JBoss Enterprise SOA Platform 5

ESB Programmers Guide

This guide is for software engineers.

Edition 5.3.1

Last Updated: 2017-10-27

JBoss Enterprise SOA Platform 5 ESB Programmers Guide

This guide is for software engineers.
Edition 5.3.1

David Le Sage
Red Hat Engineering Content Services
dlesage@redhat.com

B Long
Red Hat Engineering Content Services
belong@redhat.com

Darrin Mison

Tom Wells
twells@redhat.com

Legal Notice

Abstract

This reference document contains information about programming with the JBoss Enterprise SOA Platform product.

Table of Contents

PREFACE

CHAPTER 1. PREFACE

1.1. BUSINESS INTEGRATION

In order to provide a dynamic and competitive business infrastructure, it is crucial to have a service-oriented architecture in place that enables your disparate applications and data sources to communicate with each other with minimum overhead.

The JBoss Enterprise SOA Platform is a framework capable of orchestrating business services without the need to constantly reprogram them to fit changes in business processes. By using its business rules and message transformation and routing capabilities, JBoss Enterprise SOA Platform enables you to manipulate business data from multiple sources.

Report a bug

1.2. WHAT IS A SERVICE-ORIENTED ARCHITECTURE?

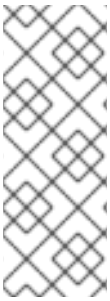
Introduction

A *Service Oriented Architecture* (SOA) is not a single program or technology. Think of it, rather, as a software design paradigm.

As you may already know, a *hardware bus* is a physical connector that ties together multiple systems and subsystems. If you use one, instead of having a large number of point-to-point connectors between pairs of systems, you can simply connect each system to the central bus. An *enterprise service bus* (ESB) does exactly the same thing in software.

The ESB sits in the architectural layer above a messaging system. This messaging system facilitates *asynchronous communications* between services through the ESB. In fact, when you are using an ESB, everything is, conceptually, either a *service* (which, in this context, is your application software) or a *message* being sent between services. The services are listed as connection addresses (known as *end-points references*.)

It is important to note that, in this context, a "service" is not necessarily always a web service. Other types of applications, using such transports as File Transfer Protocol and the Java Message Service, can also be "services."



NOTE

At this point, you may be wondering if an enterprise service bus is the same thing as a service-oriented architecture. The answer is, "Not exactly." An ESB does not form a service-oriented architecture of itself. Rather, it provides many of the tools than can be used to build one. In particular, it facilitates the *loose-coupling* and *asynchronous message passing* needed by a SOA. Always think of a SOA as being more than just software: it is a series of principles, patterns and best practices.

Report a bug

1.3. KEY POINTS OF A SERVICE-ORIENTED ARCHITECTURE

These are the key components of a service-oriented architecture:

1. the *messages* being exchanged
2. the *agents* that act as service requesters and providers
3. the *shared transport mechanisms* that allow the messages to flow back and forth.

Report a bug

1.4. WHAT IS THE JBOSS ENTERPRISE SOA PLATFORM?

The JBoss Enterprise SOA Platform is a framework for developing enterprise application integration (EAI) and service-oriented architecture (SOA) solutions. It is made up of an enterprise service bus (JBoss ESB) and some business process automation infrastructure. It allows you to build, deploy, integrate and orchestrate business services.

Report a bug

1.5. THE SERVICE-ORIENTED ARCHITECTURE PARADIGM

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 Requester

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 requester to a service provider.

Report a bug

1.6. CORE AND COMPONENTS

The JBoss Enterprise SOA Platform provides a comprehensive server for your data integration needs. On a basic level, it is capable of updating business rules and routing messages through an Enterprise Service Bus.

The heart of the JBoss Enterprise SOA Platform is the Enterprise Service Bus. JBoss (ESB) creates an environment for sending and receiving messages. It is able to apply “actions” to messages to transform them and route them between services.

There are a number of components that make up the JBoss Enterprise SOA Platform. Along with the ESB, there is a registry (jUDDI), transformation engine (Smooks), message queue (HornetQ) and BPEL engine (Riftsaw).

Report a bug

1.7. COMPONENTS OF THE JBOSS ENTERPRISE SOA PLATFORM

- A full Java EE-compliant application server (the JBoss Enterprise Application Platform)
- an enterprise service bus (JBoss ESB)
- a business process management system (jBPM)
- a business rules engine (JBoss Rules)
- support for the optional JBoss Enterprise Data Services (EDS) product.

Report a bug

1.8. JBOSS ENTERPRISE SOA PLATFORM FEATURES

The JBoss Enterprise Service Bus (ESB)

The ESB sends messages between services and transforms them so that they can be processed by different types of systems.

Business Process Execution Language (BPEL)

You can use web services to orchestrate business rules using this language. It is included with SOA for the simple execution of business process instructions.

Java Universal Description, Discovery and Integration (jUDDI)

This is the default service registry in SOA. It is where all the information pertaining to services on the ESB are stored.

Smooks

This transformation engine can be used in conjunction with SOA to process messages. It can also be used to split messages and send them to the correct destination.

JBoss Rules

This is the rules engine that is packaged with SOA. It can infer data from the messages it receives to determine which actions need to be performed.

Report a bug

1.9. FEATURES OF THE JBOSS ENTERPRISE SOA PLATFORM'S JBOSSESB COMPONENT

The JBoss Enterprise SOA Platform's JBossESB component supports:

- Multiple transports and protocols
- A listener-action model (so that you can loosely-couple services together)

- Content-based routing (through the JBoss Rules engine, XPath, Regex and Smooks)
- Integration with the JBoss Business Process Manager (jBPM) in order to provide service orchestration functionality
- Integration with JBoss Rules in order to provide business rules development functionality.
- Integration with a BPEL engine.

Furthermore, the ESB allows you to integrate legacy systems in new deployments and have them communicate either synchronously or asynchronously.

In addition, the enterprise service bus provides an infrastructure and set of tools that can:

- Be configured to work with a wide variety of transport mechanisms (such as e-mail and JMS),
- Be used as a general-purpose object repository,
- Allow you to implement pluggable data transformation mechanisms,
- Support logging of interactions.

IMPORTANT

There are two trees within the source code: **`org.jboss.internal.soa.esb`** and **`org.jboss.soa.esb`**. Use the contents of the **`org.jboss.internal.soa.esb`** package sparingly because they are subject to change without notice. By contrast, everything within the **`org.jboss.soa.esb`** package is covered by Red Hat's deprecation policy.

Report a bug

1.10. TASK MANAGEMENT

JBoss SOA simplifies tasks by designating tasks to be performed universally across all systems it affects. This means that the user does not have to configure the task to run separately on each terminal. Users can connect systems easily by using web services.

Businesses can save time and money by using JBoss SOA to delegate their transactions once across their networks instead of multiple times for each machine. This also decreases the chance of errors ocurring.

Report a bug

1.11. INTEGRATION USE CASE

Acme Equity is a large financial service. The company possesses many databases and systems. Some are older, COBOL-based legacy systems and some are databases obtained through the acquisition of smaller companies in recent years. It is challenging and expensive to integrate these databases as business rules frequently change. The company wants to develop a new series of client-facing e-commerce websites, but these may not synchronise well with the existing systems as they currently stand.

The company wants an inexpensive solution but one that will adhere to the strict regulations and security requirements of the financial sector. What the company does not want to do is to have to write and maintain “glue code” to connect their legacy databases and systems.

The JBoss Enterprise SOA Platform was selected as a middleware layer to integrate these legacy systems with the new customer websites. It provides a bridge between front-end and back-end systems. Business rules implemented with the JBoss Enterprise SOA Platform can be updated quickly and easily.

As a result, older systems can now synchronise with newer ones due to the unifying methods of SOA. There are no bottlenecks, even with tens of thousands of transactions per month. Various integration types, such as XML, JMS and FTP, are used to move data between systems. Any one of a number of enterprise-standard messaging systems can be plugged into JBoss Enterprise SOA Platform providing further flexibility.

An additional benefit is that the system can now be scaled upwards easily as more servers and databases are added to the existing infrastructure.

Report a bug

1.12. UTILISING THE JBOSS ENTERPRISE SOA PLATFORM IN A BUSINESS ENVIRONMENT

Cost reduction can be achieved due to the implementation of services that can quickly communicate with each other with less chance of error messages occurring. Through enhanced productivity and sourcing options, ongoing costs can be reduced.

Information and business processes can be shared faster because of the increased connectivity. This is enhanced by web services, which can be used to connect clients easily.

Legacy systems can be used in conjunction with the web services to allow different systems to "speak" the same language. This reduces the amount of upgrades and custom code required to make systems synchronise.

Report a bug

PART I. INTRODUCTION

CHAPTER 2. PRELIMINARIES

2.1. INTENDED AUDIENCE

This book has been written for developers needing a comprehensive guide to all the options available when building solutions with the JBoss Enterprise SOA Platform.

Report a bug

2.2. AIM OF THIS BOOK

Read this book in order to learn how to develop services and integrate your end-points with the JBoss Enterprise SOA Platform. The book guides you through theory and then practical examples, defining key terms along the way. Developing services with both pre-packaged and customized actions and the use of decoders, gateways and connectors in order to integrate your systems are explored in detail.

Report a bug

2.3. BACK UP YOUR DATA

WARNING

Red Hat recommends that you back up your system settings and data before undertaking any of the configuration tasks mentioned in this book.

Report a bug

2.4. RED HAT DOCUMENTATION SITE

Red Hat's official documentation site is at https://access.redhat.com/knowledge/docs/. There you will find the latest version of every book, including this one.

Report a bug

2.5. VARIABLE NAME: SOA_ROOT DIRECTORY

SOA Root (often written as SOA_ROOT) is the term given to the directory that contains the application server files. In the standard version of the JBoss Enterprise SOA Platform package, SOA root is the **`jboss-soa-p-5`** directory. In the Standalone edition, though, it is the **`jboss-soa-p-standalone-5`** directory.

Throughout the documentation, this directory is frequently referred to as **`SOA_ROOT`**. Substitute either **`jboss-soa-p-5`** or **`jboss-soa-p-standalone-5`** as appropriate whenever you see this name.

Report a bug

2.6. VARIABLE NAME: PROFILE

PROFILE can be any one of the server profiles that come with the JBoss Enterprise SOA Platform product: default, production, all, minimal, standard or web. Substitute one of these that you are using whenever you see "PROFILE" in a file path in this documentation.

Report a bug

CHAPTER 3. INTRODUCING THE JBOSS ENTERPRISE SOA PLATFORM

3.1. ENTERPRISE SERVICE BUS

An enterprise service bus is a concrete implementation of an abstract SOA design philosophy. An enterprise service bus (ESB) has two roles: it provides message routing functionality and allows you to register services. The enterprise service bus that lies at the center of the JBoss Enterprise SOA Platform is called JBoss ESB.

An enterprise service bus deals with infrastructure logic, not business logic (which is left to higher levels). Data is passed through the enterprise service bus on its way between two or more systems. Message queuing may or may not be involved. The ESB can also pass the data to a transformation engine before passing it to its destination.

Report a bug

3.2. CORE COMPONENTS OF THE ENTERPRISE SERVICE BUS

The enterprise service bus is built on top of four key architectural components. These are:

Message listening and message filtering code

Message listeners act as routers that 'listen' for inbound messages (such as those on a JMS queue or topic, or on the file system). They then present the message to a processing pipeline that filters and routes it (via an outbound router) to another message end-point.

Data transformation components

These are based on Smooks and XSLT.

A content-based router

This infers a message's destination from the information in its body.

A message repository

This is used to save messages and/or events that have been exchanged via the ESB.

These components, in turn, consist of a set of business classes, adapters and processors.

Client-service interaction is facilitated by a range of different approaches, including JMS, flat-file systems and e-mail.

Report a bug

3.3. INTEGRATION BETWEEN EDS AND THE JBOSS ENTERPRISE SOA PLATFORM

JBoss Enterprise Data Services is a superset of JBoss Enterprise SOA Platform. EDS augments and extends the JBoss Enterprise SOA Platform to address data access, integration and abstraction through:

- Service-oriented architecture patterns and best practices

- Service-oriented architecture patterns and best practices
- Reporting and analytics enablement
- Master data services
- Data governance and compliance
- Real-time read and write access to heterogeneous data stores
- Fast application development by simplifying access to distributed data
- Centralized access control and auditing

Report a bug

3.4. ENTERPRISE DATA SERVICES OVERVIEW

A complete Enterprise Data Services (EDS) solution consists of the following:

EDS Service

The EDS Service is positioned between business applications and one or more data sources. It coordinates integration of these data sources so they can be accessed by the business applications at runtime.

Design Tools

Various design tools are available to assist users in setting up an EDS Service for a particular data integration solution.

Administration Tools

Various management tools are available for administrators to configure and monitor a deployed EDS Service.

