JBoss Enterprise SOA Platform 4.3

Programmers Guide

A guide for developers using the JBoss Enterprise SOA Platform 4.3 CP05

JBoss Enterprise SOA Platform 4.3 Programmers Guide
A guide for developers using the JBoss Enterprise SOA Platform 4.3 CP05
Edition 4.3.5

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

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*[1] 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 keycaps 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 keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

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

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with 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; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> 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

[1] https://fedorahosted.org/liberation-fonts/

> **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.

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:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings 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 add syntax highlighting 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

Notes are tips, shortcuts or alternative approaches 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 a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings 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."

Refer to *Appendix B, Service Orientated Architecture Overview* to learn more about the principles underlying Service Oriented Architecture and the Enterprise Service Bus.

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:

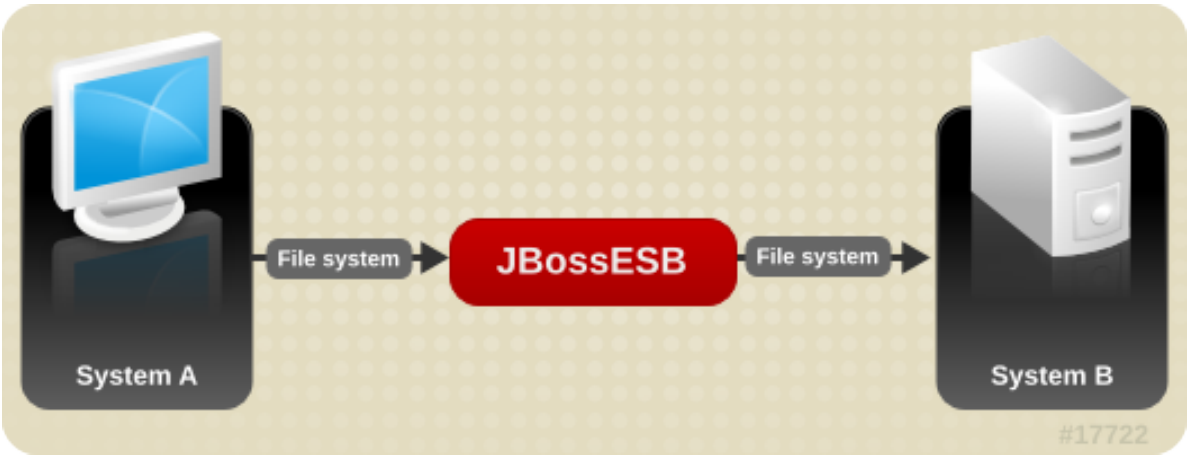


Figure 1.1. Simple File Movement Between Two Systems Without Messaging Queuing

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

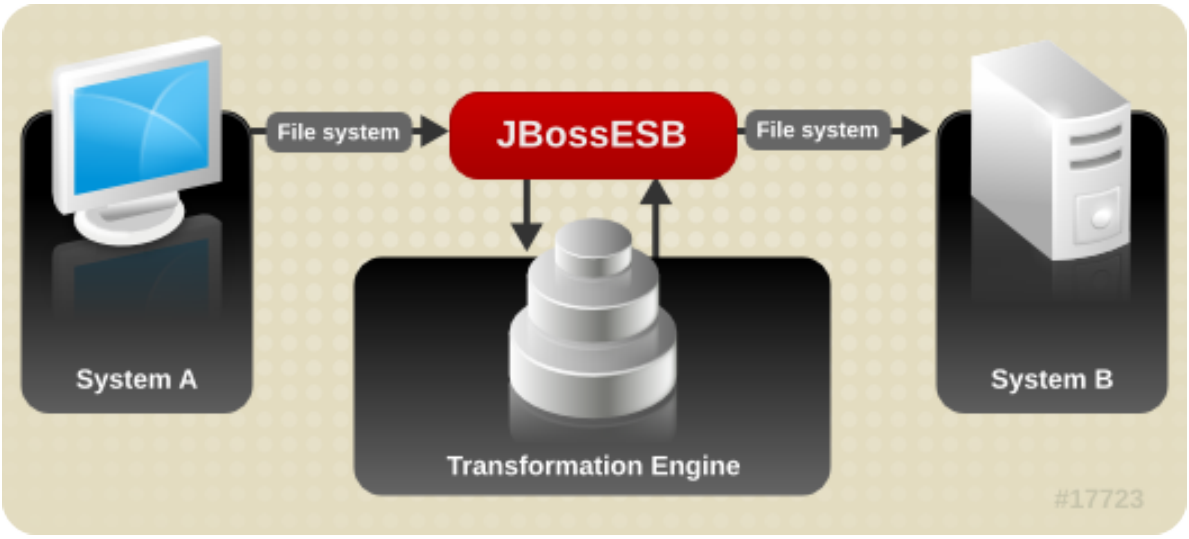


Figure 1.2. Simple File Movement with Transformation Between Two Systems Without Messaging Queuing:

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

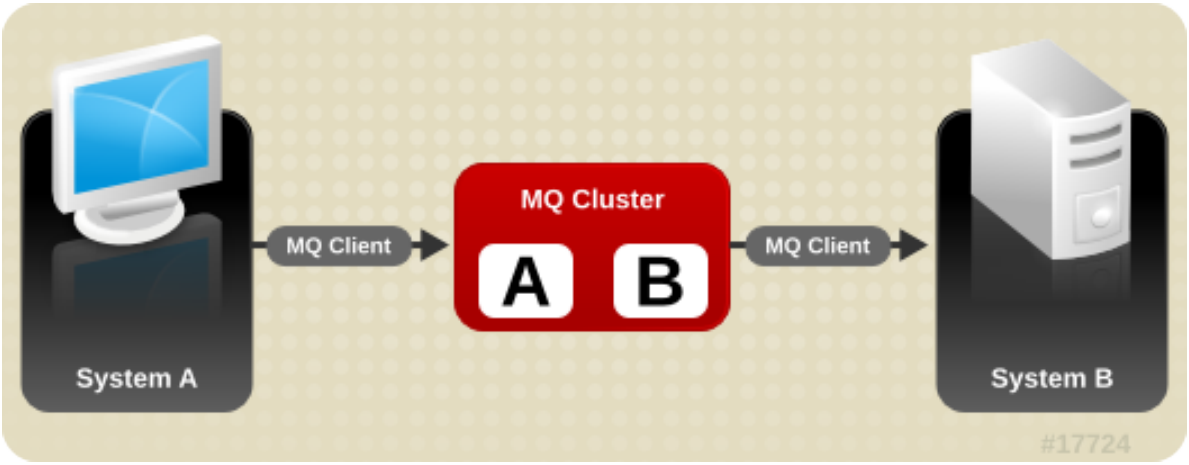


Figure 1.3. Using Messaging Queuing:

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

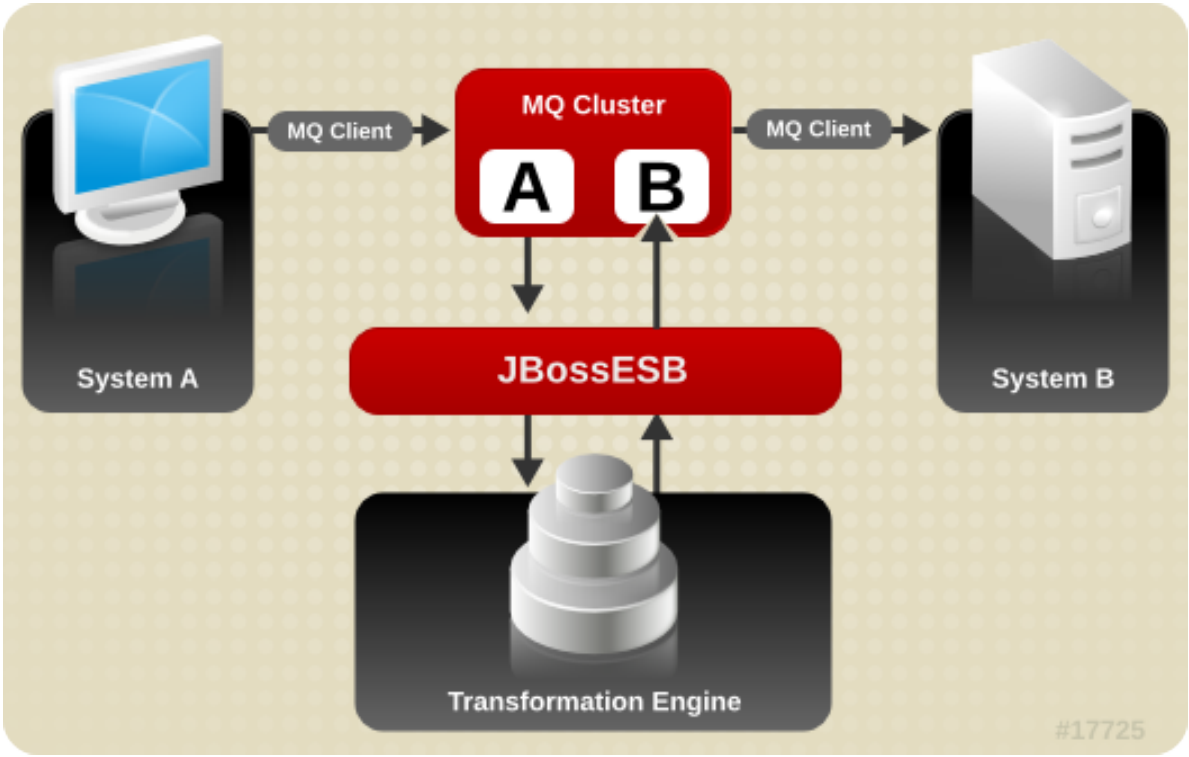


Figure 1.4. Using Messaging Queuing with Transformation

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.

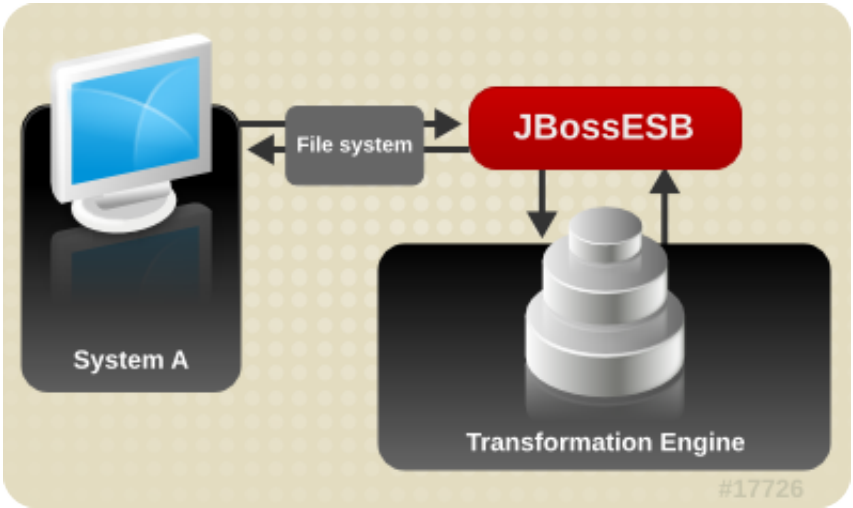


Figure 1.5. Basic Data Transformation via the ESB Using the File System

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

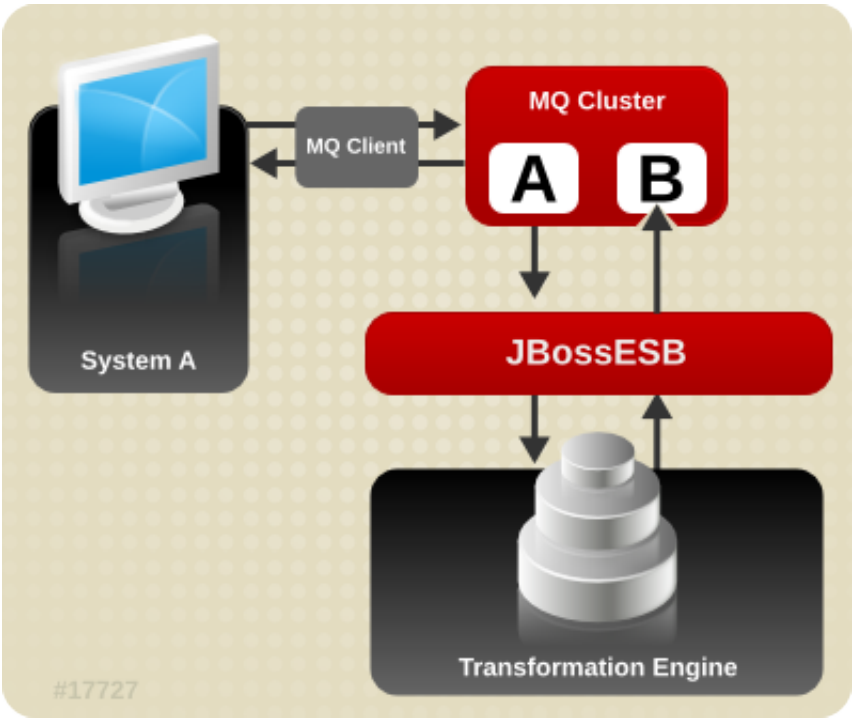


Figure 1.6. Single Party Example Using Transformation and a Queuing System:

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.

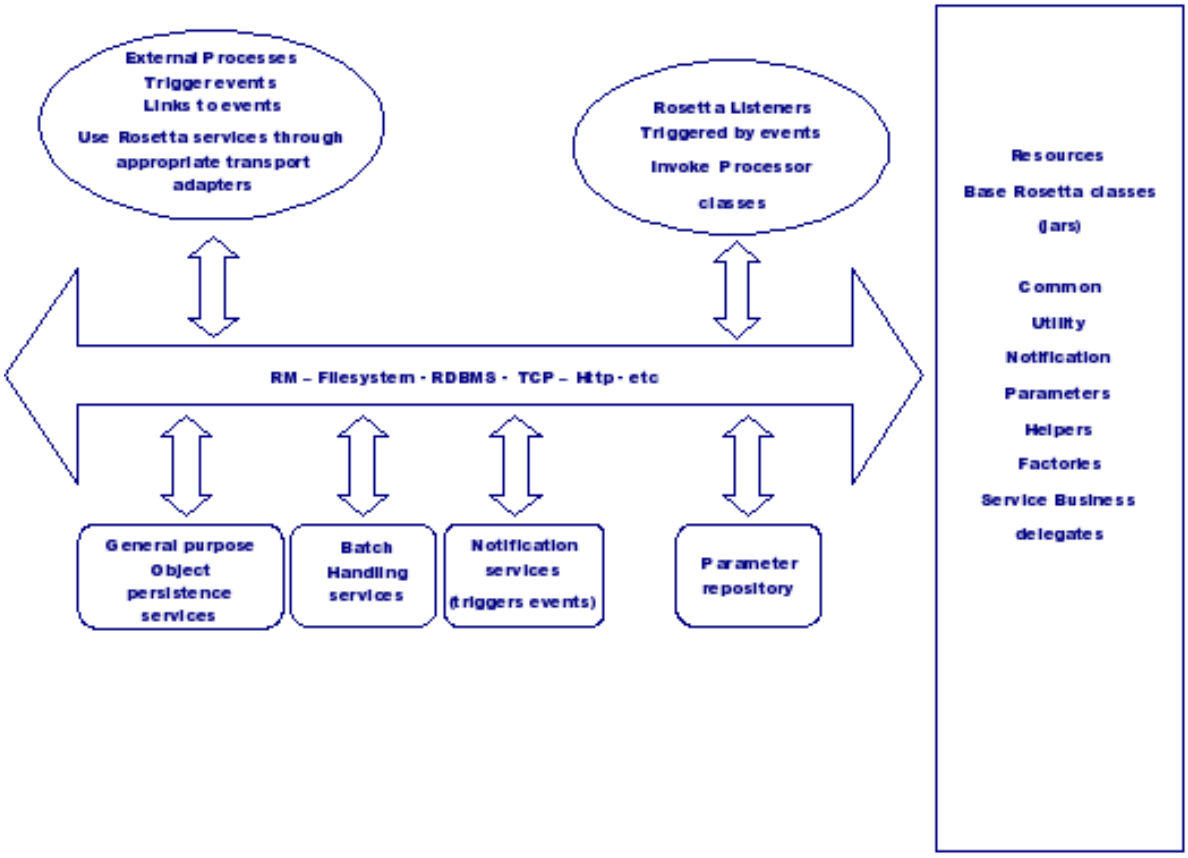


Figure 2.1. The Rosetta Architecture

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 these core architectural components:

- Message Listener and Message Filtering code
- Data transformation using the **`SmooksAction`** action processor
- A Content Based Routing service
- A Message Repository for saving messages & events exchanged within the ESB

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 ESB deployment is depicted below. This diagram shall be discussed further in subsequent sections of the book.

