactions. Therefore, a registry is more than an "e.-business directory". It is an inherent component of the SOA-P's infrastructure.

In many ways, the Registry Service can be considered to be the "heart" of the JBoss Enterprise Service Bus: services can "self-publish" their end-point references to the Registry when they are activated and then remove them when they are taken out of service. Consumers can consult the Registry in order to determine which end-point reference is that which is needed for the current service task.



# An Example

## 6.1. How to Use the Message

The *Message* is a critical component of the SOA development approach. It contains application-specific data which is sent between clients and services. The data in a *Message* represents an important aspect of the "contract" between a service and its clients. In this section, one shall learn some aspects of best practices in regard to the use of this component.

Firstly, consider the following example of a flight reservation service. This service supports the following operations:

`reserveSeat`

This takes a flight and seat number and returns a success or failure indication.

`querySeat`

This takes a flight and seat number and returns an indication of whether or not the seat is currently reserved.

`upgradeSeat`

This takes a flight number and two seat numbers (the currently reserved seat and the one to which one will move.)

When developing this service, it is likely one will use technologies such as *Enterprise Java Beans* (EJB3) and *Hibernate* in order to implement the business logic. In the example, one will not be shown how this logic is implemented. Instead, the service itself will be the focal point of the study.

The role of the *Service* is to "plug" the logic into the *Bus*. In order to configure it to do this, one must determine how the service is exposed to the bus, (that is, what type of contract it defines for the clients.) In the current version of the JBoss Enterprise Service Bus, this contract takes the form of the various messages that the clients and services exchange. Note that there is no formal specification for this contract within the ESB. In other words, at present, it is something that the developer defines and communicates to clients "out-of-band" from the Enterprise Service Bus. This will be rectified in a subsequent release.

### 6.1.1. The Message Structure

From the perspective of a service, of all the components within a *Message*, the *Body* is the most important, as it is used to convey information specific to the business' logic. In order to interact, both client and service must understand each other. This understanding takes the form of an agreement on the mode of transport (such as Java Message Service or HTTP) and the dialect to be used (for example, where and in what form will data appear in the *Message*?)

If one were to take the simple case of a client sending a *Message* directly to the example flight reservation service, then one would need to decide how the service is going to determine which of the operations is concerned with the *Message*. In this case, the developer decides that the *opcode* (operation code) will appear within the *Body* as a string ("reserve", "query", "upgrade") at the location called `org.example.flight.opcode`. Any other string value (or, indeed, the absence of any value) will result in the *Message* being considered illegal.



### Note

It is important to ensure that all of the values within a Message are given unique names. This is to avoid clashes with other clients or services.

The Message Body is the primary way in which data is exchanged between clients and services. It is flexible enough to contain any number of arbitrary data types. (The other parameters necessary for carrying out the business logic associated with each operation should also be suitably encoded.)

- `org.example.flight.seatnumber` for the seat number, which will be an integer.
- `org.example.flight.flightnumber` for the flight number, which will be a string.
- `org.example.flight.upgradenumber` for the upgraded seat number, which will be an integer.

Operation	opcode	seatnumber	flightnumber	upgradenumber
reserveSeat	String: reserve	integer	String	N/A
querySeat	String: query	integer	String	N/A
upgradeSeat	String: upgrade	integer	String	integer

Table 6.1. Operation Parameters

As has been mentioned previously, all of these operations return information to the client. Such information will, likewise, be encapsulated within a Message. Messages in response will go through the same processes as that currently being described in order for their own formats to be determined. For the purpose of simplification, the response Messages will not be considered further in this example.

The service may be built using one or more Actions. For example, one Action may pre-process the incoming Message and transform its contents in some way, before passing it on to the Action which is responsible for the main business logic. Each of these Actions may have been written in isolation (possibly by different groups within the same organisation or even by completely different organisations.) It is important that each Action has a unique view of the Message data upon which it acts. If this is not the case, it is possible that chained Actions may either overwrite or otherwise interfere with each other.

### 6.1.2. The Service

At this point, one has enough information to construct the service. For the sake of simplicity, it shall be assumed that the business logic is encapsulated within the following "pseudo-object:"

```
class AirlineReservationSystem
{
    public void reserveSeat (...);
    public void querySeat (...);
    public void upgradeSeat (...);
}
```

Example 6.1. class Airline Reservation System



### Note

One could develop one's business logic from POJOs (Plain Old Java Objects), EJBs (Enterprise Java Beans), **Spring** or so forth. The JBoss Enterprise Service Bus provides out-of-the-box support for many of these approaches. One should examine the relevant documentation and examples.

Assuming that there is no chaining of Actions and ignoring error checking procedures, the process method for the service Action then becomes the following:

```
public Message process (Message message) throws Exception
{
    String opcode = message.getBody().get("org.example.flight.opcode");

    if (opcode.equals("reserve"))
        reserveSeat(message);

    else if (opcode.equals("query"))
        querySeat(message);

    else if (opcode.equals("upgrade"))
        upgradeSeat(message);

    else
        throw new InvalidOpcode();

    return null;
}
```

Example 6.2. public Message process (Message message) throws Exception



### Note

As with WS-Addressing, one could use the Action field of the Message Header, rather than embed the opcode within the Message Body. This has a drawback in that it does not work if multiple Actions are chained together and each needs a different opcode.

## 6.1.3. Unpacking the Payload

As one can see, choosing the process method is only the start. Now methods to decode the incoming Message payload must be provided:

```
public void reserveSeat (Message message) throws Exception
{
    int seatNumber = message.getBody().get("org.example.flight.seatnumber");
    String flight =
        message.getBody().get("org.example.flight.flightnumber");

    boolean success =
        airlineReservationSystem.reserveSeat(seatNumber, flight);

    // now create a response Message
    Message responseMessage = ...

    responseMessage.getHeader().getCall().setTo(
        message.getHeader().getCall().getReplyTo()
    );

    responseMessage.getHeader().getCall().setRelatesTo(
        message.getHeader().getCall().getMessageID()
    );

    // now deliver the response Message
}
```

### Example 6.3. public void reserveSeat (Message message) throws Exception

This method illustrates how the information within the Body is extracted. It then invoke another method for business logic. A response is expected by the client in the case of the reserveSeat. This response Message is constructed using any information returned by the business logic as well as additional delivery information obtained from the Message received originally. In this example, one needs the "To" address for the response, which is taken from the ReplyTo field of the incoming Message. One also needs to relate the response to the original request. This is accomplished via both the response's RelatesTo field and the request's MessageID.

All of the other operations supported by the service will be coded in a similar manner.

### 6.1.4. The Client

As soon as the Message definitions are supported by the service, one can construct the client code. The business logic used to support the service is never directly exposed by it (as that would break one of the important principles of SOA methodology: namely, encapsulation). The client code is essentially the inverse of the service code:

```

ServiceInvoker flightService = new ServiceInvoker(...);
Message request = // create new Message of desired type

request.getBody().add("org.example.flight.seatnumber", 1);
request.getBody().add(" org.example.flight.flightnumber", "BA1234");

request.getHeader().getCall().setMessageID(1234);
request.getHeader().getCall().setReplyTo(myEPR);

Message response = null;

do
{
    response = flightService.deliverSync(request, 1000);

    if (response.getHeader().getCall().getRelatesTo() == 1234)
    {
        // it's out response!

        break;
    }
    else
        response = null; // and keep looping
} while maximumRetriesNotExceeded;

```

Example 6.4. `ServiceInvoker flightService = new ServiceInvoker`



### Note

Much of what has been outlined above will seem familiar to those readers who have worked with traditional client/server stub generators. In those systems, the low-level details (such as the opcodes and the parameters), would be hidden behind higher-level stub abstractions. In future releases of the JBoss Enterprise Service Bus, Red Hat intend to support such abstractions in order to simplify the development approach. When this happens, the ability to work with the raw Message components, such as the Body and Header, will be hidden from the majority of developers.

### 6.1.5. **Updated.** Configuring for a Remote Service Invoker

Using the **ServiceInvoker** from within the ESB's actions will work "out-of-the-box," with no additional configuration required. However, if one uses it from a remote Java Virtual Machine, (which will be the case for a stand-alone Java application, servlet, or an Enterprise Java Bean), one will need to ensure that the following JAR files are available:

- jbossesb-rosetta.jar
- jbossesb-config-model-1.0.1.jar
- jbossts-common.jar

- log4j-1.2.14.jar
- stax-1.2.0.jar
- stax-api-1.0.1.jar
- jbossall-client.jar
- scout-1.0rc2.aop.jar
- xbean-2.2.0.jar
- commons-logging-1.1.jar
- jboss-aop-jdk50-1.5.6.GA.jar
- javassist-3.6.0.GA.jar
- trove.jar
- juddi-client-2.0rc5.jar
- jboss-messaging-client.jar
- jboss-remoting.jar
- commons-codec-1.3.jar
- wstx-asl-3.2.0.jar
- xercesImpl-2.8.0.jar

One will also need to ensure that the following configuration file is on the classpath:

- **jbossesb-properties.xml**

### 6.1.6. Sample Client

The following Java program can be used to verify that a remote client's configuration is working correctly. It assumes that the **helloworld** Quick Start has been deployed and that the Enterprise Service Bus server is running.

```
package org.jboss.esb.client;

import org.jboss.soa.esb.client.ServiceInvoker;
import org.jboss.soa.esb.listeners.message.MessageDeliverException;
import org.jboss.soa.esb.message.Message;
import org.jboss.soa.esb.message.format.MessageFactory;

public class EsbClient
{
    public static void main(String[] args)
    {
        System.setProperty("javax.xml.registry.ConnectionFactoryClass",
            "org.apache.ws.scout.registry.ConnectionFactoryImpl");
        try
        {
            Message message = MessageFactory.getInstance().getMessage();
            message.getBody().add("Sample payload");
            ServiceInvoker invoker = new ServiceInvoker("FirstServiceESB",
"SimpleListener");
            invoker.deliverAsync(message);
        }
        catch (final MessageDeliverException e)
        {
            e.printStackTrace();
        }
    }
}
```

Example 6.5. Verify the Configuration of a Remote Client

### 6.1.7. Hints and Tips

One may find the following hints of use when developing one's clients and services:

- When developing Actions, ensure that any payload information specific to that Action is maintained in unique locations within the Message's Body.
- Try not to expose any back-end service implementation details within the Message, because this will make it difficult to change the implementation without affecting clients. Using Message definitions (contents, formats and so on) which are "implementation-agnostic" will help to maintain loose coupling.
- For stateless services, use the **ServiceInvoker** as it will handle fail-over "opaquely."
- When one is building request/response applications, one should use the correlation information (that is, MessageID and RelatesTo) within the Message Header.
- Consider using the Header Action field for the main service opcode.

- If one is using asynchronous interactions for which there are no delivery addresses for responses, one should consider sending any errors to the **MessageStore**. This is so that these errors can be monitored later.
- Until the JBoss ESB provides more automatic support for service contract definition and publication, one should consider maintaining a separate repository of these definitions that is available to both developers and users.



## Advanced Topics

This chapter looks at some of the more advanced concepts related to the JBoss Enterprise Service Bus.

### 7.1. Fail-Over and Load-Balancing Support

It is important have redundancy in mind when one is designing mission-critical systems. The JBoss Enterprise Service Bus includes built-in fail-over, load balancing and delayed message re-delivery, all of which will help one build a robust architecture. When one uses a SOA, it is implied that the Service has become the building unit. The JBoss Enterprise Service Bus allows one to replicate identical services across many nodes, whereby each node can be a virtual or physical machine running an instance of the ESB. The collective term for all these ESB instances is *The Bus*. Services within The Bus use different delivery channels to exchange messages. In ESB terminology, such a channel may be any one of JMS, FTP or HTTP. These different protocols are provided by systems external to the ESB, such as the JMS-provider, the FTP server and so forth. Services can be configured to listen to one or more protocols. For each protocol for which it is configured to listen, it creates an end-point reference in the Registry.

#### 7.1.1. Services, EPRs, Listeners and Actions

As previously discussed, within the `jboss-esb.xml` file, each service element consists of one or more listeners and one or more actions. The configuration fragment below is loosely based upon the configuration of the **JBossESBHelloWorld** example. When the service initializes, it registers the category, name and description in the UDDI Registry. Also, for each listener element, it will register a **ServiceBinding** to UDDI, in which it stores an EPR. In this case, it will register a JMSEPR for this service, as it is a `jms-listener`. The Java Message Service specifics, such as "queue name" are not shown. Rather, they appear at the top of the `jboss-esb.xml` file, which one can find the 'provider' section. In the `jms-listener`, one can simply reference the "quickstartEsbChannel" in the `busidref` attribute.

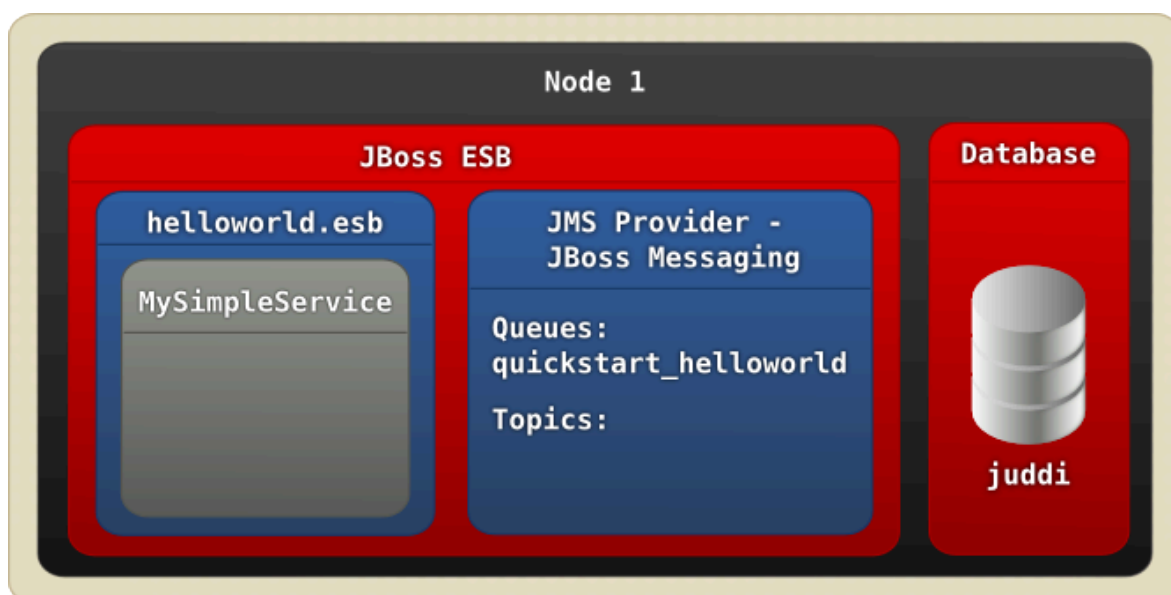


Figure 7.1. "Hello World" Quick Start Example, with One Service Instance on One Node.

```
...
<service category="FirstServiceESB" name="SimpleListener"
  description="Hello World">
  <listeners>
    <jms-listener name="helloWorld" busidref="quickstartEsbChannel"
      maxThreads="1"/>
  </listeners>
  <actions>
    <action name="action1"
      class="org.jboss.soa.esb.actions.SystemPrintln"/>
  </actions>
</service>
...
```

**Example 7.1. Hello World Quick Start example, (configuration fragment)**

Given the category and service names, another service can send a message to one's **"Hello World"** Service by looking it up in the Registry. By doing so, it will receive the JMSEPR and it can use that to send the message. All of this "heavy lifting" is undertaken in the **ServiceInvoker** class. When one's "Hello World" Service receives a message over the **quickstartEsbChannel**, it will hand it to the first action's process method in the Action Pipeline. This is the **SystemPrintln** action.



### Note

Because the **ServiceInvoker** hides much of the fail-over complexity from users, it, by necessity, only works with native ESB Messages. Additionally, not all gateways have been modified to use the **ServiceInvoker**. Therefore, incoming ESB-unaware messages for those gateway implementations may sometimes be unable to take advantage of service fail-over.

### 7.1.2. Replicated Services

In the example, the service is running on Node1. What happens if one were to simply take the **helloworld.esb** file and deploy it on Node2 as well (see the illustration below)? If one is using jUDDI as one's registry and have configured all of the nodes to access one central jUDDI database (as is recommended for clustered databases), then Node2 will find that the **FirstServiceESB - SimpleListener** Service is already registered. It will simply add a second **ServiceBinding** to this service. One will now have one's first replicated Service. If Node1 fails, Node2 will still continue to function.

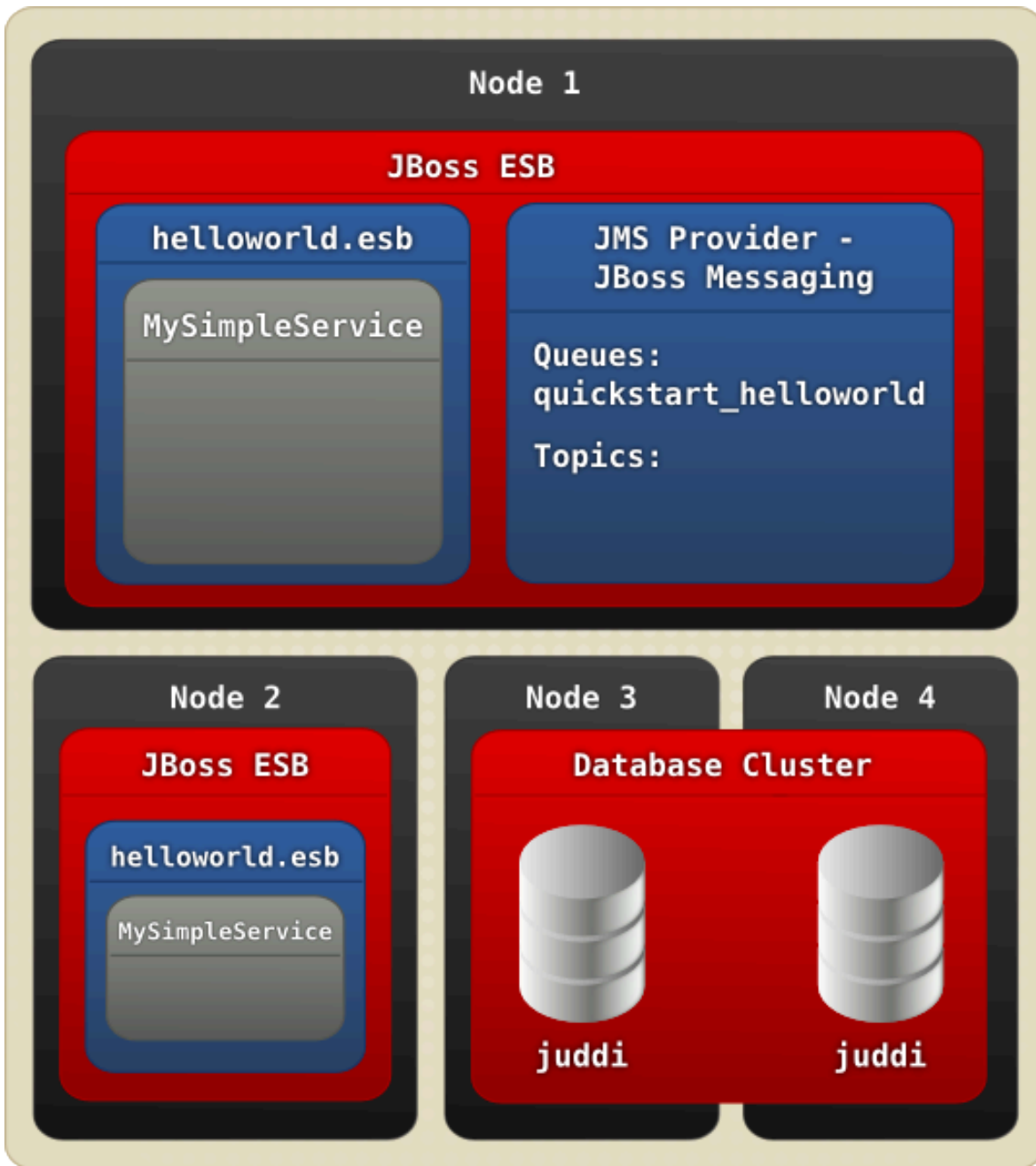


Figure 7.2. Two service instances each on a different node.

Load balancing has been achieved because both service instances listen to the same queue. However, this means that one will still have a single point of failure in one's configuration. This is where *Protocol Clustering* may be an option. It will be described in the next section.

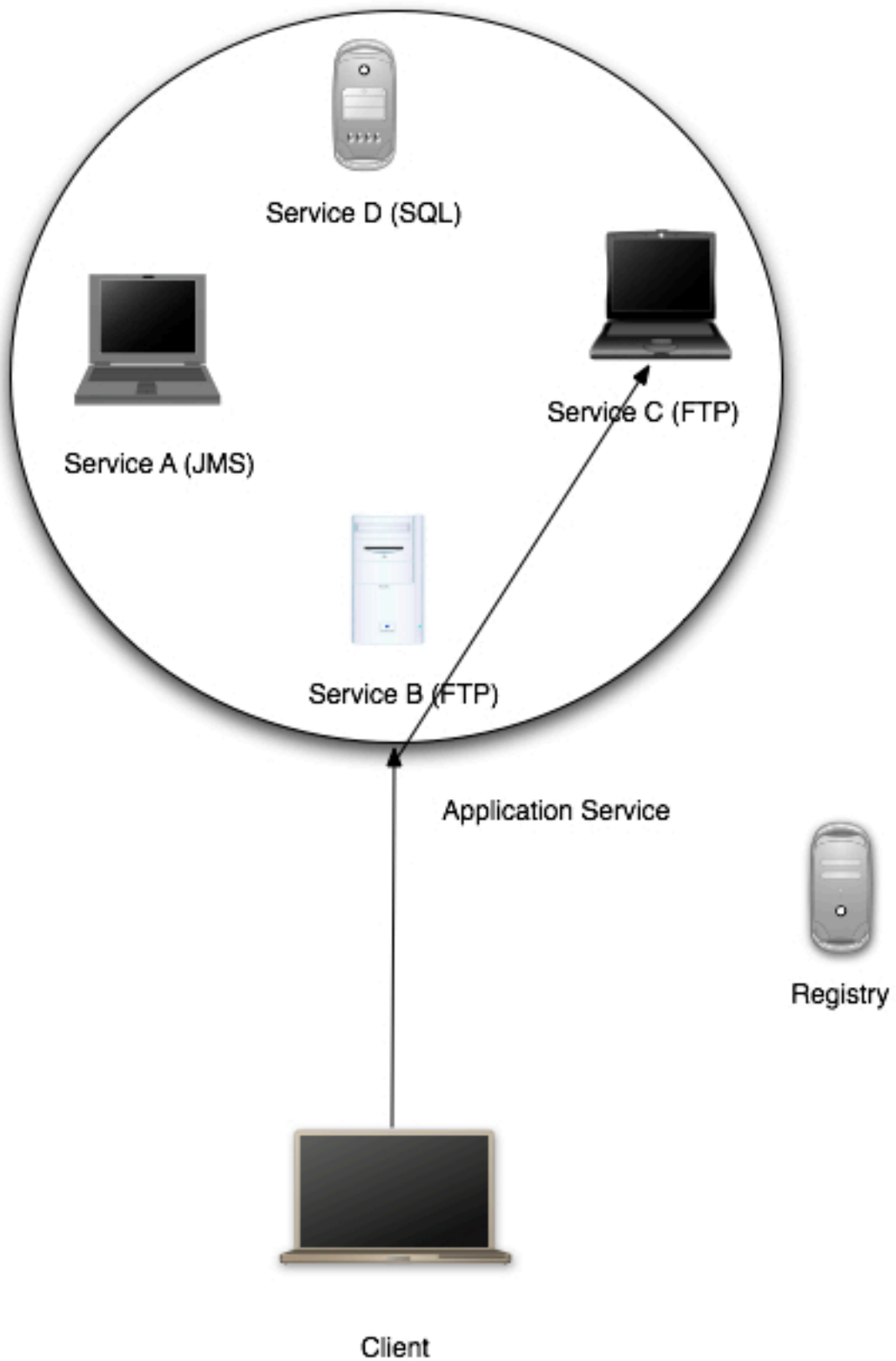
This type of replication can be used to either increase the availability of a service or to provide load balancing. To understand more fully, consider the diagram below, which has a logical service (an application service.) This logical service is actually comprised of four individual services, each of which provides the same capabilities and conforms to the same service contract. They differ only in that they do not need to share the same transport protocol. However, as far as the users of the application service are concerned, they see only a single service, which is identified by the service name and category. The **ServiceInvoker** hides the fact that the application service is actually composed of

these four other services from the clients by masking their failures. It will allow clients to "make forward progress" as long as at least one instance of the replicated service group remains available.