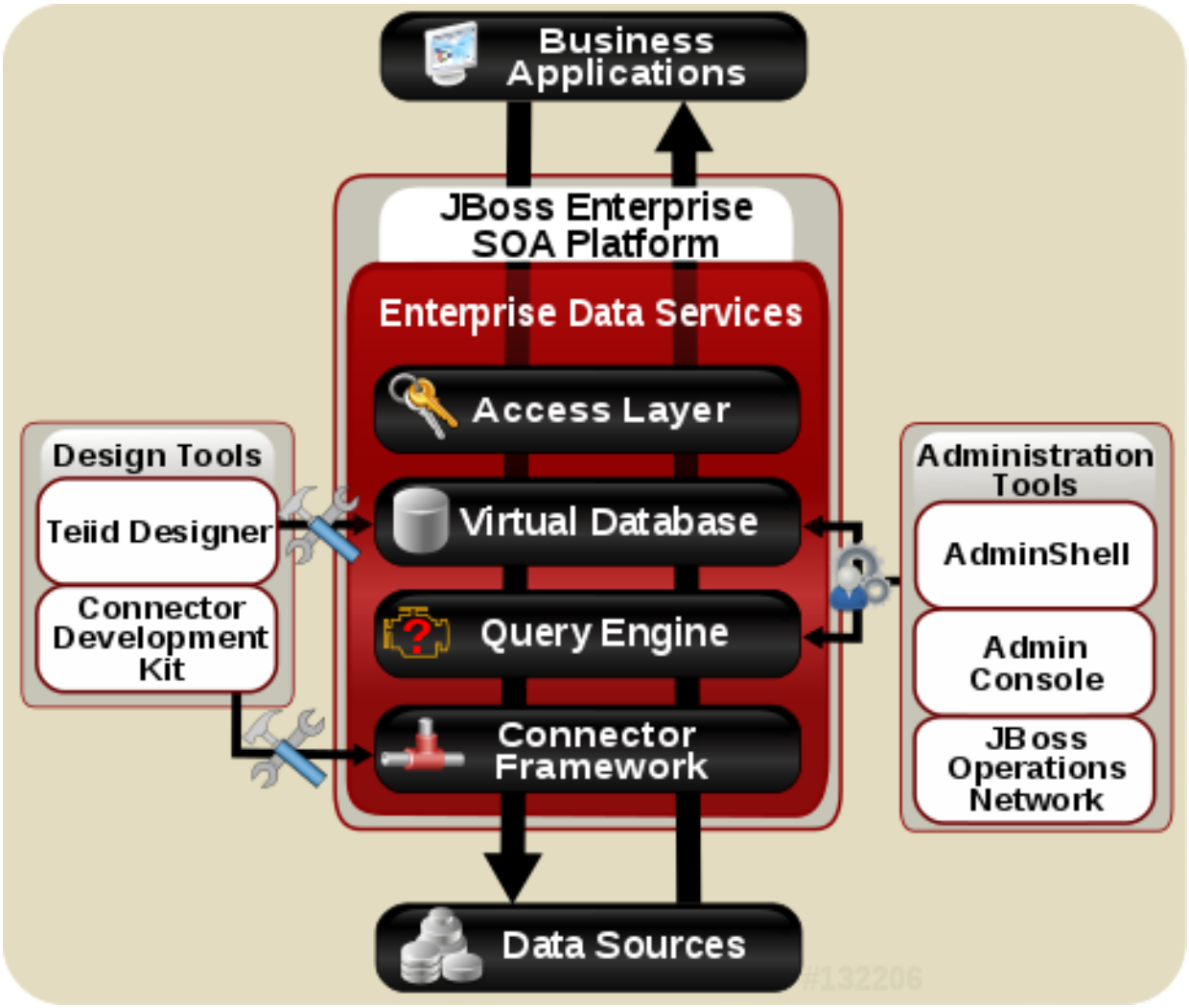


Figure 3.1. Enterprise Data Services Overview

Report a bug

3.5. DEVELOPING WITH ENTERPRISE DATA SERVICES

The JBoss Enterprise Data Services product is exposed via a JDBC driver or web service. ESB services can consume it through either method.

Note that the JDBS connection string for an EDS Virtual Database (VDB) differs from a normal JDBC connection string. This is the correct format for an EDS virtual database JBDC connection string: **`jdbc:teiid:vdb_name@mm://localhost:31000`**.

Report a bug

PART II. THEORY

CHAPTER 4. SERVICES AND MESSAGES

4.1. SERVICES

4.1.1. Service

A service is a list of action classes that process an ESB Message in a sequential manner. Each service element consists of one or more listeners and one or more actions. These are set within the **`jboss-esb.xml`** configuration file.

Report a bug

4.1.2. Action Pipeline

The action pipeline consists of a list of action classes through which messages are processed. Use it to specify which actions are to be undertaken when processing the message. Actions can transform messages and apply business logic to them. Each action passes the message on to the next one in the pipeline or, at the conclusion of the process, directs it to the end-point listener specified in the ReplyTo address.

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.

Report a bug

4.1.3. ESB-Awareness

If application clients and services are referred to as being ESB-aware, this means that they can understand the message format and transport protocols used by the SOA Platform's enterprise service bus.

Report a bug

4.1.4. Message Listeners

Message listeners encapsulate the communications end-points needed to receive SB-aware messages. Listeners are defined by services and their role is to monitor queues. They receive any messages as they land in those queues. When a listener receives a message, the ESB server calls the appropriate action class defined in the action definition. The methods in this class process the message. In other words, listeners act as inbound routers, directing messages to the action pipeline. When the message has been modified by the actions on the pipeline, the listener sends the result to the replyTo end-point.

You can configure various parameters for listeners. For instance, you can set the number of active worker threads.

There are two types of listeners: ESB-aware listeners and gateway listeners. Gateway listeners are

different from ESB-aware listeners in that they accept data in different formats (such as objects in files, SQL tables and JMS messages). They then convert them from these formats to the ESB messaging format. By contrast, ESB-aware listeners can only accept messages that are in the **`org.jboss.soa.esb.message.Message`** format. Each gateway listener must have a corresponding ESB listener defined.

With ESB-aware listeners, RuntimeExceptions can trigger rollbacks. By contrast, with a gateway listener, the transaction simply sends the message to the JBoss ESB. The message is then processed asynchronously. In this way, message failures are separated from message receipts.

Report a bug

4.1.5. ServiceInvoker

The ServiceInvoker (**`org.jboss.soa.esb.client.ServiceInvoker`**) manages the delivery of messages to the specified Services. It also manages the loading of end-point references and the selection of couriers, thereby providing a unified interface for message delivery.

The ServiceInvoker was introduced to help simplify the development effort as it hides much in the way of the lower-level details and works opaquely with the stateless service fail- over mechanisms. As such, ServiceInvoker is the recommended client-side interface for using services within the JBoss Enterprise SOA Platform.

You can create an instance of the ServiceInvoker for each service with which the client interacts. Once created, an instance examines the registry to determine the primary end-point reference and, in the case of fail-overs, any alternative end-point references.

Report a bug

4.1.6. InVM Transport

The InVM ("intra-virtual machine") Transport provides communication between services running on the same JVM.

Report a bug

4.1.7. Creating Your First Service

Here is a very simple JBoss ESB configuration that defines a single Service that outputs the contents of a message to the console.

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

<services>
    <service category="Retail" name="ShoeStore" description="Acme Shoe
Store
Service" invmScope="GLOBAL">
        <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 listeners in the Service Registry. Clients can then invoke the service using the ServiceInvoker as per this next sample:

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

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

The ServiceInvoker uses the Services Registry to look up the available endpoint addresses for the **`Retail:ShoeStore`** service. The registry automatically handles the process of sending the message from the client to one of the available endpoints. The process of transporting the message is completely transparent to the client.

The end point 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"1`**. To add additional endpoints to the service, you must add them explicitly.

Report a bug

4.1.8. Types of Message Listener

There are two types of message listener:

Gateway Listener

This type of listener configure a gateway endpoint, which is used to push ESB-unaware messages into an ESB bus. It changes the message into a form the ESB can understand by wrapping it inside an ESB Message before sending it to the action pipeline.

ESB-Aware Listener

This type of listener creates an “ESB Aware” endpoint and is used to exchange ESB Messages between ESB-aware components.

Report a bug

4.1.9. Gateway Listener

A gateway listener is used to bridge the ESB-aware and ESB-unaware worlds. It is a specialized listener process that is designed to listen to a queue for ESB-unaware messages that have arrived through an external (ESB-unaware) end-point. The gateway listener receives the messages as they land in the queue. When a gateway listener "hears" incoming data arriving, it converts that data (the non-ESB

messages) into the **`org.jboss.soa.esb.message.Message`** format. This conversion happens in a variety of different ways, depending on the gateway type. Once the conversion has occurred, the gateway listener routes the data to its correct destination.

Report a bug

4.1.10. Adding a Gateway Listener to a Service

This code demonstrates how to add a JMS Gateway listener to a service.

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

trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd">
<providers>
    <jms-provider name="JBossMQ" connection-factory="ConnectionFactory">
        <jms-bus busid="shoeStoreJMSGateway">
            <jms-message-filter dest-type="QUEUE" dest-
name="queue/shoeStoreJMSGateway"/>
        </jms-bus>
    </jms-provider>
</providers>

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

invmScope="GLOBAL">
<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>
```

Observe that a bus <providers> section has been added to the configuration. Here you can configure the transport level details for endpoints. In this case, a <jms-provider> section has been added. Its purpose is to define a single <jms-bus> for the Shoe Store JMS queue. This bus is then referenced in the <jms-listener> defined on the Shoe Store Service. The Shoe Store is now "invocable" via the InVM and JMS Gateway endpoints. (The **`ServiceInvoker`** always prefers to use a service's local InVM endpoint if one is available.)

Report a bug

4.2. MESSAGES

4.2.1. ESB Message

An ESB message is a message that takes the form defined by the **`org.jboss.soa.esb.message`** interface. This standardized format consists of a header, body (payload) and attachments. All ESB-aware clients and services communicate with one another using messages.

Report a bug

4.2.2. Components of an ESB Message

An ESB message is made up of the following components:

Header

The header contains such information as the destination end-point reference, the sender end-point reference, and where the reply goes. This is all general message-level functional information.

Context

This is additional information that further explains the message; for example, transaction or security data, the identity of the ultimate receiver or HTTP-cookie information.

Body

The actual contents of the message.

Fault

Any error information associated with the message.

Attachment

Any attachments (additional files) associated with the message.

Properties

Any message-specific properties.(For example, the jbossesb.message.id property specifies a unique value for each message).

Here is a code representation:

```
   <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>
```

Report a bug

4.2.3. How Message Objects are Sent to the Queue

Overview

Overview

The JBoss Enterprise SOA Platform product uses a properties object that is populated with parameters to identify the presence of JNDI on the local server. It is then used as the parameter for a call to create a new Naming Context which is used to obtain the ConnectionFactory. The Connection Factory, in turn, creates the QueueConnection, which creates the QueueSession. This QueueSession creates a Sender object for the Queue. The Sender object is used to create an ObjectMessage for the sender and to then send it to the Queue.

Report a bug

4.2.4. Message Interface

Each message is an implementation of 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 ();

 public Message copy () throws Exception;
}
```

Report a bug

4.2.5. Message Header

The message's header contains the address to which the message is to be sent. It also contains routing information. The address format is based on the WS-Addressing standard from W3C.

Report a bug

4.2.6. Message Header Format

This is the format of a message header:

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

The message header's contents are contained in an instance of the **`org.jboss.soa.esb.addressing.Call`** class:

```
    public class Call
```

```
{
 public Call ();
 public Call (EPR epr);
 public Call (Call copy);
 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 copy();
 public void setAction (URI uri);
 public URI getAction () throws URISyntaxException;
 public final boolean empty();
 public void setMessageID (URI uri);
 public URI getMessageID () throws URISyntaxException;
 public String toString();
 public String stringForum();
 public boolean valid();
 public void copy (Call from);
}
```

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

Table 4.1. `org.jboss.soa.esb.addressing.Call` Properties

Property	Type	Required	Description
To	EPR	Yes	The message recipient's address.
From	EPR	No	The endpoint from which the message originated.
Reply To	EPR	No	An endpoint reference that identifies the intended receiver for replies to this message. If a reply is expected, a message must contain a [ReplyTo]. The sender must use the contents of the [ReplyTo] to formulate the reply message. If the [ReplyTo] is absent, the contents of the [From] may be used to formulate a message to the source. This property may be absent if the message has no meaningful reply. If this property is present, the [MessageID] property is required.

Property	Type	Required	Description
FaultTo	EPR	No	An endpoint reference that identifies the intended receiver for faults related to this message. When formulating a fault message the sender must use the contents of the [FaultTo] of the message being replied to to formulate the fault message. If the [FaultTo] is absent, the sender may use the contents of the [ReplyTo] to formulate the fault message. If both the [FaultTo] and [ReplyTo] are absent, the sender may use the contents of the [From] to formulate the fault message. This property may be absent if the sender cannot receive fault messages (for example, it is a one-way application message). If this property is present, the [MessageID] property is required.
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 in time and space. No two messages with a distinct application intent may share a [MessageID] property. A message may be retransmitted for any purpose including communications failure and may use the same [MessageID] property. The value of this property is an opaque URI whose interpretation beyond equivalence is not defined. If a reply is expected, this property must be present.

Always consider the role of the header when developing and using services. For example, if you need a synchronous interaction pattern based upon request and response, be sure to set the ReplyTo field or a default endpoint reference will be used. Even with "request/response", the response need not go back to the original sender. Likewise, when sending one-way (no response) messages, do not set the ReplyTo field because it will be ignored.

WARNING

Users should not rely on the internal formats of EPR directly because they are specific to implementations of the API. There is no guarantee the format will remain the same for them.

NOTE

The message header is formed in conjunction with the message. It is immutable once it has been transmitted between endpoints. Although the interfaces allow the recipient to modify the individual values, the **JBoss Enterprise SOA Platform** will ignore such modifications. (In future releases it is likely that such modifications will also be disallowed by the application programming interface, in order to improve clarity.) These rules can be found in the WS-Addressing standards.

Report a bug

4.2.7. The To Field

You can specify a value for the message header's To field when you are about to send the message. This field should contain the address of the message recipient.

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.

Report a bug

4.2.8. Message Context

The message context contains session-related information. This can include transaction and security data. You can also create your own user-enhanced contexts.

Report a bug

4.2.9. Message Body

The message's body contains the "payload" of the message.

WARNING

Be extremely careful when sending serialized objects to and from the message body. Just because something can be serialized doesn't mean it will be meaningful at the receiving end. This happens with database connections.

Report a bug

4.2.10. Message Payload

The message payload is a combination of the message's body, its attachments and its properties. The payload may consist of a list of arbitrary objects. How these objects are serialized when the message is sent is determined by the object's type.

Report a bug

4.2.11. Serialize

To serialize an object is to convert it to a data object.