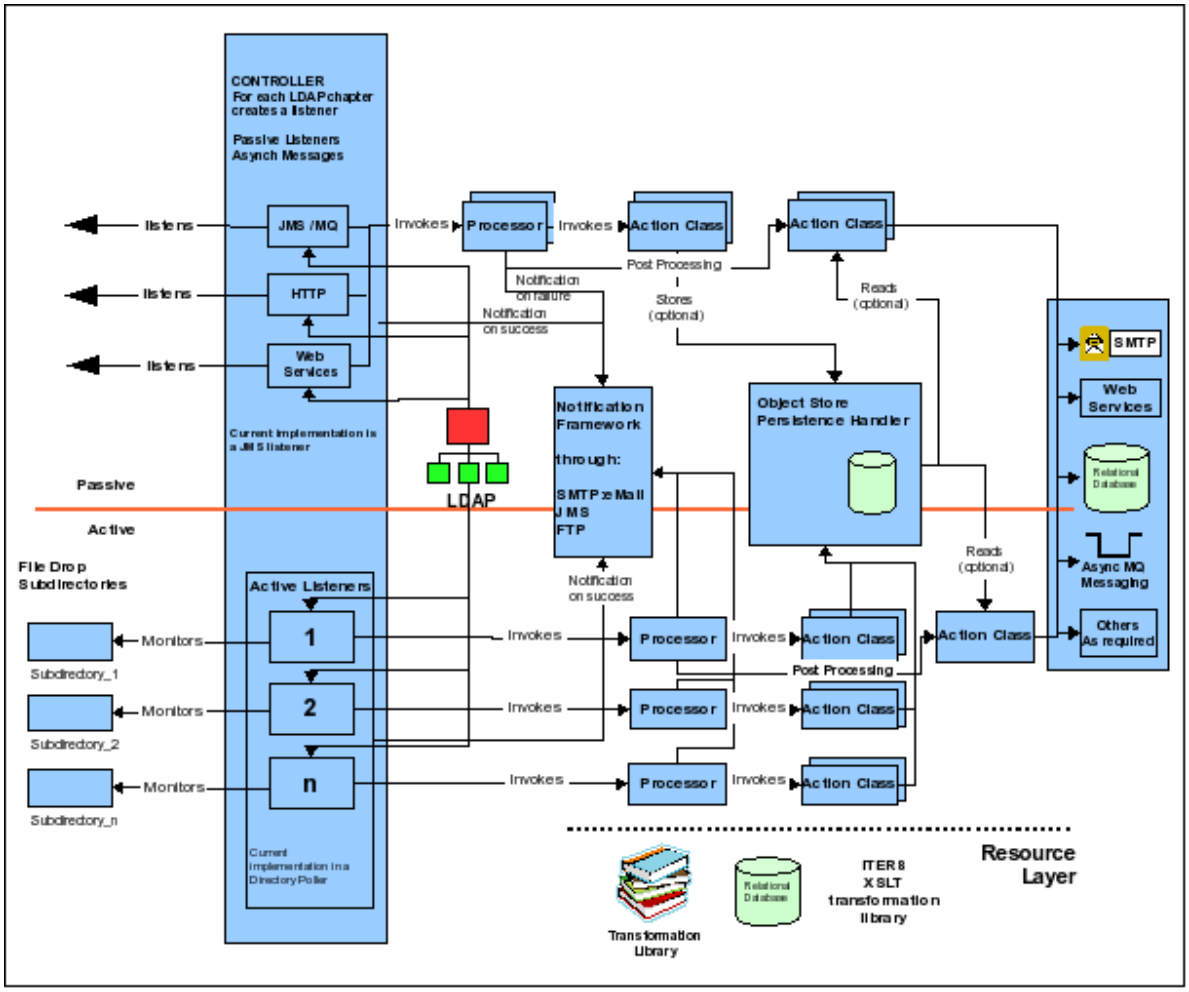


Figure 2.2. Example of a JBoss ESB deployment

Important

Some of the components in *Figure 2.2, "Example of a JBoss ESB 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 the JBoss ESB consists, and how these same components interact and can be exploited to develop service-orientated applications.

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. 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:

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

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb xmlns="http://anonsvn.jboss.org/repos/labs/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>
```

A Service has category and name attributes. When the JBoss ESB deploys the Service it uses these attributes to register the Service's listeners as *endpoints* in the Service Registry. Clients can invoke the Service using the class **`ServiceInvoker`**.

Example 3.2. Invoking the service from the client

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

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

The **`ServiceInvoker`** uses the Service Registry to lookup the available Endpoint addresses for the service "Retail:ShoeStore". It takes care of all the details of getting the message from the Client to one

of the available Service Endpoints. The message transport process is completely transparent to the client.

The Endpoint addresses made available to the ServiceInvoker will depend on the list of listeners configured on the Service, such as JMS, FTP or HTTP. No listeners are configured on the Service in the above example, but its *InVM* listener has been enabled using **`invmScope="GLOBAL"`**. The InVM transport is a new ESB feature in the SOA Platform 4.3 release that provides communication between services running on the same JVM. *Section 4.3.3, "InVM Transport"* contains more information about this feature.

You need to explicitly add listener configurations to a service to enable additional Endpoints.

The JBoss ESB supports two forms of listener configuration:

- *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 consided "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."

Example 3.3. a JMS Gateway listener added to the above ShoeStore Service example

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

    <providers>
       <jms-provider name="JBossMessaging" 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>
```

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 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's paradigm is one of services, not distributed objects.

A one-way message delivery pattern with requests and responses requires the information about where responses should be sent to be encoded in the message. That information may be present in the message body (the payload) and dealt with by the application, or as part of the initial request message and dealt with by the ESB infrastructure.

Central to the ESB is the notion of a message whose structure is similar to that found in SOA:

Example 3.4. Sample ESB Message Schema

```
<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>
```

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.)

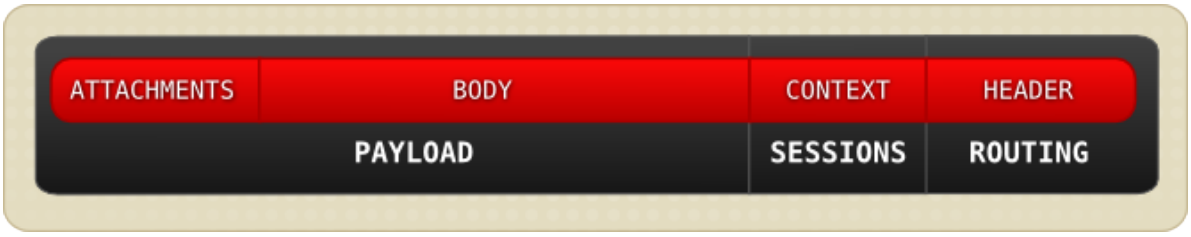


Figure 3.1. Basic Structure of a Message

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

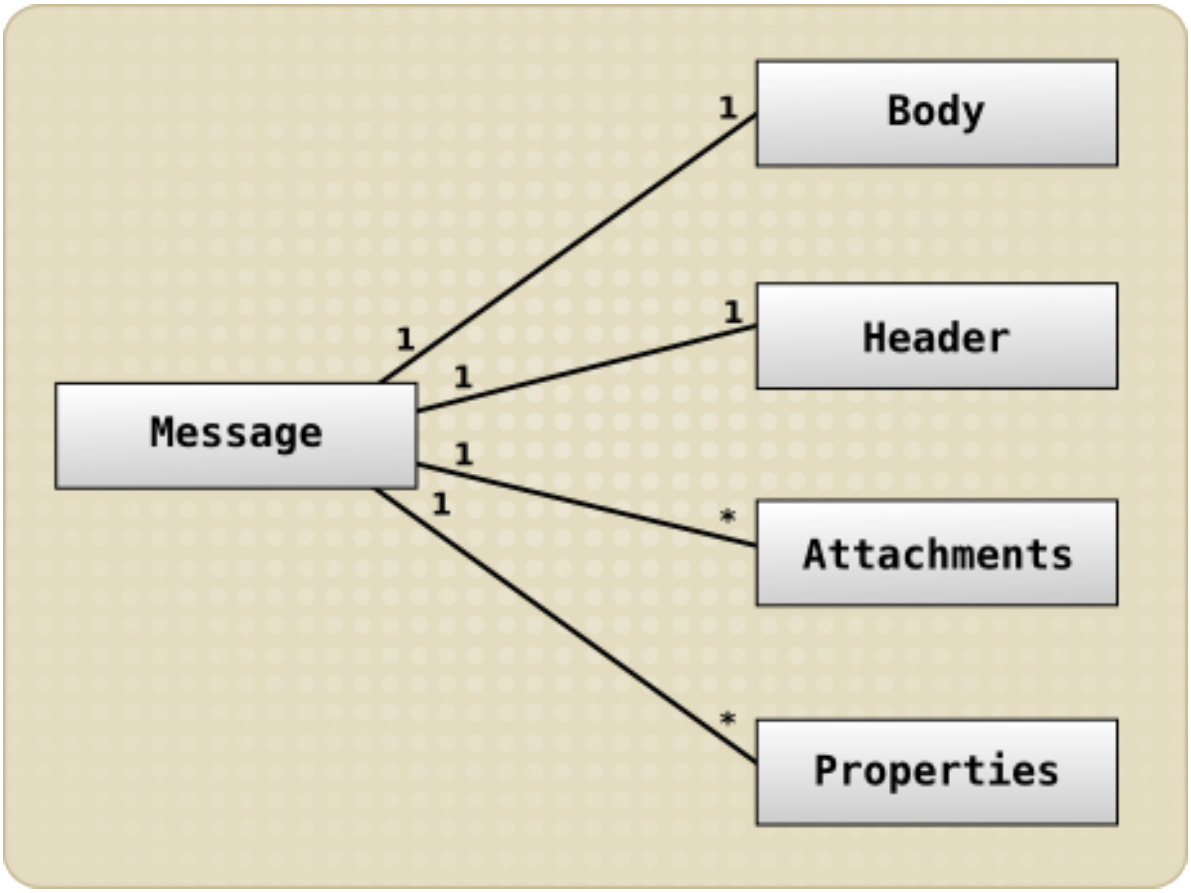


Figure 3.2. The Message Structure Represented as UML

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.)

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

```
public interface Message
```

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

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

Warning

At this time it is recommended that developers do not use Properties or Attachments.

The general concepts they embody are currently being re-evaluated and may change significantly in future releases.

It is recommended that the data for your Properties and Attachments be included as part of the Message Body.

The UML representation of the payload is shown below:

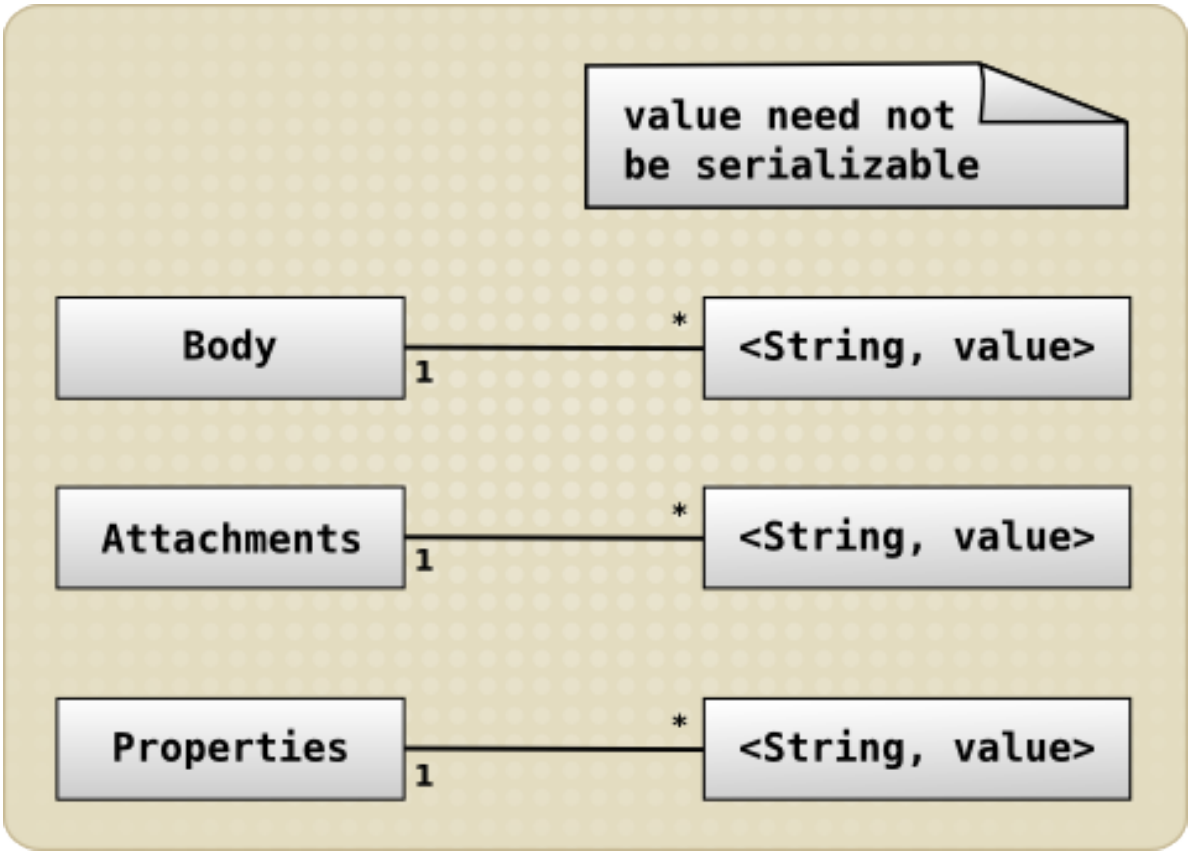


Figure 3.3. UML representation of the message payload

3.2.1. The Header

The Header contains routing and addressing information for the message as Endpoint References (EPRs) as well as information to uniquely identify the message. JBossESB uses an addressing scheme based on the WS-Addressing standard from W3C.

The relationship between the Header and the various EPRs can be illustrated in UML as:

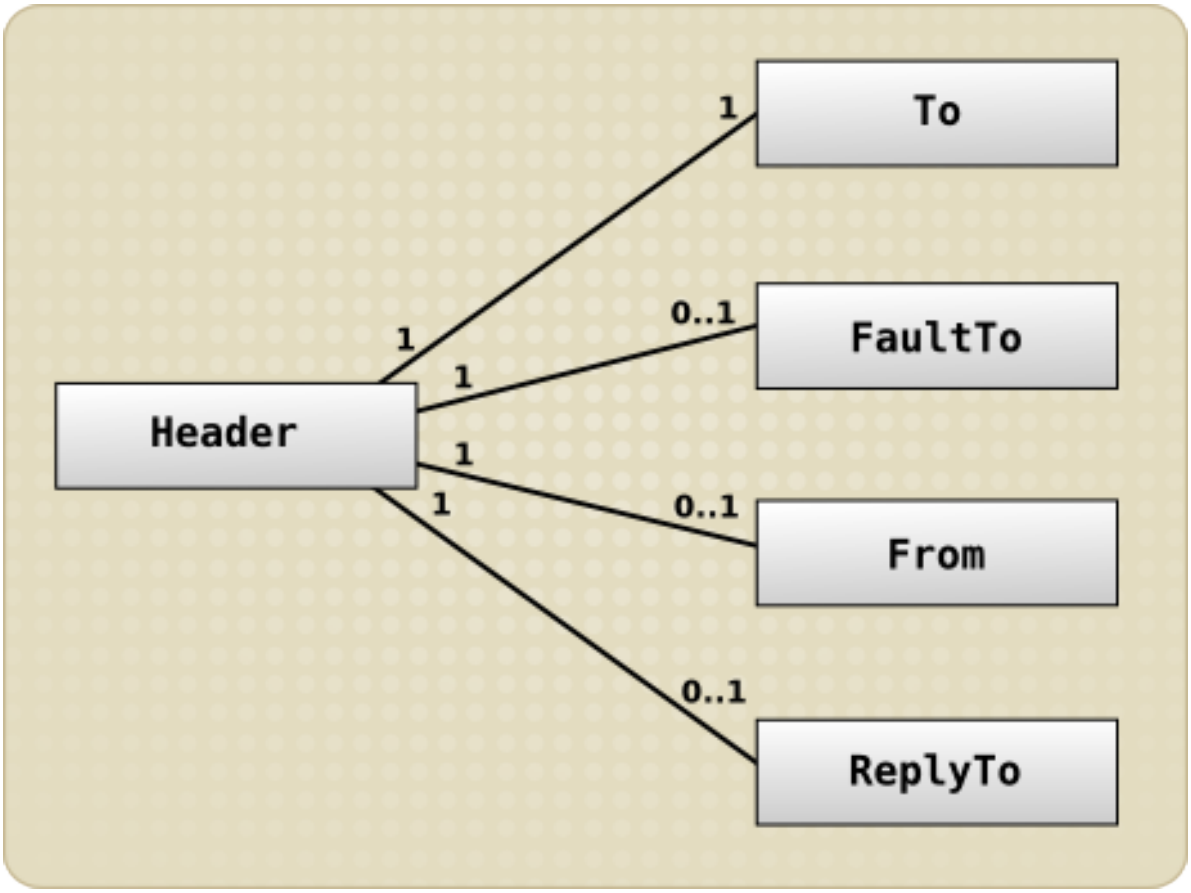


Figure 3.4. Relationship between the Header and ERPs in UML

The role of the header must be considered when developing and using your services. For example, if you require a synchronous interaction pattern based on request and response, you 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 you so choose. Likewise, when sending one-way messages (no response), you should not set the **`ReplyTo`** field because it will be ignored.

Your ReplyTo or FaultTo EPRs should always use the LogicalEPR, as opposed to one of the Physical EPRs (JMS-EPR etc). A LogicalEPR is an EPR that simply specifies the name and category of an ESB Service/Endpoint. 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 EPR (typically the ESB itself, but not necessarily). The client of the LogicalEPR can use the Service name and category details supplied in the EPR to lookup the physical endpoint details for that Service/Endpoint at the point in time when they intend making the invocation i.e. they will get relevant addressing information. The client will also be able to select an physical endpoint type that suits it.

Note

The Message Header is immutable once transmitted between endpoints.

Although the interfaces allow the Header to be modified JBossESB will ignore such changes. It is likely that in future releases the API will disallow such modifications to avoid confusion. These rules are laid down in the WS-Addressing standards.

Example 3.6. The **`org.jboss.soa.esb.message.Header`** interface

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

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

Example 3.7. **`org.jboss.soa.esb.addressing.Call`**

```
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);
}
```

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

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

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.

ReplyTo

The ReplyTo property is an EPR that identifies the intended receiver for replies to this message. The message header must contain a ReplyTo if a reply is expected.

JBossESB supports default ReplyTo values for each type of transport. This is used in situations where a response is required but the ReplyTo property has not been supplied. Some of these defaults require system administrators to configure JBossESB correctly.

Table 3.2. Default ReplyTo by transport

Transport	ReplyTo
JMS	A queue with the same name as the one used to deliver the original request with the suffix of *_reply*.
JDBC	A table in the same database with the same name as the one used to deliver the original request with the suffix of *_reply_table*. 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.

FaultTo

The FaultTo is an EPR that identifies the intended receiver for Faults related to this message. Faults are fully described in *Section 3.2.3, "The Fault"*.

The JBossESB will route any Fault to the EPR in the FaultTo property of the incoming message. If FaultTo is not set, JBossESB will check the ReplyTo and From properties in turn. If no valid EPR is obtained as a result of 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 you do not want any response at all. However it is recommended in such scenarios to use the DeadLetter Queue Service EPR as your FaultTo or any faults that do occur will be saved for later processing.

MessageID

The MessageID property is a URI that is used to uniquely identify each message.

Two different messages must not have the same MessageID, but a re-transmitted message may use the same MessageID as the original.

MessageID must be set if a reply is expected, or if either of the ReplyTo or FaultTo properties are set.

3.2.2. The Context

The Context contains session related information, such as transaction or security contexts. This release of the JBoss ESB does not support user-enhanced Contexts. This will be a feature of the 5.0 release.

3.2.3. The Fault

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

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

```
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);
}
```

3.2.4. The Body

The Body typically contains the payload of the message. You can can use the Body to send an arbitrary number of different data types. You are not restricted to sending and receiving single data items within a Body. How these objects are serialized to and from the message body is up to the specific Object type.