### Note

This type of replication should only be used for stateless services.



Although providers can replicate services independently of their consumers, in some circumstances the sender of a message will not want silent fail-over to occur. One needs to set a message

property called `org.jboss.soa.esb.exceptionOnDeliverFailure` to `true` in order to prevent automatic, silent fail-over. When this property has been set, the **ServiceInvoker** throws an **MessageDeliverException** instead of attempting to re-send the message. This property can be specified for all Messages by setting it in the Core section of the JBoss Enterprise Service Bus **property** file.

### 7.1.3. Protocol Clustering

Some Java Message Service providers can be clustered. **JBossMessaging** is one of these, which is why it was chosen as the JMS provider for the Enterprise Service Bus. When one clusters this JMS, one remove a single point of failure from one's architecture (see the next diagram.)

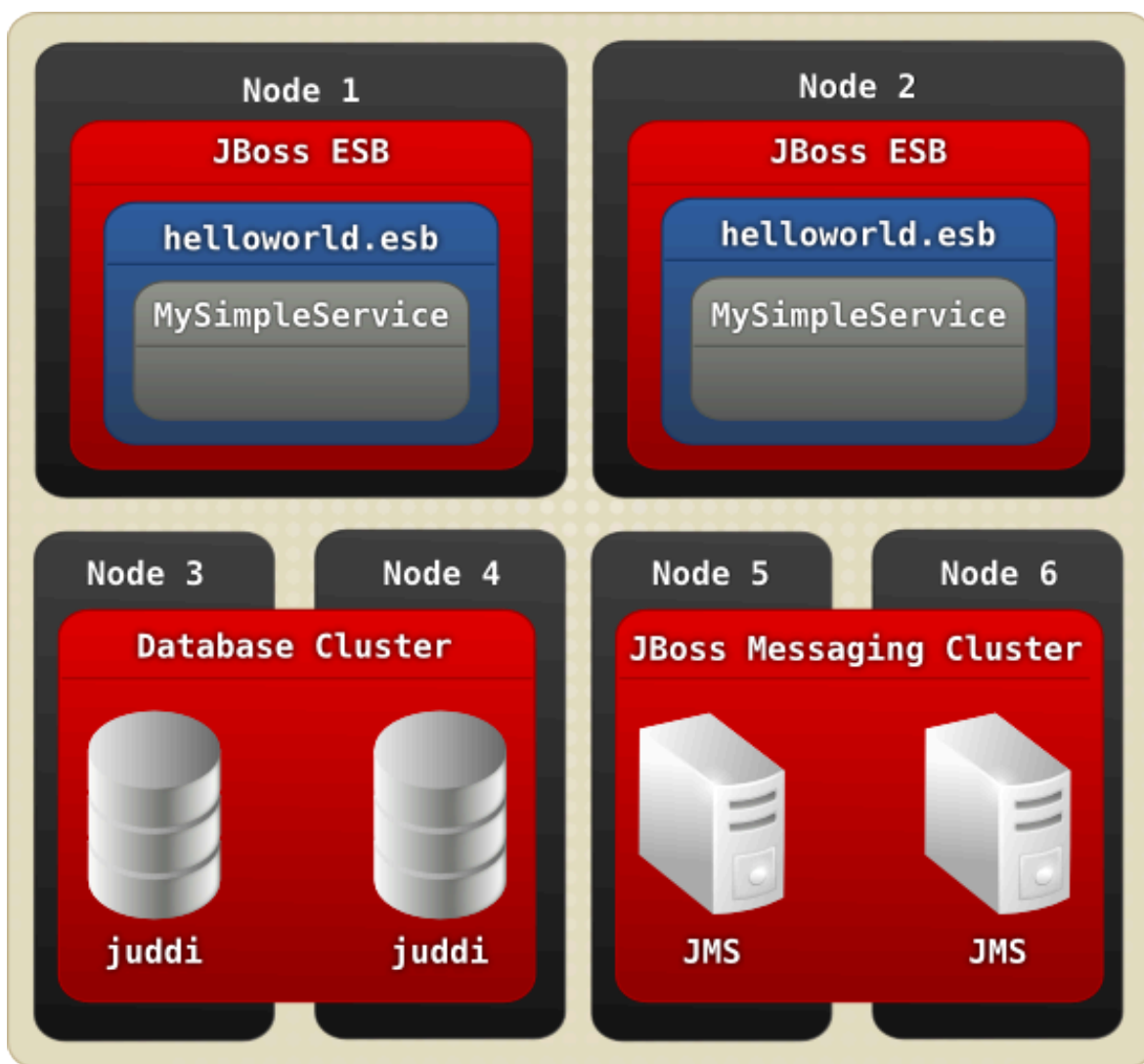
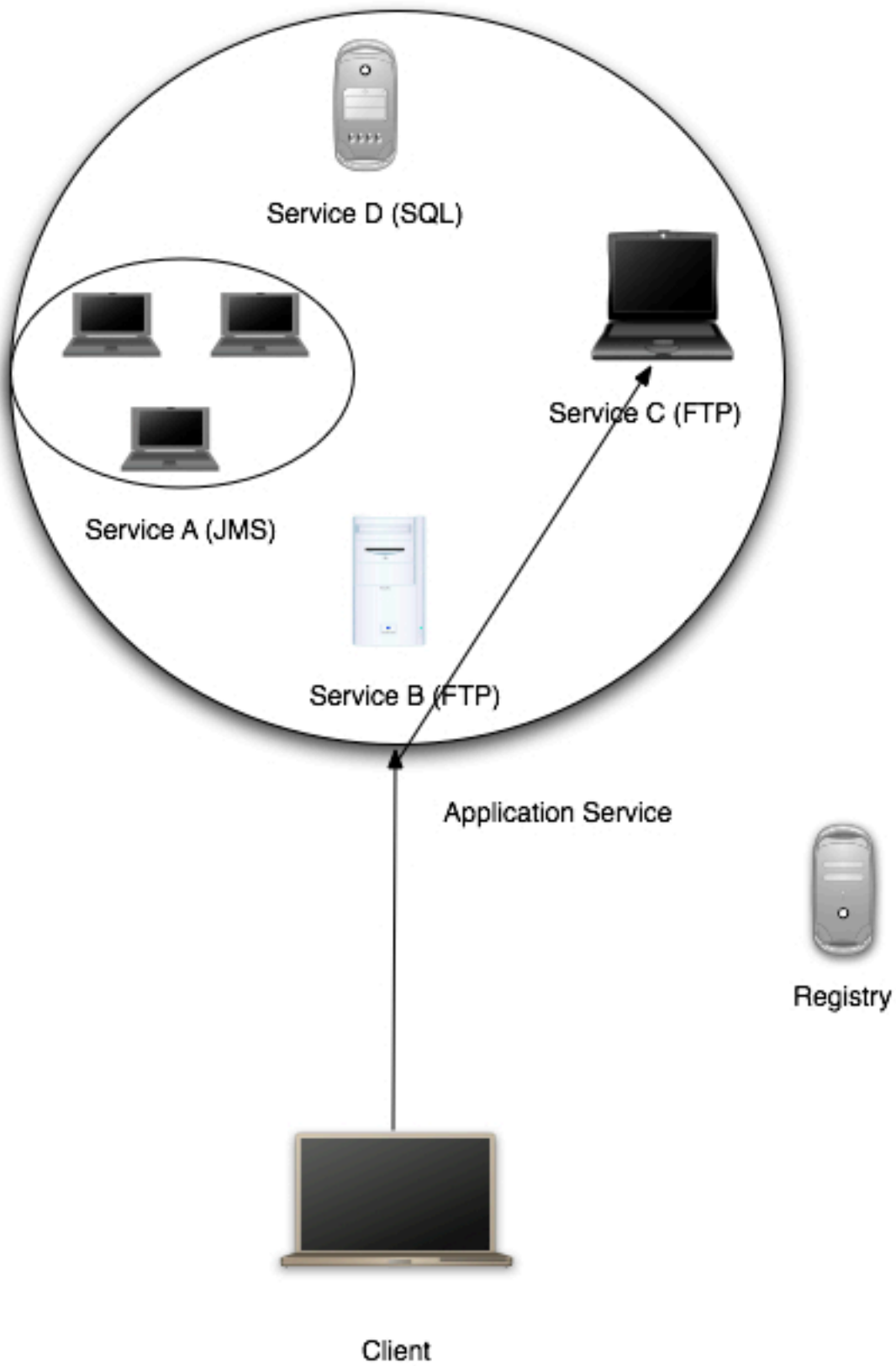


Figure 7.3. Example of Protocol Clustering Using JMS

One should read the documentation on "Clustering for JBossMessaging," if one wants to enable this functionality. Both JBoss ESB replication and JMS clustering can be used together, as shown in the following illustration. In this example, Service A is identified in the registry by a single JMS end-point reference. However, the EPR points to a clustered JMS queue, which has been separately configured to support three services. The client cannot see this. This is known as a "federated" approach to availability and load balancing. In fact, masking the replication of services from users is in accordance

with SOA principles, which dictate that one should hide the implementation details behind the service end-point and not expose them at the contract level. (This masking of replication is done to the client in the case of the JBoss ESB replication approach, and to the JBoss ESB itself in the case of the Java Message Service clustering.)







### Note

If one is using JMS clustering in this way, one will obviously need to ensure that the configuration is correct. For instance, if one place all of one's ESB services within a JMS cluster, there could be no expected benefit from replication.

Another examples of Protocol Clustering is a NAS (*Network Attached Storage*) for the **FileSystem** protocol, but what if one's provider simply cannot provide any clustering? In that case, you can add multiple listeners to your service and use multiple (JMS) providers. However, this will require fail-over and load-balancing across providers. This is discussed in the next section.

### 7.1.4. Clustering

If you would like to run the same service on more than one node in a cluster, you have to wait for service registry cache re-validation. This must occur before the service is fully working in the clustered environment. You can setup this cache re-validation timeout in **deploy/jbossesb.sar/jbossesb-properties.xml**:

```
<properties name="core">
<property name="org.jboss.soa.esb.registry.cache.life" value="60000"/>
</properties>
```

60 seconds is the default timeout.

### 7.1.5. Channel Fail-over and Load Balancing

The **HelloWorld** Service can listen to more than one protocol. An extra JMS channel has been added here:

```
...
<service category="FirstServiceESB" name="SimpleListener"
description="Hello World">
  <listeners>
    <jms-listener name="helloWorld" busidref="quickstartEsbChannel"
maxThreads="1"/>
    <jms-listener name="helloWorld2" busidref="quickstartFtpChannel2"
maxThreads="1"/>
  </listeners>
...

```

Now your Service is simultaneously listening to two JMS queues. These queues can be provided by JMS providers on different physical boxes. Therefore, you now have a redundant JMS connection between two services. You can even mix protocols in this setup, so you can also add an ftp-listener to the mix.

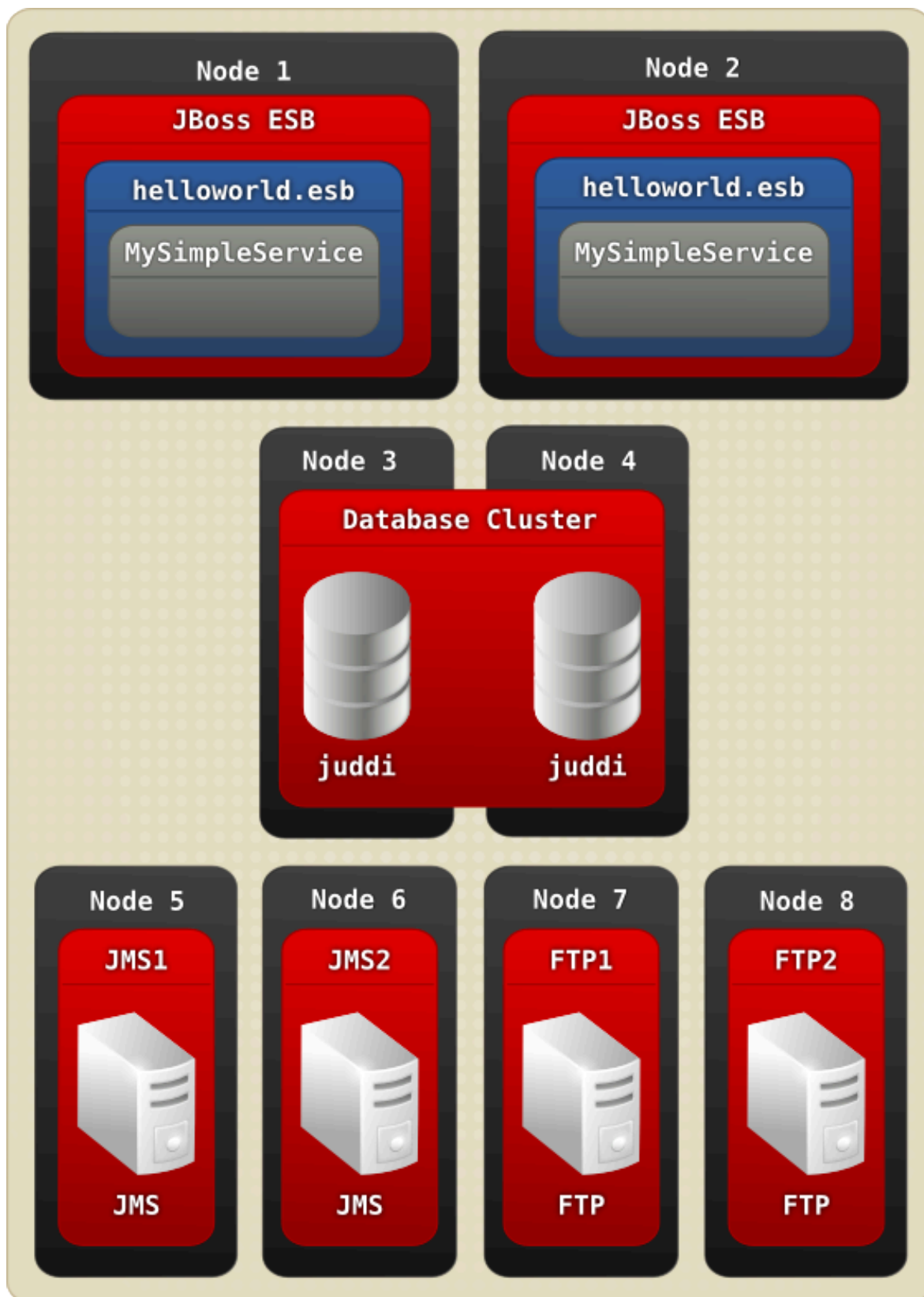


Figure 7.4. Adding two FTP servers to the mix.

```

...
<service category="FirstServiceESB" name="SimpleListener"
  description="Hello World">
  <listeners>
    <jms-listener name="helloWorld" busidref="quickstartEsbChannel"
      maxThreads="1"/>
    <jms-listener name="helloWorld2" busidref="quickstartJmsChannel2"
      maxThreads="1"/>
    <ftp-listener name="helloWorld3" busidref="quickstartFtpChannel3"
      maxThreads="1"/>
    <ftp-listener name="helloWorld4" busidref="quickstartFtpChannel3"
      maxThreads="1"/>
  </listeners>
...

```

When the **ServiceInvoker** tries to deliver a message to our Service, it will receive a choice of eight EPRs now (four each from Node1 and Node2.) How will it decide which one to use? For that, you can configure a Policy. In the **jbossesb-properties.xml**, you can set the 'org.jboss.soa.esb.loadbalancer.policy'. Right now three Policies are provided. You can also create your own. Here are the three provided policies:

- First Available. If a healthy ServiceBinding is found, then it will be used unless it dies. It will move to the next EPR in the list. This Policy does not provide any load balancing between the two service instances.
- Round Robin. This is typical Load Balance Policy where each EPR is hit in order of the list.
- Random Robin. Like the Round Robin but then random.

The EPR list with which the Policy works may become smaller over time as dead EPRs are removed from the (cached) list. When the list is emptied or the "time-to-live" of the cache is exceeded, the ServiceInvoker will obtain a fresh list of EPRs from the Registry. The **org.jboss.soa.esb.registry.cache.life** can be set in the jbossesb-properties file. Its default is 60,000 milliseconds. If you find that none of the EPRs work, you may use the Message Redelivery Service.

### 7.1.6. Updated Message Redelivery

If the list contains nothing but dead EPRs, the ServiceInvoker can do one of two things:

- If you are trying to deliver the message synchronously, it will be sent to the DeadLetterService, which, by default, will store to the DLQ MessageStore. It will then send a failure back to the caller and processing will stop. Note that you can configure the DeadLetterService in the jbossesb.esb if, for instance, you want it to go to a JMS queue or if you want to receive a notification.
- If you are trying to deliver the message asynchronously (as is recommended), it, too, will send the message to the DeadLetterService but the message will be stored in the RDLVR MessageStore. The Redeliver Service (jbossesb.esb) will keep retrying to send the message until the maximum number of redelivery attempts is exceeded. In that case, the message will be stored in the DLQ MessageStore and processing will stop.

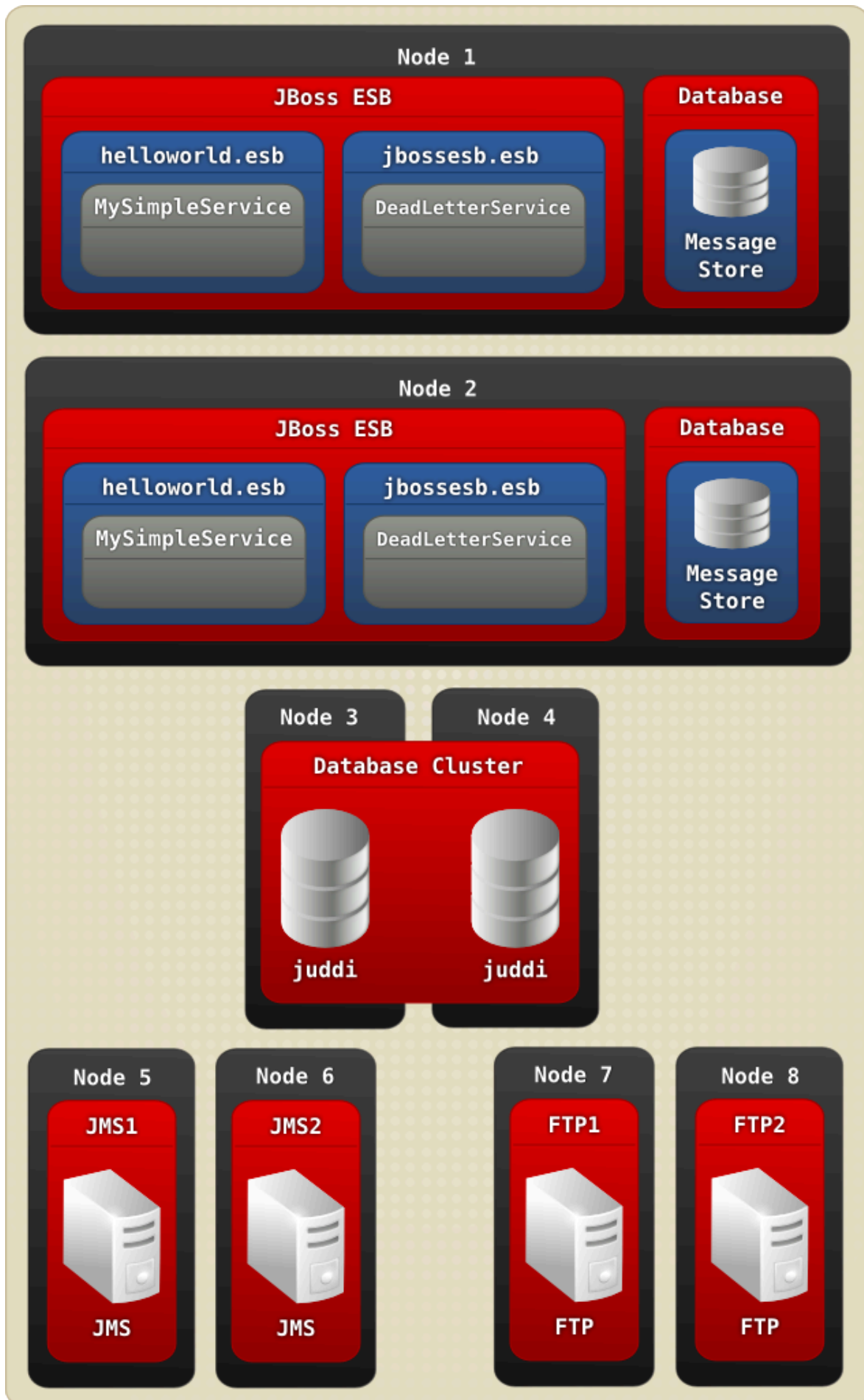


Figure 7.5. Message Re-delivery



### Note

The `DeadLetterService` is turned on by default. However in the `jbossesb-properties.xml`, you can set `org.jboss.soa.esb.dls.redeliver` to "false" to turn it off. If you want to control this on a per-message basis, set the `org.jboss.soa.esb.dls.redeliver` property in the specific Message's properties accordingly. The Message property will be used in preference to any global setting. The default is to use the value set in the configuration file.

## 7.2. Scheduling of Services

JBoss ESB 5.0 supports two types of providers:

1. Bus Providers, which supply messages to action processing pipelines via messaging protocols such as JMS and HTTP. This type is "triggered" by the underlying messaging provider.
2. Schedule Providers, which supply messages to action processing pipelines based on a schedule-driven model. (In other words, where the underlying message delivery mechanism, such as the file system, offers no support for triggering the ESB when messages are available for processing, a scheduler periodically triggers the listener to check for new messages.)

Scheduling is new to the JBoss ESB and not all of the listeners have been migrated over to this model yet.

JBoss ESB 5.0 offers a `<schedule-listener>` as well as 2 `<schedule-provider>` types: `<simple-schedule>` and `<cron-schedule>`. The `<schedule-listener>` is configured with a "composer" class, which is an implementation of the `org.jboss.soa.esb.listeners.ScheduledEventMessageComposer` interface.

### 7.2.1. Simple Schedule

This schedule type provides a simple capability based on the following attributes:

`scheduleid`

A unique identifier string for the schedule. Used to reference a schedule from a listener.

`frequency`

The frequency (in seconds) with which all schedule listeners should be triggered.

`execCount`

The number of times the schedule should be executed.

`startDate`

The schedule start date and time. The format of this attribute value is that of the XML Schema type "dateTime". See `dateTime`.

`endDate`

The schedule end date and time. The format of this attribute value is that of the XML Schema type "dateTime". See `dateTime`.

Example:

```
<providers>
  <schedule-provider name="schedule">
    <simple-schedule scheduleid="1-sec-trigger" frequency="1" execCount="5" /
  >
    </schedule-provider>
</providers>
```

### 7.2.2. Cron Schedule

This schedule type provides scheduling capability based on a CRON expression. The attributes for this schedule type are as follows:

**scheduleid**

A unique identifier string for the schedule. Used to reference a schedule from a listener

**cronExpression**

CRON expression

**startDate**

The schedule start date and time. The format of this attribute value is that of the XML Schema type “dateTime”. See `dateTime`.