Report a bug

4.2.12. Message Body Format

This is what a message body looks like:

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

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

IMPORTANT

The message body's byte array component is deprecated. To continue using a byte array in conjunction with other data stored in the body, use the **add** option and give it a unique name. If your clients and services want a location for a byte array, you can use the one that the JBoss ESB itself uses: ByteBody.BYTES_LOCATION.

WARNING

Use the default named object (DEFAULT_LOCATION) with care so that multiple services and actions do not overwrite each other's data.

Report a bug

4.2.13. Message Fault

A message fault is a problem with the information in the body of a message.

Report a bug

4.2.14. Fault Message Format

This is the format of a message indicating a fault:

```
  public interface Fault
{
 public URI getCode ();
```

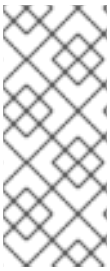
```
  public void setCode (URI code);

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

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

Report a bug

4.2.15. Message Properties

Use these message properties to add extra meta-data to the message:

```
   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();
}
```



NOTE

The properties which use **`java.util.Properties`** have not been implemented within the **JBoss Enterprise Service Bus**. This is because they would restrict the types of client and service that could be used. Web Services stacks do not implement them for the same reasons. To overcome this limitation, embed **`java.util.Properties`** within the current abstraction.

Report a bug

4.2.16. Message Attachment

Attachments are files bundled with the message. They do not appear in the message body. They can include images and documents in binary formats and compressed files.

There are several reasons to use attachments. They are generally employed to provide the message with a more logical structure. They also provide a way to improve performance when processing large messages as they can be streamed between endpoints.

Report a bug

4.2.17. Message Attachment Interface

Use this interface to add attachments to messages:

```
  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 getUnnamedCount();
 public int getNamedCount();
}
```

NOTE

At present JBossESB does not support specifying other encoding mechanisms for the Message or attachment streaming.Therefore, currently attachments are treated in the same way as named objects within the Body.

Report a bug

4.2.18. Choosing the Right Method

Users may find themselves overwhelmed when they have to choose between attachments, properties and named objects when deciding where to put the payload. However, the decision can be simplified:

- The developer defines the contract that the clients will use in order to interact with their service. As part of that contract, both functional and non-functional aspects of the service will be specified; for example, that it is an airline reservation service (functional) and that it is transactional in nature (non-functional).

  The developer will also define the operations (messages) that the service can understand. The format (such as Java Serialized Message or XML) is defined as part of the message definition. (In the example case, they will be the transaction context, seat number, customer name and so forth.) When you define the content, you can specify where in the message the service can find the payload. (This can be in the form of attachments or specific named objects, or even the default named object if one so wishes.) It is entirely up to the service developer to determine. The only restriction is that objects and attachments must have a globally-unique name, otherwise one service or action may inadvertently pick up a partial payload meant for another (if the same message body is being forwarded along on multiple "hops").

- Users can obtain the service's contract definition (either through either the UDDI registry or via an out-of-band communication) which defines where in the message the payload must be placed. Information put in other locations will almost certainly be ignored, resulting in the

incorrect operation of the service.

Report a bug

4.2.19. Advice on Adding Data to the Body of a Message

By default, every component (including actions, listeners, gateways, routers and notifiers) set data on the message via the **`MessagePayloadProxy`**. This class handles the default settings but it also allows you to over-ride the defaults. .

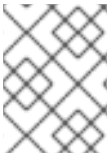
Alternatively, you can also override the "get" and "set" settings by configuring the following properties:

- get-payload-location: this is the location from which to obtain the message payload.
- set-payload-location: this where you set the location for the message payload to go.

You can use the add method to attach more complex content to the payload, which supports named objects. Using <name, Object> pairs allows for a finer granularity of data access. Arbitrary objects can be added to the payload; they do not need to be Java serializable. However, if you add non-serializable objects, you must provide the JBoss Enterprise SOA Platform with the ability to marshal and unmarshal the message when it flows across the network.

NOTE

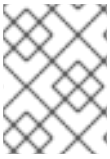
If no name has been supplied for "setting" or "getting," then that which was defined against the DEFAULT_LOCATION setting will be utilised.



NOTE

Be careful when using serialized Java objects in messages as they constrain the service implementation.

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



NOTE

In the current release of the JBoss Enterprise SOA Platform, you can only attach Java serialized objects. This restriction may be removed in a subsequent release.

Report a bug

4.2.20. Configure for Legacy Message Payload Exchange

There was no default message payload exchange pattern in place in version 4.2 of the JBoss Enterprise SOA Platform. Use this method to provide legacy support.

Procedure 4.1. Task

1. Open the **`jbossesb-properties.xml`** file in a text editor: **`vi SOA_ROOT/jboss-soa-p-5/jboss-as/server/PROFILE/deploy/jbossesb.sar/jbossesb-properties.xml`**
2. Scroll down to the section entitled "Core".
3. Set the use.legacy.message.payload.exchange.patterns property to “true”.
4. Save the file and exit.

Report a bug

4.2.21. Extensions to the Message Body

Extensions Types

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

the content of the Body is an arbitrary string, and can be manipulated via 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 via 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 via the setMap and other methods.

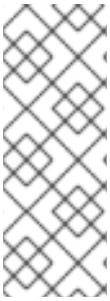
`org.jboss.soa.esb.message.body.content.BytesBody`

the content of the body is a byte stream that contains arbitrary Java data-types. It can be manipulated using the various setter and getter methods for the data-types. Once created, the BytesMessage should be placed into either a read-only or write-only mode, depending upon how it needs to be manipulated. It is possible to change between these modes (using readMode and writeMode), but each time the mode is changed the buffer pointer will be reset. In order to ensure that all of the updates have been pushed into the body, it is necessary to call flush when finished.

You can create messages that have body implementations based on one of these specific interfaces through the XMLMessageFactory or SerializedMessageFactory classes.

There is a **`create`** method associated with each of the various body types. An example is **`createTextBody`**. Use this to create and initialize a message of that specific type. Once created, manipulate the message directly by editing the raw body or by using its interface's methods. The body's structure is maintained even after transmission so that it can be manipulated by the message recipient using the methods of the interface that created it.

The XMLMessageFactory and SerializedMessageFactory are more convenient ways in which to work with Messages than the MessageFactory and associated classes.



NOTE

These extensions to the base body interface are provided in a complimentary manner to the original body. As such they can be used in conjunction with existing clients and services. Message consumers can remain unaware of these new types if necessary because the underlying data structure within the message remains unchanged. It is important to realise that these extensions do not store their data in the default location. Data should be retrieved using the corresponding getters on the extension instance.

Report a bug

4.2.22. End-Point Reference

An end-point reference (EPR) contains the address information and technical specifications for a service. Indeed, all ESB-aware services are identified using end-point references. It is through these references that services are contacted. The are stored in the registry. Services add their end-point references to the registry when they are launched and should automatically remove them when they terminate. A service may have multiple end-point references. End-point references are also known as binding templates.

End-point references can contain links to the tModels designating the interface specifications for a particular service.

Report a bug

4.2.23. Logical EPR

A Logical EPR is an end-point reference that specifies the name and category of a service. It contains no physical address information.

Report a bug

4.2.24. Logical EPR Use

It is best to use the **`Logical EPR`** because it makes no assumptions about the end-point reference user (which is usually, but not necessarily always, the Enterprise Service Bus itself.) The **`LogicalEPR`**'s client can use the service name and category details that have been provided to look up the physical endpoint for that service. The client will do so at the time it intends to use the service. The client will also be able to select a physical end-point type to suit the situation.

Report a bug

4.2.25. FaultTo Field

The FaultTo field contains the address to which messages with errors are sent. If you do not set it, then the ReplyTo field or, failing that, the From field will be used. If no valid EPR is obtained as a result of checking all of these fields, then the error will be output to the console. If you do not wish to be informed of errors, (which may be the case when you are sending a one-way message), set the FaultTo field to pointing to the DeadLetter Queue end-point reference.

Report a bug

4.2.26. Dead Letter Queue

The Dead Letter Queue is an end-point reference to which you can send failed messages. Any messages sent here will be saved for reprocessing later on.

Report a bug

4.2.27. ReplyTo Field

The ReplyTo field is an optional field in the message header. It contains the message's reply address (or end-point reference). Applications should be designed to populate this field if necessary. (Because the **JBoss Enterprise SOA Platform**'s recommended interaction pattern is based on a one-way message exchange, messages may not receive responses automatically: it is application-dependent as to whether or not a sender expects a response.)

If a response is required and the ReplyTo field has not been set, the **JBoss Enterprise SOA Platform** can automatically populate it based on default values for each type of transport. (Note that to use some of these **`ReplyTo`** defaults requires system administrators to specifically configure the **JBoss Enterprise Service Bus'** behaviour.)

Report a bug

4.2.28. Table of ReplyTo Field Settings

Table 4.2. Default ReplyTo by transport

Transport	ReplyTo
JMS	a queue with a name based on the one used to deliver the original request: <request queue name>_reply.
JDBC	A table in the same database with a name based on the one used to deliver the original request: <request table name>_reply_table. The new table needs the same columns as the request table.
files	No administration changes are required for either local or remote files. Responses are saved in the same directory as the request but with a unique suffix to ensure that only the original sender will pick up the response.

Report a bug

4.2.29. Advice on Serializing Messages

Although each enterprise service bus component treats every message as a collection of Java objects, you will find it is often necessary to serialize these messages. Do so when:

- the data is to be stored

- the message is to be sent between different ESB processes
- you are debugging.

The **JBoss Enterprise SOA Platform** does not impose a single, specific format for message serialization because requirements will be influenced by the unique characteristics of each deployment. You can obtain the various implementation of the **`org.jboss.soa.esb.message.Message`** interface from the **`org.jboss.soa.esb.message.format.MessageFactory`** class:

```
  public abstract class MessageFactory
{
 public abstract Message getMessage ();
 public abstract Message getMessage (URI type);
 public abstract void reset();
 public static MessageFactory getInstance ();
}
```

Uniform resource indicators uniquely identify message serialization implementations. Either specify the implementation when creating a new instance or use the pre-configured default.

There are two serialized message formats, JBOSS_XML and JBOSS_SERIALIZED.

MessageType.JBOSS_XML

This implementation uses an XML representation of the message. The schema for the message is defined in the **`schemas/message.xsd`** file. Arbitrary objects may be added to the message: in other words, they do not have to be serializable. Therefore, it may be necessary to provide a mechanism to marshal and un-marshal such objects to and from XML when the message needs to be serialized. Do this through the **`org.jboss.soa.esb.message.format.xml.marshal.MarshalUnmarshalPlugin`**:

```
          public interface MarshalUnmarshalPlugin
{
 public static final String MARSHAL_UNMARSHAL_PLUGIN =
     "org.jboss.soa.esb.message.format.xml.plugin";
 public boolean canPack(final Object value);
 public boolean marshal (Element doc, Object param)
          throws MarshalException;

 public Object unmarshal (Element doc) throws UnmarshalException;

 public URI type ();
}
```

MessageType.JAVA_SERIALIZED

This is the default. This implementation requires that every component of the message be serializable. It also requires that the message recipients have sufficient information (via the Java classes) to be able to de-serialize it. Its URI is urn:jboss/esb/message/type/JAVA_SERIALIZED.

It uses an XML representation of the message. The message's schema is defined in the **`schemas/message.xsd`** file. Its URI is urn:jboss/esb/message/type/JBOSS_XML.

It also requires that all of the contents be Java-serializable. Any attempt to add a non-serializable object to the message will result in an **`IllegalParameterException`** error.

Its URI is **urn:jboss/esb/message/type/JAVA_SERIALIZED**.

IMPORTANT

Be wary about using the JBOSS_SERIALIZED version of the message format because it can tie your applications to specific service implementations.

You can provide other message implementations at runtime through the **org.jboss.soa.esb.message.format.MessagePlugin**:

```
    public interface MessagePlugin
{
 public static final String MESSAGE_PLUGIN =
       "org.jboss.soa.esb.message.format.plugin";
 public Object createBodyType(Message msg, String type);
 public Message getMessage ();
 public URI getType ();
}
```

Each plug-in must uniquely identify the type of message implementation it provides (via getMessage), using the getType method. Plug-in implementations must be identified to the system via the **jbossesb-properties.xml** file. (Use the property names that have the org.jboss.soa.esb.message.format.plugin extension.)

Report a bug

4.2.30. Change the Default Message Type

Procedure 4.2. Task

1. Set the org.jboss.soa.esb.message.default.uri property to the name of the desired URI. (The default is JBOSS_XML.)
2. Save the file and exit.

Report a bug

4.2.31. Register a Marshaling Plug-In

Procedure 4.3. Task

1. Open the **jbossesb-properties.xml** file in a text editor: **vi SOA_ROOT/jboss-soa-p-5/jboss-as/server/PROFILE/deploy/jbossesb.sar/jbossesb-properties.xml**
2. Add the name of the plug-in. (It must start with this prefix: MARSHAL_UNMARSHAL_PLUGIN.)
3. Save the file and exit.

Report a bug

PART III. DEVELOPING

CHAPTER 5. BUILDING AND USING SERVICES

5.1. MESSAGE LISTENER CONFIGURATION PROPERTIES

Each **listener** configuration needs to supply information for:

- the **registry** (see the **service-category**, **service-name**, **service-description** and **EPR-description** tag names.) If you set the optional **remove-old-service** tag name to **true**, the Enterprise Service Bus will remove any pre-existing service entry from the **registry** and then add this new instance. Always use this functionality with care as the entire service will be removed, including every end-point reference.
- the instantiation of the **listener** class (see the **listenerClass** tag name).
- the endpoint reference that the **listener** will service. This is transport-specific. The following example corresponds to a Java Message Service endpoint reference (see the **connection-factory**, **destination-type**, **destination-name**, **jndi-type**, **jndi-URL** and **message-selector** tag names).
- the **action pipeline**. This needs one or more <action> elements, each of which must contain the **class** tag name. These will determine which **action** class will be instantiated for that link in the **chain**.

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

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

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