Example 3.9. The **`org.jboss.soa.esb.message.Body`** interface

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

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

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 on a location for a byte array, then you can use the one that JBossESB uses: ByteBody.BYTES_LOCATION.

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

Any type of Object can be added to the Body. If you add objects that are not Java Serializable you must provide JBossESB with the ability to marshal and unmarshal the Message. Refer to *Section 3.2.8, "The MessageFactory"* for more information.

You need to pay attention to the objects that you serialize into the Body because not all serialized objects will be meaningful or useful at the receiver. A database connection object, for example, will be of little use when received at a client which does not have access to the database server. The use of Serialized Java objects in messages can also introduce dependencies that limit possible service implementations.

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

The default behavior of all ESB components (Actions, Listeners, Gateways, Routers, Notifiers etc) is to get and set data on the message using the message's *Default Payload Location*.

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

- get-payload-location: The location from which to retrieve 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`**.

3.2.5. Extensions to Body

As well as manipulating the contents of a Message Body directly in terms of bytes or name/value pairs, there are a number of interfaces available to simplify this by providing predefined message structures and methods to manipulate them.

These interfaces are extensions on 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.

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

For each of the various Body types you will find an associated create method, such as `createTextBody` that allows you to create and initialize a Message of the specific type. Once created the Message can be manipulated directly through the raw Body or by using its interface methods. The Body structure is maintained even after transmission so it can be manipulated by the message recipient using the methods of the interface that created it.

`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.

3.2.6. Attachments

Messages may contain attachments that do not appear in the main payload body such as images, drawings, binary document formats and zip files. The `Attachment` interface supports both named and unnamed attachments. In the current release of JBossESB only Java Serialized objects may be attachments. This restriction will be removed in a subsequent release.

Attachments may be used for a number of reasons. Generally they are used to provide a more logical structure for the message. The performance of large messages can also be improved by allowing the streaming of the attachments between endpoints.

The JBossESB does not support specifying other encoding mechanisms for the Message or attachment streaming. This feature will be added in a later release and where appropriate will be tied in to the SOAP-with-attachments delivery mechanism. Currently attachments are treated in the same way as named objects within the Body.

Warning

At this time it is recommended that developers do not use Properties or Attachments.

The general concepts they embody are currently being re-evaluated and may change significantly in future releases.

It is recommended that the data for your Properties and Attachments be included as part of the Message Body.

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

```
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();
}
```

3.2.7. Properties

Message properties define additional metadata for the message. JBossESB does not implement `Properties` using *java.util.Properties* as it would place restrictions on the types of clients and services that could used. Web Services stacks also do this for the same reason. If you need to send `java.util.Properties` then you can embed them within the current abstraction.

Warning

At this time it is recommended that developers do not use Properties or Attachments.

The general concepts they embody are currently being re-evaluated and may change significantly in future releases.

It is recommended that the data for your Properties and Attachments be included as part of the Message Body.

Example 3.11. The **`org.jboss.soa.esb.message.Properties`** interface

```
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();
}
```

3.2.8. 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 undertake 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:

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

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

 public static MessageFactory getInstance ();
}
```

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.

Additional Message implementations may be provided at runtime through the **`org.jboss.soa.esb.message.format.MessagePlugin`**.

Example 3.13. **`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 ();
}
```

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.

3.2.8.1. `MessageType.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-serialise 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 `IllegalParameterException` being thrown.

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

Important

You should be wary about using the JAVA_SERIALIZED version of the Message format because it can easily tie your applications to specific service implementations.

3.2.8.2. `MessageType.JBOSS_XML`

This uses an XML representation of the Message. The schema for the message is defined in **`message.xsd`** within the schemas directory.

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

If you add non Java Serializable objects to the Message you will have to provide a mechanism for marshalling those objects to and from XML. This can be done by creating a plugin 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 ();
}
```

Marshalling plug-ins must be registered with the system through the **`jbossesb-properties.xml`** configuration file. They must have attribute names that start with the **`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.

Building and Using Services

4.1. Listeners, Routers/Notifiers and Actions

4.1.1. Listeners

Listeners encapsulate the endpoints for ESB-aware message reception. Upon receipt of a message, a Listener feeds that message into a “pipeline” of message processors that process the message before routing the result to the “replyTo” endpoint. The action processing that takes place in the pipeline may consist of steps wherein the message gets transformed in one processor, some business logic is applied in the next processor, before the result gets routed to the next step in the pipeline, or to another end-point.

Many different parameters can be configured for listeners, such as the number of active worker threads. You can refer to *Section 9.1, “ Overview ”* for a complete description of these options.

4.1.2. Routers

Routers are actions that are used to send the ESB Message, or its payload, from the action pipeline to other end-points. The SOA Platform includes several routers which cover most usage scenarios. With the exception of **`StaticWireTap`**, all the included router actions terminate processing of the action pipeline even if there are additional actions remaining in the configuration.

You can find specific details on each router in *Section 11.5, “Routing”*.

Some routers implement the unwrap property. This allows a router to send messages to endpoints that are not ESB aware by sending only the payload of the message. If this property is set to **`true`** then the ESB Message payload is extracted and sent. Setting unwrap to **`false`** will send the complete ESB Message.

There is also a router, called ContentBasedRouter, that can be used for dynamic routing based on the message content. For a detailed explanation of content-based routing you should refer to the JBoss SOA Platform Services Guide [1].

4.1.3. Notifiers

Notifiers are the way in which success or error information may be propagated to ESB-unaware endpoints. You should not use Notifiers for communicating with ESB-aware endpoints. This may mean that you cannot have ESB-aware and ESB-unaware endpoints listening on the same channel. Consider using Couriers or the ServiceInvoker within your Actions if you want to communicate with ESB-aware endpoints.

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

[1] The JBoss Enterprise SOA Platform Services Guide is provided as the file **`Services_Guide.pdf`** or can be viewed online at *http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/*

Note

JMSNotifier was sending the type of JMS message (TextMessage or ObjectMessage) depending upon the type of ESB Message (XML or Serializable, respectively). This was wrong, as the type of ESB Message should not affect the way in which the Notifier sends responses. 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 backward compatibility with previous releases, the default value depends upon the ESB Message type: object for Serializable and text for XML. However, we encourage you not to rely on defaults.

As outlined above, the responsibility of a listener is 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 **`service-category`**, **`service-name`**, **`service-description`** and **`EPR-description`** tag names). If you set the optional remove-old-service tag name to true then the ESB will remove any existing service entry from the Registry prior to adding this new instance. However, this should be used with care, because the entire service will be removed, including all EPRs.
- instantiation of the listener class (see **`listenerClass`** tag name).
- the EPR that the listener will be servicing. This is transport specific. The following example corresponds to a JMS EPR (see connection-factory, destination-type, destination-name, jndi-type, jndi-URL and message-selector tag names).
- the "action processing pipeline". One or more <action> elements that each must contain at least the 'class' tag name that will determine which action class will be instantiated for that step in the processing chain.

Example 4.1. HelloWorld Quickstart service configuration

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

<providers>
  <jms-provider name="JBossMessaging"
    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>
```

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

1. action1. **`MyJMSListenerAction`** (a trivial example follows)

2. notificationAction. An org.jboss.soa.esb.actions.SystemPrintln

The following trivial action class will prove useful for debugging your 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().getContents());
  return message;
 }
}
```

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

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

Optional public callback methods that take a Message argument will be used for notification of the result of the specific step of the processing pipeline (see items 5 and 6 below).

The **`org.jboss.soa.esb.listeners.message.ActionProcessingPipeline`** class will perform the following steps for all steps configured using <action> elements

1. Instantiate an object of the class specified in the 'class' attribute with a constructor that takes a single argument of type **`ConfigTree`**

2. Analyze contents of the 'process' attribute.

   Contents can be a comma separated list of public method names of the instantiated class (step 1), each of which must take a single argument of type **`Message`**, and 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 a single processing method called `process`

   Using a list of method names in a single <action> element has some advantages compared to using successive <action> elements, as the action class is instantiated once, and methods will be invoked on the same instance of the class. This reduces overhead and allows for state information to be kept in the instance objects.

   This approach is useful for user supplied (new) action classes, but the other alternative (list of <action> elements) continues to be a way of reusing other existing action classes.

3. Sequentially invoke each method in the list using the Message returned by the previous step

4. If the value returned by any step is null the pipeline will stop processing immediately.

5. Callback method for success in each <action> element: If the list of methods in the 'process' attribute was executed successfully, the pipeline will analyze contents of the **`okMethod`** attribute. If none is 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 3. If the pipeline succeeds then the **`okMethod`** notification will be called on all handlers from the last one back to the initial one.

6. Callback method for failure in each <action> element: If an Exception occurs then the `exceptionMethod` notification will be called on all handlers from the current (failing) handler back to the initial handler. At present time, 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 either be whatever is returned from the `getFaultMessage` of the exception, or a default **`Fault`** containing the information within the original exception.

Action classes supplied by users to tailor behavior of the ESB to their specific needs, might need extra run time configuration (for example the Notifier class in the XML above needs the <NotificationList> child element). Each <action> element will utilize the attributes mentioned above and will ignore any other attributes and optional child elements. These will be however passed through to the action class constructor in the require ConfigTree argument. Each action class will be instantiated with it's corresponding <action> element and thus does not see (in fact must not see) sibling action elements.

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 which case it is incumbent on the Action developer to choose one

or more unique locations within the Message Body for its data and communicate this to the Service consumers.

Furthermore, because Actions may be chained together it is possible that an Action earlier in the chain modifies the original input Message, or replaces it entirely.

Note

From a security perspective, you should be careful about using unknown Actions within your Service chain. We recommend encrypting information.

If Actions share data within an input Message and each one modifies the information as it flows through the chain, by default we recommend retaining the original information so that Actions further down the chain still have access to it. Obviously there may be situations where this is either not possible or would be unwise. Within JBossESB, Actions that modify the input data can place this within the **`org.jboss.soa.esb.actions.post`** named Body location. This means that if there are N Actions in the chain, Action N can find the original data where it would normally look, or 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, you should use the default named Body location with care when chaining Actions in case chained Actions 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.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.mesage.time.dod`
: The time the message left the ESB, e.g., 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.

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 4.3.

4.3. What is a Service?

JBossESB does not impose restrictions on what constitutes a service. As we discussed earlier in this document, the ideal SOA infrastructure encourages a loosely coupled interaction pattern between clients and services, where the message is of critical importance and implementation specific details are hidden behind an abstract interface. This allows for the implementations to change without requiring clients/users to change. Only changes to the message definitions necessitate updates to the clients.

As such and as we have seen, JBossESB 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 operates on the Message received. Internally a service is structured from one or more Actions, that can be chained together to process incoming the incoming Message. What an Action does is implementation dependent, e.g., update a database table entry, or call an EJB.

When developing your services, you first need to determine the conceptual interface/contract that it exposes to users/consumers. This contract should be defined in terms of Messages, e.g., what the payload looks like, what type of response Message will be generated (if any) etc.

Note

Once defined, the contract information should be published within the registry. At present JBossESB does not have any automatic way of doing this.

Clients can then use the service as long as they do so according to the published contract. How your 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. There will be the usual trade-offs to make, e.g., manageability versus re-usability.

Note

In subsequent releases we will be improving tool support to facilitate the development of services.

4.3.1. ServiceInvoker

From a clients perspective, the Courier interface and its various implementations can be used to interact with services. However, this is still a relatively low-level approach, requiring developer code to contact the registry and deal with failures. Furthermore, since JBossESB 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.

In JBossESB 4.2, the ServiceInvoker was introduced to help simplify the development effort. The ServiceInvoker hides much of the lower level details and opaquely works with the stateless service

fail-over mechanisms. As such, ServiceInvoker is the recommended client-side interface for using services within JBossESB.

```
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;
}
```

An instance of **`ServiceInvoker`** can be created for each service with which the client requires interactions. Once created, the instance contacts the registry where appropriate to determine the primary EPR and, in the case of fail-overs, any alternative EPRs.

Once created, the client can determine how to send Messages to the service: synchronously (via deliverSync) or asynchronously (via deliverAsync). In the synchronous case, a timeout must be specified which represents how long the client will wait for a response. If no response is received within this period, a MessageDeliverException is thrown.

As mentioned earlier in this document, when sending a Message it is possible to specify values for **`To`**, **`ReplyTo`**, **`FaultTo`** etc. within the **`Message`** header. When using the **`ServiceInvoker`**, because it has already contacted the registry at construction time, the **`To`** field is unnecessary. 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 JBossESB it may be possible to use any supplied To field as an alternate delivery destination should the EPRs returned by the registry fail to resolve to an active service.

4.3.2. Services and ServiceInvoker

In a client-service environment the terms client and service are used to represent roles and a single entity can be a client and a service simultaneously. As such, you should not consider 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. Or an Action could decide to route specific types of fault Messages to the Dead Letter Queue for later administration.

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

4.3.3. InVM Transport

The InVM transport is a new feature in JBossESB 4.3 that provides communication between services running on the same JVM. This means that instances of **`ServiceInvoker`** can invoke a service from within the same JVM without any networking or message serialization overhead.

Earlier versions of the ESB did not support this transport and required every service to be configured with at least one Message Aware listener. This is not longer a requirement. Services can now be configured without any <listener> configuration and still be invokable from within their VM. This makes Service configuration much simpler.

```
<service category="ServiceCat" name="ServiceName"
    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>
```

Important

It is important to realize that InVM achieves its performance benefits by optimizing the internal data structures that are used to facilitate communication between services. There are a number of limitations that are discussed below that must be taken into account when deciding to make use of this transport.

The major limitation is that the queue used to store messages is not persistent. If the service is shutdown or otherwise fails before the queue is emptied, then those messages will be lost. Further limitations are mentioned throughout this section.

Note

The JBoss ESB allows services to be invoked by multiple different transports concurrently. Choosing the appropriate transports for different messages is important when configuring your services for maximum performance and reliability.

4.3.3.1. inVM Scope

The default InVM scope for an ESB deployment is specified by setting the value of the core:jboss.esb.invm.scope.default property in the **`jbossesb-properties.xml`** file. The default configured value supplied in the JBoss SOA Platform is **`NONE`**. If this property is undefined the default scope is actually **`GLOBAL`**.

The JBossESB currently supports 2 scopes.

`NONE`
: The Service is not invokable over the InVM transport.

`GLOBAL`
: The Service is invokable over the InVM transport from within the same classloader scope.

Note

A **`LOCAL`** scope is planned for a future release, which will restrict invocation to within the same deployed **`.esb`** archive.

You can specify the InVM scope of a service using the *`invmScope`* attribute of the <service> element of the service's configuration.

Example 4.2. Enabling **GLOBAL** inVM scope for a service

```
<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>
```

4.3.3.2. InVM Transacted

The InVM listener can execute within both transacted and non-transacted scopes in the same manner as the other transports that support transactions.

This behavior can be controlled through explicit or implicit configuration and follows two basic rules:

1. The ImVM listener will be implicitly transacted if there is another transacted transport configured on the service. At present these additional transports can be JMS, scheduled or SQL.

2. If supplied, the invmTransacted attribute on the service element takes precedence.

4.3.3.3. Transaction Semantics

The InVM transport in JBoss ESB is transactional, but the message queue is held only in volatile memory. This makes the InVM transport very fast but the message queue for this transport will be lost in the case of system failure or shutdown.

Because of the volatility aspect of the InVM queue you may not be able to achieve all of the ACID semantics, particularly when used with other transactional resources such as databases. But the performance benefits of InVM can outweigh this downside in the many cases. In situations where full ACID semantics are required, we recommend that you use one of the other transactional transports, such as JMS or database.

When using InVM within a transaction, the message will not appear on the receiver's queue until the transaction commits, although the sender will get an immediate acknowledgment that the message has been accepted to be later queued. If a receiver attempts to pull a message from the queue within the scope of a transaction, then the message will be automatically placed back on the queue if that transaction subsequently rolls back. If either a sender or receiver of a message needs to know the transaction outcome then they should either monitor the outcome of the transaction directly, or register a Synchronization with the transaction.

For performance reasons when a message is placed back on the queue by the transaction manager, it may not go back into the same location. If your application relies on specific ordering of messages then you should consider a different transport or group related messages into a single "wrapper" message.

4.3.3.4. Threading

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

Example 4.3. Threading

```
<service category="HelloWorld" name="Service2" description="Service 2"
```

```
        invmScope="GLOBAL">
          <property name="maxThreads" value="100" />
           <listeners>...
           <actions>...
```

4.3.3.5. Lock-step Delivery

The InVM Transport delivers messages with low overhead to an in-memory message queue. This is very fast and the message queue can become overwhelmed if delivery is happening too quickly for the Service consuming the messages. To mitigate these situations the InVM transport provides a *Lock-Step* delivery mechanism.

The Lock-Step delivery method attempts to ensure that messages are not delivered to a service faster than the service is able to retrieve them. It does this by blocking message delivery until the receiving Service picks up the message or a timeout period expires.

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 can be enabled for a service using its property settings:

Example 4.4. Enabling "Lock-Step" delivery

```
<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>
```

`inVMLockStep`
: A boolean value controlling whether LockStep delivery is enabled

`inVMLockStepTimeout`
: The maximum number of milliseconds that message delivery will be blocked while waiting for a message to be retrieved.

Lock-step delivery is disabled within the scope of a transaction. This is because the insertion of a message into a queue depends on the commit of the enclosing transaction, which may occur any before or after the expected lock-step wait period.

4.3.3.6. Load Balancing

`ServiceInvoker` provides load balancing of invocations in situations where there are multiple endpoints available for the target service. **`ServiceInvoker`** supports several different load balancing strategies as part of this feature.

`ServiceInvoker` gives preference to invoking a service over its InVM transport if one is available. **`ServiceInvoker`**'s other load balancing strategies are only applied in the absence of an InVM endpoint for the target Service.

4.3.3.7. Pass-by-Value/Pass-by-Reference

By default, the InVM transport passes Messages "by reference". This is done for performance reasons but can result in data integrity issues and class cast issues where messages are being exchanged across **`ClassLoader`** boundaries.

Message passing "by value" can be enabled on individual services if you encounter these issues. This is done by by setting the inVMPassByValue property on the service to **`true`**

Example 4.5. setting the inVMPassByValue property

```
<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>
```

4.4. Service Contract Definition

A contract definition can be specified on a service by the inclusion of XML schema definitions representing 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 for the main body section of the message and can enforce validation of that content.

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

Table 4.1. Service Contact Attributes

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

Message validation

The contents of the request and response messages can be automatically validated providing that the associated schema has been declared on the <actions> element. The validation can be enabled by specifying the 'validate' attribute on the <actions> element with a value of 'true'.

Validation is disabled by default.

Exposing an ESB service as a web service

Declaration of the contract schemas will automatically enable the exposure of the ESB service through a web service endpoint, the contract for which can be located through the contract web application. This functionality can modified by specifying the webservice attribute, the values for which are as follows.

webservice attribute

false
: No web service endpoint will be published

true
: A web service endpoint is published (default)

The following example illustrates the declaration of a service which wishes to validate the request/response messages but without exposing the service through a web service endpoint.

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

    </actions>
</service>
```

Other Components

In this chapter we shall look at other infrastructural components and services within JBossESB. Several of these services have their own documentation which you should also read: the aim of this chapter is to simply give an overview of what else is available to developers.