**endDate**

The schedule end date and time. The format of this attribute value is that of the XML Schema type “dateTime”. See `dateTime`.

Example:

```
<providers>
  <schedule-provider name="schedule">
    <cron-schedule scheduleid="cron-trigger" cronExpression="0/1 * * * * ?" /
  >
    </schedule-provider>
</providers>
```

### 7.2.3. Scheduled Listener

The `<scheduled-listener>` can be used to perform scheduled tasks based on a `<simple-schedule>` or `<cron-schedule>` configuration.

It's configured with an **event-processor** class, which can be an implementation of one of **`org.jboss.soa.esb.schedule.ScheduledEventListener`** or **`org.jboss.soa.esb.listeners.ScheduledEventMessageComposer`**.

**ScheduledEventListener**

Event Processors that implement this interface are simply triggered through the “onSchedule” method. No action processing pipeline is executed.

### ScheduledEventMessageComposer

Event Processors that implement this interface are capable of “composing” a message for the action processing pipeline associated with the listener.

The attributes of this listener are:

1. name  
The name of the listener instance
2. event-processor  
The event processor class that is called on every schedule trigger. See above for implementation details.
3. One of:
  - scheduleidref  
The scheduleid of the schedule to use for triggering this listener.
  - schedule-frequency  
Schedule frequency (in seconds). A convenient way of specifying a simple schedule directly on the listener.

## 7.2.4. Example Configurations

The following is an example configuration involving the <scheduled-listener> and the <cron-schedule>.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/
etc/schemas/xml/jbossesb-1.0.1.xsd">

  <providers>
    <schedule-provider name="schedule">
      <cron-schedule scheduleid="cron-trigger" cronExpression="0/1 *
* * * ?" />
    </schedule-provider>
  </providers>

  <services>
    <service category="ServiceCat" name="ServiceName" description="Test
Service">

      <listeners>
        <scheduled-listener name="cron-schedule-listener"
scheduleidref="cron-trigger"
event-processor="org.jboss.soa.esb.schedule.MockScheduledEventMessageComposer" />
      </listeners>

      <actions>
        <action name="action"
class="org.jboss.soa.esb.mock.MockAction" />
      </actions>
    </service>
  </services>
</jbossesb>
```

```
        </actions>
    </service>
</services>

</jbossesb>
```

### 7.2.5. **Updated.** Quartz Scheduler Property Configuration

The Scheduling functionality in **JBossESB** is built on top of the Quartz Scheduler. The default **Quartz** Scheduler instance configuration used by **JBossESB** is as follows:

```
org.quartz.scheduler.instanceName = DefaultQuartzScheduler
org.quartz.scheduler.rmi.export = false
org.quartz.scheduler.rmi.proxy = false
org.quartz.scheduler.wrapJobExecutionInUserTransaction = false
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 2
org.quartz.threadPool.threadPriority = 5
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread
= true
org.quartz.jobStore.misfireThreshold = 60000
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

Any of these Scheduler configurations can be overridden and/or new ones can be added. You can do this simply by specifying the configuration directly on the `<schedule-provider>` configuration as a `<property>` element. For example, if you wish to increase the thread pool size to five:

```
<schedule-provider name="schedule">
    <property name="org.quartz.threadPool.threadCount" value="5" />
    <cron-schedule scheduleid="cron-trigger" cronExpression="0/1 * * *
* ?" />
</schedule-provider>
```



# Fault-Tolerance and Reliability

This chapter provides a study of the JBoss Enterprise Service Bus' reliability characteristics. The reader will learn about which failure modes he or she can expect to find "tolerated" within this release. This chapter will also provide advice on how one can improve the fault tolerance of one's applications. However, in order to proceed, some important terms must first be defined. If one already has a good knowledge of the topic and wishes to skip ahead of the introductory material, go to [Section 8.2, "Reliability Guarantees"](#).

*Dependability* is defined as "the trustworthiness of a component such that reliance can be justifiably placed on the service (the behavior as perceived by a user) it delivers." The reliability of a component is a measure of its continuous correct service delivery. A failure occurs when the service provided by the system no longer complies with its specification. An error is "that part of a system state which is liable to lead to failure" and a fault is defined as "the cause of an error."

A *fault-tolerant* system is one "which is designed to fulfill its specified purpose despite the occurrence of component failures." Techniques for providing fault-tolerance usually require mechanisms for consistent state recovery mechanisms, and detecting errors produced by faulty components. A number of fault-tolerance techniques exist, including replication and transactions.

## 8.1. Failure classification

It is necessary to formally describe the behavior of a system before the correctness of applications running on it can be demonstrated. This process establishes behavioral restrictions for applications. It also clarifies the implications of weakening or strengthening these restrictions.

Categorizing system components according to the types of faults they are assumed to exhibit is a recommended method of building such a formal description with respect to fault-tolerance.

Each component in the system has a specification of its correct behavior for a given set of inputs. A correctly working component will produce an output that is in accordance with this specification. The response from a faulty component need not be as specific. The response from a component for a given input will be considered correct if the output value is both correct and produced within a specified time limit.

Four possible classifications of failures are: Omission, value, timing, and arbitrary.

### Omission fault/failure

A component that does not respond to an input from another component and, thereby, fails by not producing the expected output is exhibiting an *omission fault*. The corresponding failure is an *omission failure*. A communication link which occasionally loses messages is an example of a component suffering from an omission fault.

### Value fault/failure

A fault that causes a component to respond within the correct time interval but with an incorrect value is termed a *value fault* (with the corresponding failure called a *value failure*). A communication link which delivers corrupted messages on time suffers from a value fault.

### Timing fault/failure

A timing fault causes the component to respond with the correct value but outside the specified interval (either too soon or too late). The corresponding failure is a *timing failure*. An overloaded processor which produces correct values but with an excessive delay suffers from a timing failure. Timing failures can only occur in systems which impose timing constraints on computations.

### Arbitrary fault/failure

The previous failure classes have specified how a component can be considered to fail in either the value or time domains. It is possible for a component to fail in both of these domains in a manner which is not covered by one of the previous classes. A failed component which produces such an output will be said to be exhibiting an *arbitrary failure* (*Byzantine failure*).

An arbitrary fault causes any violation of a component's specified behavior. All other fault types preclude certain types of fault behavior, the omission fault type being the most restrictive. Thus the omission and arbitrary faults represent two ends of a fault classification spectrum, with the other fault types placed in between. The latter failure classifications thus subsume the characteristics of those classes before them, e.g., omission faults (failures) can be treated as a special case of value, and timing faults (failures). Such ordering can be represented as a hierarchy:

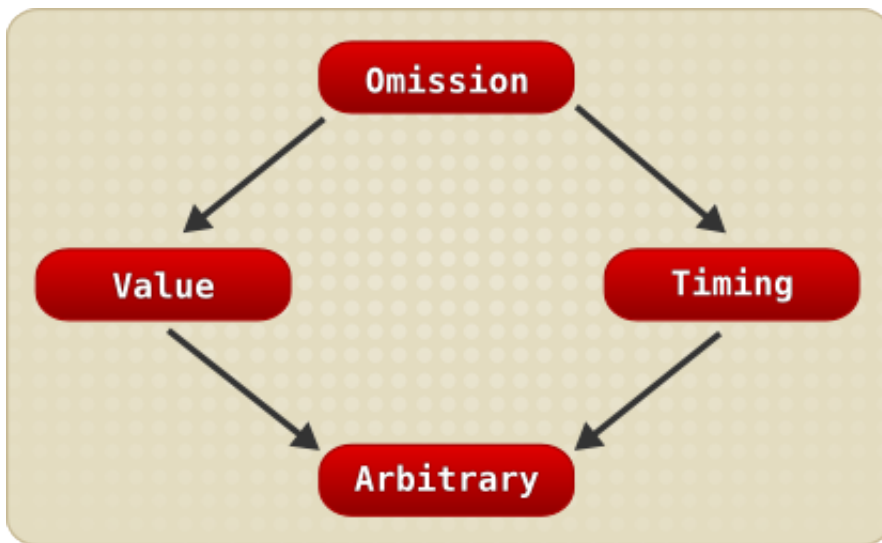


Figure 8.1. Fault classification hierarchy

### 8.1.1. JBossESB and the Fault Models

Within JBossESB there is nothing that will allow it to tolerate arbitrary failures. As you can probably imagine, these are extremely difficult failures to detect due to their nature. Protocols do exist to allow systems to tolerate arbitrary failures, but they often require multi-phase coordination or digital signatures. Future releases of JBossESB may incorporate support for some of these approaches.

Because value, timing and omission failures often require semantic information concerning the application (or specific operations), there is only so much that JBossESB can do directly to assist with these types of faults. However, by correct use of JBossESB capabilities (such as `RelatesTo` and `MessageID` within the Message header), it is possible for applications to determine, for example, whether or not a received Message is either the one for which they are waiting or a delayed Message. Unfortunately, Messages that are provided too soon by a service, (such as asynchronous one-way responses to one-way requests), may be lost due to the underlying transport implementation. For instance, if you are using a protocol such as HTTP there is a finite buffer (set at the operating system level) within which responses can be held before they are passed on to the application. If this buffer is exceeded, then information within it may be lost in favor of new Messages. Transports such as FTP or SQL do not necessarily suffer from this specific limitation but may exhibit other resource restrictions that can result in the same behavior.

Tolerating Messages that are delayed is sometimes easier than tolerating those that arrive too early. However, from an application perspective, if an early Message is lost (by a buffer overflow

for example), it is not possible to distinguish it from one that is infinitely delayed. Therefore, if you construct your applications (both consumers and services) to use a retry mechanism in the case of lost Messages, timing and omission failures should then be tolerated. However, there is following exception: When your consumer picks up an early response out of order and incorrectly processes it, resulting in a value failure. Fortunately, if you use `RelatesTo` and `MessageID` within the Message header, you can spot incorrect Message sequences without having to process the entire payload (which is obviously another option available to you).

If a response has not been received within a finite period of time within a synchronous request-response interaction pattern, many systems built upon RPC will automatically resend the request. Unfortunately, at present, JBossESB does not do this and you will have to use the timeout mechanism within `Couriers` or `ServiceInvoker` to determine when (and whether) to send the Message again. As described in the Advanced Chapter, it will retransmit the Message if it suspects a failure of the service that would affect Message delivery has occurred.



### Note

You should use care when retransmitting Messages to services. JBossESB currently has no notion of retained results and cannot detect retransmissions within the service, so any duplicate Messages will be delivered to automatically. This may mean that your service receives the same Message multiple times (if, for example, it was the initial service response that became lost rather than the initial request. As such, your service may attempt to perform the same work.) If using re-transmission (either explicitly or through the `ServiceInvoker` fail-over mechanisms), you will have to deal with multiple requests within your service to ensure it is idempotent.

The use of transactions, (such as those provided by JBossTS), and replication protocols (as provided by systems like JGroups), can help with regard to failure toleration. Furthermore, (in the case where forward progress is not possible because of a failure), by using transactions the application can roll back. The underlying transaction system will then guarantee data consistency such that it will appear as though the work was never attempted. At present, when deployed within the JBoss Application Server, JBossESB offers transactional support through JBossTS.

## 8.1.2. Failure Detectors and Failure Suspectors

An ideal failure detector is one which can allow for the unambiguous determination of the liveness of an "entity" within a distributed system (where an entity may be a process, machine and so on). However, guaranteed detection of failures in a finite period of time is not possible because one cannot differentiate between a failed system and one which is simply slow in responding.

Current failure detectors use timeout values to determine the availability of entities. For example, if a machine does not respond to an "Are-you-alive?" message within a specified time period, it is assumed to have failed. If the values assigned to such timeouts are wrong (because of a cause such as network congestion), incorrect failures may be assumed. This may potentially lead to inconsistencies, whereby some machines "detect" the failure of another, whilst others do not. Therefore, such timeouts are typically assigned based upon an assumption of the worst case scenario within the distributed environment in which they are to be used (for example, worst case network congestion and machine load.) However, distributed systems and applications rarely perform exactly as expected from one execution to another. Therefore, fluctuations from the worst case assumptions are possible, and there is always a finite probability of making an incorrect failure detection decision.

Guaranteed failure detection is not possible. However, known active entities can communicate with each other and agree that an unreachable entity may have failed. This is the work of a *failure suspector*. If one entity assumes another has failed, a protocol is executed between the remaining entities to decide or agree that it has failed or not. If it is agreed that the entity has failed, then it is excluded from the system and no further work by it will be accepted. The fact that one entity thinks it has failed does not mean that all entities will reach the same decision. If the entity has not failed and is excluded, then it must execute another protocol to be recognized as alive.

The advantage of the **failure suspector** is that all correctly functioning entities within the distributed environment will agree upon the state of an entity suspected to have failed. The disadvantage is that such failure-suspecting protocols are heavy-weight, and typically require several rounds to reach agreement. In addition, since suspected failure is still based upon timeout values, it is possible for entities which have not actually failed to be excluded, thus reducing (possibly critical) resource utilization and availability.

Some applications can tolerate the fact that failure detection mechanisms may occasionally return an incorrect answer. However, for others, the incorrect determination of the liveness of an entity may lead to problems such as data corruption, or in the case of mission-critical applications (such as aircraft control systems or nuclear reactor monitoring) could result in the loss of life.

At present, JBossESB does not support failure detectors or failure suspectors. This shortcoming will hopefully be addressed in a future release. For now, you should develop your consumers and services using the techniques previously mentioned (such as MessageID and time-out/retry) in order to attempt to determine whether or not a given service has failed. In some situations it is actually better and more efficient for the application to detect and deal with suspected failures.

## 8.2. Reliability Guarantees

As you have seen, there are a range of ways in which failures can happen within a distributed system. In this section, you will see concrete examples of how failures could affect JBossESB and applications deployed on it. In the section on Recommendations ways in which you can configure JBossESB to better tolerate these faults will be covered, along with advice on how you should approach your application development.

There are many components and services within JBossESB. The failure of some of them may go unnoticed to some or all of your applications depending upon when the failure occurs. For example, if the Registry Service crashes after your consumer has successfully obtained all necessary EPR information for the services it needs in order to function, then the crash will have no adverse affect on your application. However, if it fails before this point, your application will not be able to make forward progress. Therefore, in any determination of reliability guarantees, it is necessary to consider when failures occur as well as the types of those failures.

It is never possible to guarantee 100% reliability and fault tolerance. Hardware failure and human error is inevitable. However, you can ensure, with a high degree of probability, that a system will tolerate failures, maintain data consistency and make forward progress. Fault-tolerance techniques, such as transactions or replication, always comes at a cost to performance. This trade-off between performance and fault-tolerance is best achieved with knowledge of the application. Attempting to uniformly impose a specific approach to all applications inevitably leads to poorer performance in situations where it was not necessary. As such, you will find that many of the fault-tolerance techniques supported by JBossESB are disabled by default. You should enable them only when it makes sense to do so.

### 8.2.1. Message Loss

You have previously read how message loss or delay may adversely affect applications. You have also seen some examples of how messages could be lost within JBossESB. In this section message loss will be discussed in more detail.

Many distributed systems support reliable message delivery, either point-to-point (one consumer and one provider) or group-based (many consumers and one provider). Even in the presence of failures, the semantics imposed on reliability are typically such that the message will be delivered or the sender will be able to know with certainty that it failed to reach the receiver. It is frequently the case that systems which employ reliable messaging implementations distinguish between a message being delivered to the recipient and it subsequently being processed by the recipient. For instance, simply sending the message to a service does not mean much if a subsequent crash of that same service occurs before it has had time to work on the contents of the message.

Within JBossESB, the only transport you can use which gives the aforementioned failure semantics on Message delivery and processing, is JMS. If you use transacted sessions, (an optional part of the **JMSEpr**), it is possible to guarantee that Messages are received and processed in the presence of failures. If a failure occurs during processing by the service, the Message will be placed back on the JMS queue for later re-processing. However, transacted sessions can be significantly slower than non-transacted sessions and so should be used with caution.

Because none of the other transports supported by JBossESB come with transactional or reliable delivery guarantees, it is possible for Messages to be lost. However, in most situations, the likelihood of this occurring is small. Unless there is a simultaneous failure of both sender and receiver (possible but not probable), the sender will be informed by JBossESB about any failure to deliver the Message. If a failure of the receiver occurs whilst processing and a response was expected, then the receiver will eventually time-out and can retry.



#### Note

Using asynchronous message delivery can make failure detection/suspicion difficult (indeed, theoretically impossible to achieve). You should consider this aspect when developing your applications.

For these reasons, the Message fail-over and re-delivery protocol that was described in the Advanced Chapter is a good best-effort approach. If a failure of the service is suspected, then it will select an alternative EPR (assuming one is available,) and use it. However, if this suspicion of failure is wrong, then it is possible that multiple services will operate on the same Message concurrently. Therefore, although it offers a more robust approach to fail-over, it should be used with care. It works best where your services are stateless and idempotent, (in other words, the execution of the same message multiple times is the same as executing it once.)

For many services and applications, this type of re-delivery mechanism is fine. The robustness it provides over a single EPR can be a significant advantage. The failure modes where it does not work, (in other words, where the client and service fails or the service is incorrectly assumed to have failed), are relatively uncommon. If your services cannot be idempotent, then until such a time as JBossESB supports transactional delivery of messages or some form of retained results, you should either use JMS or code your services to be able to detect re-transmissions and cope with multiple services performing the same work concurrently.

### 8.2.2. Suspecting Endpoint Failures

It was described earlier how failure detection/suspicion is difficult to achieve. In fact, until a failed machine recovers, it is not possible to determine the difference between a crashed machine or one that is simply running extremely slowly. Networks can also become partitioned: This is a situation where the network becomes divided, and effectively acts as two or more separate networks. When this happens, consumers on different parts of the network can only see the services available in their own part. This is sometimes called "split-brain syndrome".

### 8.2.3. Supported Crash Failure Modes

JBossESB is able to recover from a catastrophic failure that shuts down the entire system when you use either transactions or a reliable message delivery protocol such as JMS.

Without these, JBossESB can only tolerate failures when the availability of the endpoints involved is guaranteed.

### 8.2.4. Component Specifics

In this section we shall look at specific components and services within JBossESB.

### 8.2.5. Gateways

Once a message is accepted by a Gateway it will not be lost unless sent within the ESB using an unreliable transport. JMS, FTP and SQL are JBossESB transports can be configured to either reliably deliver the Message or ensure it is not removed from the system. Unfortunately, HTTP cannot be configured in this way.

### 8.2.6. ServiceInvoker

The ServiceInvoker will place undeliverable Messages in the Re-delivery Queue if they have been sent asynchronously. Synchronous Message delivery that fails will be indicated to the sender immediately. In order for the ServiceInvoker to function correctly, the transport must indicate a failure to deliver to the sender unambiguously. A simultaneous failure of the sender and receiver may result in the Message being lost.

### 8.2.7. JMS Broker

Messages that cannot be delivered to the JMS broker will be placed within the Redelivery Queue. For enterprise deployments, a clustered JMS broker is recommended.

### 8.2.8. Action Pipelining

It is important to differentiate between a Message being received by the container within which services reside and it being processed by the ultimate destination. It is possible for Messages to be delivered successfully, only for an error or crash during processing within the Action pipeline to cause its loss. As mentioned previously, it is possible to configure some of the JBossESB transports so that they do not delete received Messages when they are processed. This is in order that they will not be lost in the event of an error or crash.

## 8.3. Recommendations

Given the previous overview of failure models and the capabilities within JBossESB to tolerate them, the following recommendations can be made:

- Try to develop stateless and idempotent services. If this is not possible, use MessageID to identify Messages so that your application can detect re-transmission attempts. If retrying Message transmission, use the same MessageID. Services that are not idempotent (and would suffer from re-doing the same work if they receive a re-transmitted Message), should record state transitions against the MessageID, preferably using transactions. Furthermore, applications based around stateless services tend to scale better.
- If developing stateful services, use transactions and a (preferably clustered) JMS implementation.
- Cluster your Registry and use a clustered/fault-tolerant back-end database, in order to remove any single points of failure.
- Ensure that the Message Store is backed by a highly-available database.
- Clearly identify which services and which operations on these services need higher reliability and fault tolerance capabilities than others. This will allow you to target transports other than JMS at those services, and thereby potentially improve the overall performance of applications. Because JBossESB allows services to be used through different EPRs concurrently, it is also possible to offer these different qualities of service (QoS) to different consumers, based upon application-specific requirements.
- Because network partitions can make services appear as though they have failed, avoid transports that are more prone to this for services that cannot cope with being mis-identified as having crashed.
- In some situations (for example, when using HTTP) the crash of a server after it has dealt with a message but prior to responding, could result in another server doing the same work because it is not possible to differentiate between a machine that fails after the service receives the message and process it, and one where it receives the message and does not process it.
- Using asynchronous (one-way) delivery patterns will make it difficult to detect failures of services; there is typically no notion of a lost or delayed Message if responses to requests can come at arbitrary times. If there are no responses at all, then it obviously makes failure-detection more problematic and you may have to rely upon application semantics to determine that Messages did not arrive. An example of such a semantic might be a case where the amount of money in the bank account does not match expectations.) When using either the ServiceInvoker or Couriers to deliver asynchronous Messages, a return from the respective operation (such as deliverAsync) does not mean the Message has been acted upon by the service.
- The Message Store is used by the re-delivery protocol. However, as mentioned previously, this is a best-effort protocol for improved robustness and does not use transactions or reliable message delivery. This means that certain failures may result in Messages being lost entirely (if they are not written to the store before a crash) or delivered multiple times (if the re-delivery mechanism pulls a Message from the store, delivers it successfully but there is then a crash that prevents the Message from being removed from the store; upon recovery, the Message will be delivered again).
- Some transports, such as FTP, can be configured to retain Messages that have been processed, although they will be uniquely marked to differentiate them from unprocessed Messages. The default approach is often to delete Messages once they have been processed, but you may want to

change this default to allow your applications to determine which Messages have been dealt with upon recovery from failures.

Despite the impression that you may have gained from this Chapter, failures are uncommon. Over the years, hardware reliability has improved significantly and good software development practices (including the use of formal verification tools) have reduced the chances of software problems. We have given the information within this Chapter to assist you when you determine the right development and deployment strategies for your services and applications. Not all of them will require high levels of reliability and fault tolerance, with associated reducing in performance. However, some of them undoubtedly will.



# Defining Service Configurations

## 9.1. Overview

JBoss ESB 5.0 configuration is based on the [jbossesb-1.1.0 XSD](http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd)<sup>1</sup>. This XSD is always the definitive reference for the ESB configuration.