```
    </listeners>

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

     </action>
   </actions>
  </service>
 </services>
</jbossesb>
```

This example configuration instantiates a **`listener`** object (the jms-listener tag), which will wait for those incoming ESB messages that are serialized within an interface. It then delivers each incoming message to an **`action pipeline`** consisting of two steps (<action> elements):

1. action1: **`MyJMSListenerAction`** (an example follows).

2. notificationAction: an **`org.jboss.soa.esb.actions.SystemPrintln`** class.

The reason there are two listeners is that the gateway listener is the ESB-unaware listener and its role is to encapsulate the JMS message in the ESB message used throughout the enterprise service bus.

Report a bug

5.2. CHARACTERISTICS OF FILESYSTEM GATEWAY LISTENERS

A *filesystem gateway listener* can be called when a file is modified to process the changes universally. It creates notifications of any edits that are made.

The *AbstractFileGateway* class is the base for these gateways. It works with the *AbstractScheduledManagedLifecycle* class to derive information from events. It uses a single thread to perform file operations and send ESB messages to the bus.

Inserting the max-millis-for-response property results in the ESB messages being sent synchronously. If there is an exception, the gateway will move the file to an error directory. (For asynchronous processing, withdraw this property. There will be no notifications if an exception is encountered- instead, it will be processed in a different thread.)

Report a bug

5.3. PIPELINE INTERCEPTOR

The **`org.jboss.soa.esb.listeners.message.PipelineInterceptor`** is an interface that you can use to configure an interceptor which will be passed to the service configuration and the message at the start of the pipeline, at the end of the pipeline, at service instantiation, or at the point at which the pipeline fails due to an exception.

Report a bug

5.4. WORKING WITH PIPELINE INTERCEPTORS

The Pipeline Interceptor has one method:

1. **`public void processMessage (Message msg, ConfigTree config)`** which is called at the interception points defined in the **`jbossesb-properties.xml`** configuration file.

Define your pipeline interceptors in the "interceptors" section of the **`jbossesb-properties.xml`** file (located in the **`jbossesb.sar`** archive) using the following properties:

- org.jboss.soa.esb.pipeline.failure.interceptors
- org.jboss.soa.esb.pipeline.instantiate.interceptors
- org.jboss.soa.esb.pipeline.start.interceptors
- org.jboss.soa.esb.pipeline.end.interceptors

NOTE

You will need to place any changes to your **`jbossesb-properties.xml`** file on each ESB instance that is deployed in your environment. This will ensure that every instance can process the same meta-data.

The JBoss Enterprise SOA Platform only comes with a **`org.jboss.soa.esb.listeners.message.GenericPipelineInterceptor`**, which prints the message and demonstrates the general concept. It is up to you to provide the concrete implementation to use.

Report a bug

5.5. ROUTERS

Routers are the components that send either the message itself or its payload to an end-point listener. There are two types of routers: content-based routers and static-based routers.

No further processing of the action pipeline will occur after the router action even if there are further actions in the configuration. (If this sort of splitting is required you should use the StaticWiretap action.)

Report a bug

5.6. ROUTER CONFIGURATION

Some routers support the 'unwrap' property. If this property is true then the ESB Message payload will be extracted and only the payload will be sent to the ESB-unaware endpoint. Setting 'unwrap' to false will pass the message as is and the receiving end-point must be ESB-aware so that it can handle the message.

Report a bug

5.7. CONTENT-BASED ROUTER

Content-based routers send messages that do not have destination addresses to their correct end-points. Content-based routing works by applying a set of rules (which can be defined within XPath or Drools notation) to the body of the message. These rules ascertain which parties are interested in the message. This means the sending application does not have to supply a destination address.

A typical use case is to serve priority messages in a high priority queue. The advantage here is that the routing rules can be changed on-the-fly while the service runs if it is configured in that way. (However, this has significant performance drawbacks.)

Other situations in which a content-based router might be useful include when the original destination no longer exists, the service has moved or the application simply wants to have more control over where messages go based on its content of factors such as the time of day.

Report a bug

5.8. STATIC-BASED ROUTER

A static-based router helps to coordinate information across your network. It transfers information between servers and tells them where to send their messages. You can program it to take certain routes if you feel the default is inefficient. You can only use it to route to other services.

Report a bug

5.9. NOTIFIER

Notifiers send information, such as success and error messages, to ESB-unaware endpoints. They are not to be used with ESB aware end-points. Notifiers can only transport simple chunks of data: namely either a byte[] or a String (obtained by calling toString() on the payload).

Report a bug

5.10. SERVICEINVOKER

The ServiceInvoker (**`org.jboss.soa.esb.client.ServiceInvoker`**) manages the delivery of messages to the specified Services. It also manages the loading of end-point references and the selection of couriers, thereby providing a unified interface for message delivery.

The ServiceInvoker was introduced to help simplify the development effort as it hides much in the way of

the lower-level details and works opaquely with the stateless service fail- over mechanisms. As such, ServiceInvoker is the recommended client-side interface for using services within the JBoss Enterprise SOA Platform.

You can create an instance of the ServiceInvoker for each service with which the client interacts. Once created, an instance examines the registry to determine the primary end-point reference and, in the case of fail-overs, any alternative end-point references.

Report a bug

5.11. DEVELOPING WITH THE SERVICEINVOKER

This is the ServiceInvoker's code:

```
  public class ServiceInvoker
    {
        public ServiceInvoker(Service service) throws
MessageDeliverException;
        public ServiceInvoker(String serviceCategory, String serviceName)
throws MessageDeliverException;
    public ServiceInvoker(Service service, List<PortReference.Extension>
extensions);
        public Message deliverSync(Message message, long timeoutMillis)
throws MessageDeliverException, RegistryException, FaultMessageException;
        public void deliverAsync(Message message) throws
MessageDeliverException;
   public Service getService();
   public String getServiceCategory();
    }
```

Once you have created the instance, the client determines 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. The ResponseTimeoutException is derived from MessageDeliverException.

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.

One of the features of the ServiceInvoker is that of load balancing invocations in situations where there are multiple endpoints available for the target service. The ServiceInvoker supports a number of load balancing strategies as part of this feature.

When using the ServiceInvoker, preference is always given to invoking a service over its InVM transport if one is available. Other load balancing strategies are only be applied in the absence of an InVM endpoint for the target Service.

Report a bug

5.12. REGISTRYEXCEPTION

A RegistryException is thrown if a client cannot contact the registry or if it cannot find a service. It maybe that a service has failed or is under too heavy a load and cannot respond in time. If so, a timeout value will be returned. Try again to see if it is a temporary problem.

Report a bug

5.13. FAULTMESSAGEEXCEPTION

A FaultMessageException is thrown for every failure to communicate with the service not accounted for by RegistryException.

Report a bug

5.14. MESSAGEDELIVEREXCEPTION

A MessageDeliverException is thrown if there is a problem when you are using the deliverAsync method. You will find that the Actual exceptions is embedded within this catch-all.

Report a bug

5.15. JAVA MESSAGE SERVICE

A Java Message Service (JMS) is a Java API for sending messages between two clients. It allows the different components of a distributed application to communicate with each other and thereby allows them to be loosely coupled and asynchronous. There are many different Java Message Service providers available. Red Hat recommends using HornetQ.

Report a bug

5.16. JMS TRANSACTED SESSION

In a JMS Transacted Session, a message is placed on a queue but remains undelivered until the enclosing transaction has been committed. It is then collected by the receiver in the scope of a separate transaction. Unfortunately for synchronous request/response interactions this can result in a time-out waiting for the response since the sender blocks waiting for the response before it can terminate the delivery transaction.

Report a bug

5.17. INCOMPATIBLETRANSACTIONSCOPEEXCEPTION

An IncompatibleTransactionScopeException catches blocks arising from problems with JMS transacted sessions and throws back an error report.

Report a bug

5.18. INVM

5.18.1. InVM Transport

The InVM ("intra-virtual machine") Transport provides communication between services running on the same JVM.

Report a bug

5.18.2. InVM Limitations

InVM achieves its performance benefits by optimizing the internal data structures that are used to facilitate inter-service communication. For example, the queue used to store messages is not persistent (durable) which means that messages may be lost in the event of failures.

Furthermore if a service is shut down before the queue is emptied, those messages will not be delivered. Because JBossESB allows services to be invoked across multiple different transports concurrently you should be able to design your services such that you can achieve high performance and reliability by the suitable choice of transport for specific Message types.

By default, the InVM transport passes messages “by reference”. In some situations, this can cause data integrity issues, not to mention class cast issues where messages are being exchanged across ClassLoader boundaries.

Message passing “by value” (and so avoid issues such as those listed above) can be turned on by setting the “inVMPassByValue” property on the service in question to “true”:

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

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

Report a bug

5.18.3. Developing with the InVM

Because the JBoss Enterprise SOA Platform allows services to be invoked concurrently across multiple different transports, it is straightforward to design services in such a way that they can achieve high performance and reliability. Do this by making a suitable choice of transport for each type of message.

Earlier versions of the product 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 <listeners> configuration and still be invokable from within their own virtual machines:

```
  <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>
```

This makes service configuration a little more straightforward.

Control the InVM service's invocation scope through the <service> element's “invmScope” attribute. Two scopes are supported:

1. GLOBAL: (Default) The service can be invokable over the InVM transport from within the same Classloader scope.

2. NONE: The service cannot be invoked over the InVM transport.

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

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

NOTE

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.

WARNING

The InVM transport in the JBoss Enterprise SOA Platform is not transactional and the message queue is held only in volatile memory. This means that the message queue for this transport will be lost in the case of system failure or shut down.

NOTE

You may not be able to achieve all of the ACID semantics, particularly when used in conjunction with other transactional resources such as databases, because of the volatility aspect of the InVM queue but the performance benefits of InVM should outweigh this downside in the majority of cases. In the situations where full ACID semantics are required, Red Hat recommends that you use one of the other transactional transports, such as Java Message Service 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.

When a message is placed back on the queue by the transaction manager, it may not go back into the same location. This is a deliberate choice in order to maximize performance. If your application needs specific ordering of messages then you should consider a different transport or group related messages into a single “wrapper” message.

Report a bug

5.18.4. Set an InVM Scope for an Individual Service

Procedure 5.1. Task

1. Open the file in a text editor.
2. Set the service element's invmScope attribute:

```
      <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>
```

3. Save the file and exit.

Report a bug

5.18.5. Set the Default InVM Scope for a Deployment

Procedure 5.2. Task

1. Open the **jbossesb-properties.xml** in your text editor: **vi jbossesb-properties.xml**
2. Set the “core:jboss.esb.invm.scope.default” configuration property. (If left defined, the default scope is “GLOBAL”.)
3. Save the file and exit.

Report a bug

5.18.6. Change the Number of Listener Threads Associated with an InVM Transport

Procedure 5.3. Task

1. Open the file in a text editor.
2. Edit the file like this:

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

3. Save the file and exit.

Report a bug

5.18.7. Lock-Step Delivery

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 time-out 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.

Report a bug

5.18.8. Lock-Step Delivery Settings

Configure Lock-Step Delivery via the <service>'s <property> settings:

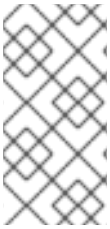
inVMLockStep

This is a Boolean value. It controls whether or not Lock-Step Delivery is enabled.

inVMLockStepTimeout

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

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

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



NOTE

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

Report a bug

5.19. LOAD BALANCING

5.19.1. Load Balancing

Load balancing is a computer networking methodology to distribute workload across multiple computers or a computer cluster, network links, central processing units, disk drives, or other resources, to achieve optimal resource utilization, maximize throughput, minimize response time, and avoid overload. Using multiple components with load balancing, instead of a single component, also increases reliability through redundancy.

Report a bug

5.19.2. Configure a Load-Balancing Policy

Procedure 5.4. Task

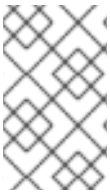
1. Open the global configuration file in a text editor: **`vi SOA_ROOT/jboss-as/server/PROFILE/deployers/esb.deployers/jbossesb-properties.xml`**.
2. Scroll down to the org.jboss.soa.esb.loadbalancer.policy property. Set it with the policy you wish to use.
3. Save the file and exit.

Report a bug

5.19.3. Load Balancing Policies

Table 5.1. Load balancing Policies Available

Policy Name	Description
first available	If a healthy service binding is found it will be used until it dies. The next end-point reference in the list will then be used. There is no load balancing between the two service instances with this policy.
round robin	A standard load-balancing policy whereby each end-point reference is utilised in list order.
random robin	This is like the round robin, but the selection is randomized.



NOTE

The end-point reference list used by the policy may become smaller over time as "dead" EPRs are removed. When the list is exhausted or the time-to-live of the list cache is exceeded, the ServiceInvoker will obtain a fresh list of EPRs from the Registry.

Report a bug

5.20. SERVICE CONTRACT DEFINITION

5.20.1. Service Contract

A service contract is a set of XML schemas that define the incoming request, outgoing response and fault message details. The schemas representing the request and response messages are used to define the format of the main body section of the message and can also enforce data validation of that content.

Report a bug

5.20.2. Declaring Service Contract Schemas

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

Table 5.2. Service Contract Attributes

Name	Description	Type
inXsd	The resource containing the schema for the request message, representing a single element.	xsd:string

Name	Description	Type
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

Report a bug

5.20.3. 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. Validation is disabled by default. Enable validation by setting the <actions> element's 'validate' attribute to 'true'.

Report a bug

5.21. EXPOSING ESB SERVICES VIA WEB SERVICE END-POINTS

5.21.1. Exposing ESB Services via Web Service End-points

When you declare your service contract schemas, the ESB service will automatically be exposed through a web service endpoint. (The contract for this web service end-point can be located through the Contract Web Application.) To modify this functionality, change the value for the 'webservice' attribute to one of the following:

Table 5.3. Web Service Attributes

Name	Description
false	No web service endpoint will be published.
true	A web service endpoint will be published (default).

By default the webservice endpoint does not support WS-Addressing but this can be enabled through use of the 'addressing' attribute.

Table 5.4. WS-Addressing Values

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

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

Table 5.5. WS-Addressing Properties

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

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

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

The following example illustrates the declaration of a service which wishes to validate the request/response messages and expose the service through a webservice endpoint. In addition the service expects the request to be provided in the named body location 'REQUEST' and will return its response in the named body location 'RESPONSE'.

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

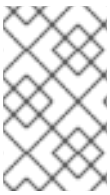
Report a bug

CHAPTER 6. OTHER COMPONENTS

6.1. MESSAGE STORE

The message store is a database persistence mechanism that has been designed to allow you to do audit-tracking. The message store reads and writes messages upon request. Each message must have a unique identification number. As with other ESB services, the message store is "pluggable", which means that you can "plug in" your own persistence mechanism should you desire to do so, though a default database persistence mechanism is supplied.

In the event of a system failure, the message store is also used as a holding place for messages that need to be re-delivered.



NOTE

If something other than a database is required, such as a file persistence mechanism, you can write your own service and then override the default behaviour with a configuration change.

Report a bug

6.2. SMOOKS

Smooks is a fragment-based data transformation and analysis tool. It is a general purpose processing tool capable of interpreting fragments of a message. It uses visitor logic to accomplish this. It allows you implement your transformation logic in XSLT or Java and provides a management framework through which you can centrally manage the transformation logic for your message-set.

Report a bug

6.3. VISITOR LOGIC IN SMOOKS

Smooks uses *visitor logic*. A "visitor" is Java code that performs a specific action on a specific fragment of a message. This enables Smooks to perform actions on message fragments.

Report a bug

6.4. DATA TRANSFORMATION

Introduction

Clients and services usually communicate using the same vocabulary. However, there are times when this will not be the case and you will require an "on-the-fly" transformation mechanism to translate from one format to another. It is unrealistic to assume that a single data format will suit every business object, particularly in a large-scale or long-running deployment. Therefore, a data transformation mechanism has been provided.

Report a bug

6.5. CONTENT-BASED ROUTER

Content-based routers send messages that do not have destination addresses to their correct end-points. Content-based routing works by applying a set of rules (which can be defined within XPath or Drools notation) to the body of the message. These rules ascertain which parties are interested in the message. This means the sending application does not have to supply a destination address.

A typical use case is to serve priority messages in a high priority queue. The advantage here is that the routing rules can be changed on-the-fly while the service runs if it is configured in that way. (However, this has significant performance drawbacks.)

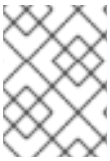
Other situations in which a content-based router might be useful include when the original destination no longer exists, the service has moved or the application simply wants to have more control over where messages go based on its content of factors such as the time of day.

Report a bug

6.6. CONTENT BASED ROUTING USING THE JBOSS RULES ENGINE

JBoss Rules is the rule provider "engine" for the content-based router. The Enterprise Service Bus integrates with this engine through three different routing **`action classes`**, these being:

- a routing rule set, written in the **`JBoss Rules`** engine's DRL language (alternatively, you can use the DSL language if you prefer it);
- the message content. This is the data that goes into the JBoss Rules engine (it comes in either XML format or as objects embedded in the message);
- the destination (which is derived from the resultant information coming out of the engine).



NOTE

When a message is sent to the **`content-based router`**, a rule-set will evaluate its content and return a set of service destinations.

- **org.jboss.soa.esb.actions.ContentBasedRouter**: This action class implements the *content-based routing* pattern. It routes a message to one or more destination services, based on the message content and the rule set against which it is evaluating that content. The content-based router throws an exception when no destinations are matched for a given rule set or message combination. This action will terminate any further pipeline processing, so always position it last in your pipeline.
- **org.jboss.soa.esb.actions.ContentBasedWiretap**: This implements the *WireTap* pattern. The **`WireTap`** is an enterprise integration pattern that sends a copy of the message to a control channel. The **`WireTap`** is identical in functionality to the standard content-based router, however it does not terminate the pipeline. It is this latter characteristic which makes it suitable to be used as a wire-tap, hence its name. For more information, see http://www.eaipatterns.com/WireTap.html.
- **org.jboss.soa.esb.actions.MessageFilter**: This implements the *message filter* pattern. The message filter pattern is used in cases where messages can simply be dropped if certain content requirements are not met. In other words, it functions identically to the content-based

router except that it does not throw an exception if the rule set does not match any destinations, it simply filters the message out. For more information, see http://www.eaipatterns.com/Filter.html.

Report a bug

6.7. SERVICE REGISTRY

A service registry is a central database that stores information about services, notably their end-point references. The default service registry for the JBoss Enterprise SOA Platform is jUDDI (Java Universal Description, Discovery and Integration). Most service registries are designed to adhere to the Universal Description, Discovery and Integration (UDDI) specifications.

From a business analyst's perspective, the registry is similar to an Internet search engine, albeit one designed to find web services instead of web pages. From a developer's perspective, the registry is used to discover and publish services that match various criteria.

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

Report a bug

6.8. JUDDI REGISTRY

The jUDDI (Java Universal Description, Discovery and Integration) Registry is a core component of the JBoss Enterprise SOA Platform. It is the product's default service registry and comes included as part of the product. In it are stored the addresses (end-point references) of all the services connected to the Enterprise Service Bus. It was implemented in JAXR and conforms to the UDDI specifications.

Report a bug

6.9. JUDDI AND THE JBOSS ENTERPRISE SOA PLATFORM

jUDDI and the JBoss Enterprise SOA Platform

The JBoss Enterprise SOA Platform product includes a pre-configured installation of a jUDDI registry. You can use a specific API to access this registry through your custom client. However, any custom client that you build will not covered by your SOA Platform support agreement. You can access the full set of jUDDI examples, documentation and APIs from: http://juddi.apache.org/.

Report a bug

CHAPTER 7. TUTORIAL ON DEVELOPING MESSAGES

7.1. OVERVIEW

Introduction

Conceptually, the message is a critical aspect of the SOA development approach. Messages contain the application-specific data 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 entails the best practices to use for this component.

Firstly, consider the following flight reservation service example. 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 the passenger will move to).

When developing a service, you will mostly use technologies such as *Enterprise Java Beans* (EJB3) and *Hibernate* to implement the business logic. This example does not teach the reader how to implement this logic. Instead, it focuses on the service itself.

The service's role is to "plug" the logic into the bus. To configure it to do so, determine how the service is exposed to the bus (that is, the 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

Currently, there is no formal specification for this contract within the ESB. It is something the developer defines and communicates to clients irrespective of the enterprise service bus. This will be rectified in a subsequent release.

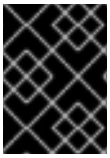
Report a bug

7.2. MESSAGE STRUCTURE

From the perspective of a service, the most important component of a message is the body. This is because it is used to convey information specific to the business logic. Both client and service must understand each other to interact. This understanding involves an agreement on the mode of transport (such as Java Message Service or HTTP) and the dialect to be used (the format in which the data is to appear in the message).

To take the simple case of a client sending a message directly to the example flight reservation service,

you 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 in the body as a string (**`reserve`**, **`query`**, **`upgrade`**) at the location called **`org.example.flight.opcode`**. Any other string value (or the absence of a value) will result in the message being considered illegal.



IMPORTANT

Ensure that every value within a message is given a unique name. This prevents clashes with other clients and services.

The Message Body is the primary way in which data should be exchanged between clients and services. It is flexible enough to contain any number of arbitrary data types. (The other parameters required to execute the business logic associated with each operation should 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 7.1. 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

All of these operations return information to the client by encapsulating it within a message. Response messages go through the same processes in order for their own formats to be determined. For simplicity, the response messages will be left out of this example.

Build the service using one or more actions. These pre-process the incoming message and transform its contents, before passing it on to the action which is responsible for the main business logic. Each of these actions can be written in isolation (possibly by different groups within the same organisation or even by completely different organisations). Always make sure that each action has a unique view of the message data upon which it acts. If this is not the case, it is possible that chained actions may either overwrite or otherwise interfere with each other.

Report a bug

7.3. DEVELOPING THE SERVICE

At this point, you have learned enough to be able to construct the service. For simplicity, it will be assumed that the business logic is encapsulated within the following "pseudo-object:"

```
    class AirlineReservationSystem
{
```

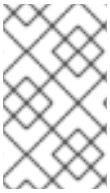
```
 public void reserveSeat (...);
 public void querySeat (...);
 public void upgradeSeat (...);
}
```

NOTE

Develop business logic using Plain Old Java Objects, Enterprise Java Beans or Spring. The **JBoss Enterprise SOA Platform** provides out-of-the-box support for many different approaches.

The service action's processing becomes this:

```
    @Process
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, you could use the message header's action field rather than embed the opcode within the message body. The drawback is that it will not work if multiple actions are chained together and if each of these needs a different opcode.

Report a bug

7.4. DECODE THE PAYLOAD

Procedure 7.1. Task

1. 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
{
 String 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
}
```

This code illustrates 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.

2. Code every operation supported by the service similarly.

Report a bug

7.5. CONSTRUCT THE CLIENT

Procedure 7.2. Task

- 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 !maximumRetriesExceeded();
```

NOTE

Much of the above will be recognizable to readers who have worked with traditional client/server *stub generators*. In those systems, the low-level details (such as the opcodes and the parameters) are hidden behind higher-level stub abstractions. SOA Platform has integration with RESTEasy that enables a user to develop annotation based REST style web services. These hide the low level details such as opcodes and parameters.

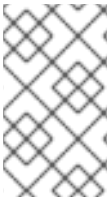
Report a bug

7.6. CONFIGURING A REMOTE SERVICE INVOKER

You do not need to configure anything to use the ServiceInvoker from within ESB actions: it will work "out of the box." However, to use it from a remote JVM, such as from a standalone Java application, a servlet or an EJB, you will need to make the following JAR files available:

commons-codec-1.3.jar
commons-collections.jar
commons-configuration-1.5.jar
commons-lang-2.4.jar
commons-logging.jar
concurrent.jar
hornetq-core-client.jar
hornetq-jms.jar
javassist.jar

jboss-aop-client.jar
jboss-common-core.jar
jboss-javaee.jar
jboss-logging-spi.jar
jboss-mdr.jar
jboss-remoting.jar
jbossall-client.jar
jbossesb-config-model-1.0.1.jar
jbossesb-config-model-1.1.0.jar
jbossesb-config-model-1.2.0.jar
jbossesb-config-model-1.3.0.jar
jbossesb-config-model-1.3.1.jar
jbossesb-registry.jar
jbossesb-rosetta.jar
jbossjmx-ant.jar
jbossts-common.jar
juddi-client-3.1.3.jar
log4j.jar
netty.jar
scout-1.2.6.jar
serializer.jar
trove.jar
uddi-ws-3.1.3.jar



NOTE

These files are found in the **$SOA_HOME/jboss-as/client/**, **$SOA_HOME/jboss-as/common/lib/** and **$SOA_HOME/jboss-as/server/$SOA_CONF/deployers/esb.deployer/lib/** directories.

The following configuration file is also required to be available on the classpath:

- **jbossesb-properties.xml**
- **META-INF/uddi.xml**

Report a bug

7.7. START THE JBOSS ENTERPRISE SOA PLATFORM

Prerequisites

The following software must be installed:

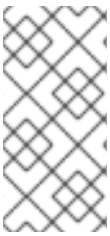
- JBoss Enterprise SOA Platform

Procedure 7.3. Start the JBoss Enterprise SOA Platform

- **Start the SOA server in a *server window***
  - **Red Hat Enterprise Linux**
    a. Open a terminal and navigate to the **bin** directory by entering the command **cd *SOA_ROOT*/jboss-as/bin**.
    b. Enter **./run.sh** to start the SOA server. (Because you are not specifying a server profile, "default" will be used.)
  - **Microsoft Windows**
    a. Open a terminal and navigate to the **bin** directory by entering the command **chdir *SOA_ROOT*\jboss-as\bin**.
    b. Enter **run.bat** to start the SOA server. (Because you are not specifying a server profile, "default" will be used.)

Result

The server starts. Note that this will take approximately two minutes, depending on the speed of your hardware.



NOTE

To verify that there have been no errors, check the server log: **less *SOA_ROOT*/jboss-as/server/*PROFILE*/log/server.log**. As another check, open a web browser and go to http://localhost:8080. Make sure you can login to the admin console with the user name and password you have set.

Report a bug

7.8. DEPLOY THE "HELLO WORLD" QUICKSTART ON YOUR TEST SERVER

Prerequisites

- Check that the setting in **SOA_ROOT/jboss-as/samples/quickstarts/conf/quickstarts.properties-example** matches the server configuration (**default** in a testing environment).