5.1. The Message Store

The message store mechanism in JBossESB is designed with audit tracking purposes in mind. As with other ESB services, it is a pluggable service, which allows for you, the developer to plug in your own persistence mechanism should you have special needs. The implementation supplied with JBossESB is a database persistence mechanism. If you require say, a file persistence mechanism, then it's just a matter of you writing your own service to do this, and override the default behavior with a configuration change.

One thing to point out with the Message Store – this is a base implementation. We will be working with the community and partners to drive the feature functionality set of the message store to support advanced audit and management requirements. This is meant to be a starting point.

5.2. Data Transformation

Often clients and services will communicate 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 from one data format to another.

In JBossESB this is the role the Transformation Service. This version of the ESB is shipped with an out-of-the-box Transformation Service based on Milyn Smooks. Smooks is a Transformation Implementation and Management framework. It allows you implement your transformation logic in XSLT, Java etc and provides a management framework through which you can centrally manage the transformation logic for your message-set.

There are a number of QuickStarts included that provide different examples of implementing transformations.

1. **`jboss-as/samples/quickstarts/transform_CSV2XML/`**

   This quickstart demonstrates how to transform a comma separated value (CSV) file to an xml. The transformation is done by configuring Smooks and performing two transformations, the first one from CSV to an intermediate xml format, and then the second from the intermediate xml format to the target xml.

2. **`jboss-as/samples/quickstarts/transform_XML2POJO/`**

   This quickstart illustrates the use of Smooks to perform a simple transformation to convert an XML file into Java POJOs.

3. **`jboss-as/samples/quickstarts/transform_XML2POJO2/`**

   This quickstart demonstrates the transform of two different XML files to a common set of POJOs.

4. **`jboss-as/samples/quickstarts/transform_XML2XML_simple/`**

This is a very basic example of how to manually define and apply a Message Transformation within JBossESB. It applies a very simple XSLT to a **`SampleOrder.xml`** message and prints the before and after XML to the console.

5. **`jboss-as/samples/quickstarts/transform_XML2XML_date_manipulation/`**

   This Quickstart continues on from the **`transformation_XML2XML_simple`** Quickstart and demonstrates how you can simplify your transformations by combining XSLT with Java. Java is used to perform the string manipulation on the SampleOrder date field (**`OrderDate.java`**) and XSLT is used for providing a template for output. The original and the transformed **`SampleOrder.xml`** messages are printed to the Java console.

6. **`jboss-as/samples/quickstarts/transform_XML2XML_stream/`**

   This is a very basic example of how to *stream* a fragment of a transformation to an ESB Service. The trick behind this is using a Smooks DOMVisitor that sends the element it is passed in its visitBefore and visitAfter.

7. **`jboss-as/samples/quickstarts/transform_EDI2XML_Groovy_XSLT/`**

   This is the most advanced of the transform Quickstarts. Be sure to go through the other transformation Quickstarts before going through this. There's an accompanying Flash demo at *http://labs.jboss.com/portal/jbossesb/resources/tutorials/xformation-demos/console-demo-03.html* which walks you through this Quickstart.

The complete Smooks documentation can be found on the Smooks project website at *http://milyn.codehaus.org/docs/v1.0/SmooksUserGuide_v1.0.html*.

5.3. Content-based Routing

Sometimes it is necessary for the ESB to dynamically route messages to their sources. For example, the original destination may no longer be available, the service may have moved, or the application simply wants to have more control over where messages go based on content, time-of-day etc. The Content-based Routing mechanism within JBossESB can be used to route Messages based on arbitrarily complex rules, which can be defined within XPath or Jboss Rules notation.

5.4. The Registry

In the context of SOA, a registry provides applications and businesses a central point to store information about their services. It 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. Ideally a registry should also facilitate the automated discovery and execution of e-commerce transactions and enabling a dynamic environment for business transactions. Therefore, a registry is more than an “e-business directory”. It is an inherent component of the SOA infrastructure.

In many ways, the Registry Service is at the heart of JBossESB: services can self-publish their endpoint references (EPRs) into the Registry when they are activated, and remove them when they are taken out of service. Consumers can consult the Registry to determine the EPR for the right service for the work at hand.

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.

Table 6.1. Operation Parameters

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

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.

From a JBossESB Action perspective, the service may be built using one or more Actions. For example, one Action may pre-process the incoming Message and transform the content 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 organization or by completely different organizations). It is important that each Action have a unique view of the Message data that it acts on. If this is not the case it is possible for chained Actions to overwrite or otherwise interfere with each other.

6.1.2. The Service

At this point we have enough information to construct the service. For simplicity, we shall assume that the business logic is encapsulated within the following pseudo-object:

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

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;
}
```

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 you can see, the process method is only the start. Now we must provide methods to decode the incoming Message payload (the Body):

```
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
}
```

What this method illustrates is how the information within the Body is extracted and then used to invoke a method on some business logic. In the case of reserveSeat, a response is expected by the client. This response Message is constructed using any information returned by the business logic as well as delivery information obtained from the original received Message. In this example, we need the To address for the response, which we take from the ReplyTo field of the incoming Message. We also need to relate the response with the original request and we accomplish this through the RelatesTo field of the response and the MessageID of the request.

All of the other operations supported by the service will be similarly coded.

6.1.4. The Client

As soon as we have the Message definitions supported by the service, we can construct the client code. The business logic used to support the service is never exposed directly by the service (that would break one of the important principles of SOA: encapsulation). This 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;
```

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. Hints and Tips

You may find the following useful when developing your 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

In this chapter we shall look at some more advanced concepts within JBossESB.

7.1. Fail-over and Load-balancing Support

It is important have redundancy in mind when 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. (It is assumed 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 we have discussed previously, within the **`jboss-esb.xml`** each service element consists of one or more listeners and one or more actions. Let's take a look at the **`JBossESBHelloworld`** example. The configuration fragment below is loosely based on the configuration of the **`JBossESBHelloworld`** example. When the service initializes it registers the category, name and description to 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 jms specific like queue name etc are not shown, but appeared at the top of the **`jboss-esb.xml`** where you can find the 'provider' section. In the jms-listener we can simply reference the "quickstartEsbChannel" in the busidref attribute.

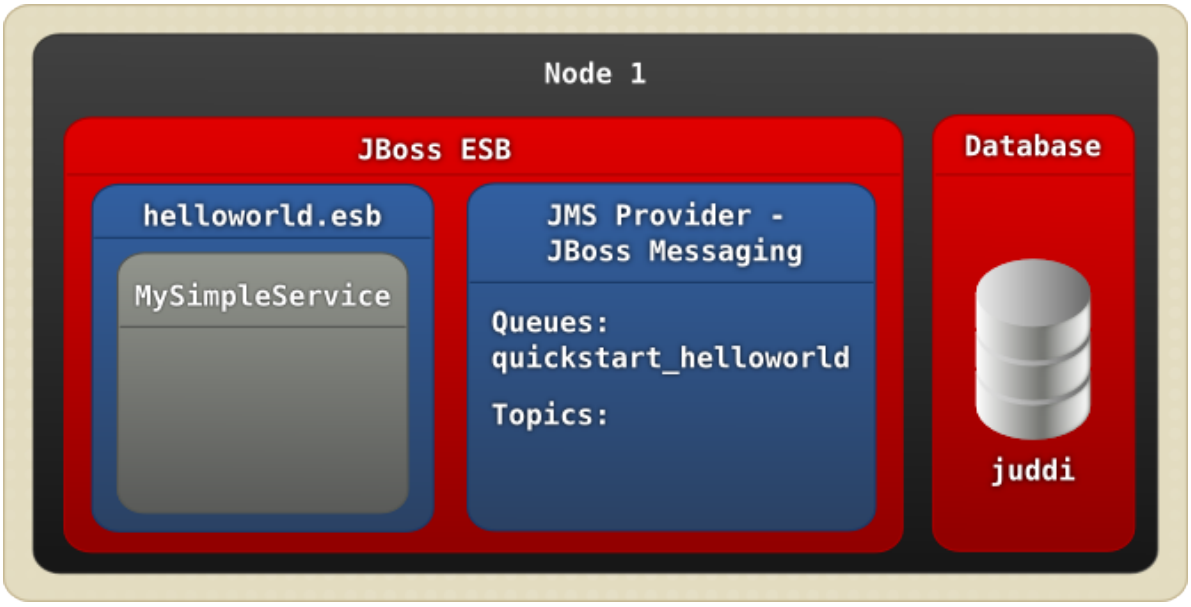


Figure 7.1. helloworld quickstart example, one service instance on one node.

Example 7.1. helloworld quickstart example, configuration fragment

```
...
```

```
<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>
...
```

Given the category and service name, another service can send a message to our Hello World Service by looking up the Service in the Registry. It will receive the JMSEPR and it can use that to send a message to. All this heavy lifting is done in the ServiceInvoker class. When our HelloWorld Service receives a message over the quickstartEsbChannel, it will hand this message to the process method of the first action in the ActionPipeline, which is the SystemPrintln action.

Note

Because **`ServiceInvoker`** hides much of the fail-over complexity from users, it necessarily only works with native ESB Messages. Additionally not all gateways have been modified to use the ServiceInvoker, so incoming ESB-unaware messages to those gateway implementations may not always be able to take advantage of service fail-over.

7.1.2. Replicated Services

In our example we have this service running on let's say Node1. What happens if we simply take the **`helloworld.esb`** and deploy it to Node2 as well (see figure 7-2)? Let's say we're using jUDDI for our Registry and we have configured all our nodes to access one central jUDDI database (it is recommended to use a clustered database for that). Node2 will find that the FirstServiceESB - SimpleListener Service is already registered! It will simply add a second ServiceBinding to this service. So now we have 2 ServiceBindings for this Service. We now have our first replicated Service! If Node1 goes down, Node2 will keep on working.

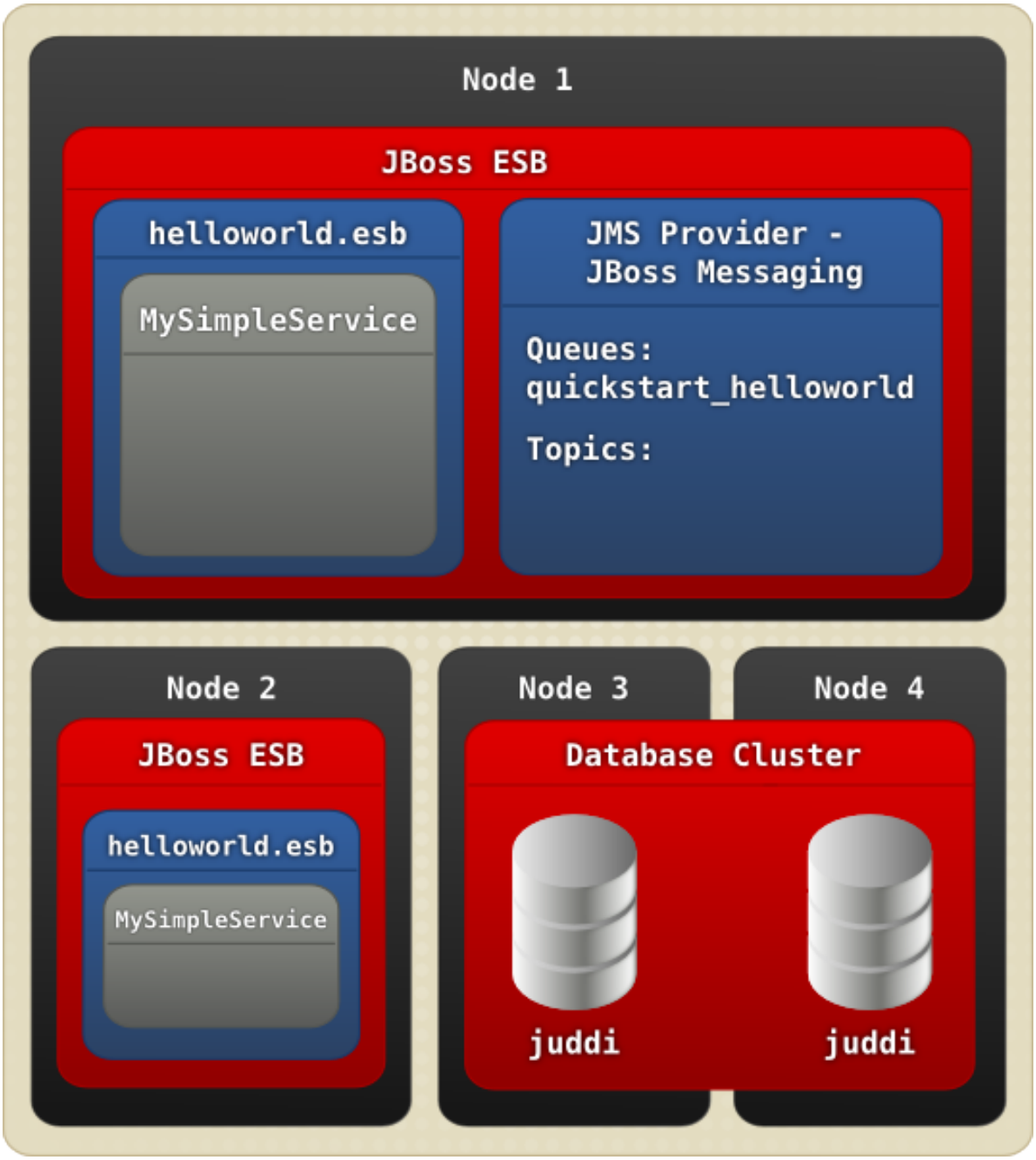


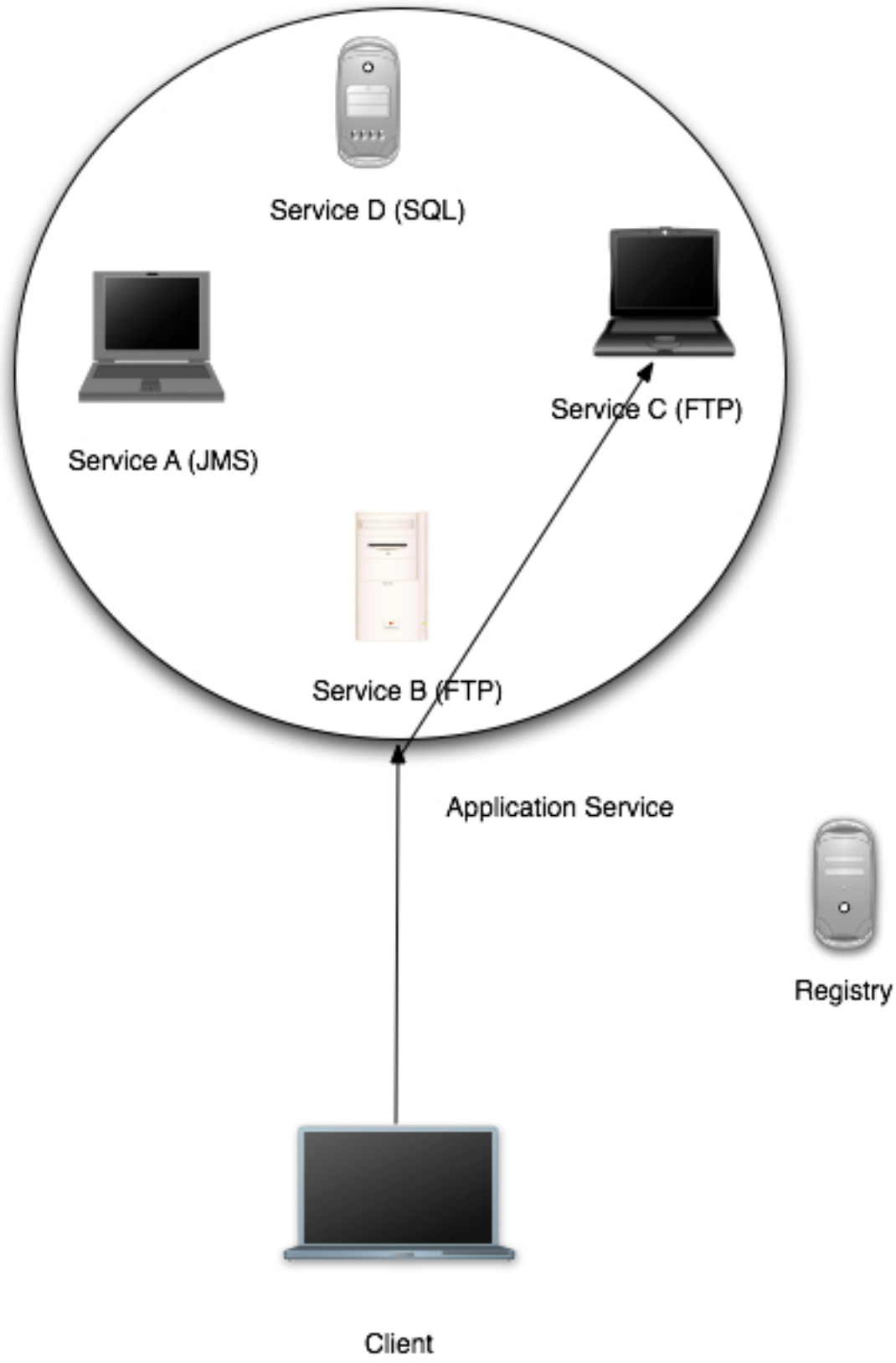
Figure 7.2. Two service instances each on a different node.

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

This type of replication can be used to increase the availability of a service or to provide load balancing. To further illustrate, consider the diagram below which has a logical service (Application Service) that is actually comprised of 4 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 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 Application Service is actually composed of 4 other services from the clients. It masks failures of the individual services and 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 service providers can replicate services independently of service consumers, in some circumstances the sender of a message will not want silent fail-over to occur. You need to set the message property org.jboss.soa.esb.exceptionOnDeliverFailure to **`true`** to prevent automatic silent fail-over. When you set this property a **`MessageDeliverException`** is thrown by the ServiceInvoker

instead of attempting to resend the message. This can be specified for all Messages by setting this property in the Core section of the JBossESB property file.