This model has two main sections:

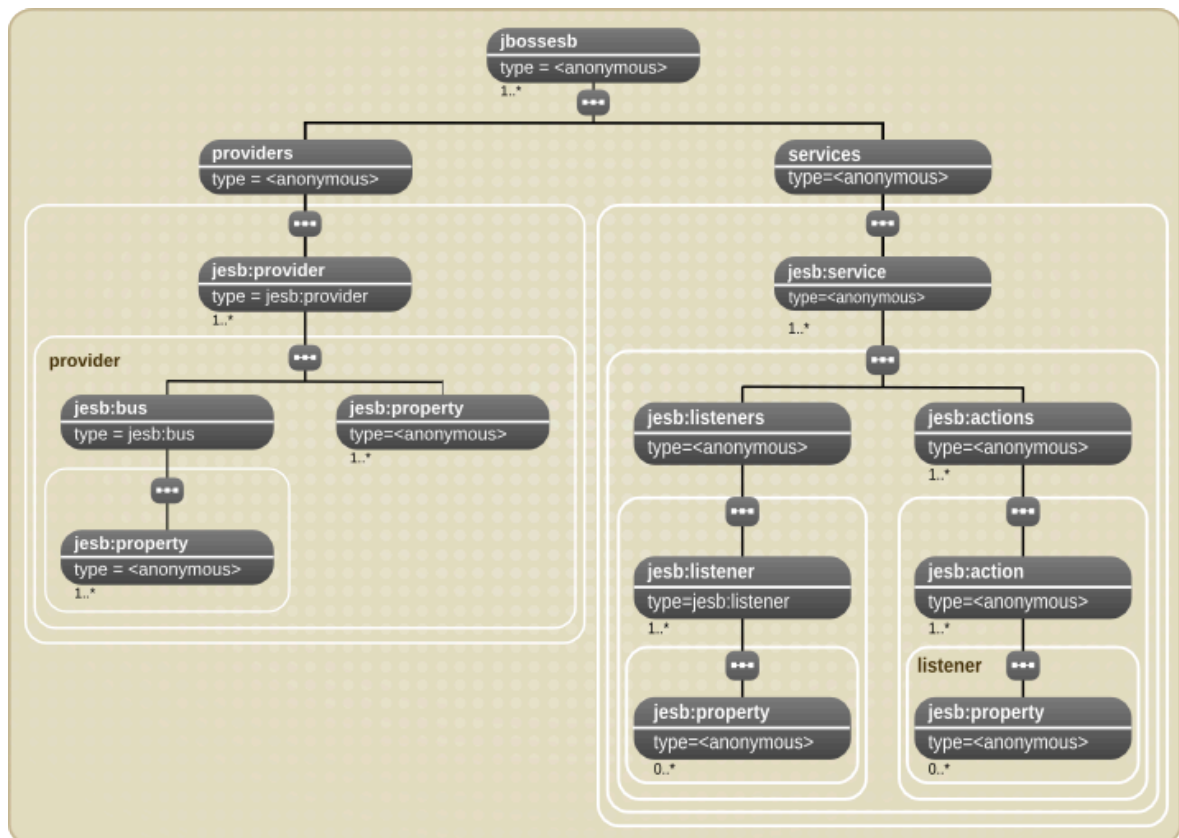


Figure 9.1. JBoss ESB Configuration Model

1. <providers>

This part of the model centrally defines all the message **<bus>** providers used by the message **<listener>**s, defined within the **<services>** section of the model.

2. <services>

This part of the model centrally defines all of the services under the control of a single instance of JBoss ESB. Each **<service>** instance contains either a “Gateway” or “Message Aware” listener definition.

Using JBoss Developer Studio is, by far, the easiest way to create configurations based on this model but you can also use an XSD-aware XML Editor such as the one found in the **Eclipse IDE**. This provides the author with auto-completion features for use when editing the configuration. Right mouse-click on the file -> **Open With -> XML Editor**.

<sup>1</sup> <http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd>

## 9.2. Providers

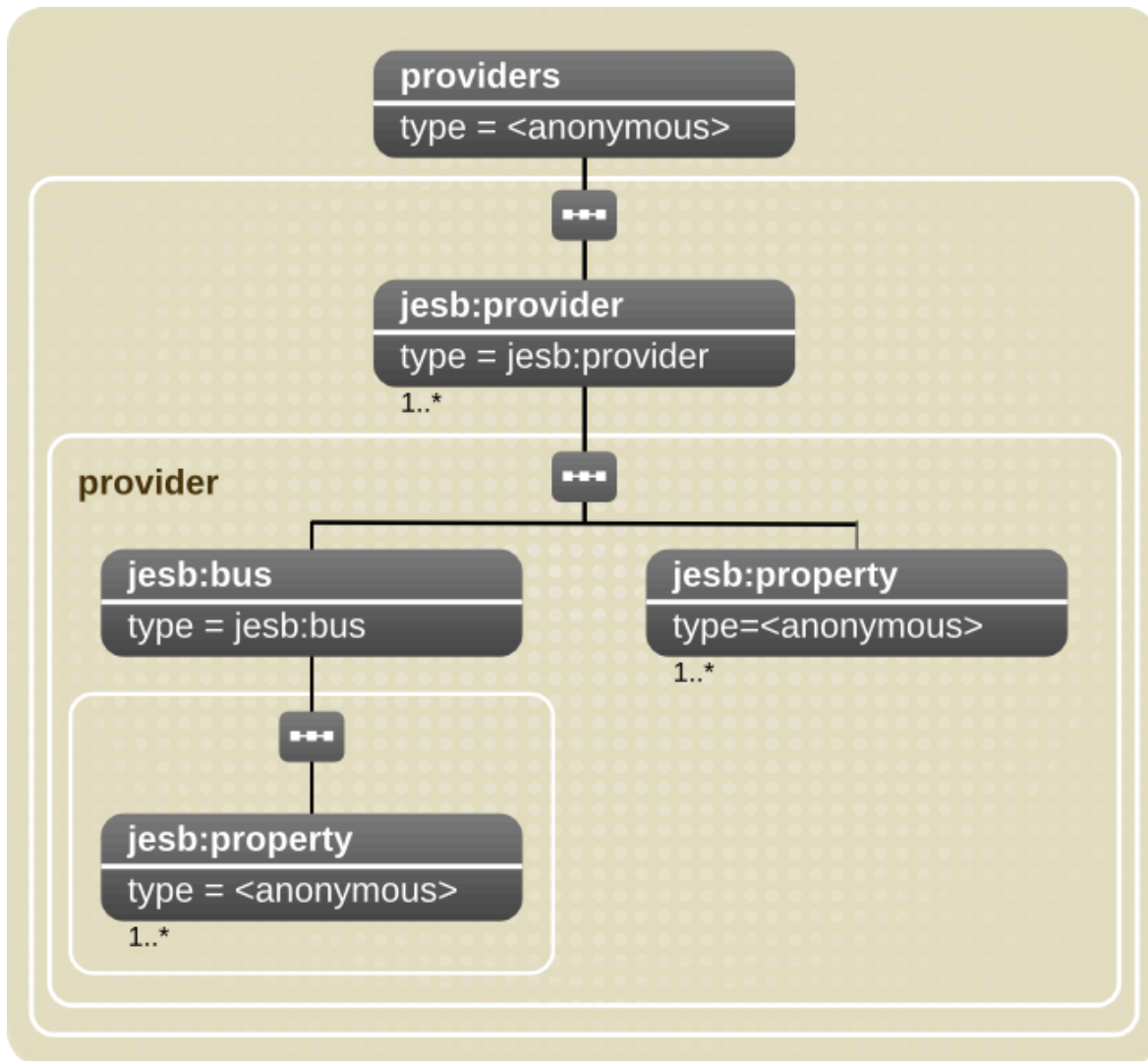


Figure 9.2. Providers Configuration Model

The `<providers>` part of the configuration defines all of the message `<provider>` instances for a single instance of the ESB. Two types of providers are currently supported:

- **Bus Providers**  
These specify provider details for “Event Driven” providers, that is, for listeners that are “pushed” messages. Examples of this provider type would be the `<jms-provider>`.
- **Schedule Provider**  
Provider configurations for schedule-driven listeners (that is, listeners that “pull” messages.)

A Bus Provider (e.g. `<jms-provider>`) can contain multiple `<bus>` definitions. The `<provider>` can also be decorated with `<property>` instances relating to provider specific properties that are common across all `<bus>` instances defined on that `<provider>`. For example, for JMS, these may be “connection-factory”, “jndi-context-factory” and so on. Likewise, each `<bus>` instance can be decorated with `<property>` instances specific to that `<bus>` instance (such as, for JMS, “destination-type”, “destination-name” and so forth.)

As an example, a provider configuration for JMS would be as follows:

```
<providers>
  <provider name="JBossMQ">
    <property name="connection-factory" value="ConnectionFactory" />
    <property name="jndi-URL" value="jnp://localhost:1099" />
    <property name="protocol" value="jms" />
    <property name="jndi-pkg-prefix" value="com.xyz"/>

    <bus busid="local-jms">
      <property name="destination-type" value="topic" />
      <property name="destination-name" value="queue/B" />
      <property name="message-selector" value="service='Reconciliation'" />
      <property name="persistent" value="true"/>
    </bus>
  </provider>
</providers>
```

The above example uses the “base” `<provider>` and `<bus>` types. This is perfectly legal, but we recommend use of the specialized extensions of these types for creating real configurations, namely `<jms-provider>` and `<jms-bus>` for JMS. The most important part of the above configuration is the `busid` attribute defined on the `<bus>` instance. This is a required attribute on the `<bus>` element/type (including all of its specializations - `<jms-bus>` etc). This attribute is used within the `<listener>` configurations to refer to the `<bus>` instance, on which the listener receives its messages. There will be more discussion of this later.

### 9.3. Services

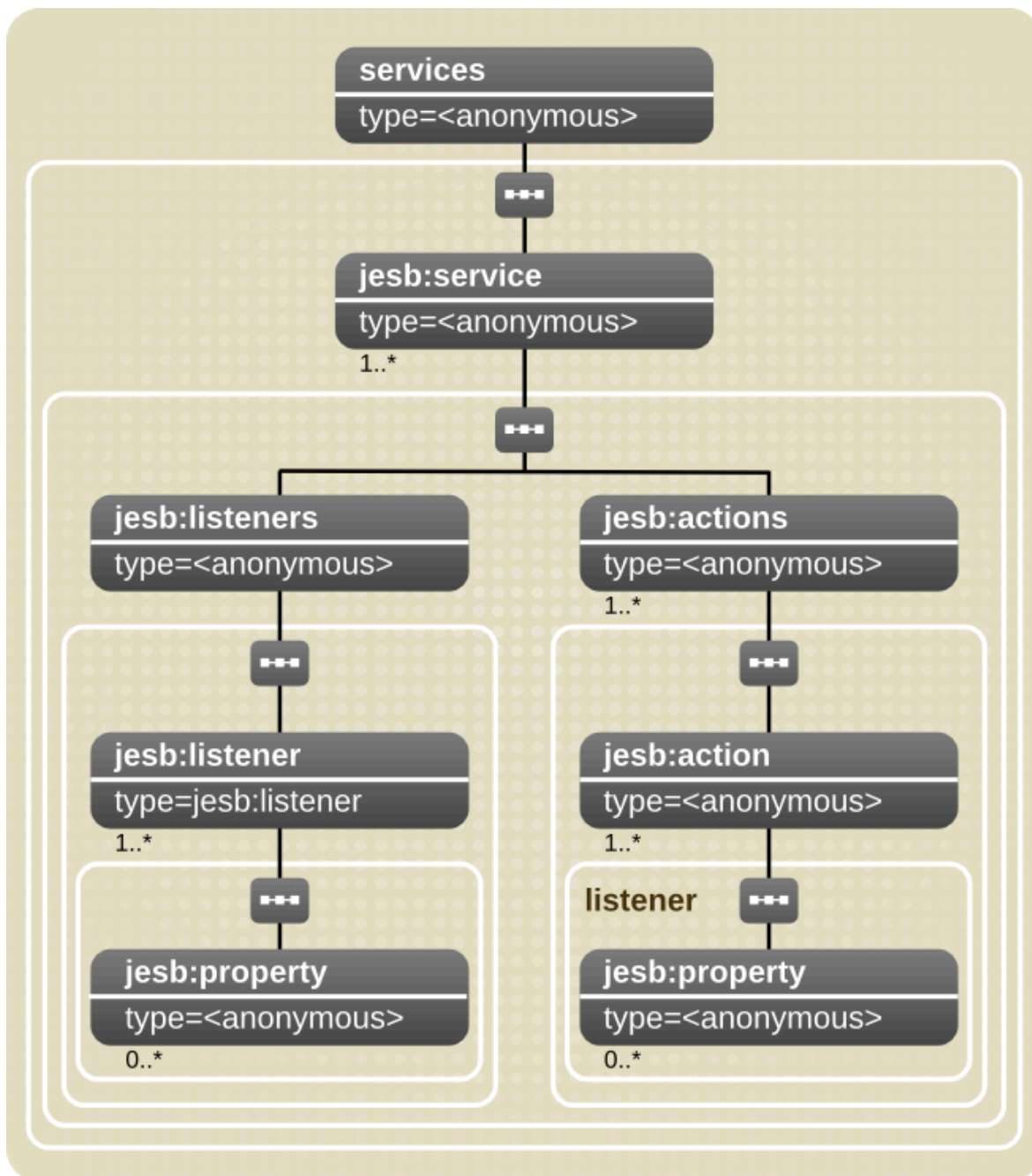


Figure 9.3. Services Configuration Model

The <services> part of the configuration defines each of the Services under the management of this instance of the ESB. It defines them as a series of <service> configurations. A <service> can also be decorated with the following attributes.

Name	Description	Type	Required
name	The Service Name under which the Service is Registered in the Service Registry.	xsd:string	true

Name	Description	Type	Required
category	The Service Category under which the Service is Registered in the Service Registry.	xsd:string	true
description	Human readable description of the Service. Stored in the Registry.	xsd:string	true

Table 9.1. Service Attributes

A `<service>` may define a set of `<listeners>` and a set of `<actions>`. The configuration model defines a “base” `<listener>` type, as well as specializations for each of the main supported transports (`<jms-listener>`, `<sql-listener>` etc.)

The “base” `<listener>` defines the following attribute. These attribute definitions are inherited by all `<listener>` extensions. As such, they can be set for all of the listeners and gateways supported by JBossESB, including the InVM transport.

Name	Description	Type	Required
name	The name of the listener. This attribute is required primarily for logging purposes.	xsd:string	true
busrefid	Reference to the busid of the <b>&lt;bus&gt;</b> through which the listener instance receives messages.	xsd:string	true
maxThreads	The max number of concurrent message processing threads that the listener can have active.	xsd:int	True
is-gateway	Whether or not the listener instance is a “Gateway”. <sup>1</sup>	xsd:boolean	true

<sup>1</sup> A message bus defines the details of a specific message channel/transport.

Table 9.2. Listener Attributes

Listeners can define a set of zero or more `<property>` elements (just like the `<provider>` and `<bus>` elements/types). These are used to define listener specific properties.



### Note

For each gateway listener defined in a service, an ESB-aware (or “native”) listener must also be defined. This is because gateway listeners do not define bidirectional endpoints but, rather, “startpoints” into the ESB. You cannot send a message to a Gateway from within the ESB. Also, note that, since a gateway is not an endpoint, it does not have an Endpoint Reference (EPR) persisted in the registry.

An example of a `<listener>` reference to a `<bus>` can be seen in the following illustration (using “base” types only).

```

1 <?xml version = '1.0' encoding = "UTF-8"?>
2 <jbossesb xmlns='http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.xsd'>
3
4   <providers>
5     <provider name="JBossMQ">
6       <property name="connection-factory" value="ConnectionFactory" />
7       <property name="jndi-URL" value="jnp://localhost:1099" />
8       <property name="protocol" value="jms" />
9
10      <bus busid="local-jms">
11        <property name="destination-type" value="topic" />
12        <property name="destination-name" value="queue/B" />
13        <property name="message-selector" value="service='Reconciliation'" />
14      </bus>
15    </provider>
16  </providers>
17  <services>
18    <service category="Bank" name="Reconciliation"
19      description="Bank Reconciliation Service" is-gateway="false">
20
21      <listeners>
22        <listener name="Bank-Listener"
23          busidref="local-jms"
24          maxThreads="2">
25
26        </listener>
27      </listeners>
28
29      <actions>
30        ....
31      </actions>
32    </service>
33  </services>
34 </jbossesb>

```

A Service will do little without a list of one or more <actions>. <action>s typically contain the logic for processing the payload of the messages received by the service (through it's listeners). Alternatively, it may contain the transformation or routing logic for messages to be consumed by an external Service/ entity.

The <action> element/type defines the following attributes.

Name	Description	Type	Required
name	The name of the action. This attribute is required primarily for logging purposes.	xsd:string	true
class	The <b>org.jboss.soa.esb.actions.ActionProcessor</b> implementation class name.	xsd:string	true
process	The name of the "process" method that will be reflectively called for message processing.(Default is the "process" method as defined on the <b>ActionProcessor</b> class).	xsd:int	false

Table 9.3. Action Attributes

In a list of <action> instances within an <actions> set, the actions are called (that is, their "process" method is called) in the order in which the <action> instances appear in the <actions> set. The message returned from each <action> is used as the input message to the next <action> in the list.

Like a number of other elements/types in this model, the <action> type can also contain zero or more <property> element instances. The <property> element/type can define a standard name-value-pair, or contain free form content (xsd:any). According to the XSD, this free form content is valid as

child content for the <property> element/type, no matter where it is in the configuration (on any of <provider>, <bus>, <listener> and any of their derivatives). However, it is only on <action>- defined <property> instances that this free-form child content is used.

As stated in the <action> definition above, actions are implemented through implementing the **org.jboss.soa.esb.actions.ActionProcessor** class. All implementations of this interface must contain a public constructor of the following form:

```
public ActionZ(org.jboss.soa.esb.helpers.ConfigTree configuration);
```

It is the Constructor that supplies an instance of a ConfigTree with the action attributes. The free-form content from the action property instances is also included in this.

So an example of an <actions> configuration might be as follows:

```
<actions>
  <action name="MyAction-1" class="com.acme.MyAction1"/>
  <action name="MyAction-2" class="com.acme.MyAction2">
    <property name="propA" value="propAVal" />
  </action>
  <action name="MyAction-3" class="com.acme.MyAction3">
    <property name="propB" value="propBVal" />
    <property name="propC">
      <!-- Free form child content... -->
      <some-free-form-element>zzz<some-free-form-element>
    </property>
  </action>
</actions>
```

## 9.4. Transport Specific Type Implementations

The JBoss ESB configuration model defines transport specific specializations of the “base” types <provider>, <bus> and <listener> (JMS, SQL and so forth.) This allows you to have stronger validation on the configuration. It also makes configuration easier if you use an XSD-aware XML Editor (such as the Eclipse XML Editor). These specializations explicitly define the configuration requirements for each of the transports supported by JBoss ESB out-of-the-box. It is recommended that you use these specialized types instead of the “base” types when creating JBoss ESB configurations. The only alternative is whereby a new transport is being supported outside an official JBoss ESB release.

The same basic principals that apply when creating configurations from the “base” types also apply when creating configurations from the transport-specific alternatives:

1. Define the provider configuration e.g. <jms-provider>.
2. Add the bus configurations to the new provider (for example, <jms-bus>), and assign a unique busid attribute value.
3. Define your <services> as “normal,” adding transport-specific listener configurations (such as <jms-listener>) that reference (using busidref) the new bus configurations you just made. (For example, <jms-listener> referencing a <jms-bus>.)

The only rule that applies when using these transport-specific types is that you cannot cross-reference from a listener of one type, to a bus of another. In other words, you can only reference a `<jms-bus>` from a `<jms-listener>`. A runtime error will result where cross-references are made.

The transport specific implementations that are in place in this release are:

### JMS

`<jms-provider>`, `<jms-bus>`, `<jms-listener>` and `<jms-message-filter>`: The `<jms-message-filter>` can be added to either the `<jms-bus>` or `<jms-listener>` elements. Where the `<jms-provider>` and `<jms-bus>` specify the JMS connection properties, the `<jms-message-filter>` specifies the actual message QUEUE/TOPIC and selector details.

### SQL

`<sql-provider>`, `<sql-bus>`, `<sql-listener>` and `<sql-message-filter>`: The `<sql-message-filter>` can be added to either the `<sql-bus>` or `<sql-listener>` elements. Where the `<sql-provider>` and `<ftp-bus>` specify the JDBC connection properties, the `<sql-message-filter>` specifies the message/row selection and processing properties.

### FTP

`<ftp-provider>`, `<ftp-bus>`, `<ftp-listener>` and `<ftp-message-filter>`: The `<ftp-message-filter>` can be added to either the `<ftp-bus>` or `<ftp-listener>` elements. Where the `<ftp-provider>` and `<ftp-bus>` specify the FTP access properties, the `<ftp-message-filter>` specifies the message/file selection and processing properties

### Hibernate

`<hibernate-provider>`, `<hibernate-bus>`, `<hibernate-listener>` : The `<hibernate-message-filter>` can be added to either the `<hibernate-bus>` or `<hibernate-listener>` elements. Where the `<hibernate-provider>` specifies file system access properties like the location of the hibernate configuration property, the `<hibernate-message-filter>` specifies what classnames and events should be listened to.

### File System

`<fs-provider>`, `<fs-bus>`, `<fs-listener>` and `<fs-message-filter>` The `<fs-message-filter>` can be added to either the `<fs-bus>` or `<fs-listener>` elements. Where the `<fs-provider>` and `<sql-bus>` specify the File System access properties, the `<fs-message-filter>` specifies the message/file selection and processing properties.

### Schedule

`<schedule-provider>`. This is a special type of provider and differs from the bus based providers listed above. See Scheduling for more.

### JMS/JCA Integration

`<jms-jca-provider>`: This provider can be used in place of the `<jms-provider>` to enable delivery of incoming messages using JCA inflow. This introduces a transacted flow to the action pipeline, and thereby encompasses actions within a JTA transaction.



As you will notice, all of the currently implemented transport-specific types include an additional type not present in the “base”, that being `<*-message-filter>`. This element/type can be added inside either the `<*-bus>` or `<*-listener>`. Allowing this type to be specified in both places means that you can specify message-filtering globally for the bus (for all listeners using that bus) or locally on a listener-by-listener basis.



### Note

In order to list and describe the attributes for each transport specific type, you can use the `jbosessb-1.1.0 XSD`. Using an XSD-aware XML Editor such as the **Eclipse XML Editor** makes working with these types far easier.

Property Name	Description	Required
dest-type	The type of destination, either QUEUE or TOPIC	Yes
dest-name	The name of the Queue or Topic	Yes
selector	Allows multiple listeners to register with the same queue/topic but they will filter on this message-selector.	No
persistent	Indicates if the delivery mode for JMS should be persistent or not. True or false. Default is true	No
acknowledge-mode	The JMS Session acknowledge mode. Can be one of AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE. Default is AUTO_ACKNOWLEDGE	No
jms-security-principal	JMS destination user name. This will be used when creating a connection to the destination.	No
jms-security-credential	JMS destination password. This will be used when creating a connection to the destination.	No

Table 9.4. JMS Message Filter Configuration

Example configuration:

```
<jms-bus busid="quickstartGwChannel">
  <jms-message-filter
    dest-type="QUEUE"
    dest-name="queue/quickstart_jms_secured_Request_gw"
    jms-security-principal="esbuser"
    jms-security-credential="esbpassword"/>
</jms-bus>
```

## 9.5. FTP Provider Configuration

Property	Description	Required
hostname	Can be a combination of <code>&lt;host : port&gt;</code> of just <code>&lt;host&gt;</code> which will use port 21.	Yes
username	Username that will be used for the FTP connection.	Yes
password	Password for the above user	Yes

Property	Description	Required
directory	The FTP directory that is monitored for incoming new files	Yes
input-suffix	The file suffix used to filter files targeted for consumption by the ESB (note: add the dot, so something like '.esbIn'). This can also be specified as an empty string to specify that all files should be retrieved.	Yes
work-suffix	The file suffix used while the file is being process, so that another thread or process won't pick it up too. Defaults to <b>.esbInProgress</b> .	No
post-delete	If true, the file will be deleted after it is processed. Note that in that case post-directory and post-suffix have no effect. Defaults to true.	No
post-directory	The FTP directory to which the file will be moved after it is processed by the ESB. Defaults to the value of directory above.	No
post-suffix	The file suffix which will be added to the file name after it is processed. Defaults to <b>.esbDone</b> .	No
error-delete	If true, the file will be deleted if an error occurs during processing. Note that in that case error-directory and error-suffix have no effect. This defaults to "true."	No
error-directory	The FTP directory to which the file will be moved after when an error occurs during processing. This defaults to the value of directory above.	No
error-suffix	The suffix which will be added to the file name after an error occurs during processing. Defaults to <b>.esbError</b> .	No
protocol	The protocol, can be one of: <ul style="list-style-type: none"> <li>• sftp (SSH File Transfer Protocol)</li> <li>• ftps (FTP over SSL)</li> <li>• ftp (default).</li> </ul>	No
passive	Indicates that the FTP connection is in passive mode. Setting this to "true" means the FTP client will establish two connections to the ftpserver. Defaults to false, meaning that the client will tell the FTP Server the port to which it should connect. The FTP Server then establishes the connection to the client.	No
read-only	If true, the FTP Server does not permit write operations on files. Note that, in this case, the following properties have no effect: work-suffix, post-delete, post-directory, post-suffix, error-delete, error-directory, and error-suffix. Defaults to false. See section "Read-only FTP Listener for more information.	No
certificate-url	The URL to a public server certificate for FTPS server verification or to a private certificate for SFTP client verification. An SFTP certificate can be located as a resource embedded within a deployment artifact	No
certificate-name	The common name for a certificate for FTPS server verification	No

Property	Description	Required
certificate-passphrase	The pass-phrase of the private key for SFTP client verification.	No

Table 9.5. FTP Provider Configuration

## 9.6. FTP Listener Configuration

Schedule Listener that polls for remote files based on the configured schedule (scheduleidref). See Service Scheduling.