Procedure 7.4. Deploy the "Hello World" Quickstart

1. Check that the server has fully launched.
2. Open a second terminal window and navigate to the directory containing the quick start: **cd SOA_ROOT/jboss-as/samples/quickstarts/helloworld** (or **chdir SOA_ROOT\jboss-as\samples\quickstarts\helloworld** in Microsoft Windows).
3. Run **ant deploy** to deploy the quickstart. Look for messages such as this to confirm if the deployment was successful:

```
deploy-esb:
     [copy] Copying 1 file to
/jboss/local/53_DEV2/jboss-soa-p-5/jboss-as/server/default/deploy

deploy-exploded-esb:

quickstart-specific-deploys:
     [echo] No Quickstart specific deployments being made.

display-instructions:
     [echo]
     [echo] ******************
     [echo] Quickstart deployed to target JBoss ESB/App Server at
'/jboss/local/53_DEV2/jboss-soa-p-5/jboss-as/server/default/deploy'.
     [echo] 1.  Check your ESB Server console to make sure the
deployment was
executed without errors.
     [echo] 2.  Run 'ant runtest' to run the Quickstart.
     [echo] 3.  Check your ESB Server console again.  The Quickstart
should
have produced some output.
     [echo] ******************

deploy:

BUILD SUCCESSFUL
```

Also, check for this in the **SOA_ROOT/jboss-as/server/default/log/server.log**:

```
10:58:52,998 INFO  [NamingHelper] JNDI InitialContext properties:{}
11:00:58,154 INFO  [QueueService]
```

```
Queue[/queue/quickstart_helloworld_Request_esb] started,
fullSize=200000,
pageSize=2000, downCacheSize=2000
11:00:58,186 INFO  [QueueService]
Queue[/queue/quickstart_helloworld_Request_gw] started,
fullSize=200000,
pageSize=2000, downCacheSize=2000
11:00:58,427 INFO  [EsbDeployment] Starting ESB Deployment
'Quickstart_helloworld.esb'
```

4. Run the quickstart by issuing this command: **`ant runtest`**. When the quickstart is run, messages such as this are written to the **`SOA_ROOT/jboss-as/server/default/log/server.log`**:

```
11:03:02,190 INFO  [STDOUT]
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
11:03:02,191 INFO  [STDOUT] Body: Hello World
11:03:02,191 INFO  [STDOUT]
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
11:03:02,192 INFO  [STDOUT] Message structure:
11:03:02,192 INFO  [STDOUT] [Hello World].
```

Result

The words "Hello World" will appear on the server terminal. This message will also be appended to the **`SOA_ROOT/jboss-as/server/default/log/server.log`** file.

Report a bug

7.9. TEST THE CONFIGURATION OF THE REMOTE CLIENT

Prerequisites

- The server must have been started and the Hello World quick start must have been deployed

Procedure 7.5. Task

- Run this test code:

```
package org.jboss.esb.client;
import
import
import
import
org.jboss.soa.esb.client.ServiceInvoker;
org.jboss.soa.esb.listeners.message.MessageDeliverException;
org.jboss.soa.esb.message.Message;
org.jboss.soa.esb.message.format.MessageFactory;
public class EsbClient
{
public static void main(String[] args)
{
```

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

Report a bug

7.10. VERIFY THAT A REMOTE CLIENT'S CONFIGURATION IS CORRECT

Prerequisites

- The JBoss Enterprise SOA Platform must be running and the HelloWorld quick start must be deployed.

Procedure 7.6. Task

- Run this code:

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

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

public class EsbClient
{
    public static void main(String[] args)
    {

System.setProperty("javax.xml.registry.ConnectionFactoryClass",

"org.apache.ws.scout.registry.ConnectionFactoryImpl");
        try
        {
            Message message =
MessageFactory.getInstance().getMessage();
            message.getBody().add("Sample payload");
            ServiceInvoker invoker = new
ServiceInvoker("FirstServiceESB", "SimpleListener");
```

```
            invoker.deliverAsync(message);
        }
        catch (final MessageDeliverException e)
        {
            e.printStackTrace();
        }
```

Report a bug

7.11. FURTHER ADVICE FOR WHEN BUILDING CLIENTS AND SERVICES

These hints may be of help when you are building clients and services:

- when developing an action, ensure any payload information specific to it is maintained in unique message body locations.
- try not to expose any backend service implementation details within the message as this will make it difficult to change the implementation without affecting clients. Use message definitions (contents, formats and so on) which are "implementation-agnostic" as this will help to keep the coupling loose.
- for stateless services, use the **`ServiceInvoker`** as it handles fail-over "opaquely."
- When building request/response applications, use the correlation information (MessageID and RelatesTo) within the Message Header.
- consider using the Header Action field for the main service opcode.
- If using asynchronous interactions in which there is no delivery address for responses, consider sending any errors to the MessageStore so that they can be monitored later.
- until the **JBoss Enterprise SOA Platform** provides better automatic support for service contract definition and publication, consider maintaining a separate repository of these definitions, and make it available to both developers and users.

Report a bug

CHAPTER 8. ADVANCED TOPICS

8.1. NODE

When you use the JBoss Enterprise SOA Platform, it is implied that the service has become the building unit. This allows you to replicate identical services across many nodes, where each node is a virtual or physical machine running an instance of the enterprise service bus.

Report a bug

8.2. THE BUS

The collective name for a group of nodes is the Bus. Services within the bus use different delivery channels to exchange messages.

Report a bug

8.3. DELIVERY CHANNEL

A delivery channel is a protocol provided by a system external to the enterprise service bus. A channel maybe JMS, HTTP or FTP. Services can be configured to listen to more than one protocol. For each protocol to which a service is configured to listen, it creates an end point reference in the registry.

Report a bug

8.4. RUN THE SAME SERVICE ON MORE THAN ONE NODE IN A CLUSTER

Procedure 8.1. Task

- To run the same service on more than one node in a cluster, wait until the Registry's cache revalidates.

Report a bug

8.5. REMOVE FAILED END-POINT REFERENCES FROM THE REGISTRY

Procedure 8.2. Task

1. Open the **`jbossesb-properties.xml`** in a text editor: **`vi SOA_ROOT/jboss-as/server/PROFILE/deployers/esb.deployers/jbossesb-properties.xml`**.
2. Scroll down to the section that contains org.jboss.soa.esb.failure.detect.removeDeadEPR. Set this property to true.
3. Save the file and exit.

WARNING

Note that the default setting is false because this feature should be used with extreme care. If it is employed, the end-point reference for a service that is simply overloaded and, therefore, slow to respond may, inadvertently, be removed by mistake. There will be no further interactions with these "orphaned" services you may have to restart them.

Report a bug

8.6. HOW SERVICES WORK

Introduction

Read this section to learn in more detail how services, end-point references, listeners and actions actually work.

The following code fragment is loosely based on the configuration settings for the JBossESBHelloworld example:

```
  ...
<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>
...
```

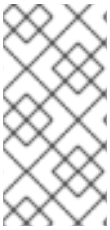
When the service initializes it registers the category, name and description to the UDDI registry. Also, for each listener element, it will register a ServiceBinding to UDDI, in which it stores an end-point reference. (In this case it will register a JMSEPR for this service, as it is a jms-listener.)

The specific details for JMS like the queue name are not shown, but appeared at the top of the jboss-esb.xml file where you can find the 'provider' section.

In the jms-listener you can simply reference the "quickstartEsbChannel" in the busidref attribute.

If it is given the category and service name, another service can look up your service in the registry. It will then receive the JMSEPR which it can use to send a message to your service. (All of this work is done by the ServiceInvoker class.)

When your 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, in this case, is the SystemPrintln action.



NOTE

Because ServiceInvoker hides much of the fail-over complexity from users, by necessity, it can only work with native ESB Messages. Furthermore, not all gateways have been modified to use the ServiceInvoker, so ESB-unaware messages coming to those kinds of gateways may not always be able to take advantage of service fail-over.

Report a bug

8.7. APPLICATION SERVICE

An application service is a logical service compromised of multiple individual services.

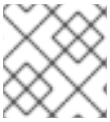
Report a bug

8.8. HOW SERVICE REPLICATION WORKS

What happens if you take a service like the **`helloworld.esb`** and deploy it to Node2 as well as on Node1? Assume you are using jUDDI for your registry and that you have configured all our nodes to access one central jUDDI database. Node2 will find that the **`FirstServiceESB - SimpleListener Service`** is already registered. Therefore, it will add a second ServiceBinding to this service so there are now two ServiceBindings for this service. Hence, if Node1 goes down, Node2 will keep on working.

Note that you will have load balancing as both service instances will be listening to the same queue.

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

Replication of services may be defined by service providers outside of the control of service consumers. As such, there may be times when the sender of a message does not want to silently fail-over to using an alternative service if one is mentioned within the Registry. If you set the org.jboss.soa.esb.exceptionOnDeliverFailure message property to true then no retry attempt will be made by the ServiceInvoker and the MessageDeliverException will be thrown. If you want to specify this approach for all messages then define the same property in the Core section of the JBossESB property file.

Report a bug

8.9. JBOSSMESSAGING

JBossMessaging is open source software that provides clustering functionality. It is the default provider of clustering in the JBoss Enterprise SOA Platform product as it allows you to cluster JMS services.

Report a bug

8.10. CLUSTER

A cluster is a group of loosely-connected computers that work together in such a way that they can be viewed as a single system. The components (or nodes) of a cluster are usually connected to each other through fast local area networks, each node running its own operating system. Clustering middleware orchestrates collaboration between the nodes, allowing users and applications to treat the cluster as a single processor. Clusters are effective for scalable enterprise applications, since performance is improved by adding more nodes as required. Furthermore, if at anytime one node fails, others can take on its load.

Report a bug

8.11. STATELESS SERVICE FAILOVER

The JBoss Enterprise SOA Platform provides "failover" capabilities for stateless services. If one node fails, the service will be resumed on another. The ServiceInvoker hides much of the fail-over complexity from users but it only works with native Enterprise Service Bus messages and, furthermore, not every gateway has been programmed to take advantage of it. Hence, non-ESB Aware messages sent to these gateway implementations may not be able to use service fail-over.

Report a bug

8.12. ENABLE JMS CLUSTERING

Procedure 8.3. Task

- Read the documentation on clustering for JBoss Messaging at http://community.jboss.org/wiki/JBossMessaging.

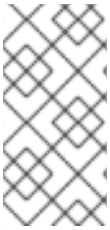
Report a bug

8.13. PROTOCOL CLUSTERING

Introduction

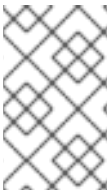
When you cluster your JMS you remove a single point of failure from your architecture.

Both JBossESB replication and JMS clustering can be used together. For example, Service A is identified in the registry by a single JMS end-point reference. However, invisibly to the client, the JMS end-point reference is pointing to a clustered JMS queue, which has been separately configured to support three services. This is a federated approach to availability and load balancing.



NOTE

In fact, masking the replication of services from users (the client in the case of the JBoss ESB replication approach, and JBossESB itself in the case of the JMS clustering) is in line with the SOA principle of hiding these implementation details behind the service endpoint and not exposing them at the contract level.



NOTE

If using JMS clustering in this way you will 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 will not benefit from ESB replication.

If your provider simply cannot provide any clustering, 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.

Report a bug

8.14. RUNNING THE SAME SERVICE ACROSS DIFFERENT NODES IN A CLUSTER

Introduction

If you would like to run the same service on more than one node in a cluster you have to wait for the service registry cache to revalidate. After that, the service will run across the clustered environment.

Report a bug

8.15. CONFIGURE THE REGISTRY CACHE TIME-OUT VALUE

Procedure 8.4. Task

1. Open the **`deploy/jbossesb.sar/jbossesb-properties.xml`** file in your text editor: **`vi deploy/jbossesb.sar/jbossesb-properties.xml`**

2. Set the cache time-out value:

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

   Note that 60 000 milliseconds (sixty seconds) is the default value.

3. Save the file and exit.

Report a bug

8.16. CHANNEL FAIL-OVER

The HelloWorld Service can listen to multiple protocols. Here we have added a JMS 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>
...
```

The service is simultaneously listening to two JMS queues which can be provided by JMS providers on different physical boxes. This makes a JMS connection between two services redundant. You can also mix protocols in this configuration. For instance, this code show how to add an FTP listener.

```
...
<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>
...
```

NOTE

When the ServiceInvoker tries to deliver a message to the Service it will get a choice of eight end-point references (four EPRs from Node1 and four EPRs from Node2). To define which one will be used, configure a load-balancing policy.

Report a bug

8.17. DEACTIVATE AUTOMATIC FAIL-OVER

Procedure 8.5. Task

1. Open the JBossESB property file in a text editor.
2. Set the org.jboss.soa.esb.exceptionOnDeliverFailure property to true
3. Save the file and exit.

NOTE

Alternatively this can be configured on a per message basis by setting the same property in the specific message in question to true. In both cases the default is false.

Report a bug

8.18. LOAD BALANCING

Load balancing is a computer networking methodology to distribute workload across multiple computers or a computer cluster, network links, central processing units, disk drives, or other resources, to achieve optimal resource utilization, maximize throughput, minimize response time, and avoid overload. Using multiple components with load balancing, instead of a single component, also increases reliability through redundancy.

Report a bug

8.19. CONFIGURE A LOAD-BALANCING POLICY

Procedure 8.6. Task

1. Open the global configuration file in a text editor: **`vi SOA_ROOT/jboss-as/server/PROFILE/deployers/esb.deployers/jbossesb-properties.xml`**.
2. Scroll down to the org.jboss.soa.esb.loadbalancer.policy property. Set it with the policy you wish to use.
3. Save the file and exit.

Report a bug

8.20. LOAD BALANCING POLICIES

Table 8.1. Load balancing Policies Available

Policy Name	Description
first available	If a healthy service binding is found it will be used until it dies. The next end-point reference in the list will then be used. There is no load balancing between the two service instances with this policy.
round robin	A standard load-balancing policy whereby each end-point reference is utilised in list order.
random robin	This is like the round robin, but the selection is randomized.

NOTE

The end-point reference list used by the policy may become smaller over time as "dead" EPRs are removed. When the list is exhausted or the time-to-live of the list cache is exceeded, the ServiceInvoker will obtain a fresh list of EPRs from the Registry.

Report a bug

8.21. TRANSACTIONS AND THE ACTION PIPELINE

The action pipeline is capable of running either in a transaction or by itself. To perform the former, a service must have a JCA listener attached to it with the *transact* property set to *true*. It must also be told how to retrieve the TransactionManager when running in an application server. This can be accomplished by running JNDI look-up.

When working in the action pipeline, transactions can be rolled back during the commit phase which occurs after all the *onSuccess* methods of the actions are called. The entire pipeline can still fail and rollback during commit phase which would cause the message to retry the pipe.

Report a bug

8.22. ROLLBACKS

Rollbacks will return a service action to its original state before modifications. For a service action to rollback, the user should ensure that the *setRollbackOnly* property is present on the current transaction.