7.1.3. Protocol Clustering

Some JMS providers can be clustered. JBossMessaging is one of these providers, which is why we use this as our default JMS provider in JBossESB. When you cluster JMS you remove a single point of failure from your architecture, see Figure 7-3.

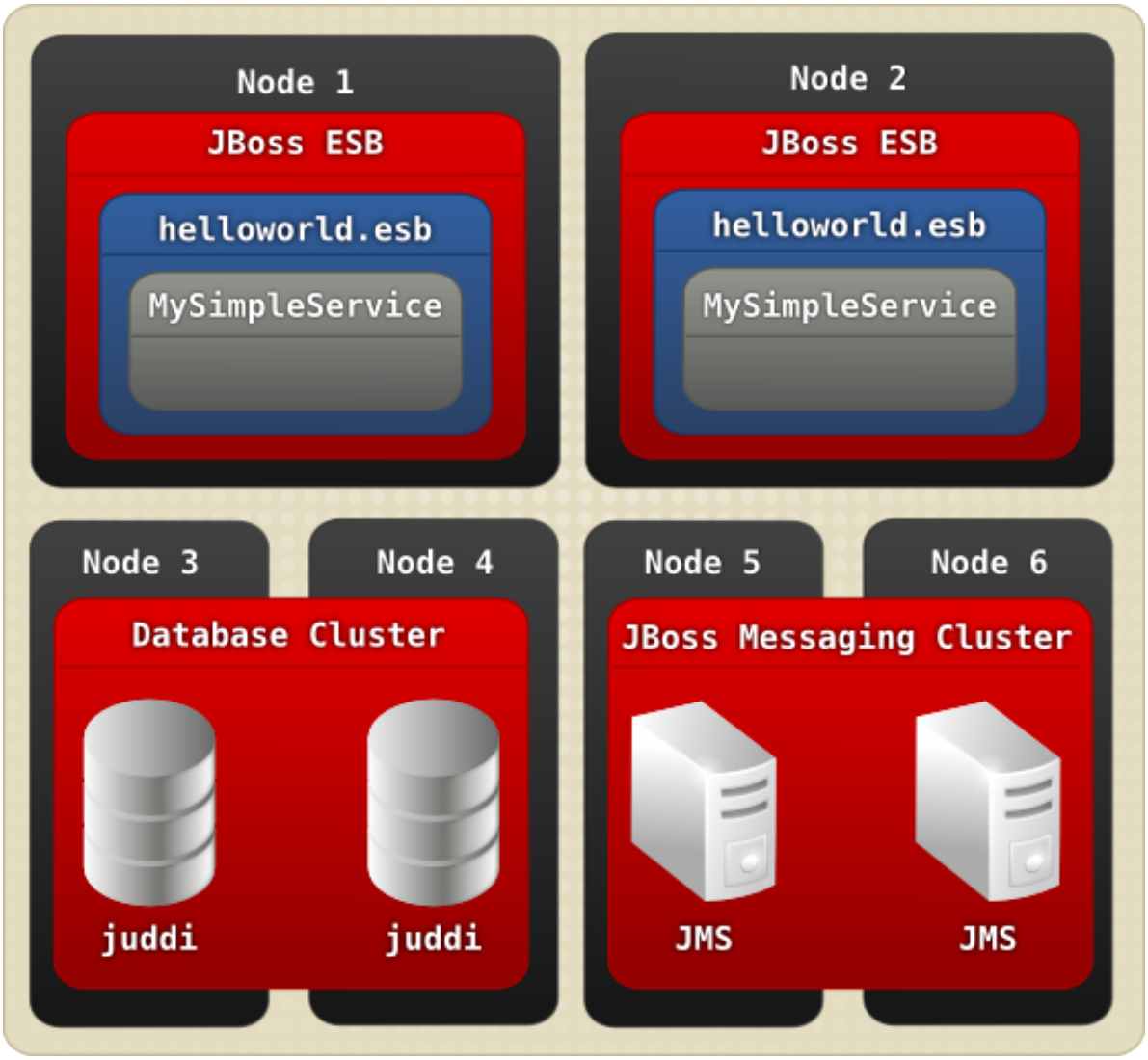
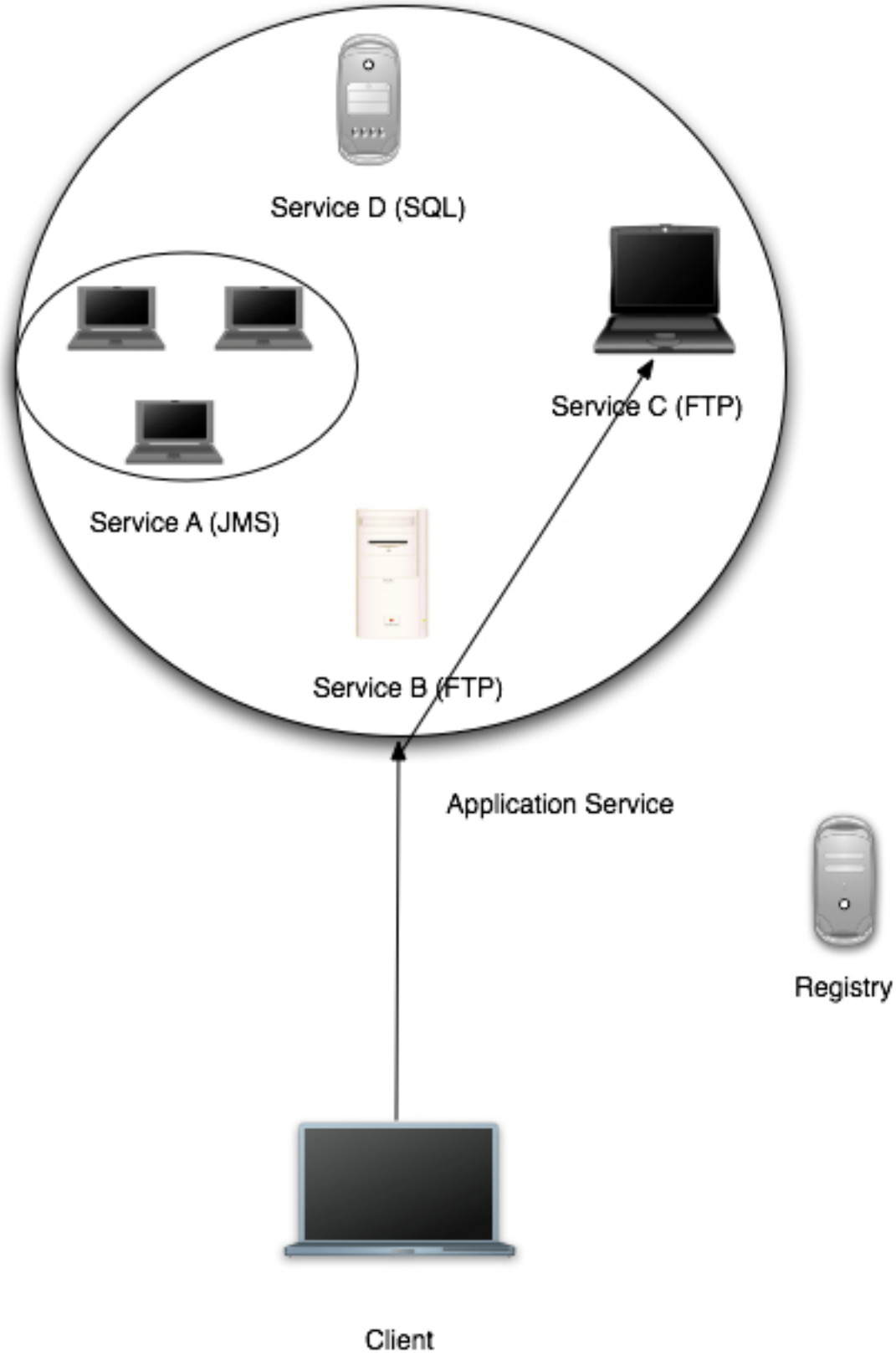


Figure 7.3. Protocol Clustering example using JMS

Please read the documentation on Clustering for JBossMessaging if you want to enable JMS clustering. Both JBossESB replication and JMS clustering can be used together, as illustrated in the following figure. In this example, Service A is identified in the registry by a single JMSEpr. However, opaquely to the client, that JMSEpr points to a clustered JMS queue, which has been separately configured to support 3 services. This is a federated approach to availability and load balancing. In fact masking the replication of services from users (the client in the case of the JBossESB replication approach, and JBossESB in the case of the JMS clustering) is in line with SOA principles: hiding these implementation details behind the service endpoint and not exposing them at the contract level.


Service D (SQL)
Service C (FTP)
Service A (JMS)
Service B (FTP)
Application Service
Registry
Client

Note

If using JMS clustering in this way you will obviously need to ensure that your configuration is correctly configured. For instance, if you place all of your ESB services within a JMS cluster then you cannot expect to benefit from ESB replication.

Other examples of Protocol Clustering would be a NAS for the FileSystem protocol, but what if your provider simply cannot provide any clustering? Well 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 which leads us to 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 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

Our **`HelloWorld`** Service can listen to more then 1 protocol. Here we have added an ftp channel.

```
...
<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 our Service is simultaneously listening to two JMS queues. Now these queues can be provided by JMS providers on different physical boxes! So we now have a made a redundant JMS connection between two services. We can even mix protocols in this setup, so we can also add and ftp-listener to the mix.

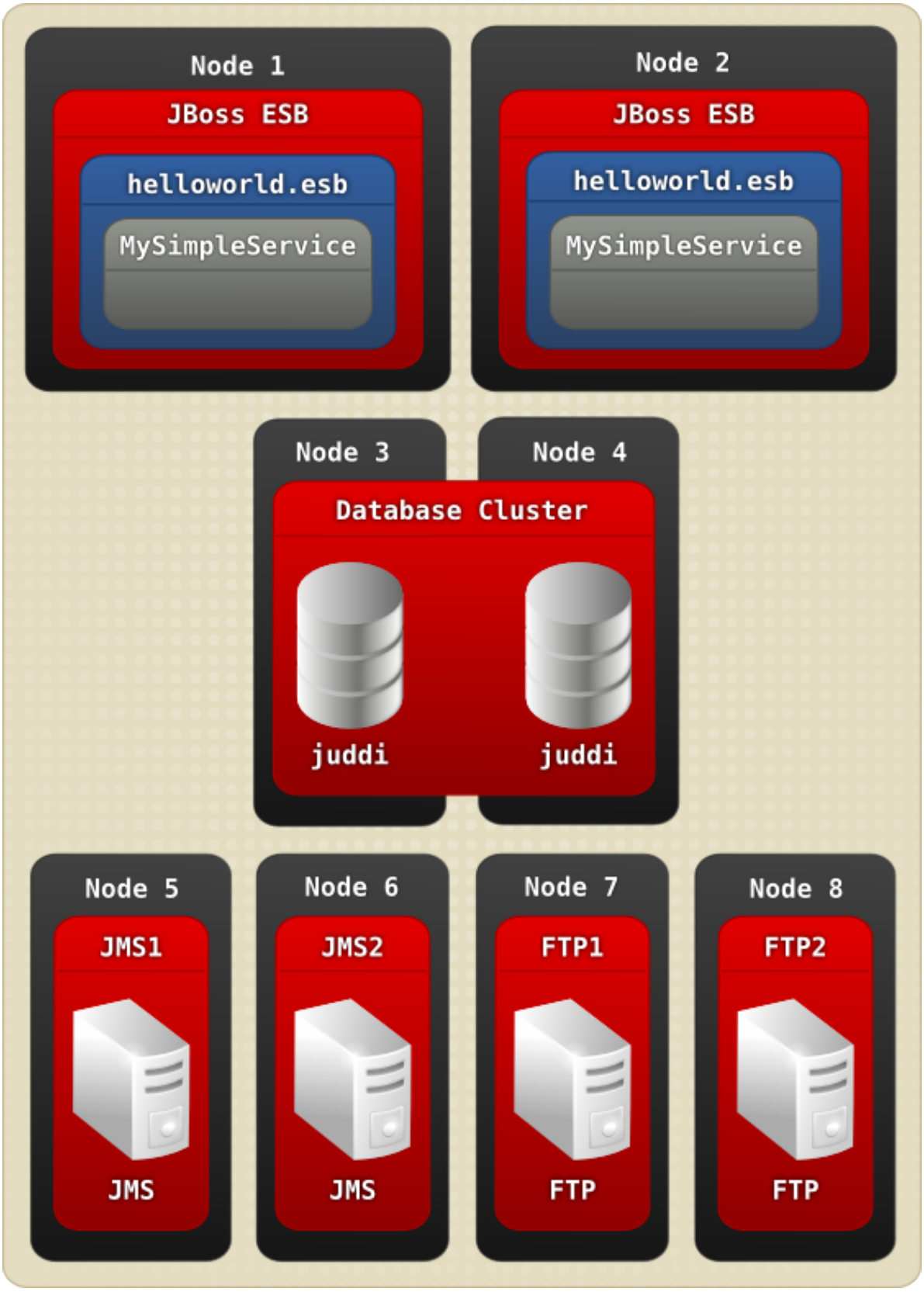


Figure 7.4. Adding two FTP servers to the mix.

```
...
<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 get a choice of 8 EPRs now (4 EPRs from Node1 and 4 EPRs from 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, or you can create your own.

- First Available. If a healthy ServiceBinding is found it will be used unless it dies, and 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. Typical Load Balance Policy where each EPR is hit in order of the list.
- Random Robin. Like the other Robin but then random.

The EPR list the Policy works with may get smaller over time as dead EPRs will be removed from the (cached) list. When the list is emptied or the time-to-live of the list 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, and is defaulted to 60,000 milliseconds. What if none of the EPRs work at the moment? This is where we may use Message Redelivery Service.

7.1.6. Message Redelivery

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

- If you are trying to deliver the message synchronously it will send the message to the DeadLetterService, which by default will store to the DLQ MessageStore, and it will send a failure back to the caller. 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 (recommended), it too will send the message to the DeadLetterService, but the message will get stored to the RDLVR MessageStore. The Redeliver Service (jbossesb.esb) will retry sending the message until the maximum number of redelivery attempts is exceeded. In that case the message will get stored to the DLQ MessageStore and processing will stop.

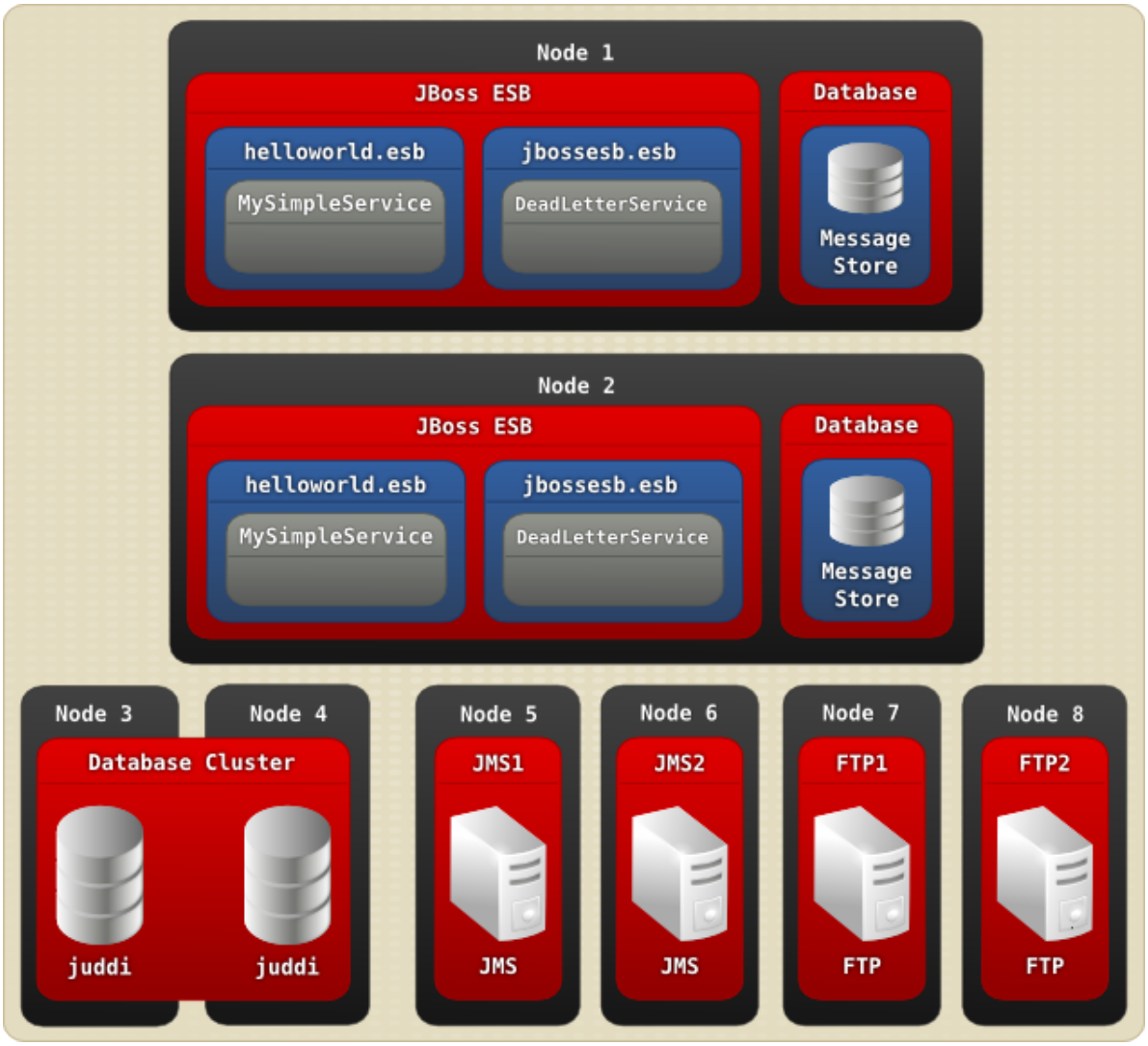


Figure 7.5. Message Re-delivery

Note

The **`DeadLetterService`** is turned on by default, however in the **`jbossesb-properties.xml`** you could set **`org.jboss.soa.esb.dls.redeliver`** to false to turn off its use.

7.2. Scheduling of Services

JBoss ESB 4.3 supports two types of providers.