### 9.6.1. Read-Only FTP Listener

Setting the ftp-provider property "read-only" to "true" will tell the system that the remote file system does not allow write operations. This is often the case when the FTP server is running on a mainframe computer where permissions are given to a specific file.

The read-only implementation uses JBoss TreeCache to hold a list of the file names that have been retrieved. It is also used to fetch only those that have not previously been retrieved. The cache should be configured to use a cacheloader in order to persist to stable storage.

Please note that a strategy must exist for removing the file names from the cache. This might take the form of an archiving process on the mainframe that moves the files to a different location on a regular basis. The removal of file names from the cache could be undertaken by having a database procedure that removes them from the cache every few days. Another strategy would be to specify a TreeCacheListener that, upon evicting file names from the cache, also removes them from the cacheloader. The eviction period would then be configurable. This can be set via a property (removeFilesystemStrategy-cacheListener) in the ftp-listener configuration.

Name	Description
scheduleidref	Schedule used by the FTP listener. See Service Scheduling.
removeFilesystemStrategy-class	Override the remote file system strategy with a class that implements: <b>org.jboss.soa.esb.listeners.gateway.remotestrategies.RemoteFilesystemStrategy</b> . Defaults to <b>org.jboss.soa.esb.listeners.gateway.remotestrategies.ReadOnlyRemoteFilesystemStrategy</b>
removeFilesystemStrategy-configFile	Specify a JBoss TreeCache configuration file on the local file system or one that exists on the classpath. Defaults to looking for a file named <b>/ftplib-cache-config.xml</b> which it expects to find in the root of the classpath
removeFilesystemStrategy-cacheListener	Specifies an JBoss <b>TreeCacheListener</b> implementation to be used with the TreeCache. Default is no TreeCacheListener.
maxNodes	The maximum number of files that will be stored in the cache. 0 denotes no limit
timeToLiveSeconds	Time to idle (in seconds) before the node is swept away. 0 denotes no limit
maxAgeSeconds	Time an object should exist in TreeCache (in seconds) regardless of idle time before the node is swept away. 0 denotes no limit

Table 9.6. Read-only FTP Listener Configuration

Example configuration:

```
<ftp-listener name="FtpGateway"
  busidref="helloFTPChannel"
  maxThreads="1"
  is-gateway="true"
  schedule-frequency="5">
  <property name="remoteFileSystemStrategy-configFile" value="./ftpfile-
cache-config.xml"/>
  <property name="remoteFileSystemStrategy-cacheListener"
  value="org.jboss.soa.esb.listeners.gateway.remotestrategies.cache.DeleteOnEvictTreeC
>
</ftp-listener>
```

Example snippet from JBoss cache configuration:

```
<region name="/ftp/cache">
  <attribute name="maxNodes">5000</attribute>
  <attribute name="timeToLiveSeconds">1000</attribute>
  <attribute name="maxAgeSeconds">86400</attribute>
</region>
```

The read-only configuration is demonstrated by `helloworld_ftp_action` Quick Start. Run 'ant help' in the `helloworld_ftp_action quickstart` directory for instructions on running the Quick Start. Please refer to the JBoss Cache documentation for more information about the configuration options available (<http://labs.jboss.com/jboss-cache/docs/index.html>).

## 9.7. Updated. UDP Gateway

The **UDP Gateway** is an implementation for receiving ESB-unaware messages sent via the **UDP** protocol. The payload will be passed along to the action chain in the default ESB Message object location. Actions can call `esbMessage.getBody().get()` to retrieve the byte array payload.

Property	Description	Comments
Host	The hostname/ip to which to listen.	Mandatory.
Port	The port to which to listen.	Mandatory.
handlerClass	A concrete implementation of org.jboss.soa.esb.listeners.gateway.mina.MessageHandler.	Optional. Default is org.jboss.soa.esb.listeners.gateway.mina.DefaultMessageHandler.
is-gateway	UDPGatewayListener can only act as a gateway.	Mandatory.

Table 9.7. UDP Gateway Configuration

Here is an example configuration:

```
<udp-listener
  name="udp-listener"
  host="localhost"
  port="9999"

  handlerClass="org.jboss.soa.esb.listeners.gateway.mina.DefaultMessageHandler"
  is-gateway="true"
</udp-listener/>
```

Example 9.1. **UDP** Gateway Configuration

## 9.8. **Updated** JBoss Remoting (JBR) Configuration

The JBoss Remoting Gateway hooks JBoss Remoting (JBR) into JBoss ESB as a transport option. It leverages support for HTTP(S) and Socket (+SSL) via JBR.

The basic configuration of the JBR Provider is as follows:

```
<jbr-provider name="socket_provider" protocol="socket" host="localhost">
  <jbr-bus busid="socket_bus" port="64111"/>
</jbr-provider>
```

As you can see, the basic **jbr-provider** and **jbr-bus** configuration is very simple. The **jbr-bus** can then be referenced from a service configuration through the **jbr-listener**:

```
<listeners>
  <jbr-listener name="soc" busidref="socket_bus" is-gateway="true"/>
</listeners>
```

The **jbr-listener** is only supported as a gateway (in other words, setting **is-gateway** to "false" will result in a Service deployment error.)

## 9.9. Transitioning from the Old Configuration Model

This section is aimed at developers who are familiar with the old JBoss ESB non-XSD based configuration model.

The old model used free-form XML, with ESB components receiving their configurations via an instance of **org.jboss.soa.esb.helpers.ConfigTree**. The new configuration model is XSD-based. However, the underlying component configuration pattern is still via an instance of **org.jboss.soa.esb.helpers.ConfigTree**. This means that, at the moment, the XSD-based configurations are mapped/transformed into **ConfigTree**-style configurations.

Developers who have been accustomed to the old model now need to keep the following in mind:

1. Do not assume you can infer the new configurations based on your knowledge of the old. Read all of the documentation on the new configuration model.
2. The only location where free-form markup is supported in the new configuration is on the `<property>` element/type. This type is allowed on `<provider>`, `<bus>` and `<listener>` types (and sub-types). However, the only location in which `<property>` based free form markup is mapped into the ConfigTree configurations is where the `<property>` exists on an `<action>`. In this case, the `<property>` content is mapped into the target ConfigTree `<action>`. Note however, if you have 1+ `<property>` elements with-free form child content on an `<action>`, all this content will be concatenated together on the target ConfigTree `<action>`.
3. You must ensure that the ConfigTree-based configuration upon which these components depend can be mapped from the new XSD-based configurations, when developing new Listener/Action components. For example, in the ConfigTree configuration model, you could decide to supply the settings to a listener component via either the attributes on the listener node or the child nodes within the listener configuration. This type of free-form configuration on `<listener>` components is not supported on the XSD-to-ConfigTree mapping. In other words, the child content in the above example would not be mapped from the XSD to the ConfigTree-style configuration. In fact, the XSD configuration simply would not accept the arbitrary content, unless it was in a `<property>` and, even in that case, if it were on a `<listener>`, it would simply be ignored by the mapping code.

### 9.10. Configuration

All components within the core receive their configuration parameters as XML. How these parameters are provided to the system is hidden by the `org.jboss.soa.esb.parameters.ParamRepositoryFactory`:

```
public abstract class ParamRepositoryFactory
{
    public static ParamRepository getInstance();
}
```

**Example 9.2. Enabling GLOBAL** public abstract class `ParamRepositoryFactory`

This returns implementations of the `org.jboss.soa.esb.parameters.ParamRepository` interface (which allows for different implementations):

```
public interface ParamRepository
{
    public void add(String name, String value) throws
        ParamRepositoryException;
    public String get(String name) throws ParamRepositoryException;
    public void remove(String name) throws ParamRepositoryException;
}
```

**Example 9.3. Enabling GLOBAL** public interface `ParamRepository`

Within this version of the JBossESB, there is only a single implementation, (the `org.jboss.soa.esb.parameters.ParamFileRepository`), which expects to be able to



---

load the parameters from a file. The implementation to use this may be over-ridden using the `org.jboss.soa.esb.paramsRepository.class` property.



### Note

It is recommended that you construct your ESB configuration file using **Eclipse** or some other XML editor. The JBossESB configuration information is supported by an annotated XSD, which should help if you are using a more basic editor.



# Web Services Support

## 10.1. JBossWS

The JBoss Enterprise Service Bus has several components that are used for exposing and invoking Webservice end points.

### SOAPProcessor

The **SOAPProcessor** action lets one expose **JBossWS 2.x** and higher *Webservice Endpoints* through listeners running on the ESB. This can be achieved even if the end points do not provide web-service interfaces of their own. The JBossWS Webservice Endpoints exposed by the **SOAPProcessor** action are *ESB Message-Aware*. Therefore, they can be used to invoke other Webservice Endpoints over any transport channel that is supported by the Enterprise Service Bus.

Note that the **SOAPProcessor** action is sometimes also referred to as *SOAP on the bus*.

### SOAPClient

The **SOAPClient** action allows one to make invocations on Webservice Endpoints.

The **SOAPClient** action is also referred to as *SOAP off the bus* by some.

In order to learn more about these components, one should refer to the *Services Guide*. It explains, in detail, how to configure them.

More information about this topic can also be found within the "Wiki" pages that are been shipped with the JBoss ESB documentation.



# Out-of-the-box Actions

This section provides a catalog of all Actions that are included by default in the JBoss ESB.

## 11.1. Updated Out-of-the-Box Actions

This section provides a catalog of all of the Actions that are included by default in the **JBoss** ESB.

### 11.1.1. Updated Transformers & Converters

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

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

#### ByteArrayToString

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

Input Type	<b>byte[]</b>
Class	<b>org.jboss.soa.esb.actions.converters.ByteArrayToString</b>
Properties	1. "encoding": The binary data encoding on the message byte array. Defaults to "UTF-8" when not specified.
Sample Configuration	<pre>&lt;action name="transform" class="org.jboss.soa.esb.actions.converters.ByteArrayToString"&gt;   &lt;property name="encoding" value="UTF-8" /&gt; &lt;/action&gt;</pre>

#### LongToDateConverter

Takes a long based message payload and converts it into a **java.util.Date** object instance.

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

#### ObjectInvoke

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

Input Type	User Object
Output Type	User Object
Class	<b>org.jboss.soa.esb.actions.converters.ObjectInvoke</b>
Properties	<ol style="list-style-type: none"> <li>1. "class-processor": The runtime-classname of the processor class used on the message payload.</li> <li>2. "class-method": The name of the method on the processor class used.</li> </ol>
Sample Configuration	<pre>&lt;action name="invoke" class="org.jboss.soa.esb.actions.converters.ObjectInvoke"&gt; &lt;property name="class-processor" value="org.jboss.MyXXXProcessor"&gt; &lt;property name="class-method" value="processXXX"&gt; &lt;/action&gt;</pre>

### 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.

Input Type	User Object
Output Type	java.lang.String
Class	<b>org.jboss.soa.esb.actions.converters.ObjectToCSVString</b>
Properties	<ol style="list-style-type: none"> <li>1. "bean-properties": List of Object bean property names used to obtain CSV values for the output CSV String. The Object should support a getter method for each listed properties.</li> <li>2. "fail-on-missing-property": Flag indicating whether or not the action should fail if a property is missing from the Object (in other words, if the Object does not support a getter method for the property.) The default value is "false".</li> </ol>
Sample Configuration	<pre>&lt;action name="transform" class="org.jboss.soa.esb.actions.converters.ObjectToCSVString"&gt; &lt;property name="bean-properties" value="name, address, phoneNumber"&gt; &lt;property name="fail-on-missing-property" value="true" /&gt; &lt;/action&gt;</pre>

### ObjectToXStream

Takes the Object bound as the Message payload and converts it into XML using the *XStream*<sup>1</sup> processor.

Input Type	User Object
------------	-------------

<sup>1</sup> <http://xstream.codehaus.org/>

Output Type	<b>java.lang.String</b>
Class	<b>org.jboss.soa.esb.actions.converters.ObjectToXStream</b>
Properties	<ol style="list-style-type: none"> <li>1. "class-alias": Class alias used in call to <i>XStream.alias(String, Class)</i><sup>2</sup> prior to serialisation. Defaults to the input Object's class name.</li> <li>2. "exclude-package": Exclude the package name from the generated XML. Default is "true". Not applicable if a "class-alias" is specified.</li> <li>3. "aliases": Optional. Specify additional aliases in order to help XStream to convert the XML elements into Objects.</li> <li>4. "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.</li> <li>5. "xstream-mode": Optional. Specify the XStream mode to use. Possible values are <b>XPATH_RELATIVE_REFERENCES</b> (the default), <b>XPATH_ABSOLUTE_REFERENCES</b>, <b>ID_REFERENCES</b> or <b>NO_REFERENCES</b>.</li> </ol>
Sample Configuration	<pre> &lt;action name="transform"   class="org.jboss.soa.esb.actions.converters.ObjectToXStream"&gt;   &lt;property name="class-alias" value="MyAlias" /&gt;   &lt;property name="exclude-package" value="true" /&gt;   &lt;property name="aliases"&gt;     &lt;alias name="alias1" value="com.acme.MyXXXClass1"/&gt;     &lt;alias name="alias2" value="com.acme.MyXXXClass2"/&gt;     &lt;alias name="xyz" value="com.acme.XyzValueObject"/&gt;     &lt;alias name="x" value="com.acme.XValueObject"/&gt;     ...   &lt;/property&gt;   &lt;property name="namespaces"&gt;     &lt;namespace namespace-uri="http://www.xyz.com" local-       part="xyz"/&gt;     &lt;namespace namespace-uri="http://www.xyz.com/x" local-       part="x"/&gt;     ...   &lt;/property&gt; &lt;/action&gt; </pre>
Input Type	User Object
Output Type	<b>java.lang.String</b>
Class	<b>org.jboss.soa.esb.actions.converters.ObjectToXStream</b>
Properties	<ol style="list-style-type: none"> <li>1. "class-alias": Class alias used in call to <i>XStream.alias(String, Class)</i><sup>3</sup> prior to serialization. Defaults to the input Object's class name.</li> <li>2. "exclude-package": Exclude the package name from the generated XML. Default is "true". Not applicable if a "class-alias" is specified.</li> </ol>
Sample Configuration	<pre> &lt;action name="transform" </pre>

```
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.

Input Type	<b>java.lang.String</b>
Output Type	User Object (specified by "incoming-type" property)
Class	<b>org.jboss.soa.esb.actions.converters.XStreamToObject</b>
Properties	<ol style="list-style-type: none"> <li>"class-alias": Class alias used during serialisation. Defaults to the input Object's class name.</li> <li>"exclude-package": Flag indicating whether or not the XML includes a package name.</li> <li>"incoming-type": Class type.</li> <li>"root-node": Optional. Specify a different root node then the actual root node in the XML. Takes an <b>XPath</b> expression.</li> <li>"aliases": Optional. Specify additional aliases to help Xstream to convert the xml elements to Objects</li> <li>"attribute-aliases": Optional. Specify additional attribute aliases to help Xstream to convert the xml attributes to Objects</li> <li>"converters": Optional. Specify converters to help Xstream to convert the xml elements and attributes to Objects. For more information about converters see <a href="http://xstream.codehaus.org/converters.html">http://xstream.codehaus.org/converters.html</a></li> </ol>
Sample Configuration	<pre>&lt;action name="transform" class="org.jboss.soa.esb.actions.converters.XStreamToObject"&gt; &lt;property name="class-alias" value="MyAlias" /&gt; &lt;property name="exclude-package" value="true" /&gt; &lt;property name="incoming-type" value="com.acme.MyXXXClass" /&gt; &lt;property name="root-node" value="/rootNode/MyAlias" /&gt; &lt;property name="aliases"&gt; &lt;alias name="alias1" value="com.acme.MyXXXClass1"/&gt; &lt;alias name="alias2" value="com.acme.MyXXXClass2"/&gt; ... &lt;/property&gt; &lt;property name="attribute-aliases"&gt; &lt;attribute-alias name="alias1" value="com.acme.MyXXXClass1"/&gt; &lt;attribute-alias name="alias2" value="com.acme.MyXXXClass2"/&gt; ... &lt;/property&gt;</pre>



```

<property name="converters">
  <converter class="com.acme.MyXXXConverter1"/>
  <converter class="com.acme.MyXXXConverter2"/>
  ...
</property>
</action>

```

### XsltAction

Performs transformation on entire documents. If you need per-fragment transformations, please refer to the **SmooksAction**.

```

<action name="transform"
  class="org.jboss.soa.esb.actions.transformation.xslt.XsltAction">
  <property name="templateFile" value="/template.xml" />
  <property name="resultType" value="STRING" />
  <property name="failOnWarning" value="true" />
</action>

```

#### Example 11.1. XsltActionConfiguration

<b>get-payload-location</b>	Message Body location containing the message payload.	Default Payload Location
<b>set-payload-location</b>	Message Body location where result payload is to be placed.	Default Payload Location
<b>resultType</b>	<p>The type of Result to be set as the result Message payload.</p> <p>This property controls the output result of the transformation.</p> <p>The following values are currently available:</p> <p>STRING - will produce a string.</p> <p>BYTES - will produce a <b>byte[]</b>.</p> <p>DOM - will produce a <b>DOMResult</b>.</p> <p>SAX - will produce a <b>SAXResult</b>.</p> <p>If the above does not suit your needs, then you have the option of specifying both the Source and Result by creating a SourceResult object instance. This is a simple object that holds both a Source and a Result.</p> <p>You need to create this object prior to calling this action. The type of Result returned will be that which was used to create the SourceResult.</p>	STRING

<b>failOnWarning</b>	If "true", this will make a transformation warning cause an exception to be thrown. If "false", the failure will be logged.	True
<b>uriResolver</b>	Fully qualified class name of a class that implements <b>URIResolver</b> . This will be set on the "transformation factory."	Not applicable
<b>factory.feature</b>	Factory features that will be set for the transformation factory. The feature names, which are fully qualified URIs, should be specified after the ' <b>factory.feature.</b> ' prefix. For example: <b>factory.feature.http://javax.xml.XMLConstants/feature/secure-processing</b>	Not applicable
<b>Factory.attribute</b>	Factory attributes that will be set for the transformation factory. The attribute name should be specified after the ' <b>factory.attribute.</b> ' prefix. For example: <b>factory.attribute.someVendorAttributename</b>	Not applicable.

### SmooksTransformer

Message Transformation on the **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) and target (XML, Java, CSV, EDI) data formats are supported by the **SmooksTransformer** component. Various Transformation Technologies are also supported, all within a single framework.

Class	<b>org.jboss.soa.esb.actions.converters.SmooksTransformer</b>
Properties	<p><b>Smooks</b> Resource Configuration:</p> <ol style="list-style-type: none"> <li>"<b>resource-config</b>": The <b>Smooks</b> resource configuration file.</li> </ol> <p>Message Profile Properties (Optional):</p> <ol style="list-style-type: none"> <li>"<b>from</b>": Message Exchange Participant name. Message Producer.</li> <li>"<b>from-type</b>": Message type/format produced by the "from" message exchange participant.</li> <li>"<b>to</b>": Message Exchange Participant name. Message Consumer.</li> <li>"<b>to-type</b>": Message type/format consumed by the "to" message exchange participant.</li> </ol> <p>Note: All the above properties can be overridden by supplying them as "properties to the message" (<b>Message.Properties</b>).</p>
Sample Configuration	<pre> Default Input/Output: &lt;action name="transform" class="org.jboss.soa.esb.actions.converters.SmooksTransformer"&gt; &lt;property name="resource-config" value="/smooks/ config-01.xml" /&gt; &lt;/action&gt; Named Input/Output: &lt;action name="transform" </pre>

```

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>

```

**Java** Objects are bound to the **Message.Body** under their **beanId**.

### SmooksAction

The **SmooksAction** class (**org.jboss.soa.esb.smooks.SmooksAction**) is the second-generation ESB action class for executing **Smooks** “processes” (it can do more than just transform messages, such as splitting.) The **SmooksTransformer** action will be deprecated in a future release of the ESB (and it will eventually be removed).

The **SmooksAction** class can, using **Smooks** PayloadProcessor, process a wider range of ESB Message payloads such as Strings, byte arrays, InputStreams, Readers, POJOs and more. 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 message stream, including content-based payload Splitting and Routing (though not ESB Message routing). The **SmooksAction** enables the full range of **Smooks** capabilities from within **JBossESB**.



### Note

Users should be aware that **Smooks** does not detect (nor report errors on) certain types of configuration errors for resource settings made through the base resource-config. If, for example, the resource is a **Smooks** Visitor implementation, and you misspell the name of the Visitor class, **Smooks** will not raise this as an error simply because it does not know that the misspelling was supposed to be a class. Remember, **Smooks** supports many different types of resources, not just **Java** Visitor implementations.

The easiest way to avoid this issue (in **JBoss** ESB 4.5.GA+) is to use the extended **Smooks** configuration namespaces for all “out-of-the-box” functionality. For example, instead of defining **Java** binding configurations by defining **org.milyn.javabean.BeanPopulator** resource-config configurations, you should use the <http://www.milyn.org/xsd/smooks/javabean-1.2.xsd> configuration namespace. (This is the **jb:bean** configuration.)

If you have implemented new **Smooks Visitor** functionality, the easiest way to avoid this issue is to define an extended configuration namespace for this new resource type. This also has the advantage of making this new resource easier to configure, because you are able to leverage the schema support built into your Integrated Development Environment.

See the **Smooks v1.1+** documentation for examples of these extended namespace configurations.

### SmooksAction Configuration

The following illustrates the basic **SmooksAction** configuration:

```
<action name="transform" class="org.jboss.soa.esb.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
<b>set-payload-location</b>	Message Body location where result payload is to be placed.	Default Payload Location
<b>excludeNonSerializables</b>	Exclude non-Serializable Objects when mapping the contents of the Smooks <i>ExecutionContext</i> <sup>4</sup> back onto the ESB Message.	true
resultType	The type of Result to be set as the result Message payload.	STRING
<b>javaResultBeanId</b>	Note: Only relevant when <b>resultType=JAVA</b>  The <b>Smooks bean</b> context <b>beanId</b> to be mapped when the <b>resultType</b> is "JAVA." If not specified, the whole context bean Map is mapped as the <b>JAVA</b> result.	
<b>reportPath</b>	The path and file name for generating a <i>Smooks Execution Report</i> <sup>5</sup> . 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 (by using `Message.getBody().get()` and `Message.getBody().set(Object)`).

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

### XML, EDI and CSV Input Payloads

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

1. String
2. `InputStream`<sup>6</sup>
3. `Reader`<sup>7</sup>
4. byte array

Apart from that, you just need to perform the standard **Smooks** configurations (in the **Smooks** configuration, not the ESB configuration) for processing the message type in question to, for example, configure a parser if it's not an XML Source (such as EDI or CSV.)