Report a bug

8.23. ROLLBACKS AND THE JMS JCA LISTENER

When using JMS JCA listener to orchestrate transactions, the retries will occur from the message queue. The maximum amount of retries and interval between retries can be set on the queue. If you are using HornetQ, you can include a *hornetq-configuration.xml* covering the rety policy for the specific queue the service is running from:

```
<configuration xmlns="urn:hornetq"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:hornetq /schema/hornetq-
configuration.xsd">
   <address-settings>
     <address-setting match="jms.queue.MyServiceQueue">
        <max-delivery-attempts>5</max-delivery-attempts>
        <redelivery-delay>1000</redelivery-delay>
        <max-size-bytes>10240</max-size-bytes>
     </address-setting>
   </address-settings>
</configuration>
```

Report a bug

8.24. MESSAGE RE-DELIVERY

If the list of end-point references 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.

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. If you want to control this on a per message basis then set the org.jboss.soa.esb.dls.redeliver property in the specific Message properties accordingly. The Message property will be used in preference to any global setting. The default is to use the value set in the configuration file.

Report a bug

8.25. SCHEDULING

8.25.1. Quartz Scheduler

Quartz Scheduler is an open source project that provides the foundation upon which the JBoss Enterprise SOA Platform's service scheduling functionality is built.

Report a bug

8.25.2. Configuring Quartz Scheduler

These are the default settings for Quartz Scheduler in the JBoss Enterprise SOA Platform:

```
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
```

You can override these properties or add new ones by specifying the configuration directly on the <schedule-provider> configuration as a <property> element. For example, this is how you would 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>
```

Report a bug

8.25.3. Scheduling Services

The JBoss Enterprise SOA Platform supports two types of scheduling functionality providers:

Bus Providers

These supply messages to action processing pipelines via messaging protocols such as JMS and HTTP. This provider type is “triggered” by the underlying messaging provider.

Schedule Providers

These supply messages to action processing pipelines based on a schedule driven model such as when the underlying message delivery mechanism (for example, the file system) offers no support for triggering the enterprise service bus when messages are available for processing, a scheduler periodically triggers the listener to check for new messages.

The JBoss Enterprise SOA Platform offers a <schedule-listener> as well as two <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..

IMPORTANT

Scheduling functionality is new to the JBoss Enterprise SOA Platform and not all of the listeners have been migrated over to this model yet.

Report a bug

8.25.4. Simple Schedule

This type of scheduler provides capabilities derived from these 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 http://books.xmlschemata.org/relaxng/ch19-77049.html.

endDate

The schedule end date and time. The format of this attribute value is that of the XML Schema type “dateTime”. See http://books.xmlschemata.org/relaxng/ch19-77049.html.

Here is some example code:

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

Report a bug

8.25.5. Cron Schedule

This type of schedular provides capabilities based on CRON expressions. Its attributes 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 http://books.xmlschemata.org/relaxng/ch19-77049.html.

endDate

The schedule end date and time. The format of this attribute value is that of the XML Schema type “dateTime”. See http://books.xmlschemata.org/relaxng/ch19-77049.html.

Here is a code example:

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

```
/>
        </schedule-provider>
</providers>
```

Report a bug

8.25.6. Scheduled Listener

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

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

ScheduledEventListener

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

ScheduledEventMessageComposer

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

The attributes of this listener are:

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

Report a bug

8.25.7. Sample Configuration Combining the Scheduled Listener and Cron Scheduler

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

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

Report a bug

CHAPTER 9. FAULT TOLERANCE AND RELIABILITY

9.1. SYSTEM RELIABILITY

Introduction

There are many components and services within the JBoss Enterprise SOA Platform. 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 the EPR information for the services it needs to function, the crash will have no adverse affect on your application. However, if it fails before this point, your application will not be able to progress. Therefore, in any determination of reliability guarantees, it is necessary to consider when failures occur and what type of failures they could be.

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

Report a bug

9.2. FAULT TOLERANCE

A fault-tolerant system is one which is designed to fulfill its specified purpose even with 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.

Report a bug

9.3. DEPENDABILITY

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".

Report a bug

9.4. MESSAGE LOSS

Many distributed systems support reliable message delivery, either point-to-point (one consumer and one provider) or group-based (many consumers and one provider). Even in the presence of failures, the semantics imposed on reliability usually mean the message will be delivered or the sender will be informed that it failed to reach the receiver. Often systems which employ reliable messaging

implementations distinguish between a message being delivered to the recipient and it subsequently being processed by the recipient. For instance, simply sending the message to a service does not mean much if a subsequent crash of that same service occurs before it has had time to work on the contents of the message.

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

Because none of the other transports supported by the product 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 the JBoss Enterprise SOA Platform about any failure to deliver the message. If a failure of the receiver occurs while processing and a response was expected, the receiver will eventually time-out and can retry.

NOTE

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

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

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

Report a bug

9.5. FAILED END-POINTS

Until/unless a failed machine recovers, it is not possible to determine the difference between a crashed machine or one that is simply running extremely slowly. Furthermore, because networks can become partitioned, it is entirely possible that different consumers have different views of which services are available.

Report a bug

9.6. SUPPORTED CRASH RECOVERY MODES

Unless you are using transactions or a reliable message delivery protocol such as JMS, the JBoss Enterprise SOA Platform will only tolerate crash failures that do not result in total system failure and allow the application to reason without ambiguity about whether the end-points involved are still "alive". If services crash or shut down cleanly before receiving messages, then it is safe to use transports other than JMS.

Report a bug

9.7. MESSAGE FAILURE, COMPONENT BY COMPONENT

Gateways

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

ServiceInvoker

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

JMS Broker

Messages that cannot be delivered to the JMS broker will be placed within the re-delivery queue. For enterprise deployments, a clustered JMS broker is recommended.

Action Pipelining

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

Report a bug

9.8. WAYS IN WHICH YOU CAN MINIMIZE THE RISK OF FAILURES

Here are some ways in which you can minimize the risk of data loss through failures:

- Try to develop stateless and idempotent services. If this is not possible, use MessageID to identify Messages so your application can detect retransmission attempts. If retrying Message transmission, use the same MessageID. Services that are not idempotent and would suffer from redoing the same work if they receive a retransmitted Message, should record state transitions against the MessageID, preferably using transactions. Applications based around stateless services tend to scale better as well.
- If developing stateful services, use transactions and a (preferably clustered) JMS implementation.

- Cluster your Registry and use a clustered/fault-tolerant back-end database, 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 services need higher reliability and fault tolerance capabilities than others. This will allow you to target transports other than JMS at those services, potentially improving 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 on application specific requirements.
- Because network partitions can make services appear as though they have failed, avoid transports that are more prone to this type of failure for services that cannot cope with being misidentified as having crashed.
- In some situations (for example, HTTP) the crash of a server after it has dealt with a message but before it has responded could result in another server doing the same work. This is 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 doesn't 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 problematical and you may have to rely upon application semantics to determine that Messages did not arrive (for example, the amount of money in the bank account does not match expectations). When using either the ServiceInvoker or Couriers to delivery asynchronous Messages, a return from the respective operation (e.g., deliverAsync) does not mean the Message has been acted upon by the service.
- The message store is used by the redelivery protocol. However, as mentioned 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 (they do not get written to the store before a crash), or delivered multiple times (the redelivery mechanism pulls a message from the store, delivers it successfully but there is 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 un-processed 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.

Report a bug

CHAPTER 10. DEFINING SERVICE CONFIGURATIONS

10.1. INTRODUCTION TO CONFIGURING THE PRODUCT

Introduction

The JBoss Enterprise SOA Platform's configuration settings are based on the jbossesb-1.3.0 XSD schema (http://anonsvn.jboss.org/repos/labs/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.3.0.xsd). This XSD is always the definitive reference for configuring the product.Introductory text goes here.

This model has two main sections:

1. <globals>
2. <providers>: Defines all the message <bus> providers used by the message <listener>s, defined within the <services> section of the model.
3. <services>: 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.

NOTE

By far the easiest way to create configurations based on this model is using JBoss Developer Studio (http://www.jboss.com/products/devstudio/) This provides the author with auto-completion features when editing the configuration. Right mouse-click on the **File** -> **Open With** -> **XML Editor**.

Report a bug

10.2. PROVIDERS

The <providers> part of the configuration file defines all of the message <provider> instances for a single instance of the ESB.

Report a bug

10.3. TYPES OF PROVIDERS

Two types of providers are currently supported:

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

A bus provider, such as **<jms-provider>** can contain multiple **<bus>** definitions. The **<provider>** can also be decorated with **<property>** instances relating to provider specific properties that are common in **<bus>** instances defined on that **<provider>**. For example, JMS may have “connection-

factory", "jndi-context-factory" and so on. Likewise, each **<bus>** instance can be decorated with **<property>** instances specific to that **<bus>** instance. (For example, JMS has "destination-type", "destination-name" and so forth.)

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

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

IMPORTANT

Red Hat recommends using the specialized extensions of these types for creating configurations, namely <jms-provider> and <jms-bus> for JMS. The most important part of the above configuration is the busid attribute defined on the <jms-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.

Report a bug

10.4. SERVICES

The <services> part of the configuration defines each of the services under the management of this instance of the Enterprise Service Bus. It defines them as a series of <service> configurations.

Report a bug

10.5. ATTRIBUTES OF A SERVICE

A <service> can possess the following attributes:

Table 10.1. Service Attributes

Name	Description	Type	Required
name	The 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 registry.	xsd:string	true

Name	Description	Type	Required
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 (<jms-listener>, <sql-listener> and so forth.)

Report a bug

10.6. ATTRIBUTES OF A LISTENER

The “base” <listener> possesses the following attributes. These attribute definitions are inherited by all <listener> extensions. As such, they can be set for all of the listeners and gateways supported by the JBoss Enterprise SOA Platform, such as InVM.

Table 10.2. 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 maximum 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” or “Message Aware” Listener. A message bus defines the details of a specific message channel or transport.	xsd:boolean	true

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

NOTE

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

Here is an example of a <listener> referencing a <bus>:

```
<?xml version = "1.0" encoding = "UTF-8"?>
```

```
<jbossesb
xmlns="http://anonsvn.jboss.org/repos/labs/labs/jbossesb/trunk/product/etc
/schemas/xml/jbossesb-1.3.0.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://anonsvn.jboss.org/repos/labs/labs/jbossesb/trun
k/product/etc/schemas/xml/jbossesb-1.3.0.xsd
http://anonsvn.jboss.org/repos/labs/labs/jbossesb/trunk/product/etc/schema
s/xml/jbossesb-1.3.0.xsd"
    parameterReloadSecs="5">
    <providers>
          <jms-provider name="JBossMQ" connection-
factory="ConnectionFactory">
               <jms-bus busid="Reconciliation">
                  <jms-message-filter
                      dest-type="QUEUE"
                      dest-name="queue/B"
                   />
               </jms-bus>
<!-- busid --> <jms-bus busid="ReconciliationEsb">
                  <jms-message-filter
                      dest-type="QUEUE"
                      dest-name="queue/C"
               </jms-bus>
          </jms-provider>
      </providers>
      <services>
          <service category="Bank" name="Reconciliation"
                   description="Bank Reconciliation Service">
              <listeners>
<!-- busidref --> <jms-listener name="Bank-Listener"
                      busidref="Reconciliation"
                      is-gateway="true"/>
                  <jms-listener name="Bank-Esb"
                      busidref="ReconciliationEsb"/>
              </listeners>
              <actions>
                   ....
              </actions>
          </service>
      </services>
</jbossesb>
```

Report a bug

10.7. ACTIONS

Actions usually contain the logic for processing the payload of the messages received by the service (through its listeners). Alternatively, an action may contain the transformation or routing logic for messages to be consumed by an external service or entity. A service must always have a list of one or more actions. Actions generally act as templates, therefore requiring you to set external configuration options in order for them to work.

Report a bug

10.8. ATTRIBUTES OF AN ACTION

The <action> element possesses the following attributes.

Table 10.3. 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 action class must extend one of: org.jboss.soa.esb.actions.AbstractActionLifecycle, org.jboss.soa.esb.actions.ActionPipelineProcessor.	xsd:string	true
process	The name of the “process” method that will be reflectively called for message processing.(Default is the “process” method.)	xsd:int	false

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

Like a number of other elements/types in this model, the <action> type can also contain zero or more <property> element instances. The <property> element/type can define a standard name-value-pair, or contain free form content (xsd:any). According to the XSD, this free form content is valid as child content for the <property> element/type, no matter where it is in the configuration (on any of <provider>, <bus>, <listener> and any of their derivatives). However, it is only on <action> defined <property> instances that this free-form child content is used.

Actions are implemented via 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 through this constructor-supplied ConfigTree instance that all of the action attributes are made available, including the free-form content from the action <property> instances. The free form content is supplied as child content on the ConfigTree instance.

Here is an example of an <actions> configuration:

```
<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>
```

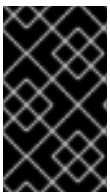
Report a bug

10.9. IMPLEMENTING A TRANSPORT-SPECIFIC CONFIGURATION

The JBoss Enterprise SOA Platform configuration model has transport-specific variants for **<provider>**, **<bus>** and **<listener>** (that is, JMS, SQL and so forth.) Using one of these special variants allows you to have stronger validation on the configuration. These specializations explicitly define the configuration requirements for each of the transports supported by the product out-of-the-box.

NOTE

Using one of these specific implementations also makes configuration easier if you are using an XSD-aware XML editor (such as JBoss Developer Studio).



IMPORTANT

Red Hat recommends you use these specialized types instead of the “base” types when creating JBoss ESB configurations. The only variation is when a new transport is being supported outside an official product release.

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

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

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

So the transport specific implementations that are in place in this product are as follows:

1. 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.
2. 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 <sql-bus> specify the JDBC connection properties, the <sql-message-filter> specifies the message/row selection and processing properties.
3. 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

4. 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.

5. 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 <fs-bus> specify the File System access properties, the <fs-message-filter> specifies the message/file selection and processing properties.

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

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

Each of the transport-specific types include an additional element not present in the “base”, this being <*-message-filter>. This element can be added inside either the <*-bus> or <*-listener>. If you use this element in both places you can specify message-filtering globally for the bus (in other words, for every listeners using that bus) or locally (that is, on a listener-by-listener basis).

Table 10.4. 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 which will filter on this message-selector.	No
persistent	Indicates if the delivery mode for JMS should be persistent or not. Set as 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 username. This will be used when creating a connection to the destination.	No
jms-security-credential	JMS destination password. This will be used when creating a connection to the destination.	No

Here is an 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>
```

Table 10.5. JMS Listener configuration

Property Name	Description	Required
name	The listener name.	Yes
busrefid	The ID of the JMS bus on which to listen.	No
is-gateway	Is this JMS Listener instance a gateway, or is it a message aware listener.	No. Default is false.
maxThreads	The maximum number of threads to use when listening for messages on the JMS bus. Only relevant if is-gateway is false.	No. Default is 1
clientId	Client ID to be associated with the JMS connection. Used to associate a connection and its objects with state maintained on behalf of the client by a provider. (For example, durable subscriptions.)	No. If a clientId is required (e.g. when a durableSubscriptionName is specified), but is not specified, it will default to the listener name.
durableSubscriptionName	Durable subscription name.	No. Only relevant for JMS Topics.