1. Bus Providers, which supply messages to action processing pipelines via messaging protocols such as JMS and HTTP. This provider type is “triggered” by the underlying messaging provider.
2. Schedule Providers, which supply messages to action processing pipelines based on a schedule driven model i.e. where the underlying message delivery mechanism (e.g. 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 4.3 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 scheduling capability based on a 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's called on every schedule trigger. Se 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.jboss.org/repos/labs/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>
```

7.2.5. 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, or/and new ones can be added. You can do this by simply 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 5:

```
<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:

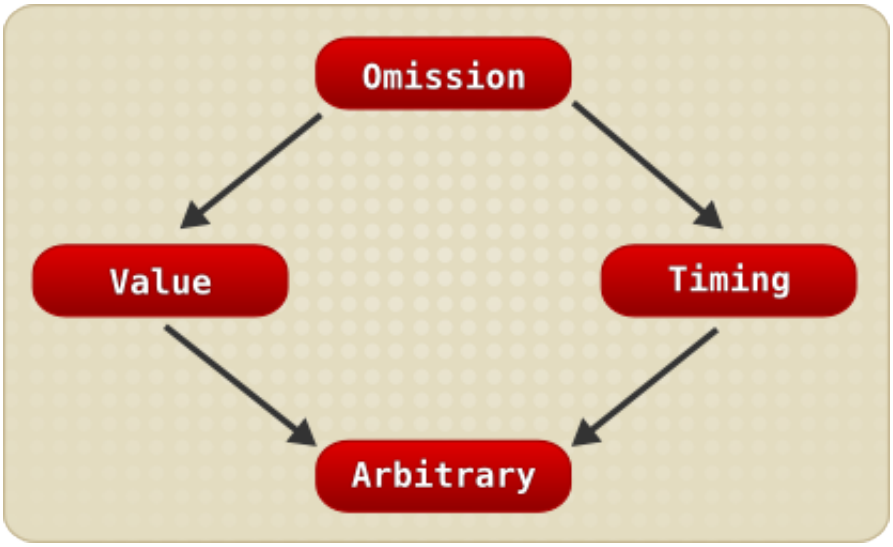


Figure 8.1. Fault classification hierarchy

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 whether or not a received Message is the one they are waiting for or a delayed Message, for example. Unfortunately Messages that are provided too soon by a service, e.g., asynchronous one-way responses to one-way requests, may be lost due to the underlying transport implementation. For instance, if 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 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 (e.g., by buffer overflow) it is not possible to distinguish it from one that is infinitely delayed. Therefore, if you construct your applications (consumers and services) to use a retry mechanism in the case of lost Messages, timing and omission failures should be tolerated, with the following exception: your consumer picks up an early response

out of order and incorrectly processes it (which then becomes 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).

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

Note

You should use care when retransmitting Messages to services. JBossESB currently has no notion of retained results or detecting retransmissions within the service, so any duplicate Messages will be delivered to the service automatically. This may mean that your service receives the same Message multiple times (e.g., it was the initial service response that got 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 to tolerate many of these failure models. Furthermore, in the case where forward progress is not possible because of a failure, using transactions the application can then roll back and the underlying transaction system will guarantee data consistency such that it will appear as though the work was never attempted. At present JBossESB offers transactional support through JBossTS when deployed within the JBoss Application Server.

8.1.2. Failure Detectors and Failure Suspectors

An ideal failure detector is one which can allow for the unambiguous determination of the liveliness of an entity, (where an entity may be a process, machine etc.,), within a distributed system. However, guaranteed detection of failures in a finite period of time is not possible because it is not possible to 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 (e.g., because of network congestion), incorrect failures may be assumed, potentially leading to inconsistencies when some machines “detect” the failure of another machine while others do not. Therefore, such timeouts are typically assigned given what can be assumed to be the worst case scenario within the distributed environment in which they are to be used, e.g., 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 either agree whether 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 being alive.

The advantage of the *failure suspector* is that all correctly functioning entities within the distributed environment will agree upon the state of another suspected failed entity. The disadvantage is that such failure suspecting protocols are heavy-weight, typically requiring several rounds of agreement. In addition, since suspected failure is still based upon timeout values, it is possible for non-failed entities 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 other applications the incorrect determination of the liveliness of an entity may lead to problems such as data corruption, or in the case of mission critical applications (e.g., aircraft control systems or nuclear reactor monitoring) could result in loss of life.

At present JBossESB does not support failure detectors or failure suspectors. We hope to address this shortcoming in future releases. For now you should develop your consumers and services using the techniques previously mentioned (e.g., MessageID and time-out/retry) to attempt to determine whether or not a given service has failed. In some situations it is better and more efficient for the application to detect and deal with suspected failures.

8.2. Reliability Guarantees

As we have seen, there are a range of ways in which failures can happen within a distributed system. In this section we will translate those into concrete examples of how failures could affect JBossESB and applications deployed on it. In the section on Recommendations we shall cover ways in which you can configure JBossESB to better tolerate these faults, or 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 it 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, ensure data consistency and make forward progress. Fault-tolerance techniques such as transactions or replication always comes at the cost of 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 when it makes sense to do so.

8.2.1. Message Loss

We have previously discussed how message loss or delay may adversely affect applications. We have also shown some examples of how messages could be lost within JBossESB. In this section we shall discuss message loss 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). Typically the semantics imposed on reliability are that the message will be delivered or the sender will be able to know with certainty that it did not get to the receiver, even in the presence of failures. It is frequently the case that systems employing reliable messaging implementations distinguish between a message being delivered to the recipient and it being processed by the recipient: for instance, simply getting the message to a service

does not mean much if a subsequent crash of the service occurs before it has time to work on the contents of the message.

Within JBossESB, the only transport you can use which gives the above mentioned 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 to the JMS queue for later re-processing. However, transacted sessions can be significantly slower than non-transacted sessions 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 (theoretically impossible to achieve). You should consider this aspect when developing your applications.

For these reasons, the Message fail-over and redelivery 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 failure suspicion is wrong, then it is possible that multiple services will get to 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, i.e., the execution of the same message multiple times is the same as executing it once.

For many services and applications this type of redelivery mechanism is fine. The robustness it provides over a single EPR can be a significant advantage. The failure modes where it does not work, i.e., 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 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 retransmissions and cope with multiple services performing the same work concurrently.

8.2.2. Suspecting Endpoint Failures

We saw 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 - 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 part of the network. This is sometimes called "split-brain syndrome".

8.2.3. Supported Crash Failure Modes

When using transactions or a reliable message delivery protocol such as JMS, JBossESB is able to recover from a catastrophic failure that shuts down the entire system.

Without these, JBossESB can only tolerate failures when the availability of the endpoints involved can be 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. All of the following JBossESB transports can be configured to either reliably deliver the Message or ensure it is not removed from the system: JMS, FTP, SQL. Unfortunately HTTP cannot be so configured.

8.2.6. ServiceInvoker

The ServiceInvoker will place undeliverable Messages to the Redelivery Queue if sent asynchronously. Synchronous Message delivery that fails will be indicated immediately to the sender. In order for the ServiceInvoker to function correctly the transport must indicate an unambiguous failure to deliver to the sender. 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 queued within the Redelivery Queue. For enterprise deployments a clustered JMS broker is recommended.

8.2.8. Action Pipelining

As with most distributed systems, we 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 but for an error or crash during processing within the Action pipeline to cause it to be lost. As mentioned previously, it is possible to configure some of the JBossESB transports to they do not delete received Messages when they are processed, so 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 4.3 configuration is based on the *jbossesb-1.0.1 XSD*[1]. This XSD is always the definitive reference for the Enterprise Service Bus configuration.

This model has two main sections:

<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.

<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.

By far the easiest way to create configurations based on this model, is to use an XSD-aware XML Editor such as the XML Editor in JBoss Developer Studio. This provides the author with auto-completion features when editing the configuration. Another tool is the **JBDS ESB Editor** (*http://www.redhat.com/docs/en-US/JBoss_Developer_Studio/3.0/html-single/ESB_Tools_Reference_Guide/index.html*)

9.2. Providers

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 i.e. for listeners that are "pushed" messages. Examples of this provider type would be the <jms-provider>.

Schedule Provider
: Provider configurations for schedule driven listeners i.e. 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> (e.g. for JMS - "connection-factory", "jndi-context-factory" etc). Likewise, each <bus> instance can be decorated with <property> instances specific to that <bus> instance (e.g. for JMS - "destination-type", "destination-name" etc).

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

```
<providers>
    <provider name="JBossMessaging" connection-factory="ConnectionFactory">
 <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">
```

[1] http://anonsvn.jboss.org/repos/labs/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd

```
    <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. More on this later.

9.3. Services

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.

Table 9.1. Service Attributes

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

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 i.e. <jms-listener>, <sql-listener> etc.

The base <listener> defines the following attributes. These attribute definitions are inherited by all <listener> extensions. They can be set for all of the listeners and fateways supported by JBoss ESB including the InVM transport.

Table 9.2. Listener Attributes

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 **<bus>** 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”. [1]	xsd:boolean	true

[1] A message bus defines the details of a specific message channel/transport.

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 listener (or “native”) listener must also be defined as gateway listeners do not define bidirectional endpoints, but rather “startpoints” into the ESB. From within the ESB you cannot send a message to a Gateway. 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).

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

    <providers>
        <provider name="JBossMQ">
            <property name="connection-factory" value="ConnectionFactory" />
            <property name="jndi-URL" value="jnp://localhost:1099" />
            <property name="protocol" value="jms" />

            <bus busid="local-jms">
                <property name="destination-type" value="topic" />
                <property name="destination-name" value="queue/B" />
                <property name="message-selector" value="service='Reconciliation'" />
            </bus>
        </provider>
    </providers>
    <services>
        <service category="Bank" name="Reconciliation"
                 description="Bank Reconciliation Service" is-gateway="false">

            <listeners>
                <listener name="Bank-Listener"
                          busidref="local-jms"
                          maxThreads="2">
                </listener>
            </listeners>

            <actions>
                ....
            </actions>

        </service>
    </services>
</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 its 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.

Table 9.3. Action Attributes

Name	Description	Type	Required
name	The name of the action. This attribute is required primarily for logging purposes.	xsd:string	true
class	The **`org.jboss.soa.esb.actions.ActionProcessor`** implementation class name.	xsd:string	true

Name	Description	Type	Required
process	The name of the “process” method that will be reflectively called for message processing. (Default is the “process” method as defined on the **`ActionProcessor`** class).	xsd:int	false

In a list of <action> instances within an <actions> set, the actions are called (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 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 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 etc). This allows us to have stronger validation on the configuration, as well as making configuration easier for those that use an XSD aware XML Editor (e.g. the JBDS 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 to use these specialized types instead of the “base” types when creating JBoss ESB configurations, the only alternative being where 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 (e.g. <jms-bus>), assigning a unique busid attribute value.
3. Define your <services> as normal, adding transport specific listener configurations (e.g. <jms-listener> that reference (using busidref) the new bus configurations you just made e.g. <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 type i.e. you can only reference a <jms-bus> from a <jms-listener>. A runtime error will result where cross references are made.

So 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, encompassing actions within a JTA transaction.

As you'll notice, all of the currently implemented transport specific types include an additional type not present in the “base” types, 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 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 *http://anonsvn.jboss.org/repos/labs/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd*, which is fully annotated with descriptions of each of the attributes. Using an XSD aware XML Editor such as the **JBDS** XML Editor makes working with these types far easier.

Table 9.4. JMS Message Filter Configuration

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. Will be used when creating a connection to the destination.	No
jms-security-credential	JMS destination password. Will be used when creating a connection to the destination.	No

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

Table 9.5. FTP Provider Configuration

Property	Description	Required
hostname	Can be a combination of **<host:port>** of just **<host>** which will use port 21.	Yes
username	Username that will be used for the ftp connection.	Yes
password	Password for the above user	Yes
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 **.esbInProcess**.	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 **.esbDone**.	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. Defaults to true.	No
error-directory	The ftp directory to which the file will be moved after when an error occurs during processing. Defaults to the value of directory above.	No
error-suffix	The file suffix which will be added to the file name after an error occurs during processing. Defaults to **.esbError**.	No
protocol	The protocol, can be one of: • sftp (SSH File Transfer Protocol) • ftps (FTP over SSL) • ftp (default).	No
passive	Indicates that the ftp connection is in passive. 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 ftpserver which port the ftpserver should connect to. The ftpserver 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

Property	Description	Required
certificate-name	The common name for a certificate for ftps server verification	No
certificate-passphrase	The pass-phrase of the private key for sftp client verification.	No

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 and only fetch those that have not previously been retrieved. The cache should be configured to use a cacheloader to persist the cache to stable storage.

Please note that there must exist a strategy for removing the file names from the cache. There might be an archiving process on the mainframe that moves the files to a different location on a regular basis. The removal of fil enames from the cache could be done by having a database procedure that removes all file names from the cache every couple of 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 configured by setting a property (removeFilesystemStrategy-cacheListener) in the ftp-listener configuration.

Table 9.6. Read-only FTP Listener Configuration

Name	Description
scheduleidref	Schedule used by the FTP listener. See Service Scheduling.
remoteFilesystemStrategy-class	Override the remote file system strategy with a class that implements: **`org.jboss.soa.esb.listeners.gateway.remotestrategies.RemoteFileSystemStrategy`**. Defaults to **`org.jboss.soa.esb.listeners.gateway.remotestrategies.ReadOnlyRemoteFileSystemStrategy`**
remoteFilesystemStrategy-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 **`/ftpfile-cache-config.xml`** which it expects to find in the root of the classpath
removeFilesystemStrategy-cacheListener	Specifies an JBoss **`TreeCacheListener`** 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

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.DeleteOnEvictTreeCacheListener"/
>

</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 helloworld_ftp_action quickstart demonstrates the read-only configuration. Run 'ant help' in the helloworld_ftp_action quickstart directory for instructions on running the quickstart. Please refer to the JBoss Cache documentation for more information about the configuration options available (http://labs.jboss.com/jbosscache/docs/index.html).

9.7. Transitioning from the Old Configuration Model

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

The old configuration model used a free form XML configuration 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 that were used to using the old model now need to keep the following in mind:

1. Read all of the docs on the new configuration model. Don't assume you can infer the new configurations based on your knowledge of the old.

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. When developing new Listener/Action components, you must ensure that the ConfigTree based configuration these components depend on can be mapped from the new XSD based configurations. An example of this is how in the ConfigTree configuration model, you could decide to supply the configuration to a listener component via attributes on the listener node, or you could decide to do it based on child nodes within the listener configuration. This type of free form configuration on <listener> components is not supported on the XSD to ConfigTree mapping i.e.

the child content in the above example would not be mapped from the XSD configuration 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 (on a <listener>), it would simply be ignored by the mapping code.

9.8. 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();
}
```

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;
}
```

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 may be overridden using the org.jboss.soa.esb.paramsRepository.class property.

Note

Red Hat recommends that you construct your ESB configuration file using **JBDS** or some other XML editor such as the **JBDS ESB Editor** (*http://www.redhat.com/docs/en-US/JBoss_Developer_Studio/3.0/html-single/ESB_Tools_Reference_Guide/index.html*). The JBossESB configuration information is supported by an annotated XSD which should help if using a 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. Transformers and Converters

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

Unless state otherwise, all of these actions use **`MessagePayloadProxy`** for accessing the message payload.

11.1.1. ByteArrayToString

Input Type	byte[ ]
Class	**`org.jboss.soa.esb.actions.converters.ByteArrayToString`**

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

Table 11.1. ByteArrayToString Properties

Property	Description	Required
encoding	The binary data encoding on the message byte array. Defaults to **`UTF-8`** when not specified.	No

Example 11.1. ByteArrayToString

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

11.1.2. ObjectInvoke

Input Type	User Object
Output Type	User Object
Class	**`org.jboss.soa.esb.actions.converters.ObjectInvoke`**

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

Table 11.2. ObjectInvoke Properties

Property	Description	Required
class-processor	The runtime class name of the processor class used to process the message payload.	Yes
class-method	The name of the method on the processor class used to process the method. The default value is the name of the action.	No

Example 11.2. ObjectInvoke

```
<action name="invoke"
```

```
  class="org.jboss.soa.esb.actions.converters.ObjectInvoke">
    <property name="class-processor"
        value="org.jboss.MyXXXProcessor"/>
    <property name="class-method" value="processXXX" />
</action>
```

11.1.3. ObjectToCSVString

Input Type	User Object
Output Type	**`java.lang.String`**
Class	**`org.jboss.soa.esb.actions.converters.ObjectToCSVString`**

Takes the Object bound as the message payload and converts it into a Comma Separated Value (CSV) string based on the supplied message object and a comma-separated bean-properties list property.

Table 11.3. ObjectToCSVString Properties

Property	Description	Required
bean-properties	List of Object beanproperty names used to get CSV values for the output CSV String. The Object should support a getter method for eachof listed properties.	Yes
fail-on-missing-property	Flag indicating if the action should fail if the Object have support a getter method for the property. The default value is **`false`**.	No

Example 11.3. ObjectToCSVString

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.ObjectToCSVString">
    <property name="bean-properties" value="name,address,phoneNumber"/>
    <property name="fail-on-missing-property" value="true" />
</action>
```

11.1.4. ObjectToXStream

Input Type	User Object
Output Type	**`java.lang.String`**
Class	**`org.jboss.soa.esb.actions.converters.ObjectToXStream`**

Takes the Object bound as the Message payload and concerts it into XML using the XStream processor.

Table 11.4. ObjectToXStream Properties

Property	Description	Required
class-alias	Class alias used in call to `XStream.alias(String, Class)` prior to serialization. The default is the input Object's class name.	No
exclude-package	Boolean flag that indicates where the package name should be excluded from the generated XML. The default is **`true`**. This property does not apply if a class-alias is specified.	No

Property	Description	Required
aliases	Specifies additional aliases to help XStream to covert the xml elements to objects.	No
namespaces	Specifies 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.	No
xstream-mode	Specifies the XStream mode to use.Possible values are **`XPATH_RELATIVE_REFERENCES`**, **`XPATH_ABSOLUTE_REFERENCES>`**, **`ID_REFERENCES`**, **`NO_REFERENCES`**.	No

Example 11.4. ObjectToXStream

```
<action name="transform"
    class="org.jboss.soa.esb.actions.converters.ObjectToXStream">
  <property name="class-alias" value="MyAlias" />
  <property name="exclude-package" value="true" />
</action>
```

Example 11.5. ObjectToXStream using aliases and namespaces

```
<action name="transform"
    class="org.jboss.soa.esb.actions.converters.ObjectToXStream">
  <property name="class-alias" value="MyAlias" />
  <property name="exclude-package" value="true" />
  <property name="aliases">
    <alias name="alias1" value="com.acme.MyXXXClass1/">
    <alias name="alias2" value="com.acme.MyXXXClass2/">
    <alias name="xyz" value="com.acme.XyzValueObject"/>
    <alias name="x" value="com.acme.XValueObject"/>
     ...
  </property>
  <property name="namespaces">
    <namespace namespace-uri="http://www.xyz.com" local-part="xyz"/>
    <namespace namespace-uri="http://www.xyz.com/x" local-part="x"/>
     ...
  </property>
</action>
```

11.1.5. XStreamToObject

Input Type	**`java.lang.String`**
Output Type	User Object
Class	**`org.jboss.soa.esb.actions.converters.XStreamToObject`**

Takes the XML bound as the message payload and converts it into an Object using the XStream processor.