### 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** or **Java-to-Java** transformations.

### 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 is bound back in the ESB Message payload location.

By default, the **ResultType** is “**STRING**”, but it 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.smooks.SmooksAction">
<property name="smooksConfig" value="/smooks/order-to-java.xml" />
<property name="resultType" value="JAVA" />
<property name="javaResultBeanId" value="order" />
</action>
```

Example 11.2. Specifying the Result Type

### PersistAction

This is used to interact with the **MessageStore**, where necessary.

Input Type	Message
Output Type	The input Message
Class	<b>org.jboss.soa.esb.actions.MessagePersister</b>
Properties	<ol style="list-style-type: none"> <li>1. <b>classification</b>: used to classify where the Message will be stored. If the Message Property <b>org.jboss.soa.esb.messagestore.classification</b> is defined on the Message then that will be used instead. Otherwise, a default may be provided at instantiation time.</li> <li>1. <b>message-store-class</b>: the implementation of the <b>MessageStore</b>.</li> <li>2. <b>terminal</b>: if the Action is to be used to terminate a pipeline then this should be "true" (the default). If not, then set this to "false" and the input message will be returned from processing.</li> </ol>
Sample Configuration	<pre>&lt;action name="PersistAction"   class="org.jboss.soa.esb.actions.MessagePersister"&gt; &lt;property name="classification" value="test"/&gt; &lt;property name="message-store-class" value="org.jboss.internal.soa.esb.persistence.format.db.DBMessageStoreImpl"/ &gt; &lt;/action&gt;</pre>

## 11.2. Updated. Business Process Management

### jBPM BpmProcessor

**JBosTjBPM**. The **BpmProcessor** action uses the jBPM command API to make calls into jBPM. The following jBPM commands have been implemented:

- **NewProcessInstanceCommand**
- **StartProcessCommand**
- **CancelProcessInstanceCommand**

Input Type	<b>org.jboss.soa.esb.message.Message</b> generated by <b>AbstractCommandVehicle.toCommandMessage()</b>
Output Type	Message – same as the input message
Class	<b>org.jboss.soa.esb.services.jbpm.actions.BpmProcessor</b>
Properties	<ol style="list-style-type: none"> <li>1. <b>command</b> - required property. Needs to be one of: <b>NewProcessInstanceCommand</b>, <b>StartProcessInstanceCommand</b>, <b>SignalCommand</b> or <b>CancelProcessInstanceCommand</b></li> <li>2. <b>processdefinition</b> – required property for the New- and Start-ProcessInstanceCommands if the process-definition-id property is not used.</li> </ol>

	<p>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.</p> <ol style="list-style-type: none"> <li>1. 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 processdefinition id in jBPM of which you want to create a new instance. This property does not apply to the Signal- and CancelProcessInstanceCommands.</li> <li>2. actor – optional property to specify the jBPM actor id. This applies to the New- and StartProcessInstanceCommands only.</li> </ol>
Properties	<ol style="list-style-type: none"> <li>1. 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.</li> <li>2. transition-name – optional property. This property only applies to the <b>StartProcessInstance-</b> and <b>Signal</b> 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 <b>processdefinition.xml</b>.</li> </ol> <ol style="list-style-type: none"> <li>1. <b>esbToBpmVars</b> - optional property for the New- and <b>StartProcessInstanceCommands</b> and the <b>SignalCommand</b>. This property defines a list of variables that need to be extracted from the <b>EsbMessage</b> and set into jBPM context for the particular process instance. The list consists of mapping elements. Each mapping element can have the following attributes: <ol style="list-style-type: none"> <li>2. esb – required attribute which can contain an MVEL expression to extract a value anywhere from the EsbMessage.</li> <li>3. bpm – optional attribute containing the name which be used on the jBPM side. If omitted the esb name is used.</li> <li>4. default – optional attribute which can hold a default value if the esbMVEL expression does not find a value set in the <b>EsbMessage</b>.</li> </ol> </li> </ol>
Message variables	<p>Finally, some variables can be set on the body of the <b>EsbMessage</b>:</p> <ol style="list-style-type: none"> <li>1. <b>jbpmProcessInstId</b> – required parameter which applies to the <b>Cancel-ProcessInstanceCommand</b> only. It is up to the user make sure this value is set as a named parameter on the <b>EsbMessage</b> body.</li> <li>2. <b>jbpmTokenId</b> or <b>jbpmProcessInstId</b> – either one is a required parameter and applies to the <b>SignalCommand</b> only. The <b>SignalCommand</b> 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 identification and obtain the root token. It is up to</li> </ol>

	the user make sure either the <b>jbpmTokenId</b> or the <b>jbpmProcessInstId</b> is set on the <b>EsbMessage</b> body.
Sample Configuration	<pre>&lt;action name="create_new_process_instance" class="org.jboss.soa.esb.services.jbpm.actions.BpmProcessor"&gt; &lt;property name="command" value="StartProcessInstanceCommand" / &gt; &lt;property name="process-definition-name" value="processDefinition2"/&gt; &lt;property name="actor" value="FrankSinatra"/&gt; &lt;property name="esbToBpmVars"&gt; &lt;!-- esb-name maps to getBody().get("eVar1") --&gt; &lt;mapping esb="eVar1" bpm="counter" default="45" /&gt; &lt;mapping esb="BODY_CONTENT" bpm="theBody" /&gt; &lt;/property&gt; &lt;/action&gt;</pre>

### 11.3. Updated. Scripting

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

#### GroovyActionProcessor

Executes a [Groovy](http://groovy.codehaus.org/)<sup>8</sup> action processing script, receiving the message, **payloadProxy**, action configuration and logger as variable input.

Script Variable Bindings	<ol style="list-style-type: none"> <li>1. <b>message</b>: The message.</li> <li>2. <b>payloadProxy</b>: Utility for message payload (<b>MessagePayloadProxy</b>).</li> <li>3. <b>config</b>: The action configuration (<b>ConfigTree</b>).</li> </ol>
Class	4. <b>logger</b> : The <b>GroovyActionProcessor</b> 's static Log4J Logger (Logger). <b>org.jboss.soa.esb.actions.scripting.GroovyActionProcessor</b>
Properties	<ol style="list-style-type: none"> <li>1. <b>script</b>: Path (on classpath) to <a href="http://groovy.codehaus.org/">Groovy</a><sup>9</sup> script.</li> <li>2. <b>supportMessageBasedScripting</b>: Allow scripts within the message.</li> <li>3. <b>cacheScript</b>: Should the script be cached. Default is "true."</li> </ol>
Sample Configuration	<pre>&lt;action name="process" class="org.jboss.soa.esb.scripting.GroovyActionProcessor"&gt; &lt;property name="script" value="/scripts/myscript.groovy"/&gt; &lt;/action&gt;</pre>

<sup>8</sup> <http://groovy.codehaus.org/>



## ScriptingAction

Executes a script using the Bean Scripting Framework ([BSF<sup>10</sup>](#)), receiving the message, **payloadProxy**, action configuration and logger as variable input. Some notes:

1. JBoss ESB 4.4 contains [BSF<sup>11</sup>](#) 2.3.0, which has less language support than [BSF<sup>12</sup>](#) 2.4.0 (for example: no [Groovy<sup>13</sup>](#), and non-functioning [Rhino<sup>14</sup>](#)). A future version will contain [BSF<sup>15</sup>](#) 2.4.0, which will support [Groovy<sup>16</sup>](#) and [Rhino<sup>17</sup>](#).
2. [BSF<sup>18</sup>](#) does not provide an API to pre-compile, cache and reuse scripts. Because of this, each execution of the **ScriptingAction** will go through the compile step again. Please keep this in mind while evaluating your performance requirements.
3. When including [BeanShell<sup>19</sup>](#) scripts in your application, it is advised to use a **.beanshell** extension instead of **.bsh**, otherwise the **JBoss BSHDeployer<sup>20</sup>** might pick it up.

Script Variable Bindings	<ol style="list-style-type: none"> <li>1. "message": The message.</li> <li>2. "payloadProxy": Utility for message payload (MessagePayloadProxy).</li> <li>3. "config": The action configuration (ConfigTree).</li> </ol>
Class	<ol style="list-style-type: none"> <li>4. "logger": The ScriptingAction's static Log4J Logger (Logger).</li> </ol> <b>org.jboss.soa.esb.actions.scripting.ScriptingAction</b>
Properties	<ol style="list-style-type: none"> <li>1. "script": Path (on classpath) to script.</li> <li>2. <b>"supportMessageBasedScripting"</b>: Allow scripts within the message.</li> <li>3. "language": Optional script language (overrides extension deduction).</li> </ol>
Sample Configuration	<pre>&lt;action name="process" class="org.jboss.soa.esb.scripting.GroovyActionProcessor"&gt; &lt;property name="script" value="/scripts/myscript.groovy"/&gt; &lt;/action&gt;</pre>

## 11.4. Services

Actions defined within the ESB Services.

### 11.4.1. EJBProcessor

Input Type	EJB method name and parameters
Output Type	EJB specific object
Class	<b>org.jboss.soa.esb.actions.EJBProcessor</b>

Takes an input Message and uses the contents to invoke a Stateless Session Bean. This action supports EJB2.x and EJB3.x.

<sup>10</sup> <http://jakarta.apache.org/bsf/>

Property	Description	Required
ejb3	Boolean flag to indication if the call is to an EJB3.x Session Bean	
ejb-name	The identity of the EJB. Optional when <code>ejb3</code> is <code>true</code> .	
jndi-name	Relevant JNDI lookup.	
initial-context-factory	JNDI lookup mechanism.	
provider-url	Relevant provider.	
method	EJB method name to invoke.	
ejb-params	The list of parameters to use when invoking the method and where in the input Message they reside.	
esb-out-var	The location of the output. Default value is <b>DEFAULT_EJB_OUT</b> .	No

Table 11.1. EJBProcessor Properties

```
<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>
```

Example 11.3. Sample Configuration for EJB 2.x

```

<action name="EJBTest" class="org.jboss.soa.esb.actions.EJBProcessor">
  <property name="ejb3" value="true" />
  <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>

```

Example 11.4. Sample Configuration for EJB 3.x

## 11.5. Updated. 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*<sup>21</sup>.

Class	<b>org.jboss.soa.esb.actions.Aggregator</b>
Properties	1. "timeoutInMillies": This is optional. The timeout time in milliseconds before the aggregation process times out.
Sample Configuration	<pre> &lt;action class="org.jboss.soa.esb.actions.Aggregator" name="Aggregator"&gt;   &lt;property name="timeoutInMillies" value="60000"/&gt; &lt;/action&gt; </pre>

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.

<sup>21</sup> <http://www.enterpriseintegrationpatterns.com/Aggregator.html>

### EchoRouter

Simply echos the incoming message payload to the info log stream and returns the input Message from the process method

### HttpRouter

Currently, there are two **HttpRouter** actions in the code base, one that uses JBoss Remoting to perform the HTTP invocation and other that uses **Apache Commons HttpClient**. This section will describe both. Please note that the **JBoss Remoting HttpRouter** is now deprecated, in order to avoid the confusion that having two could potentially cause.

### JBoss Remoting HttpRouter (Deprecated)

This instance will forward the incoming message to a Uniform Resource Locator for further processing.

Class	<b>org.jboss.soa.esb.actions.routing.HttpRouter</b>
Properties	1. <b>"routeUrl"</b> the endpoint to forward the message. If not set, then <b>localhost:5400</b> will be used.

### Apache Commons HttpRouter

This action allows you to invoke external (ESB unaware) HTTP endpoints from an ESB action pipeline. This action uses **Apache CommonsHttpClient** under the bonnet.

Class	<b>org.jboss.soa.esb.actions.routing.http.HttpRouter</b>
Properties	<p>"endpointUrl" the endpoint to forward the message. Required.</p> <p>"http-client-property" Optional. The <b>HttpRouter</b> uses the <b>HttpClientFactory</b> to create and configure the <b>HttpClient</b> instance. You can specify the configuration of the factory by using the file property which will point to a properties file on the local file system, classpath or URI based. See example below to see how this is done. For more information about the factory properties please refer to: <a href="http://www.jboss.org/community/docs/DOC-9969">http://www.jboss.org/community/docs/DOC-9969</a></p> <p>"method" Currently only supports GET and POST. Required.</p> <p>"responseType" Specifies in what form the response should be sent back. Either <b>STRING</b>(default) or <b>BYTES</b>.</p> <p>"headers" To be added to the request. Supports multiple header name="test" value="testvalue" elements. Optional.</p> <p><b>"http-client-property"</b> The <b>HttpRouter</b> uses the <b>HttpClientFactory</b> to create and configure the <b>HttpClient</b> instance. You can specify the configuration of the factory by this setting.</p>
Sample Configuration	<pre>&lt;action name="httprouter" class="org.jboss.soa.esb.actions.routing.http.HttpRouter"&gt;   &lt;property name="endpointUrl" value="http://host:80/blah"&gt;     &lt;http-client-property name="file" value="/ht.props"/&gt;   &lt;/property&gt;   &lt;property name="method" value="GET"/&gt; &lt;/action&gt;</pre>

```
<property name="responseType" value="STRING"/>
<property name="headers">
  <header name="blah" value="blahval" ></header>
</property>
</action>
```

## JMSRouter

Routes the incoming message on to JMS.

Class	<b>org.jboss.soa.esb.actions.routing.JMSRouter</b>
Properties	<ol style="list-style-type: none"> <li>1. <b>"unwrap"</b>: Setting this to <b>"True"</b> will extract the message payload from the Message object before sending. <b>"False"</b> (the default) will send the serialised Message object.</li> <li>2. <b>jndi-context-factory</b>: The JNDI context factory to use. The default is <b>"org.jnp.interfaces.NamingContextFactory"</b>.</li> <li>3. <b>jndi-URL</b>: The JNDIURL to use. The default is 127.0.0.1:1099.</li> <li>4. <b>jndi-pkg-prefix</b>: The JNDI naming package prefixes to use. The default is <b>org.jboss.naming:org.jnp.interfaces</b></li> <li>5. <b>connection-factory</b>: The name of the <b>ConnectionFactory</b> to use. Default is <b>"ConnectionFactory"</b>.</li> <li>6. <b>persistent</b>: The <b>JMSDeliveryMode</b> which can be set to <b>"true"</b> (the default) or <b>"false."</b></li> <li>7. <b>priority</b>: The JMS priority to be used. Default is <b>javax.jms.Message.DEFAULT_PRIORITY</b>.</li> <li>8. <b>time-to-live</b>: The JMS Time-To-Live to be used. The default is <b>javax.jms.Message.DEFAULT_TIME_TO_LIVE</b>.</li> <li>9. <b>security-principal</b>: The security principal to use when creating the JMS connection.</li> <li>10. <b>security-credentials</b>: The security credentials to use when creating the JMS connection.</li> <li>11. <b>property-strategy</b>: The implementation of the <b>JMSPropertiesSetter</b> interface, if over-riding the default.</li> <li>12. <b>message-prop</b>: Properties to be set on the message are prefixed with <b>"message-prop."</b></li> </ol>

## EmailRouter

Routes the incoming message to a configured e.-mail account.

Class	<b>org.jboss.soa.esb.actions.routing.email.EmailRouter</b>
-------	--

Properties	<p><b>"unwrap"</b>: Setting to this to <b>"true"</b> will extract the message payload from the Message object before sending, whilst <b>"false"</b> (the default) will send the serialised Message object.</p> <p><b>"host"</b>: The host name of the SMTP server. If not specified will default to the property <b>"org.jboss.soa.esb.mail.smtp.host"</b> in the <b>jbossesb-properties.xml</b> file.</p> <p><b>"port"</b>: The port for the SMTP server. If not specified will default to the property <b>"org.jboss.soa.esb.mail.smtp.port"</b> in the <b>jbossesb-properties.xml</b> file.</p> <p><b>"username"</b>: The username for the SMTP server. If not specified will default to the property <b>"org.jboss.soa.esb.mail.smtp.user"</b> in the <b>jbossesb-properties.xml</b> file.</p> <p><b>"password"</b>: The password for the above username on the SMTP server. If not specified will default to the property <b>"org.jboss.soa.esb.mail.smtp.password"</b> in the <b>jbossesb-properties.xml</b> file.</p> <p><b>"auth"</b>: If set to "true," this will attempt to authenticate the user using the AUTH command. If not specified will default to the property <b>"org.jboss.soa.esb.mail.smtp.auth"</b> in the <b>jbossesb-properties.xml</b> file.</p> <p><b>"from"</b>: The "from" e.-mail address.</p> <p><b>"sendTo"</b>: The destination e.-mail account.</p> <p><b>"subject"</b>: The subject of the e.-mail.</p> <p><b>"messageAttachmentName"</b>: This is the filename of an attachment containing the message payload (optional). If not specified the message payload will be included in the message body.</p> <p><b>"message"</b>: A string to be prepended to the ESB message contents of which the e-mail message consists (optional).</p> <p><b>"ccTo"</b>: Comma-separated list of e.-mail addresses (optional)</p> <p><b>"attachment"</b>: Child elements that contain file that will be added as attachments to the e.-mail sent.</p>
Sample Configuration	<pre>&lt;action name="send-email"   class="org.jboss.soa.esb.actions.routing.email.EmailRouter"&gt;   &lt;property name="unwrap" value="true" /&gt;   &lt;property name="host" value="smtpHost" /&gt;   &lt;property name="port" value="25" /&gt;   &lt;property name="username" value="smtpUser" /&gt;   &lt;property name="password" value="smtpPassword" /&gt;   &lt;property name="from" value="jbossesb@xyz.com" /&gt;   &lt;property name="sendTo" value="system2@xyz.com" /&gt;   &lt;property name="subject" value="Message Subject" /&gt;</pre>

```
</action>
```

## ContentBasedRouter

This is a content (and rules)-based message routing action.

Class	<b>org.jboss.soa.esb.actions.ContentBasedRouter</b>
Properties	<ol style="list-style-type: none"> <li>1. <b>"ruleSet"</b>: JBoss Rules ruleset.</li> <li>2. <b>"ruleLanguage"</b>: CBR-evaluation Domain Specific Language (DSL) file.</li> <li>3. <b>"ruleReload"</b>: Flag indicating whether or not the rules file should be reloaded each time. The default is <b>"false"</b>.</li> <li>4. <b>"destinations"</b>: Container property for the &lt;route-to&gt; configurations. <ol style="list-style-type: none"> <li>1. &lt;route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/&gt;</li> </ol> </li> </ol>
"process" methods	<ol style="list-style-type: none"> <li>1. <b>"process"</b>: Do not append aggregation data to the message.</li> <li>2. <b>"split"</b>: Append aggregation data to the message.</li> </ol> <p>See the Aggregator action.</p>
Sample Configuration	<pre>&lt;action process="split" name="ContentBasedRouter" class="org.jboss.soa.esb.actions.ContentBasedRouter"&gt; &lt;property name="ruleSet" value="MyESBRules-XPath.drl"/&gt; &lt;property name="ruleLanguage" value="XPathLanguage.dsl"/&gt; &lt;property name="ruleReload" value="true"/&gt; &lt;property name="destinations"&gt; &lt;route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/&gt; &lt;route-to destination-name="normal" service-category="NormalShipping" service-name="NormalShippingService"/&gt; &lt;/property&gt; &lt;/action&gt;</pre> <p>See the "What is Content-Based Routing?" chapter in the <i>Services Guide</i> for more details on Content-Based Routing.</p>

## 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	<b>org.jboss.soa.esb.actions.StaticRouter</b>
Properties	<ol style="list-style-type: none"> <li>1. <b>"destinations"</b>: Container property for the &lt;route-to&gt; configurations.</li> </ol>

	<ol style="list-style-type: none"> <li>1. <code>&lt;route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/&gt;</code></li> </ol>
"process" methods	<ol style="list-style-type: none"> <li>1. "process": Don't append aggregation data to message.</li> <li>2. "split": Append aggregation data to message.</li> </ol> <p>See the Aggregator action.</p>
Sample Configuration	<pre>&lt;action name="routeAction" class="org.jboss.soa.esb.actions.StaticRouter"&gt; &lt;property name="destinations"&gt; &lt;route-to service-category="ExpressShipping" service-name="ExpressShippingService"/&gt; &lt;route-to service-category="NormalShipping" service-name="NormalShippingService"/&gt; &lt;/property&gt; &lt;/action&gt;</pre>

### StaticWiretap

The **StaticWiretap** action differs from the **StaticRouter** in that the **StaticWiretap** "listens in" on the action chain and allows actions below it to be executed. By contrast, the **StaticRouter** action terminates the action chain at the point it is used. A **StaticRouter** should, therefore, be the last action in a chain.

Class	<b>org.jboss.soa.esb.actions.StaticWiretap</b>
Properties	<ol style="list-style-type: none"> <li>1. "destinations": Container property for the <code>&lt;route-to&gt;</code> configurations.</li> </ol> <ol style="list-style-type: none"> <li>1. <code>&lt;route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/&gt;</code></li> </ol>
"process" methods	<ol style="list-style-type: none"> <li>1. "process": Do not append aggregation data to message.</li> </ol> <p>See the Aggregator action.</p>
Sample Configuration	<pre>&lt;action name="routeAction" class="org.jboss.soa.esb.actions.StaticWiretap"&gt; &lt;property name="destinations"&gt; &lt;route-to service-category="ExpressShipping" service-name="ExpressShippingService"/&gt; &lt;route-to service-category="NormalShipping" service-name="NormalShippingService"/&gt; &lt;/property&gt; &lt;/action&gt;</pre>

### EmailWiretap

The **EmailWiretap** will publish the ESB Message payload to a configured e.-mail account.



Class	<b>org.jboss.soa.esb.actions.routing.email.EmailWiretap</b>
Properties	<p><b>“host”</b>: The host name of the SMTP server. If not specified, it will default to the property <b>'org.jboss.soa.esb.mail.smtp.host'</b> in <b>jbossesb-properties.xml</b>.</p> <p><b>“port”</b>: The port for the SMTP server. If not specified, it will default to the property <b>'org.jboss.soa.esb.mail.smtp.port'</b> in <b>jbossesb-properties.xml</b>.</p> <p><b>“username”</b>: The username for the SMTP server. If not specified, it will default to the property <b>"org.jboss.soa.esb.mail.smtp.user"</b> in the <b>jbossesb-properties.xml</b> file.</p> <p><b>“password”</b>: The password for the above username on the SMTP server. If not specified will default to the property <b>"org.jboss.soa.esb.mail.smtp.password"</b> in the <b>jbossesb-properties.xml</b> file.</p> <p><b>“auth”</b>: If true will attempt to authenticate the user using the AUTH command. If not specified, it will default to the property <b>'org.jboss.soa.esb.mail.smtp.auth'</b> in the <b>jbossesb-properties.xml</b> file.</p> <p><b>“from”</b>: The "from" e.-mail address.</p> <p><b>“sendTo”</b>: The destination e.-mail account.</p> <p><b>“subject”</b>: The subject of the e.-mail.</p> <p><b>“messageAttachmentName”</b>: This is the filename of an attachment containing the message payload (optional). If not specified, the message payload will be included in the message body.</p> <p><b>“message”</b>: A string to be pre-pended to the ESB message contents which make up the e.-mail message (optional)</p> <p><b>“ccTo”</b>: Comma-separated list of email addresses (optional)</p> <p><b>“attachment”</b>: Child elements that contain file that will be added as attachments to the email sent.</p>
Sample Configuration	<pre>&lt;action name="send-email"   class="org.jboss.soa.esb.actions.routing.email.EmailWiretap"&gt;   &lt;property name="host" value="smtpHost" /&gt;   &lt;property name="port" value="25" /&gt;   &lt;property name="username" value="smtpUser" /&gt;   &lt;property name="password" value="smtpPassword" /&gt;   &lt;property name="from" value="jbossesb@xyz.com" /&gt;   &lt;property name="sendTo" value="systemX@xyz.com" /&gt;   &lt;property name="subject" value="Important message" /&gt; &lt;/action&gt;</pre>

## 11.6. Notifier

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

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

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

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

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

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

```

</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	<ol style="list-style-type: none"> <li>1. append – if value is true, append the notification to an existing file</li> <li>2. URI – any valid URI specifying a file</li> </ol>
Sample Configuration	<pre> &lt;target class="NotifyFiles" &gt;   &lt;file append="true" URI="anyValidURI"/&gt;   &lt;file URI="anotherValidURI"/&gt; &lt;/target&gt; </pre>
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	<ol style="list-style-type: none"> <li>1. driver-class</li> <li>2. connection-url</li> <li>3. user-name</li> <li>4. password</li> <li>5. table – table in which notification record is stored</li> <li>6. dataColumn – name of table column in which ESB message contents are stored</li> </ol>

Child	column
Child Attributes	<ol style="list-style-type: none"> <li>1. name – name of table column in which to store additional value</li> <li>2. value – value to be stored</li> </ol>
Sample Configuration	<pre> &lt;target class="NotifySQLTable" driver-class="com.mysql.jdbc.Driver" connection-url="jdbc:mysql://localhost/db" user-name="user" password="password" table="table" dataColumn="messageData"&gt; &lt;column name="aColumnName" value="aColumnValue"/&gt; &lt;/target&gt; </pre>
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	<ol style="list-style-type: none"> <li>1. URL – a valid FTP URL</li> <li>2. filename – the name of the file to contain the ESB message content on the remote system</li> </ol>
Sample Configuration	<pre> &lt;target class="NotifyFTP" &gt; &lt;ftp URL="ftp://username:pwd@server.com/remote/dir" filename="someFile.txt" /&gt; &lt;/target&gt; </pre>

### 11.6.1. Webservices/SOA-P

**Updated.** SOAPProcessor

**JBoss** Webservices SOA-P Processor.

This action supports invocation of a **JBossWS**-hosted webservice endpoint through any **JBossESB**-hosted listener. This means that the ESB can be used to expose Webservice endpoints for Services that do not already do so. You can do this by writing a thin Service Wrapper Webservice, (such as a JSR 181 implementation), that wraps calls to the target Service (that does not 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 (such as HTTP, FTP or JMS.)