Table 11.5. XStreamToObject Properties

Property	Description	Required
class-alias	Class alias used during serialization. Defaults to the input object's class name.	No

Property	Description	Required
exclude-package	Boolean flag indicating whether or not the XML includes a package name.	Required
incoming-type	The object type that the XML represents, and the type of object that will be returned.	Yes
root-node	An XPath expression specifying a different root node then the actual root node in the XML.	No
aliases	Additional aliases to help XStream to convert the xml elements to Objects.	No
attribute-aliases	Additional attribute aliases to help XStream to convert the XML attributes to Objects.	No
converters	Used to specify converters to help XStream to convert the XML elements and attributes to Objects. For additional information about converters refer to *http://xstream.codehaus.org/converters.html*.	No

```
<action name="transform" class="org.jboss.soa.esb.actions.converters.XStreamToObject">
    <property name="class-alias" value="MyAlias" />
    <property name="exclude-package" value="true" />
    <property name="incoming-type" value="com.acme.MyXXXClass" />
    <property name="root-node" value="/rootNode/MyAlias" />
    <property name="aliases">
        <alias name="alias1" class="com.acme.MyXXXClass1/>
        <alias name="alias2" class="com.acme.MyXXXClass2/>
        ...
    </property>
    <property name="attribute-aliases">
        <attribute-alias name="alias1" class="com.acme.MyXXXClass1"/>
        <attribute-alias name="alias2" class="com.acme.MyXXXClass2"/>
        ...
    </property>
    <property name="fieldAliases">
        <field-alias alias="aliasName" definedIn="className" fieldName="fieldName"/>
        <field-alias alias="aliasName" definedIn="className" fieldName="fieldName"/>
        ...
    </property>
    <property name="implicit-collections">
        <implicit-colletion class="className" fieldName="fieldName" fieldType="fieldType"
            itemType="itemType"/>
        ...
    </property>
    <property name="converters">
        <converter class="className" fieldName="fieldName" fieldType="fieldType"/>
        ...
    </property>
</action>
```

11.1.6. SmooksTransformer

Class	**`org.jboss.soa.esb.actions.converters.SmooksTransformer`**

Message Transformation on the JBoss ESB 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 data formats (XML, Java, CSV, EDI etc.) are supported by the SmooksTransformer component for both input and output. A wide range of Transformation Technologies are also supported, all within a single framework.

Important

The **`SmooksTransformer`** will be deprecated in a future release. You should refer to **`SmooksAction`** for a more general purpose and flexible Smooks action class.

Table 11.6. SmooksTransformer Properties

Property	Description	Required
resource-config	The Smooks resource configuration file.	Yes
from	Message Exchange Participant name. Message Producer.	No
from-type	Message type/format produced by the from message exchange participant.	No
to	Message Exchange Participant name. Message Consumer.	No
to-type	Message type/format consumed by the to message exchange participant.	No

All the above properties can be overridden by supplying them as properties to the message (**`Message.Properties`**).

Example 11.6. Default Input/Output

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

Example 11.7. Named Input/Output

```
<action name="transform"
  class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
  <property name="resource-config" value="/smooks/config-01.xml" />
  <property name="get-payload-location" value="get-order-params" />
  <property name="set-payload-location" value="get-order-response" />
</action>
```

Example 11.8. 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 (*http://milyn.codehaus.org/javadoc/v1.0/smooks-cartridges/javabean/org/milyn/javabean/BeanPopulator.html*). Additional

information can be found in the JBoss SOA Services Guide [1] , or the *http://wiki.jboss.org/wiki/Wiki.jsp?page=MessageTransformation*.

11.1.7. 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. The SmooksTransformer action will be deprecated (and eventually removed) in a future release of the ESB in favor of SmooksAction.

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

The 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>
```

Table 11.7. SmooksAction Optional Configuration Properties

Property	Description	Default
get-payload-location	Message Body location containing the message payload.	Default Payload Location
set-payload-location	Message Body location where result payload is to be placed.	Default Payload Location
excludeNonSerializables	Exclude non Serializable Objects when mapping the contents of the Smooks *ExecutionContext*[2] back onto the ESB Message.	true
resultType	The type of Result to be set as the result Message payload.	STRING
javaResultBeanId	Note: Only relevant when resultType=JAVA The Smooks bean context beanId to be mapped as the result when the resultType is "JAVA". If not specified, the whole bean context bean Map is mapped as the JAVA result.	
reportPath	The path and file name for generating a *Smooks Execution Report*[3]. 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”

[1] The JBoss Enterprise SOA Platform Services Guide is provided as the file **`Services_Guide.pdf`** or can be viewed online at *http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/*

and “set-payload-location” action properties, the SmooksAction gets and sets the Message payload on the default message location (i.e. using Message.getBody().get() and Message.getBody().set(Object)).

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

XML, EDI, CSV etc Input Payloads

To process these message types using the SmooksAction, simply supply the Source message as a String, *InputStream*[4], *Reader*[5], or byte array.

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

Java Input Payload

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

Specifying the Result Type

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

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

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

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

11.1.8. PersistAction

Input Type	Message
Output Type	The input Message
Class	**`org.jboss.soa.esb.actions.MessagePersister`**

This is used to interact with the **`MessageStore`**.

[4] http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html
[5] http://java.sun.com/j2se/1.5.0/docs/api/java/io/Reader.html

Table 11.8. PersistAction Properties

Property	Description	Required
classification	Used to classify where the Message will be stored. If the Message Property **`org.jboss.soa.esb.messagestore.classification`** is defined on the **`Message`** then that will be used instead. Otherwise a default may be provided at instantiation time.	Yes
message-store-class	The implementation of the MessageStore.	Yes
terminal	If this is set to **`true`** then this action will terminate the processing pipeline and the input message will be returned from processing. Default value is **`false`**.	No

Example 11.9. PersistAction

```
<action name="PersistAction"
  class="org.jboss.soa.esb.actions.MessagePersister" >
  <property name="classification" value="test"/>
  <property name="message-store-class" value=
  "org.jboss.internal.soa.esb.persistence.format.db.DBMessageStoreImpl"/>
</action>
```

11.2. Business Process Management

11.2.1. jBPM - BpmProcessor

Input Type	**`org.jboss.soa.esb.message.Message`** generated by **`AbstractCommandVehicle.toCommandMessage()`**
Output Type	Message – same as the input message
Class	**`org.jboss.soa.esb.services.jbpm.actions.BpmProcessor`**

JBoss ESB Services can invoke jBPM commands using the **`BpmProcessor`** action. The **`BpmProcessor`** action uses the jBPM command API to access jBPM.

You should refer to the JBoss Enterprise SOA Platform Services Guide [6] for additional information regarding the jBPM integration including how to access ESB services from jBPM.

The following jBPM commands have been implemented:

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

Table 11.9. BpmProcessor Properties

Property	Description	Required
command	The jBPM command being invoked. *Required* Allowable values: • **`NewProcessInstanceCommand`**	Yes

[6] The JBoss Enterprise SOA Platform Services Guide is provided as the file **`Services_Guide.pdf`** or can be viewed online at *http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/*

Property	Description	Required
	• **StartProcessInstanceCommand** • **CancelProcessInstanceCommand**	
processdefinition	This is required property for the **NewProcessInstanceCommand** and **StartProcessInstanceCommand** commands if the process-definition-id property is not supplied. The value of this property must reference a process definition that is already deployed to jBPM that you want to create a new instance of. This property does not apply to the **CancelProcessInstanceCommand**.	Depends
process-definition-id	Only required for the **NewProcessInstanceCommand** and **StartProcessInstanceCommand** commands if the processdefinition property is not supplied. The value of this property must reference a process defintion id in jBPM that you want to create a new instance of. This property does not apply to the **CancelProcessInstanceCommand**.	Depends
actor	Specifies the jBPM actor id. Only applies to the **NewProcessInstanceCommand** and **StartProcessInstanceCommand** commands only.	No
key	Specify the value of the jBPM 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 and process definition must be unique if a business key is supplied. For example, a unique invoice id could be used as the value for this key. 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 **NewProcessInstanceCommand** and **StartProcessInstanceCommand** commands only.	No
transition-name	This property only applies to the **StartProcessInstanceCommand** and specifies the transition to use out of the node if more than one is defined. If this property is not specified the default transition out of the node will be taken. The default transition is the first transition in the list of transition defined for that node in the jBPM **processdefinition.xml**.	No
esbToBpmVars	This property defines a list of variables that need to be extracted from the **EsbMessage** and set into jBPM context for the particular process instance. The list consists of mapping elements. Each mapping element can have the following attributes: • **esb**: required attribute which can contain an MVEL expression to extract a value anywhere from the **EsbMessage**. • **bpm**: optional attribute containing the name which be used on the jBPM side. If omitted the esb name is used.	No

Property	Description	Required
	• **`default`**: optional attribute which can hold a default value if the esb MVEL expression does not find a value set in the **`EsbMessage`**. • **`reply-to-originator`**: optional property for **`NewProcessInstanceCommand`** and **`StartProcessInstanceCommands`**. If this property is specified, with a value of **`true`**, then the creation of the process instance will store the **`ReplyTo`** and **`FaultTo`** EPRs of the invoking message within the process instance. These values can then be used within subsequent **`EsbNotifier`** and **`EsbActionHandler`** invocations to deliver a message to the **`ReplyTo`** and **`FaultTo`** addresses. This is only used by **`NewProcessInstanceCommand`** and **`StartProcessInstanceCommand`**.	

Example 11.10. jBPM - BpmProcessor

```
<action name="create_new_process_instance"
  class="org.jboss.soa.esb.services.jbpm.actions.BpmProcessor">
    <property name="command" value="StartProcessInstanceCommand" />
    <property name="process-definition-name" value="processDefinition2"/>
    <property name="actor" value="FrankSinatra"/>
    <property name="esbToBpmVars">
        <!-- esb-name maps to getBody().get("eVar1") -->
        <mapping esb="eVar1" bpm="counter" default="45" />
        <mapping esb="BODY_CONTENT" bpm="theBody" />
    </property>
</action>
```

11.3. Scripting

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

11.3.1. GroovyActionProcessor

Class	**`org.jboss.soa.esb.actions.scripting.GroovyActionProcessor`**

This action executes a Groovy action processing script, receiving the message and action configuration as input. You can find additional information about Groovy at *http://groovy.codehaus.org/*.

Table 11.10. GroovyActionProcessor Properties

Property	Description	Required
script	Path within classpath for the script.	
supportMessageBasedScripting	Allow scripts within the message.	
cacheScript	Should the script be cached. Defaults to **`true`**.	No

Table 11.11. GroovyAction Processor Script Binding Variables

Variable	Description
message	The Message

Variable	Description
payloadProxy	Utility for message payload (MessagePayloadProxy)
config	The action configuration (ConfigTree)
logger	The GroovyActionProcessor's static Log4J logger (Logger)

Example 11.11. GroovyActionProcessor

```
<action name="process"
   class="org.jboss.soa.esb.scripting.GroovyActionProcessor">
   <property name="script" value="/scripts/myscript.groovy"/>
</action>
```

11.3.2. ScriptingAction

Class	**org.jboss.soa.esb.actions.scripting.ScriptingAction**

Executes a script using the Bean Scripting Framework (BSF), receiving the message, payloadProxy, action configuration and logger as variable input. You can find additional information about at *http://jakarta.apache.org/bsf/*.

JBoss ESB 4.4 includes BSF 2.3.0, which does not support Groovy or Rhino. A future version will contain BSF 2.4.0, which will support those languages.

BSF does not provide an API to precompile, cache and reuse scripts. Each execution of the ScriptingAction will compile the scripts again. You will need to take this into account when considering the performance requirements of your application.

You should use the filename extention of **`.beanshell`** for you BeanShell scripts instead of **`.bsh`** to avoid the JBoss BSHDeployer detecting them and attempting to deploy.

Table 11.12. ScriptingAction Properties

Property	Description	Required
script	Path within classpath for the script.	
supportMessageBasedScripting	Allow scripts within the message.	
language	This value specifies the language that the script is written in. If not supplied then it will be deduced based on the filename extension.	No

Table 11.13. ScriptingAction Processor Script Binding Variables

Variable	Description
message	The Message
payloadProxy	Utility for message payload (MessagePayloadProxy)
config	The action configuration (ConfigTree)
logger	The ScriptingAction's static Log4J logger (Logger)

Example 11.12. ScriptingAction

```
<action name="process"
   class="org.jboss.soa.esb.scripting.ScriptingAction">
   <property name="script" value="/scripts/myscript.beanshell"/>
```

```
</action>
```

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	**`org.jboss.soa.esb.actions.EJBProcessor`**

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

Table 11.14. EJBProcessor Properties

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 ejb3 is **`true`**.	
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 **`DEFAULT_EJB_OUT`**.	No

Example 11.13. Sample Configuration for EJB 2.x

```
<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 11.14. Sample Configuration for EJB 3.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>
```

11.5. Routing

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

11.5.1. Routing Actions and the Action Pipeline

Table 11.15. Impact of Routing Actions Upon the Action Pipeline

Routing Action	Does It Terminate the Pipeline?
org.jboss.soa.esb.actions.Aggregator	No
org.jboss.soa.esb.actions.EchoRouter	No
org.jboss.soa.esb.actions.routing.http.HttpRouter	No
org.jboss.soa.esb.actions.routing.JMSRouter	No
org.jboss.soa.esb.actions.routing.email.EmailRouter	No
org.jboss.soa.esb.actions.StaticWiretap	No
org.jboss.soa.esb.actions.routing.email.EmailWiretap	No
org.jboss.soa.esb.actions.ContentBasedRouter	Yes
org.jboss.soa.esb.actions.StaticRouter	Yes

11.5.2. Aggregator

Class	**`org.jboss.soa.esb.actions.Aggregator`**

Message aggregation action. An implementation of the Aggregator Enterprise Integration Pattern. You can find more information on this development pattern at *http://www.enterpriseintegrationpatterns.com/Aggregator.html*.

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 also *Section 11.5.6, "ContentBasedRouter"* and *Section 11.5.7, "StaticRouter"*.

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.

Table 11.16. Aggregator Properties

Property	Description	Required
timeoutInMillis	Timeout time in milliseconds before the aggregation process times out.	No

Example 11.15. Aggregator

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

11.5.3. EchoRouter

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

11.5.4. HttpRouter

Currently there are two HttpRouter actions in the code base. One that uses JBoss Remoting to perform the HTTP invocation and one that uses Apache Commons HttpClient. This section will describe both.

11.5.4.1. JBoss Remoting HttpRouter

Class	**`org.jboss.soa.esb.actions.routing.HttpRouter`**

Important

JBoss Remoting HttpRouter is now deprecated to avoid confusion.

This action forwards the incoming message to a URL for further processing.

Table 11.17. JBoss Remoting HttpRouter Properties

Property	Description	Required
routeUrl	The endpoint that the message will be forwarded to. The default value is **`localhost:5400`**.	No

11.5.4.2. Apache Commons HttpRouter

Class	**`org.jboss.soa.esb.actions.routing.http.HttpRouter`**

This action allows invocation of external, ESB unaware, Http endpoints from an ESB action pipeline. This action is implemented using the Apache Commons HttpClient.

Table 11.18. Apache Commons HttpRouter

Property	Description	Required
endpointUrl	The endpoint that the message will be forwarded to.	Yes

Property	Description	Required
http-client-property	The HttpRouter uses the HttpClientFactory to create and configure the HttpClient 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: *http://www.jboss.org/community/docs/DOC-9969*.	No
method	Currently only supports GET and POST.	Yes
responseType	Specifies in what form the response should be sent back. Either STRING or BYTES. Default value is STRING.	No
headers	Http headers To be added to the request. Supports multiple <header name=”blah” value=”blahvalue”/> elements.	No

Example 11.16. httprouter

```
<action name="httprouter"
  class="org.jboss.soa.esb.actions.routing.http.HttpRouter">
    <property name="endpointUrl"value="http://host:80/blah">
        <http-client-property name="file" value="/ht.props"/>
    </property>
    <property name="method" value="GET"/>
    <property name="responseType" value="STRING"/>
    <property name="headers">
        <header name="blah" value="blahval" ></header>
    </property>
</action>
```

11.5.5. JMSRouter

Class	**org.jboss.soa.esb.actions.routing.JMSRouter**

Routes the incoming message on to JMS.

Table 11.19. JMSRouter

Property	Description	Required
unwrap	When set to **true**, the message payload from the Message object will be extracted and sent. Otherwise the entire Message object is sent. Default is **false**.	No
jndi-context-factory	The JNDI context factory to use. The default is **org.jnp.interfaces.NamingContextFactory**.	No
jndi-URL	The JNDI URL to use. The default is 127.0.0.1:1099	No
jndi-pkg-prefix	The JNDI naming package prefixes to use. The default is **org.jboss.naming:org.jnp.interfaces**.	No
connection-factory	The name of the ConnectionFactory to use. Default is **ConnectionFactory**.	No
persistent	The JMS DeliveryMody. Default value is **true**.	No
priority	The JMS priority to be used. Default is **javax.jms.Message.DEFAULT_PRIORITY**.	No

Property	Description	Required
time-to-live	The JMS Time-To-Live to be used. The default is **`javax.jms.Message.DEFAULT_TIME_TO_LIVE`**.	No
security-principal	The security principal to use when creating the JMS connection.	Yes
security-credentials	The security credentials to use when creating the JMS connection.	Yes
property-strategy	The implementation of the **`JMSPropertiesSetter`** interface, if overriding the default.	No
message-prop:	The properties to be set on the message are prefixed with **`message-prop:`**.	

11.5.6. ContentBasedRouter

Class	**`org.jboss.soa.esb.actions.ContentBasedRouter`**

Content (plus rules) based message routing action.

ContentBasedRouter has two `process` methods. `process` doesn't append aggregation data to message. `split` does append aggregation data to message. See *Section 11.5.2, "Aggregator"* for more details.

You can refer to Content Based Routing in the JBoss Enterprise SOA Platform Services Guide [7] for more details.

Table 11.20. ContentBasedRouter

Property	Description	Required
ruleSet	JBoss Rules ruleset.	
ruleLanguage	CBR evaluation Domain Specific Language (DSL) file.	
ruleReload	Flag indicating whether or not the rules file should be reloaded each time. Default is "false".	
destinations	Container property for the <route-to> configurations.	

Example 11.17. ContentBasedRouter

```
<action process="split" name="ContentBasedRouter"
  class="org.jboss.soa.esb.actions.ContentBasedRouter">
    <property name="ruleSet" value="MyESBRules-XPath.drl"/>
    <property name="ruleLanguage" value="XPathLanguage.dsl"/>
    <property name="ruleReload" value="true"/>
    <property name="destinations">
        <route-to destination-name="express"
                  service-category="ExpressShipping"
                  service-name="ExpressShippingService"/>
        <route-to destination-name="normal"
                  service-category="NormalShipping"
```

[7] The JBoss Enterprise SOA Platform Services Guide is provided as the file **`Services_Guide.pdf`** or can be viewed online at *http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/*

```
                    service-name="NormalShippingService"/>
    </property>
</action>
```

11.5.7. StaticRouter

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

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

Table 11.21. StaticRouter Properties

Property	Description	Required
destinations	Container property for the <route-to> configurations. `<route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/>`	Yes

Table 11.22. StaticRouter Process Methods

method	Description
process	Don't append aggregation data to message.
split	Append aggregation data to message.

See *Section 11.5.2, "Aggregator"*.

Example 11.18. StaticRouter

```
<action name="routeAction"
    class="org.jboss.soa.esb.actions.StaticRouter">
    <property name="destinations">
       <route-to service-category="ExpressShipping"
        service-name="ExpressShippingService"/>
    <route-to service-category="NormalShipping"
        service-name="NormalShippingService"/>
    </property>
</action>
```

11.5.8. StaticWireTap

The StaticWiretap differs from the StaticRouter in that it does not cause the pipeline processing to end.

Class	**org.jboss.soa.esb.actions.StaticWireTap**

Table 11.23. StaticWireTap Properties

Property	Description	Required
destinations	Container property for the <route-to> configurations. `<route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/>`	Yes

Table 11.24. StaticWireTap Process Methods

method	Description
process	Don't append aggregation data to message.
split	Append aggregation data to message.

See *Section 11.5.2, "Aggregator"*.

Example 11.19. StaticWireTap

```
<action name="routeAction"
 class="org.jboss.soa.esb.actions.StaticWiretap">
 <property namjbosse="destinations">
  <route-to service-category="ExpressShipping"
   service-name="ExpressShippingService"/>
  <route-to service-category="NormalShipping"
   service-name="NormalShippingService"/>
 </property>
</action>
```

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 processsing; a NotificationList of type "err" specifies targets to receive notifications upon exceptional action pipeline processing, according to the action pipeline processing semantics mentioned earlier. Both "err" and "ok" are case insensitive.

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

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

Class	org.jboss.soa.esb.actions.Notifier
Properties	NotificationList subtree indicating targets
Sample Configuration	<action class="org.jboss.soa.esb.actions.Notifier" okMethod="notifyOK"> <property name="destinations"> <NotificationList type="OK">

```
      <target class="NotifyConsole" />
      <target class="NotifyFiles" >
        <file name=”@results.dir@/goodresult.log” />
      </target>
    </NotificationList>
    <NotificationList type="err">
      <target class="NotifyConsole" />
      <target class="NotifyFiles" >
        <file name=”@results.dir@/badresult.log” />
      </target>
    </NotificationList>
  </property>
</action>
```

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

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

Class	NotifyFiles
Purpose	Performs a notification by writing the contents of the ESB message to a specified set of files.
Attributes	none
Child	file
Child Attributes	1. append – if value is true, append the notification to an existing file 2. URI – any valid URI specifying a file
Sample Configuration	<target class="NotifyFiles" > <file append=”true” URI=”anyValidURI”/> <file URI=”anotherValidURI”/>

	</target>

Class	NotifySQLTable
Purpose	Performs a notification by inserting a record into an existing database table. The database record contains the ESB message contents and, optionally, other values specified using nested <column> elements.
Attributes	1. driver-class 2. connection-url 3. user-name 4. password 5. table – table in which notification record is stored 6. dataColumn – name of table column in which ESB message contents are stored
Child	column
Child Attributes	1. name – name of table column in which to store additional value 2. value – value to be stored
Sample Configuration	<target class="NotifySQLTable" driver-class=”com.mysql.jdbc.Driver” connection-url=”jdbc:mysql://localhost/db” user-name=”user” password=”password” table=”table” dataColumn=”messageData”> <column name=”aColumnlName” value=”aColumnValue”/> </target>

Class	NotifyFTP
Purpose	Performs a notification by creating a file containing the ESB message content and transferring it via FTP to a remote file system.
Attributes	none
Child	ftp
Child Attributes	1. URL – a valid FTP URL 2. filename – the name of the file to contain the ESB message content on the remote system
Sample Configuration	<target class="NotifyFTP" > <ftp URL=”ftp://username:pwd@server.com/remote/dir” filename=”someFile.txt” />

	</target>

Class	NotifyQueues
Purpose	Performs a notification by translating the ESB message (including its attached properties) into a JMS message and sending the JMS message to a list of Queues. Additional properties may be attached using the <messageProp> element.
Attributes	none
Child	queue
Child Attributes	1. jndiName – the JNDI name of the Queue 2. jndi-URL – the JNDI provider URL (optional) 3. jndi-context-factory – the JNDI initial context factory (optional) 4. jndi-pkg-prefix – the JNDI package prefixes (optional) 5. connection-factory – the JNDI name of the JMS connection factory (by default, “ConnectionFactory”)
Child	messageProp
Child Attributes	1. name – name of the new property to be added 2. value – value of the new property
Sample Configuration	<target class="NotifyQueues" > <messageProp name=”aNewProperty” value=”theValue”/> <queue jndiName=”queue/quickstarts_notifications_queue” /> </target>

Class	NotifyTopics
Purpose	Performs a notification by translating the ESB message (including its attached properties) into a JMS message and publishing the JMS message to a list of Topics. Additional properties may be attached using the <messageProp> element.
Attributes	none
Child	topic
Child Attributes	1. jndiName – the JNDI name of the Queue 2. jndi-URL – the JNDI provider URL (optional) 3. jndi-context-factory – the JNDI initial context factory (optional) 4. jndi-pkg-prefix – the JNDI package prefixes (optional) 5. connection-factory – the JNDI name of the JMS connection factory (by default, “ConnectionFactory”)
Child	messageProp
Child Attributes	1. name – name of the new property to be added 2. value – value of the new property
Sample Configuration	<target class="NotifyTopics" >

	<messageProp name=”aNewProperty” value=”theValue”/> <queue jndiName=”topic/quickstarts_notifications_topic” /> </target>

Class	NotifyEmail
Purpose	Performs a notification by sending an email containing the ESB message content and, optionally, any file attachments.
Attributes	1. from – email address (javax.email.InternetAddress) 2. sendTo – comma-separated list of email addresses 3. ccTo – comma-separated list of email addresses (optional) 4. subject – email subject 5. message – a string to be prepended to the ESB message contents which make up the e-mail message (optional) 6. msgAttachmentName - filename of an attachment containing the message payload (optional). If not specified the message payload will be included in the message body.
Child	Attachment (optional)
Child Text	the name of the file to be attached
Sample Configuration	<target class="NotifyEmail" from=”person@somewhere.com” sendTo=”person@elsewhere.com” subject=”theSubject”> <attachment>attachThisFile.txt</attachment> </target>

11.7. Webservices/SOAP

11.7.1. JBoss Webservices SOAPProcessor

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

Dependencies

1. JBoss Application Server 4.2.0GA or higher.
2. JBossWS 2.0.x or higher

3. The **soap.esb** Service. This is deployed as part of the **production** server configuration by default.

A JBossWS console is located at *http://localhost:8080/jbossws*. and provides access to a list of all deployed JBossWS endpoints. Additional information about JBossWS can be found at *http://jbossws.jboss.org/mediawiki/index.php?title=JBossWS*

11.7.1.1. "ESB Message Aware" Webservice Endpoints

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

11.7.1.2. Webservice Endpoint Deployment

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

11.7.1.3. WSDL

WSDLs for Webservices exposed via JBossESB are available through the "Contracts" application (deployed with the ESB components). This application can be accessed through your JBoss SOA Platform Server at *http://localhost:8080/contract*. This application lists URLs that can be used by your Webservice Client (e.g. soapUI) for accessing a Service's WSDL, enabling WSDL based invocation of the Service through an ESB Endpoint.

See the “Contract Publishing” section of the Administration Guide [8] for details on Endpoint Publishing.

11.7.1.4. JAXB Annotation Introductions

The native JBossWS SOAP stack uses JAXB to bind to and from SOAP. This means that an un-annotated typeset cannot be used to build a JBossWS endpoint. To overcome this we provide a JBossESB and JBossWS feature called "JAXB Annotation Introductions" which basically means you can define an XML configuration to "Introduce" the JAXB Annotations.

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 enable this feature in JBossWS 2.0.0, see *Appendix A, Writing JAXB Annotation Introduction Configurations*.

11.7.1.5. Action Configuration

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

[8] The JBoss Enterprise SOA Administration Guide is provided as the file **Administration_Guide.pdf** or can be viewed online at *http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/*

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

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

11.7.1.6. Quickstarts

A number of quickstarts demonstrating how to use this action are available. See the **`webservice_producer`** quickstart.

11.7.2. SOAPCLIENT - WISE

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>
```

Optional Properties

Property Name	Description
wsdl	The WSDL to be used.
SOAPAction	The endpoint operation.
EndPointName	The EndPoint invoked. Webservices can have multiple endpoint. If it's not specified the first specified in wsdl will be used.
SmooksRequestMapper	Specifies a smooks config file to define the java-to-java mapping defined for the request.
SmooksResponseMapper	Specifies a smooks config file to define the java-to-java mapping defined for the response
ServiceName	A symbolic service name used by wise to cache object generation and/or use already generated object. If it isn't provided wise uses the servlet name of wsdl.
UserName	Username used if the webservice is protected by BASIC Authentication HTTP.
Password	Password used if the webservice is protected by BASIC Authentication HTTP.
smooksTransform	It's often necessary to be able to transform the SOAP request or response, especially in header. This may be to simply add some standard SOAP handlers. Wise support JAXWS Soap Handler, both custom or a predefined one based on smooks.

	Transformation of the SOAP request (before sending) is supported by configuring the SOAPClient action with a Smooks transformation configuration property.
custom-handlers	It's also possible to provide a set of custom standard JAXWS Soap Handler. The parameter accept a list of classes implementing SoapHandler interface. Classes have to provide full qualified name and be separated by semi-columns.
LoggingMessages	It's useful for debug purpose to view soap Message sent and response received. Wise achieve this goal using a JAX-WS handler printing all messages exchanged on System.out. Boolean value.

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

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

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

1. With a set of Objects of any type. In this case a Smooks config 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 SOAP parameter as specified in wsdls (or in generated class) to be populated with the key's value. SOAP Response Message Consumption

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

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

The response object instance can also be populated (from the SOAP response) in one of 2 ways:

1. With a set of Objects of any type. In this case a smooks config have 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 SOAP answer as specified in wsdls (or in generated class) to be populated with the key's value. JAX-WS Handler for the SOAP Request/Response

For examples of using the SOAPClient please refer to the following quickstarts:

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_consomer_wise4`**, shows usage of 'custom-handlers'.

More information about Wise can be found on their website *http://www.javalinuxlabs.org/wise*.

11.7.3. SOAPClient - SOAPUI

Uses the soapUI Client Service to construct and populate a message for the target service. This action then routes that message to that service. You can find additional details about soapUI at *http://www.soapui.org*.

11.7.3.1. HTTP Connection Configuration.

This "Configurator" is always applied to `HttpClient` instances created by the **`HttpClientFactory`**. To make the connection configurations in a different way, create a custom configurator that can generate and attach a different **`HttpConnectionManager`** instance with its own settings.

Table 11.25. Properties

Property	Description
max-total-connections	Maximum total number of connection for the `HttpClient` instance.
max-connections-per-host	Maximum connection per target host for the `HttpClient` instance. Note that the default value for this configuration is **2**. (Hence, configuring max-total-connections without also configuring this property will have little impact on performance for a single host because it will only ever open a maximum of two connections.)

11.7.3.2. 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>
```

11.7.3.3. SOAP Request Message Construction

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

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

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

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

As stated above, OGNL is the mechanism we use for selecting the SOAP parameter values to be injected into the SOAP message from the supplied parameter Map. The OGNL expression for a

specific parameter within the SOAP message depends on that the position of that parameter within the SOAP body. You can find additional information about OGNL at *http://www.opensymphony.com/ognl/*.

In the following message:

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

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

Once the OGNL expression has been calculated for a parameter, this class will check the supplied parameter map for an Object keyed off the full OGNL expression (Option 1 above). If no such parameter Object is present on the map, this class will then attempt to load the parameter by supplying the map and OGNL expression instances to the OGNL toolkit (Option 2 above). If this doesn't yield a value, this parameter location within the SOAP message will remain blank.

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

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

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

```
</soapenv:Envelope>
```

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

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

11.7.3.4. SOAP Response Message Consumption

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

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

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

1. As an Object Graph created and populated by the XStream toolkit.

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

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

Using XStream as a mechanism for populating an Object Graph (Option 1 above) is straightforward and works well, as long as the XML and Java object models are in line with each other. You can find more information about XStream at *http://xstream.codehaus.org*[9].

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

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

[9] http://xstream.codehaus.org/

```
  <alias name="orderheader" class="com.acme.order.Header"
   namespace="http://schemas.acme.com/services/CustomerOrder.xsd" />
  <alias name="item" class="com.acme.order.OrderItem"
   namespace="http://schemas.acme.com/services/CustomerOrder.xsd" />
 </property>
</action>
```

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

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

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

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

11.7.3.5. 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>
```

The file property value will be evaluated, in order, as a filesystem, classpath or URI based resource.

The following is an example of this property set:

```
# Configurators
configurators=HttpProtocol,AuthBASIC

# HttpProtocol config...
protocol-socket-factory=org.apache.commons.httpclient.contrib.ssl.EasySSLPr
otocolSocketFactory
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
```

Properties may also be set directly in the action configuration.

```
<property name="http-client-properties">
    <http-client-property name="http.proxyHost" value="localhost"/>
    <http-client-property name="http.proxyPort" value="8080"/>
</property>
```

Additional information about the available configuration options is available at *https://www.jboss.org/community/wiki/SOAPClient*.

11.8. Miscellaneous

Miscellaneous Action Processors.

SystemPrintln

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

Will attempt to format the message contents as XML.

Input Type	java.lang.String
Class	org.jboss.soa.esb.actions.SystemPrintln
Properties	1. “message”: A message prefix. 1. “printfull”: If true then the entire message is printed, otherwise just the byte array and attachments. 2. “outputstream”: if true then System.out is used, otherwise System.err.
Sample Configuration	<action name="print-before" class="org.jboss.soa.esb.actions.SystemPrintln"> <property name="message" value="Message before action XXX" /> </action>

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 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...
    }
}
```

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>
```

The default method for loading property values in an action implementation is the use of a **`ConfigTree`** instance. The ConfigTree provides a DOM-like view of the action XML. By default, actions are expected to have a public constructor that takes a **`ConfigTree`** as a parameter. For example:

```
public class PrintMessage extends AbstractActionPipelineProcessor {

 private String information;
 private Integer repeatCount;

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

 public Message process(Message message) throws
```

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

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);
  }
 }
}
```

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 clients and services of JBossESB will be able to understand the protocols and Message formats it uses natively. As such 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 in to” 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, to 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: specialized servers 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 etc (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, which will be delivered to a target service that must be an ESB aware service. 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; same idea for JMS messages. Default message composing class for a SQL table row is to package contents of all columns specified in configuration, into a java.util.Map.

Although these default composer classes will be enough for most use cases, it is relatively straightforward for users to provide their own message composing classes. The only requirements are a) they must have a constructor that takes a single ConfigTree argument, and b) they must provide a 'Message composing' method (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.

13.2.1. 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 JMS **`TextMessage`**, 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);
}
```

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);
  }
 }
}
```

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. Then it 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>
```

JCA gateways are defined in <jca-gateway> elements. These are the configurable attributes of this XML element.

Table 13.1. jca-gateway Configuration Attributes

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., **`jms-ra.rar`**
endpointClass	yes	The name of your endpoint class
messagingType	no	The message interface for the adapter. If you do not specify one, ESB will guess based on the endpoint class.
transacted	no	Default to true. Whether or not you want to invoke the message within a JTA transaction.

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. 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.

Table 13.2. Activation Properties

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 inflow adapters.

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 using the **`jbossesb-properties.xml`** file and 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 class name 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>
```

Specifying the ActivationMapper within the deployment will override any global setting. The mapper can be specified within the listener, the bus or the provider in that 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>
```

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>
```

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>
```

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/jbossws/projects/jaxbintros/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 becomr 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 requires changes to the way in which people work and interact as much as assistance from underlying infrastructures, such as JBossESB. With the SOA Platform, Red Hat is providing a base SOA infrastructure upon which SOA applications can be developed. The Platform will continue to evolve, with out-of-the-box improvements around tooling, runtime management, service life-cycle and so forth.

Appendix C. Revision History

Revision 1.5 **Mon Mar 21 2011** **David Le Sage** *dlesage@redhat.com*

Updated for 4.3.CP05 Release

Revision 1.4 **Tue Apr 27 2010** **David Le Sage** *dlesage@redhat.com*

Updated for 4.3.CP03 Release
SOA-2003 - added information about routing actions that terminate the pipeline. Section 11.5.1
SOA-1959 - corrected property name. Section 11.5.4
SOA-1506 - removed BPMProcesor commands no longer in use. Section 11.2.1 and 11.7.3
SOA-1470 - added information about HTTP Connection Configuration. Section 11.7.3.1

Revision 1.3 **Tue Apr 20 2010** **David Le Sage** *dlesage@redhat.com*

Updated for SOA 4.3.CP03

Revision 1.2 **Wed Jul 1 2009** **Darrin Mison** *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 webservices 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*

Updated for 4.3.CP01 Release

Revision 1.0 **Fri Jan 23 2009** **Darrin Mison** *dmison@redhat.com*

Created