### Dependencies

1. **JBoss** Application Server
2. The `soap.esb` Service. This is available in the `lib` folder of the distribution.

### "ESB Message Aware" Webservice Endpoints

Please note that a Webservice endpoint exposed via this action will 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 (like the Webservice `.war` is bundled inside the `.esb`) and outside (such as standalone Webservice `.war` deployments, Webservice `.war` deployments bundled inside an `.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. In other words, it is not supported on the **JBossESB** Server.

### SOAPClient

The **SOAPClient** action uses the **Wise** Client Service to generate a JAXWS client class and call the target service.

Example configuration:

```
<action name="soap-wise-client-action"
  class="org.jboss.soa.esb.actions.soap.wise.SOAPClient">
  <property name="wsdl" value="http://host:8080/OrderManagement?wsdl"/>
  <property name="SOAPAction" value="http://host/OrderMgmt/SalesOrder"/>
</action>
```

#### Example 11.5. Wise Client Service

Optional Properties

Property Name	Description
<code>wsdl</code>	The WSDL to be used.
<b><code>operationName</code></b>	The name of the operation as specified in the webservice <b>WSDL</b> .
<b><code>SOAPAction</code></b>	The endpoint operation. This is now superceded by <b><code>operationName</code></b> .
<b><code>EndPointName</code></b>	The EndPoint invoked. Webservices can have multiple endpoints. If this is not specified, then the first designated in WSDL will be used.
<b><code>SmooksRequestMapper</code></b>	Specifies a <b>Smooks</b> configuration file to define the <b>Java-to-Java</b> mapping for the request.

<b>SmooksResponseMapper</b>	Specifies a <b>Smooks</b> configuration file to define the <b>Java-to-Java</b> mapping defined for the response
<b>ServiceName</b>	A symbolic service name used by <b>Wise</b> to cache object generation and/or use an already-generated object. If it is not provided, <b>Wise</b> uses the "servlet" name of WSDL.
<b>UserName</b>	This is the user name used if the webservice is protected by BASIC Authentication HTTP.
Password	Password used if the webservice is protected by BASIC Authentication HTTP.
<b>smooks-handler-config</b>	It is often necessary to be able to transform the SOA-P request or response, especially in header. This may be to simply add some standard SOA-P handlers. <b>Wise</b> supports two JAXWS SOA-P Handlers, both custom and predefined ones based upon <b>Smooks</b> .  Transformation of the SOA-P request (before sending) is supported by configuring the <b>SOAPClient</b> action with a <b>Smooks</b> transformation configuration property.
<b>custom-handlers</b>	It is also possible to provide a set of custom standard <b>JAX-WS Soap Handler</b> . The parameter accept a list of classes implementing <b>SoapHandler</b> interface. Classes have to provide full qualified name and be separated by semi-columns.
<b>LoggingMessages</b>	It is useful for debugging purposes to view <b>SOA-P</b> Message sent and response received. <b>Wise</b> achieve this goal using a JAX-WS handler printing all messages exchanged on <b>System.out</b> . This is a Boolean value.

### SOA-P Operation Parameters

The SOA-P operation parameters are supplied in one of two ways:

- As a Map instance set on the default body location (**Message.getBody().add(Map)**)
- As a Map instance set on the named body location (**Message.getBody().add(String, Map)**), whereby 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 two ways:

1. With a set of Objects of any type. In this case, a **Smooks** configuration has to be specified in action attribute "**SmooksRequestMapper**" and **Smooks** is used to make the **Java-to-java** conversion.
2. With a set of String based key-value pairs(<String, Object>), where the key is the name of the SOA-P parameter as specified in wsdl (or in generated class) to be populated with the key's value. SOA-P Response Message Consumption.

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

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

2. On a named body location (**Message.getBody().add(String, Map)**), where the name of that body location is specified as the value of the "**responseLocation**" action property.

The response object instance can also be populated (from the SOA-P response) in one of two ways:

1. With a set of Objects of any type. In this case, a **Smooks** configuration has to be specified in action attribute **SmooksResponseMapper** and **Smooks** is used to make the **Java-to-Java** conversion
2. With a set of String based key-value pairs(<String, Object>), where the key is the name of the SOA-P answer as specified in wsdl (or in generated class) to be populated with the key's value. JAX-WS is the handler for the SOA-P request/response.

For examples of using the **SOAPClient**, please refer to the following Quick Starts:

1. **webservice\_consumer\_wise**, shows basic usage.
2. **webservice\_consumer\_wise2**, shows how to use '**SmooksRequestMapper**' and '**SmooksResponseMapper**.'
3. **webservice\_consumer\_wise3**, shows how to use '**smooks-handler-config**.'
4. **webservice\_consumer\_wise4**, shows usage of '**custom-handlers**.'

More information about **Wise** can be found on this website: <http://www.javainuxlabs.org/wise><sup>22</sup>.

### JAXB Annotation Introductions

The native **JBossWS** SOA-P stack uses JAXB to bind with SOA-P. This means that an unannotated typeset cannot be used to build a **JBossWS** endpoint. To overcome this, there is a **JBossESB** and **JBossWS** feature called "JAXB Annotation Introductions" which allows you to 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.

### Quick Starts

A number of Quick Starts demonstrating how to use this action are available in the **JBossESB** distribution (samples/quickstarts). See the "**webservice\_bpel**" Quick Start for more information.

### SOAPClient

SOA-P Client action processor.

Uses the [soapUI](http://www.soapui.org/)<sup>23</sup> Client Service to construct and populate a message for the target service. This action then routes that message to that service.

<sup>22</sup> <http://www.javainuxlabs.org/wise/index.html>

<sup>23</sup> <http://www.soapui.org/>

### 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>
```

#### Example 11.6. Endpoint Operation Specification

### SOA-P Request Message Construction

The SOA-P operation parameters are supplied in one of two 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 two ways:

1. Option 1: With a set of Objects that are accessed (for SOA-P message parameters) using the [OGNL](http://www ognl.org/)<sup>24</sup> framework. There is more information about 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 SOA-P parameter to be populated with the key's value. There is more information about the use of OGNL below.

As stated above, [OGNL](http://www ognl.org/)<sup>25</sup> is the mechanism we use for selecting the SOA-P parameter values to be injected into the SOA-P message from the supplied parameter Map. The OGNL expression for a specific parameter within the SOA-P message depends on the position of that parameter within the SOA-P body. In the following message:

---

<sup>25</sup> <http://www.ognl.org/>



```
<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>
```

### Example 11.7. OGNL

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 does not yield a value, this parameter location within the SOA-P message will remain blank.

Taking the sample message above and using the "Option 1" approach to populating the "**customerNumber**" requires an object instance (such as 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 (in other words, a "Header" object instance). The object instance behind the "header" property (via 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>

```

#### Example 11.8. OGNL Associated with a Collection

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 Two (above) provides a "quick-and-dirty" way to populate a SOA-P message without having to create an Object model as per Option One. The OGNL expressions that correspond with the SOA-P operation parameters are exactly the same as for Option One, with the exception that there is no 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 SOA-P message template as it is 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 switched on outside of development usage.

## SOA-P Response Message Consumption

The SOA-P 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 SOA-P response) in one of three ways:

1. Option One: As an Object Graph created and populated by the *XStream*<sup>26</sup> toolkit<sup>27</sup>.
2. Option Two: As a set of String based key-value pairs(<String, String>), where the key is an OGNL expression identifying the SOA-P response element and the value is a String representing the value from the SOA-P message.
3. Option Three: If Options One or Two are not specified in the action configuration, the raw SOA-P response message (String) is attached to the message.

Using *XStream*<sup>28</sup> as a mechanism for populating an Object Graph (Option One 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 One) is configured on the action as follows:

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

### Example 11.9. XStream

In the above example, there is also a demonstration of how to specify non-default named locations for the request parameters Map and response object instance.

<sup>28</sup> <http://xstream.codehaus.org/>

You will also find provided, 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>
```

### Example 11.10. XStream Annotated

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 SOA-P response data extracted into an OGNL keyed map (Option Two 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>
```

### Example 11.11. SOA-P Response Data Extract

To return the raw SOA-P message as a String (Option Three), simply omit both the "**responseXStreamConfig**" and "**responseAsOgnlMap**" properties.

## 11.6.2. Updated. HttpClient Configuration

The **SOAPClient** uses Apache Commons **HttpClient** to execute **SOAP** requests. It uses the **HttpClientFactory** to create and configure the **HttpClient** instance. Specifying the **HttpClientFactory** configuration on the **SOAPClient** is very easy. Just add an additional property to the "**wsdl**" property as follows:

```
<property name="wsdl"
  value="https://localhost:18443/active-bpel/services/RetailerCallback?
wsdl">
  <http-client-property name="file" value="/localhost-
https-18443.properties" ></http-client-property>
</property>
```

### Example 11.12. Adding a "wsdl" Property

The "file" property value will be evaluated as a filesystem, classpath or URI-based resource (in that order).

The following is an example of this property set:

```
# Configurators
configurators=HttpProtocol,AuthBASIC

# HttpProtocol config...
protocol-socket-
factory=org.apache.commons.httpclient.contrib.ssl.EasySSLProtocolSocketFactory
keystore=/packages/jakarta-tomcat-5.0.28/conf/chap8.keystore
keystore-passw=xxxxxx
https.proxyHost=localhost
https.proxyPort=443

# AuthBASIC config...
auth-username=tomcat
auth-password=tomcat
authscope-host=localhost
authscope-port=18443
authscope-realm=ActiveBPEL security realm
```

#### Example 11.13. Configuring a "file" Property

Properties may also be set directly on the action configuration like so:

```
<property name="http-client-properties">
  <http-client-property name="http.proxyHost" value="localhost"/>
  <http-client-property name="http.proxyPort" value="8080"/>
</property>
```

#### Example 11.14. Setting Properties on the Action Configuration

For more information about the configuration options available, please refer to the wiki page.

### 11.6.3. **Updated.** SOAPProxy

The intention of the WS Proxy is to analyse the consumption of an external Web Service endpoint, (whether it be one that is hosted on **.NET**, another external **Java**-based Application Server or LAMP), and re-publication of such an endpoint via the **ESB**. The **ESB** sits between the ultimate consumer/client (for instance, a **.NET** WinForm application) and the ultimate producer (which could be a RoR-hosted WS). The purpose of this intermediary is to provide an abstraction layer that provides a number of advantages:

1. It results in looser coupling between the client and service, making them mutually unaware of each other.
2. The client no longer has a direct connection to the remote service's hostname/IP address.

3. The client will see modified **WSDL** which changes the inbound/outbound parameters. At a minimum, the **WSDL** must be tweaked so that the client is pointed to the **ESB's** exposed endpoint, instead of the original, now proxied, endpoint.
4. A transformation of the **SOA-P** envelope/body can be introduced via the **ESB** action chain both for the inbound request and outbound response (see **XsltAction** or **SmooksAction**.)
5. Service versioning is possible, since clients can connect to two or more proxy endpoints on the **ESB**, each of which can have its own **WSDL** and/or transformations and routing requirements. The **ESB** will be able to send the appropriate message to the appropriate endpoint and provide an ultimate response.

Other mechanisms for doing this are inappropriate or inadequate:

1. **SOAPClient** is used to invoke external web services, not to mirror them.
2. **SOAPProducer** only executes internally-deployed **JBoss WS** services.
3. **HttpRouter** requires too much manual configuration for easy proxying.
4. "message": A message prefix.
5. **EBWS** strips out the **SOA-P** Envelope and only passes along the body.

A SOAPProxy action:

1. Is both a producer and consumer of web services.
2. Only requires a property pointing to the external wsdl.
3. **wSDL** can be automatically transformed via the optional **wSDLTransform** property.
4. Understands that **SOA-P** is not tied to **http**. The **wSDL** is read, and if an **http** transport is defined, that will be used. Other transports (**jms**) will need future consideration.
5. If using **http**, any of the **HttpRouter** properties can also be applied optionally as overrides.

Class	<b>org.jboss.soa.esb.actions.soap.proxy.SOAPProxy</b>
Properties	<p>"<b>wSDL</b>" (required): The uniform resource locator of the target web service's wsdl.</p> <p>"<b>wSDLTransform</b>" (optional): A <b>smooks-resource-list</b> XML configuration file, allowing for flexible wsdl transformation.</p> <p>* (optional): Any of the <b>HttpRouter</b> properties can be applied.</p>
Sample Configuration	<pre>&lt;action name="wsproxy"   class="org.jboss.soa.esb.actions.soap.proxy.SOAPProxy"&gt;   &lt;property name="wSDL" value="http://127.0.0.1:8080/path/HelloWorldWebService?wSDL"/&gt; &lt;/action&gt;</pre>

### 11.6.4. 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	<b>org.jboss.soa.esb.actions.SystemPrintln</b>
Properties	<ol style="list-style-type: none"><li>1. "message": A message prefix.</li><li>1. "<b>printfull</b>": If true then the entire message is printed, otherwise just the byte array and attachments.</li><li>2. "<b>outputstream</b>": if true then <b>System.out</b> is used, otherwise <b>System.err</b>.</li></ol>
Sample Configuration	<pre>&lt;action name="print-before"   class="org.jboss.soa.esb.actions.SystemPrintln"&gt;   &lt;property name="message" value="Message before action XXX" /&gt; &lt;/action&gt;</pre>





## Developing Custom Actions

In order to implement a custom Action Processor, simply use the `org.jboss.soa.esb.actions.ActionPipelineProcessor` interface.

This interface supports the implementation of *managed life-cycle* stateless actions. A single instance of a class that implements this interface is instantiated on a "per-pipeline" basis (in other words, per-action configuration.) This means that one can cache the resources needed by the action in the `initialise` method, and then clean them up by using the `destroy` method.

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

It should be convenient to simply extend the

**`org.jboss.soa.esb.actions.AbstractActionPipelineProcessor`**:

```
public class ActionXXXProcessor extends AbstractActionPipelineProcessor {
    public void initialise() throws ActionLifecycleException {
        // Initialize resources...
    }
    public Message process(final Message message) throws
    ActionProcessingException {
        // Process messages in a stateless fashion...
    }
    public void destroy() throws ActionLifecycleException {
        // Cleanup resources...
    }
}
```

Example 12.1. `public class ActionXXXProcessor extends AbstractActionPipelineProcessor`

### 12.1. Configuring Actions Using Properties

Actions generally act as templates. They require external configuration in order to perform their tasks. For example, a **PrintMessage** action might use a property named `message` to indicate what to print and another property called `repeatCount` to indicate the number of times to print it. If so, the action configuration in the `jboss-esb.xml` file would look something like this:

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

Example 12.2. Hello World Example

A **ConfigTree** instance is the default method to use in order to load property values into an action implementation. The **ConfigTree** provides a DOM-like view of the action XML code. 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);
        }
    }
}
```

### Example 12.3. ConfigTree Example

One may take another approach to setting properties by adding "setters" on the action that will correspond to the property names. This will thereby allow the framework to populate them automatically. The action class must implement the **org.jboss.soa.esb.actions.BeanConfiguredAction** marker interface in order to make the action Bean populate automatically. The following class has the same behavior as that shown above, in order to demonstrate this:

```
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);
        }
    }
}
```

### Example 12.4. Action Bean Auto-Populate

**Note**

The Integer parameter in the `setRepeatCount ( )` method 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 a better option in situations when one needs to deal with the XML representation directly.



# Connectors and Adapters

## 13.1. Introduction

Not all of the JBoss Enterprise Service Bus' clients and services will be able to understand the protocols and Message formats that it uses natively. Because of this, there is a need to be able to bridge between ESB-aware endpoints (those that understand JBossESB) and ESB-unaware endpoints (those that do not understand JBossESB.) Such bridging technologies have existed for many years in a variety of distributed systems and are often referred to as Connectors, Gateways or Adapters.

One of the aims of JBossESB is to allow a wide variety of clients and services to interact. JBossESB does not require that all such clients and services be written using JBossESB or any ESB for that matter. There is an abstract notion of an Interoperability Bus within JBossESB, such that endpoints that may not be JBossESB-aware can still be "plugged into" the bus.



### Note

In what follows, the terms "within the ESB" or "inside the ESB" refer to ESB-aware endpoints.

All JBossESB-aware clients and services communicate with one another using Messages, (these will be described later.) A Message is simply a standardized format for information exchange, containing a header, body (payload), attachments and other data. Additionally, all JBossESB-aware services are identified using Endpoint References (EPRs).

It is important for legacy interoperability scenarios that a SOA infrastructure such as JBossESB allow ESB-unaware clients to use ESB-aware services, or ESB-aware clients to use ESB-unaware services. The concept that JBossESB uses to facilitate this interoperability is through Gateways. A gateway is a service that can bridge between the ESB-aware and ESB-unaware worlds and translate to/from Message formats and to/from EPRs.

JBossESB currently supports Gateways and Connectors. In the following sections we shall examine both concepts and illustrate how they can be used.

## 13.2. The Gateway

Not all users of JBossESB will be ESB-aware. In order to facilitate those users interacting with services provided by the ESB, JBossESB has the concept of a "Gateway," which is a specialized server that can accept messages from non-ESB clients and services and route them to the required destination.

A Gateway is a specialized listener process, that behaves very similarly to an ESB-aware listener. There are some important differences however:

- Gateway classes can pick up arbitrary objects contained in files, JMS messages, SQL tables and so forth (each 'gateway class' is specialized for a specific transport), whereas JBossESB listeners can only process JBossESB normalized Messages as described in "The Message" section of this document. However, those Messages can contain arbitrary data.

- Only one action class is invoked to perform the 'message composing' action. ESB listeners are able to execute an action processing pipeline.
- Objects that are 'picked up' will be used to invoke a single 'composer class' (the action) that will return an ESB Message object. This will then be delivered to a target service that must be ESB-aware. The target service defined at configuration time, will be translated at runtime into an EPR (or a list of EPRs) by the Registry. The underlying concept is that the EPR returned by the Registry is analogous to the 'toEPR' contained in the header of ESB Messages but because incoming objects are 'ESB-unaware' and there is, thus, no dynamic way to determine the toEPR, this value is provided to the gateway at configuration time and included in all outgoing messages.

There are a few off-the-shelf composer classes; the default 'file' composer class will just package the file contents into the Message body; with the same process occurring for JMS messages. The default message composing class for an SQL table row is to package contents of all columns specified in the configuration, into a `java.util.Map`.

Although these default composer classes will be adequate for most uses, it is relatively straightforward for users to provide their own. The only requirements are that, firstly, they must have a constructor that takes a single `ConfigTree` argument, and, secondly, they must provide a 'Message composing' method (whereby the default name is 'process' but this can be configured differently in the 'process' attribute of the `<action>` element within the `ConfigTree` provided at constructor time.) The processing method must take a single argument of type `Object`, and return a `Message` value.

From JBossESB 4.5 onwards, the `FileGateway` accepts the `file-filter-class` configuration attribute which allows you to define a `FileFilter` implementation that may be used to select the files used by the gateway in question. Initialisation of user-defined `FileFilter` instances is performed by the gateway if the instance is also of type `org.jboss.soa.esb.listeners.gateway.FileGatewayListener.FileFilterInit`, in which case the `init` method will be called and passed to the gateway `ConfigTree` instance.

By default, the following `FileFilter` implementations are defined and used by the `FileGateway`: if an input suffix is defined in the configuration then files matching that suffix will be returned; alternatively if there is no input suffix then any file is accepted as long as it does not match the work suffix, error suffix and post suffix.

### 13.2.1. **This section has changed.** Gateway Data Mappings

When a non-JBossESB message is received by a Gateway, it must be converted to a `Message`. How this is done and where in the `Message` the received data resides, depends upon the type of Gateway. How this conversion occurs depends upon the type of Gateway; the default conversion approach is described below:

#### JMS Gateway

If the input message is a `JMSTextMessage`, then the associated `String` will be placed in the default named `Body` location; if it is an `ObjectMessage` or a `BytesMessage` then the contents are placed within the `BytesBody.BYTES_LOCATION` named `Body` location.

#### Local File Gateway

The contents are placed within the `BytesBody.BYTES_LOCATION` named `Body` location.

#### Hibernate Gateway

The contents are placed within the `ListenerTagNames.HIBERNATE_OBJECT_DATA_TAG` named `Body` location.

### Remote File Gateway

The contents are placed within the *BytesBody*. *BYTES\_LOCATION* named Body location.



#### Note

With the introduction of the InVM transport, it is now possible to deploy services within the same address space (VM) as a gateway, improving the efficiency of gateway-to-listener interactions.

## 13.2.2. How to change the Gateway Data Mappings

If you want to change how this mapping occurs then it will depend upon the type of Gateway:

### File Gateways

Instances of the `org.jboss.soa.esb.listeners.message.MessageComposer` interface are responsible for performing the conversion. To change the default behavior, provide an appropriate implementation that defines your own `compose` and `decompose` methods. The new `MessageComposer` implementation should be provided in the configuration file using the `composer-class` attribute name.

### JMS and Hibernate Gateways

These implementations use a reflective approach for defining composition classes. Provide your own `Message` composer class and use the `composer-class` attribute name in the configuration file to inform the Gateway which instance to use. You can use the `composer-process` attribute to inform the Gateway which operation of the class to call when it needs a `Message`; this method must take an `Object` and return a `Message`. If not specified, a default name of process is assumed.



#### Note

Whichever of the methods you use to redefine the `Message` composition, it is worth noting that you have complete control over what is in the `Message` and not just the `Body`. For example, if you want to define `ReplyTo` or `FaultTo` EPRs for the newly created `Message`, based on the original content, sender etc., then you should consider modifying the header too.

## 13.3. Connecting via JCA

You can use JCA Message Inflow as an ESB Gateway. This integration does not use MDBs, but rather ESB's lightweight inflow integration. To enable a gateway for a service, you must first implement an endpoint class. This class is a Java class that must implement the `org.jboss.soa.esb.listeners.jca.InflowGateway` class:

```
public interface InflowGateway
{
    public void setServiceInvoker(ServiceInvoker invoker);
}
```

Example 13.1. `public interface InflowGateway`

The endpoint class must either have a default constructor, or a constructor that takes a `ConfigTree` parameter. This Java class must also implement the messaging type of the JCA adapter you are binding to. Here's a simple endpoint class example that hooks up to a JMS adapter:

```
public class JmsEndpoint implements InflowGateway, MessageListener
{
    private ServiceInvoker service;
    private PackageJmsMessageContents transformer = new
    PackageJmsMessageContents();
    public void setServiceInvoker(ServiceInvoker invoker)
    {
        this.service = invoker;
    }
    public void onMessage(Message message)
    {
        try
        {
            org.jboss.soa.esb.message.Message esbMessage =
            transformer.process(message);
            service.postMessage(esbMessage);
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

### Example 13.2. `public class JmsEndpoint implements InflowGateway`

One instance of the `JmsEndpoint` class will be created per gateway defined for this class. This is not like an MDB that is pooled. Only one instance of the class will service each and every incoming message, so you must write thread safe code.

At configuration time, the ESB creates a **ServiceInvoker** and invokes the `setServiceInvoker` method on the endpoint class. The ESB then activates the JCA endpoint and the endpoint class instance is ready to receive messages. In the `JmsEndpoint` example, the instance receives a JMS message and converts it to an ESB message type. It then uses the `ServiceInvoker` instance to invoke on the target service.



#### Note

The JMS Endpoint class is provided for you with the ESB distribution under `org.jboss.soa.esb.listeners.jca.JmsEndpoint`. It is quite possible that this class would be used over and over again with any JMS JCA inflow adapters.

### 13.3.1. Configuration

A JCA inflow gateway is configured in a `jboss-esb.xml` file. Here's an example:



```

<service category="HelloWorld_ActionESB"
  name="SimpleListener"
  description="Hello World">
  <listeners>
    <jca-gateway name="JMS-JCA-Gateway"
      adapter="jms-ra.rar"
      endpointClass="org.jboss.soa.esb.listeners.jca.JmsEndpoint">
    <activation-config>
      <property name="destinationType" value="javax.jms.Queue"/>
      <property name="destination" value="queue/esb_gateway_channel"/>
    </activation-config>
    </jca-gateway>
  ...
</service>

```

### Example 13.3. JCA InflowGateway

```

<service category="HelloWorld_ActionESB"
  name="SimpleListener"
  description="Hello World">
  <listeners>
    <jca-gateway name="JMS-JCA-Gateway"
      adapter="jms-ra.rar"
      endpointClass="org.jboss.soa.esb.listeners.jca.JmsEndpoint">
    <activation-config>
      <property name="destinationType" value="javax.jms.Queue"/>
      <property name="destination" value="queue/esb_gateway_channel"/>
    </activation-config>
    </jca-gateway>
  ...
</service>

```

### Example 13.4. HelloWorld Action ESB

JCA gateways are defined in `<jca-gateway>` elements. These are the configurable attributes of this XML element.

Attribute	Required	Description
name	yes	The name of the gateway
adapter	yes	The name of the adapter you are using. In JBoss it is the file name of the RAR you deployed, e.g., <b>jms-ra.rar</b>
endpointClass	yes	The name of your end point class
messagingType	no	The message interface for the adapter. If you do not specify one, ESB will guess based on the endpoint class.

Attribute	Required	Description
transacted	no	Default to true. Whether or not you want to invoke the message within a JTA transaction.

Table 13.1. jca-gateway Configuration Attributes

You must define an <activation-config> element within <jca-gateway>. This element takes one or more <property> elements which have the same syntax as action properties. The properties under <activation-config> are used to create an activation for the JCA adapter that will be used to send messages to your endpoint class. This is really no different than using JCA with MDBs.

You may also have as many <property> elements as you want within <jca-gateway>. This option is provided so that you can pass additional configuration to your endpoint class. You can read these through the ConfigTree passed to your constructor.

### 13.3.2. Updated. Mapping Standard Activation Properties

A number of ESB properties are automatically mapped onto the activation configuration using an **ActivationMapper**. The properties, their location and their purpose are described in the following table:

Attribute	Location	Description
maxThreads	jms-listener	The maximum number of messages which can be processed concurrently
dest-name	jms-message-filter	The JMS destination name.
dest-type	jms-message-filter	The JMS destination type, QUEUE or TOPIC
selector	jms-message-filter	The JMS message selector
providerAdapterJNDI	jms-jca-provider	The JNDI location of a Provider Adapter which can be used by the JCA inflow to access a remote JMS provider. This is a JBoss-specific interface, supported by the default JCA inflow adapter and may be used, if necessary, by other in-flow adapters.

Table 13.2. Mapping Standard Activation Properties

The mapping of these properties onto an activation specification can be overridden by specifying a class which implements the ActivationMapper interface and can be declared globally or within each ESB deployment configuration.

Specifying the **ActivationMapper** globally is done through the **jbossesb-properties.xml** file. It defines the default mapper used for the specified JCA adapter. The name of the property to be configured is **org.jboss.soa.esb.jca.activation.mapper."adapter name"** and the value is the classname of the ActivationMapper.

The following snippet the configuration of the default ActivationMapper used to map the properties on the the activation specification for the JBoss JCA adapter, **jms-ra.rar**:

```
<properties name="jca">
  <property name="org.jboss.soa.esb.jca.activation.mapper.jms-ra.rar"
    value="org.jboss.soa.esb.listeners.jca.JBossActivationMapper"/>
</properties>
```

#### Example 13.5. Default Activation Manager

Specifying the **ActivationMapper** within the deployment will over-ride any global setting. The mapper can be specified within the listener, the bus or the provider with the precedence being the same order.

The following snippet shows an example specifying the mapper configuration within the listener configuration:

```
<jms-listener name="listener" busidref="bus" maxThreads="100">
  <property name="jcaActivationMapper" value="TestActivationMapper"/>
</jms-listener>
```

#### Example 13.6. Mapping Configurations

The following snippet shows an example specifying the mapper configuration within the bus configuration:

```
<jms-bus busid="bus">
  <property name="jcaActivationMapper" value="TestActivationMapper"/>
  <jms-message-filter dest-type="TOPIC" dest-name="DestName"/>
</jms-bus>
```

#### Example 13.7. Bus Configurations

The following snippet shows an example specifying the mapper configuration within the provider configuration.

```
<jms-jca-provider name="provider" connection-factory="ConnectionFactory">
  <property name="jcaActivationMapper" value="TestActivationMapper"/>
  <jms-bus busid="bus">
    <jms-message-filter dest-type="TOPIC" dest-name="DestName"/>
  </jms-bus>
</jms-jca-provider>
```

#### Example 13.8. Provider Configurations

---

---

# Appendix A. Writing JAXB Annotation Introduction Configurations

The configurations for the JAXB (*Java Architecture for XML Binding*) Annotation Introduction are very easy to write. If the reader is already familiar with the JAXB Annotations, there should be no difficulty in writing a *JAXB Annotation Introduction* configuration.

The *XML Schema Definition* (XSD) for the configuration is available online at <http://anonsvn.jboss.org/repos/jboss/ws/projects/jaxb/intros/tags/1.0.0.GA/src/main/resources/jaxb-intros.xsd>. One must register this XSD against the <http://www.jboss.org/xsd/jaxb/intros> namespace in one's IDE.

At present, only three annotations are supported:

## @XmlType

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

## @XmlElement

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

## @XmlAttribute

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

The basic structure of the configuration file follows the that of a Java class (that is, a "Class" containing "Fields" and "Methods".) The <Class>, <Field> and <Method> elements all require a "name" attribute. This attribute provides the name of the Class, Field or Method. This name attribute's value is able to support regular expressions. This allows a single Annotation Introduction configuration to be targeted at more than one Class, Field or Member (by, for example, setting the name-space for a field in a Class, or for all Classes in a package.)

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. (One should use the XSD and an IDE to edit the configuration.)

Finally, here is 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>
```

---

# Appendix B. Service Orientated Architecture Overview

JBossESB is a Service Oriented Architecture (SOA) infrastructure. SOA represents a popular architectural paradigm for applications development. While the principles behind SOA have existed for many years and it does not necessarily require the use of web services, it is these that popularised it.

Web services implement capabilities that are available to other applications (or even other web services) via industry standard network and application interfaces and protocols. SOA advocates an approach in which one software component provides its functionality as a service that can be leveraged by others. Components (or services) thus represent reusable software building blocks.

SOA allows the integration of existing systems, applications and users into a flexible architecture that can easily accommodate changing needs. Integrated design, reuse of existing IT investments and, above all, industry standards are the elements needed to create a robust SOA.

As enterprises slowly emerge from the mad rush of cost reduction into a more stable period of cost management, many of them find themselves in unfamiliar territory. Prior to the economic slow down, most firms understood the options they had for IT investment. Many embarked on major package implementations (installing products such as Siebel, PeopleSoft and so on), while others built on the legacy systems they had trusted for years. Either way, most firms recognized the return promised and made the investment. Today, the appetite for such large investment is gone.

However, enterprises still need to make progress and keep ahead of the competition. SOA (and Web Services as a concrete implementation of those principles) make this possible. The result is dramatic improvements in collaboration between users, applications and technology components, generating significant value for any business creating competitive advantage.

Imagine a company that has existing software from a variety of different vendors, such as SAP and PeopleSoft. Some of these software packages may be used to conduct business with other companies (customers, suppliers, etc.) and therefore what the company would like to do is to take those existing systems and make them available to other firms, by exposing them as services. A service here is some software component with a stable, published interface that can be invoked by "clients" ( which are other software components). So, requesting and executing services involves software components owned by one company talking to components owned by another company, in other words, business-to-business (B2B) transactions.

Conventional distributed system infrastructures (middleware) are not sufficient for these cross-organizational exchanges. For instance

- You would need agreement between the parties involved on the middleware platform.
- There is an implicit (and sometimes explicit) lack of trust between the parties involved.
- Business data is confidential and should only to be seen by the intended recipient.
- Many assumptions of conventional middleware are invalid in cross-organizational interactions. Transactions, for instance, last longer, possibly for hours or days, so conventional transaction protocols such as "two phase commit" are not applicable.

So, in B2B exchanges, the lack of standardization across middleware platforms makes point-to-point solutions costly to realize in practice. The Internet alleviated some of these problems by providing

standard interaction protocols (HTTP) and data formats (XML) but, by themselves, these standards are not enough to support application integration. They do not define interface definition languages, name and directory services, transaction protocols and so forth. It is the gap between that which the Web provides and what application integration requires that Web services are trying to fill.

However, whilst the ultimate goal of SOA is inter-company interactions, services do not need to be accessed using the Internet. They can be easily made available to clients residing on a local network. It is common for web services to be used in this context to provide integration between systems operating within a single company.

As demonstration of how web services can connect applications to each other both within and between companies, consider a stand-alone inventory system. If you do not connect it to anything else, it is not as valuable as it could otherwise be. The system can track inventory but not do much more. Inventory information may have to be entered separately in the accounting and customer relationship management systems. The inventory system may be unable to automatically place orders to suppliers. The benefits of such an inventory system are diminished by high overhead costs.

However, if you connect your inventory system to your accounting software with XML, it becomes more interesting. Now, whenever you buy or sell something, the implications for your inventory and your cash flow can be tracked in one step. If you go further, and connect your warehouse management system, customer ordering system, supplier ordering systems, and your shipping company with XML, suddenly that inventory management system is worth a lot. You can do end-to-end management of your business while dealing with each transaction only once, instead of over and over for every system it affects. This results in a lot less work and opportunities for errors. These connections can be made easily using Web services.

Businesses are "waking up" to the benefits of SOA. These include:

- opening the door to new business opportunities by making it easy to connect with partners;
- saving time and money by cutting software development time and consuming a service created by others;
- increasing revenue streams by easily making your own services available.

### **B.1. Why SOA?**

The problem space can be categorized by past IT investments in the area of eProcurement, eSourcing, Supply Chain Management, Customer Relationship Management (CRM) and, indeed, Internet computing in general. All of these investments were made in a "silo." The decisions made in this space, (along with the incremental growth of these systems to meet short-term tactical requirements), hurt the long-term viability of the applications and infrastructure.

The three key drivers for implementing an SOA approach are:

#### Cost Reduction

Achieved by the ways in which services talk to each other. The direct cost effect is delivered through enhanced operations productivity, effective sourcing options and a significantly improved ability to shift ongoing costs to a variable model.

#### Delivering IT solutions faster and smarter

A standards-based approach will allow organizations to connect and share information and business processes much faster and easier than before. IT delivery productivity is markedly



improved through simplification of the developer's role by providing standard frameworks and interfaces. Delivery timescales have been drastically reduced by easing the integration load of individual functionality, and applying accelerated delivery techniques within the environment.

#### Maximizing return on investment

Implementation of Web Services opens the way for new business opportunities by enabling new organisational models. Web Services present the ability to measure value and discrete return much differently than traditional functional-benefit methods. Typical Total Cost of Ownership (TCO) models do not take into account the lifetime value generated by historical investment. This cost-centric view destroys many opportunities to exploit these past investments and most enterprises end up building redundancy into their architecture, not out of necessity but of perceived need. These same organizations focus the value proposition of their IT investment on a portfolio of applications, balanced by the overhead of infrastructure. An approach based on Web Services takes into account the lifetime contribution of legacy IT investment and promotes an evolution of these systems rather than a planned replacement.

SOA/Web Services fundamentally changes the way enterprise software is developed and deployed. SOA has evolved to the point where new applications will not be developed using monolithic approaches but, instead, become a virtualized on-demand execution model that breaks the current economic and technological bottleneck that has been caused by traditional approaches.

Software-as-a-service has become a pervasive model for forward-looking enterprises wishing to streamline operations, as it leads to lower cost of ownership and provides competitive differentiation in the marketplace. Using Web Services gives enterprises a viable opportunity to drive significant costs out of software acquisitions, react to rapidly changing market conditions and conduct transactions with business partners at will. Loosely coupled, standards-based architectures are one approach to distributed computing. They allow software resources available on the network to be leveraged. Applications that provide separate business processes, presentation rules, business rules and data access in separate, loosely-coupled layers will not only assist in the construction of better software but also make it more adaptable to future change.

SOA allows you to combining existing functions with new development efforts, resulting in composite applications. The re-use of existing functionality in this way reduces the overall project risk and delivery time-frame. It also improves the overall quality of the software.

Loose coupling helps preserve the future by allowing parts to be changed at their own pace without the risks linked to the costly which occur with monolithic approaches. SOA allows business users to focus on business problems at hand without worrying about technical constraints. SOA helps the individuals who develop the solutions, in the following manner:

- Business analysts can focus on higher-order responsibilities in the development life-cycle while increasing their own knowledge of the business domain.
- Parallel development is enabled by separating functionality into component-based services that can be tackled by multiple teams.
- Quality assurance and unit testing become more efficient; errors can be detected earlier in the development life-cycle.
- Development teams can deviate from initial requirements without incurring additional risk.
- Components within architecture can become reusable assets so that the business can avoid reinventing the wheel.

- The flexibility, future maintainability and ease of integration efforts is preserved by functional decomposition of services and their underlying components with respect to the business process
- Security rules are implemented at the service level. They can, therefore, solve many security considerations within the enterprise

### B.2. Basics of SOA

Traditional distributed computing environments have been tightly coupled, in the sense that they do not deal with a changing environment at all well. For instance, if one application is interacting with another, how do they handle data types or data encoding if data formats in one system change? How are incompatible data-types handled?

The service-oriented architecture (SOA) consists of three roles: requester, provider, and broker.

#### Service Provider

A service provider allows access to services, creates a description of a service and publishes it to the service broker.

#### Service Requestor

A service requester is responsible for discovering a service by searching through the service descriptions given by the service broker. A requester is also responsible for binding to services provided by the service provider.

#### Service Broker

A service broker hosts a registry of service descriptions. It is responsible for linking a requestor to a service provider.

### B.3. Advantages of SOA

SOA provide several significant benefits for distributed enterprise systems. Some of the most notable benefits include: interoperability, efficiency, and standardization. We will briefly explore each of these in this section.

#### B.3.1. Interoperability

Interoperability is the ability of software on different systems to communicate by sharing data and functionality. SOA and web services are as much about interoperability as they are about the Web and Internet scale computing. Most companies will have numerous business partners throughout the life of the company. Instead of writing a new addition to your applications every time you gain a new partner, you can write one interface using web service technologies like SOAP. So now your partners can dynamically find the services they need using UDDI and bind to them using SOAP. You can also extend the interoperability of your systems by implementing web services within your own network. With the addition of web services to your own systems, you can reduce the cost integration, increase communication and increase your customer base.

It is also important to note that the industry has even established the Web Services Interoperability Organization.

“The Web Services Interoperability Organization is an open industry effort chartered to promote Web Services interoperability across platforms, applications, and programming languages. The organization brings together a diverse community of Web services leaders to respond to customer needs by

---

providing guidance, recommended practices, and supporting resources for developing interoperable Web services.” (www.ws-i.org)

The WS-I will actually determine whether a Web service conforms to WS-I standards as well as industry standards. In order to establish integrity and acceptance, companies will seek to build their Web services in compliance with the WS-I standards.

### **B.3.2. Efficiency**

SOA will enable you to reuse your existing applications. Instead of creating totally new applications, you can create them using various combinations of services exposed by your existing applications. Developers can be more efficient because they can focus on learning industry standard technology. They will not have to spend a lot of time learning every new technology that arises. For a manager this means a reduction in the cost of buying new software and having to hire new developers with new skill sets. This approach will allow developers to meet changing business requirements and reduce the length of development cycles for projects. Overall, SOA provides for an increase in efficiency by allowing applications to be reused, decreasing the learning curve for developers and speeding up the total development process.

### **B.3.3. Standardization**

For something to be a true standard, it must be accepted and used by the majority of the industry. One vendor or small group of vendors must not control the evolution of the technology or specification. Most if not all of the industry leaders are involved in the development of Web service specifications. Almost all businesses use the Internet and World Wide Web in one form or another. The underlying protocol for the WWW is of course HTTP. The foundation of Web services is built upon HTTP and XML. Although SOA does not mandate a particular implementation framework, interoperability is important and SOAP is one of the few protocols that all good SOA implementations can agree on.

### **B.3.4. Stateful and Stateless Services**

Most proponents of Web Services agree that it is important that its architecture is as scalable and flexible as the Web. As a result, the current interaction pattern for Web Services is based on coarse-grained services or components. The architecture is deliberately not prescriptive about what happens behind service endpoints: Web Services are ultimately only concerned with the transfer of structured data between parties, plus any meta-level information to safeguard such transfers (e.g., by encrypting or digitally signing messages). This gives flexibility of implementation, allowing systems to adapt to changes in requirements, technology etc. without directly affecting users. Furthermore, most businesses will not want to expose their back-end implementation decisions and strategies to users for a variety of reasons.

In distributed systems such as CORBA, J2EE and DCOM, interactions are typically between stateful objects that reside within containers. In these architectures, objects are exposed as individually referenced entities, tied to specific containers and therefore often to specific machines. Because most web services applications are written using object-oriented languages, it is natural to think about extending that architecture to Web Services. Therefore a service exposes web services resources that represent specific states. The result is that such architectures produce tight coupling between clients and services, making it difficult for them to scale to the level of the World Wide Web.

Right now, there are two primary models for the session concept that are being defined by companies participating in defining Web services: the WS-Addressing EndpointReferences with ReferenceProperties/ReferenceParameters and the WS-Context explicit context structure. Both of

these models are supported within JBossESB. The WS-Addressing session model provides coupling between the web service endpoint information and the session data, which is analogous to object references in distributed object systems.

WS-Context provides a session model that is an evolution of the session models found in HTTP servers, transaction, and MOM systems. On the other hand, WS-Context allows a service client to more naturally bind the relationship to the service dynamically and temporarily. The client's communication channel to the service is not impacted by a specific session relationship.

This has important implications as we consider scaling Web services from internal deployments to general services offered on the Internet. The current interaction pattern for web services is based on "coarse-grained" services or components. The architecture is deliberately not prescriptive about what happens behind service endpoints: web services are ultimately only concerned with the transfer of structured data between parties, plus any meta-level information to safeguard such transfers (e.g., by encrypting or digitally signing messages). This gives flexibility of implementation, allowing systems to adapt to changes in requirements, technology etc. without directly affecting users. It also means that issues such as whether or not a service maintains state on behalf of users or their (temporally bounded) interactions, has been an implementation choice not typically exposed to users.

If a session-like model based on WS-Addressing were to be used when interacting with stateful services, then the tight coupling between state and service would impact on clients. As in other distribution environments where this model is used (e.g., CORBA or J2EE), the remote reference (address) that the client has to the service endpoint must be remembered by the client for subsequent invocations. If the client application interacts with multiple services within the same logical session, then it is often the case that the state of a service has relevance to the client only when used in conjunction with the associated states of the other services. This necessarily means that the client must remember each service reference and somehow associate them with a specific interaction; multiple interactions will obviously result in different reference sets that may be combined to represent each sessions.

For example, if there are N services used within the same application session, each maintaining m different states, the client application will have to maintain  $N*m$  reference endpoints. It is worth remembering that the initial service endpoint references will often be obtained from some bootstrap process such as UDDI. But in this model, these references are stateless and of no use beyond starting the application interactions. Subsequent visits to these sites that require access to specific states must use different references in the WS-Addressing model.

This obviously does not scale to an environment the size of the Web. However, an alternative approach is to use WS-Context and continue to embrace the inherently loosely-coupled nature of Web Services. As we have shown, each interaction with a set of services can be modeled as a session, and this in turn can be modeled as a WS-Context activity with an associated context. Whenever a client application interacts with a set of services within the same session, the context is passed on to the services and they map this context to the necessary states that the client interaction requires.

How this mapping occurs is an implementation specific choice that need not be exposed to the client. Furthermore, since each service within a specific session gets the same context, upon later revisiting these services and providing the same context again, the client application can be sure to return to a consistent set of states. So for the N services and m states in our previous example, the client need only maintain N endpoint references and as we mentioned earlier, typically these will be obtained from the bootstrap process anyway. Thus, this model scales much better.

## **B.4. JBossESB and its Relationship with SOA**

SOA is more than technology; it does not come in a shrink-wrapped box and it requires changes to the way in which people work and interact as much as it does assistance from underlying infrastructures, such as JBossESB. With JBossESB 4.3 GA, Red Hat is providing a base SOA infrastructure upon which SOA applications can be developed. With the 4.2.1 release, most of the necessary hooks for SOA development are in place and Red Hat is working with its partners to ensure that their higher level platforms leverage these hooks appropriately. However, the baseline platform (JBossESB) will continue to evolve, with out-of-the-box improvements around tooling, runtime management, service life-cycle etc. In JBossESB 4.3 GA, it may be necessary for developers to leverage these hooks themselves, using low-level API and patterns.

---

---

# Appendix C. Revision History

Revision 3.0    Wed 14 Oct 2009                      David Le Sage [dlesage@redhat.com](mailto:dlesage@redhat.com)  
Revised for JBoss SOA Platform Release 5.0

Revision 1.2    Wed Jul 1 2009                      Darrin Mison [dmison@redhat.com](mailto:dmison@redhat.com)  
Updated for 4.3.CP02 Release  
SOA-1358 - corrected misplaced section of text. Section 11.7  
SOA-1341 - removed reference to a quickstart that is not longer included. Section 11.7.1  
SOA-1335 - removed old webservises configuration details. Section 4.4  
SOA-1001 - updated the JAXB XSD url reference. Appendix A

Revision 1.1    Tue Jan 27 2009                      Darrin Mison [dmison@redhat.com](mailto:dmison@redhat.com)  
Updated for 4.3.CP01 Release

Revision 1.0    Tue 9 Sep 2008                      Darrin Mison [dmison@redhat.com](mailto:dmison@redhat.com)  
Created

---