Report a bug

10.10. CONFIGURING THE FILE SYSTEM PROVIDER

The following file system provider configuration options are available within a file system message filter (fs-message-filter) which itself is contained in an fs-bus. For a good example, see the helloworld_file_action quick start.

NOTE

For the directory options below, each directory specified must exist and the application server's user must have both read and write permissions on the directory in order to move and rename files through them.

Table 10.6. File System Message Filter Configuration

Property	Description	Required
directory	The directory that will be monitored for incoming files.	Yes
input-suffix	Suffix used to filter for incoming files. Must be one character or greater, with a ".", such as ".esbIn".	Yes
work-suffix	The suffix used when a file is being processed by the ESB. The default is ".esbInProcess".	No
post-suffix	The suffix used when a file has been successfully processed by the ESB. The default is ".esbDone".	No
post-delete	If true, the file will be deleted after it is processed. In this case, post-directory and post-suffix have no effect. Defaults to true.	No
post-directory	The directory to which the file will be moved after it is processed by the ESB. Defaults to the value of directory above.	Yes
post-rename	If true, the file will be renamed after it is processed. Note that the post-rename and post-delete options are mutually exclusive.	No
error-delete	If true, the file will be deleted if an error occurs during processing. In this case, error-directory and error-suffix have no effect. This defaults to "true."	No
error-directory	The FTP directory to which the file will be moved after when an error occurs during processing. This defaults to the value of directory above.	Yes
error-suffix	The suffix which will be added to the file name after an error occurs during processing. Defaults to `.esbError`.	No

Report a bug

10.11. CONFIGURING AN FTP PROVIDER

Table 10.7. 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. This defaults to "true."	No
error-directory	The FTP directory to which the file will be moved after when an error occurs during processing. This defaults to the value of directory above.	No
error-suffix	The suffix which will be added to the file name after an error occurs during processing. Defaults to **`.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 mode. Setting this to "true" means the FTP client will establish two connections to the ftpserver. Defaults to false, meaning that the client will tell the FTP Server the port to which it should connect. The FTP Server then establishes the connection to the client.	No

Property	Description	Required
read-only	If true, the FTP Server does not permit write operations on files. Note that, in this case, the following properties have no effect: work-suffix, post-delete,post-directory, post-suffix, error-delete, error-directory, and error-suffix. Defaults to false. See section Read-only FTP Listener for more information.	No
certificate-url	The URL to a public server certificate for FTPS server verification or to a private certificate for SFTP client verification. An SFTP certificate can be located as a resource embedded within a deployment artifact	No
certificate-name	The common name for a certificate for FTPS server verification	No
certificate-passphrase	The pass-phrase of the private key for SFTP client verification.	No

You can configure a schedule listener that polls for remote files based on the configured schedule (scheduleidref).

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 filenames 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.

NOTE

There must exist a strategy for removing the filenames 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 filenames from the cache could be done by having a database procedure that removes all filenames from the cache every couple of days. Another strategy would be to specify a TreeCacheListener that upon evicting filenames 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 10.8. Read-only FTP Listener Configuration

Name	Description
scheduleidref	Schedule used by the FTP listener. See Service Scheduling.
remoteFileSystem Strategy-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`**

Name	Description
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

Here is an 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.DeleteOnEvictT
reeCacheListener"/>

</ftp-listener>
```

Here is some more sample code, demonstrating how to configure the JBoss Cache component:

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

Table 10.9. Configuration

Property	Description	Comments
maxNodes	The maximum number of files that will be stored in the cache.	0 denotes no limit

Property	Description	Comments
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

NOTE

The helloworld_ftp_action quick start demonstrates the read-only configuration. Run 'ant help' in the helloworld_ftp_action quick start directory for instructions on running the quick start. .

Report a bug

10.12. UDP GATEWAY

The UDP Gateway allows you to receive ESB-unaware messages sent using the User Datagram Protocol. The payload of messages arriving via this gateway will be passed along to the action pipeline in the default ESB Message object location.

Report a bug

10.13. CONFIGURING THE UDP GATEWAY

Call the esbMessage.getBody().get() method from within your actions to retrieve the byte array payload from messages arriving via the UDP Gateway.

Here are the options for configuring the gateway:

Table 10.10. UDP Gateway Configuration

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

Here is an example configuration:

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

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

Report a bug

10.14. JBOSS REMOTING GATEWAY

The JBoss Remoting Gateway hooks the open source JBoss Remoting component into the JBoss Enterprise SOA Platform, providing another a transport option. The Gateway leverages support for HTTP(S) and Socket (+SSL) via the JBoss Remoting component.

Report a bug

10.15. CONFIGURING THE JBOSS REMOTING GATEWAY

Here is the basic configuration of the JBoss Remoting provider:

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

The <jbr-bus> can then be referenced from a <service> configuration through the <jbr-listener>:

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

IMPORTANT

The <jbr-listener> is only supported as a gateway. If you set is-gateway to false an error will occur.

You can set the following configuration options for the JBoss Remoting Gateway on any of the <jbr-provider>, <jbr-bus> or <jbr-listener> elements (set them as <property> elements).

Table 10.11. Configuration

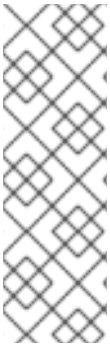
Name	Description	Default

Name	Description	Default
synchronous	Is the target Service to be invoked Synchronously.	True
serviceInvokerTimeout	Asynchronous invocation timeout.	20000
asyncResponse	Asynchronous response.	"<ack/>
securityNS	This is the namespace for the version of Web Service Security that should be used. This namespace is used to match security headers in SOAP messages. This is to allow the Enterprise Service Bus to extract security information from these headers.	http://docs.oasis-open.org/wss/2004/01/oasis-200401http-wss-wssecurity-secext-1.0.xsd

Also note that you can set JBoss Remoting-specific configuration properties. This can be done by prefixing the property name with "jbr-". Consult the JBoss Remoting documentation (http://www.jboss.org/jbossremoting/) for details.

Here is an example of a configuration that uses JBoss Remoting- specific settings to configure a keystore and client authentication mode for HTTPS:

```
<jbr-provider name="https_provider" protocol="https" host="localhost">
    <!-- Https/SSL settings -->
    <property name="jbr-KeyStoreURL" value="/keys/myKeystore" />
    <property name="jbr-KeyStorePassword" value="keys_ssl_pass" />
    <property name="jbr-TrustStoreURL" value="/keys/myKeystore" />
    <property name="jbr-TrustStorePassword" value="keys_ssl_pass" />
    <property name="jbr-ClientAuthMode" value="need" />
    <property name="serviceInvokerTimeout" value="20000" />

    <jbr-bus busid="https_bus" port="9433"/>
</jbr-provider>
```



NOTE

The JBoss Remoting Gateway expects all response headers to be located in the **`Message.Properties`** as instances of **`org.jboss.soa.esb.message.ResponseHeader`** class. If you require the Gateway to set specific response headers, the enterprise service bus message provided to the gateway response decompose (for example, after a synchronous invocation of the target service) must contain instances of the **`ResponseHeader`** class, set on the **`Message.Properties`**.

Report a bug

10.16. HTTP GATEWAY

The HTTP Gateway allows you to expose ESB-unaware HTTP end-points.

Report a bug

10.17. CONFIGURING THE HTTP GATEWAY

The HTTP Gateway uses the JBoss Enterprise SOA Platform's Application Server's HTTP Container to expose HTTP end-points, hence many of the configurations are managed at the container level. These are the bind/port address, SSL and so forth.

The following code shows you the way easiest to configure the <http-gateway> on a service (no provider configuration is required):

```
<service category="Vehicles" name="Cars" description=""
invmScope="GLOBAL">
    <listeners>
        <http-gateway name="Http" />
    </listeners>
    <actions mep="RequestResponse">
        <!-- Service Actions.... -->
    </actions>
</service>
```

The above configuration uses the **default** HTTP bus provider. This is because it does not define a busrefid attribute. It uses the service category and name to construct the HTTP end-point address of the following format:

```
        http://<host>:<port>/<.esbname>/http/Vehicles/Cars
```

The <.esbname> token is the name of the .esb deployment, without the ".esb" extension. Note also the "http" token in the address. This is a hardcoded name-space prefix used for all <http-gateway> end-points.

The <http-gateway> also supports a urlPattern:

```
<service category="Vehicles" name="Cars" description=""
invmScope="GLOBAL">
    <listeners>
        <http-gateway name="Http" urlPattern="esb-cars/*" />
    </listeners>
    <actions mep="RequestResponse">
        <!-- Service Aactions.... -->
    </actions>
</service>
```

This will expose an HTTP end-point for the service, capturing all HTTP requests found at the following address:

```
        http://<host>:<port>/<.esbname>/http/esb-cars/*
```

You can use the allowedPorts property to confine certain services to one or more HTTP ports. To do so, create a comma-separated list of the ports in question like this:

-

```
<http-gateway name="Http" urlPattern="esb-cars/*">
    <property name="allowedPorts" value="8080,8081">
</http-gateway>
```

This will expose a HTTP end-point for the service, capturing all HTTP requests under the following ports only (all other port's request would receive HTTP Status code 404 – Not Found).

- http://<host>:8080/*
- http://<host>:8081/*

The <http-gateway> is typically able to decode a HTTP request payload based on the request MIME type. It uses the **`jbossesb-properties.xml`** file's “core:org.jboss.soa.esb.mime.text.types” configuration property to decide whether the payload is to be decoded for the service as a string or whether it is to remain as a byte array, with the service handling the decoding itself through an action.

The “core:org.jboss.soa.esb.mime.text.types” configuration property is a semi-colon separated list of “text” (character) MIME types, with the default set being as follows (note the support for wildcards):

- text/*
- application/xml
- application/*-xml

The <http-gateway> uses the character encoding from the request when decoding text payloads.

The <http-gateway> also supports the payloadAs attribute, which can be used as an override for the default MIME type based behavior described above. With this attribute, you can explicitly tell the gateway to treat the payload as “BYTES” or “STRING”.

The HTTP Request contains a lot of other information in addition to the data payload that may be required by the service. This information is stored, by the gateway, in a HttpRequest object instance on the message. Use this code inside an action to access that information:

```
HttpRequest requestInfo = HttpRequest.getRequest(message);
```

HttpRequest exposes the following set of properties (via getter methods)

Table 10.12. Properties

Property	Description
queryParams	A java.util.Map<String, String[]> containing the query parameters. Note the values are String[] so as to support multi-valued parameters.
headers	A java.util.List<HttpHeader> containing the request headers.
authType	The name of the authentication scheme used to protect the endpoint, or null if not authenticated. Same as the value of the CGI variable AUTH_TYPE.
characterEncoding	The name of the character encoding used in the body of this request, or null if the request does not specify a character encoding.

Property	Description
contentType	Content Type (MIME Type) of the body of the request, or null if the type is not known. Same as the value of the CGI variable CONTENT_TYPE.
contextPath	The portion of the request URI that indicates the context of the request. The context path always comes first in a request URI. The path starts with a "/" character but does not end with a "/" character. For endpoints in the default (root) context, this returns "". The container does not decode this string. (See Servlet Spec.)
pathInfo	Any extra path information associated with the URL the client sent when it made this request. The extra path information follows the endpoint path but precedes the query string and will start with a "/" character. This method returns null if there was no extra path information. Same as the value of the CGI variable PATH_INFO. (See Servlet Spec.)
pathInfoToken	A List<String> containing the tokens of the pathInfo.
queryString	Query String (See Servlet Spec)
requestURI	The part of this request URL from the protocol name up to the query string. The web container does not decode this String. (See Servlet Spec)
requestPath	The part of this request URL that calls the endpoint. Does not include any additional path information or a query string. Same as the value of the CGI variable SCRIPT_NAME. This method will return just "http") if the urlPattern was "/*". (See Servlet Spec)
localAddr	The IP address of the interface on which the request was received.
localName	The host name of the IP interface on which the request was received.
method	HTTP Method
protocol	Name and version of the HTTP protocol
remoteAddr	The IP address of the client or last proxy that sent the request. Same as the value of the CGI variable REMOTE_ADDR.
remoteHost	The fully qualified name of the client or the last proxy that sent the request. If the engine cannot or chooses not to resolve the hostname (to improve performance), this will be the dotted-string form of the IP address. Same as the value of the CGI variable REMOTE_HOST.
remoteUser	The login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated. Whether the username is sent along with each subsequent request depends on the client and type of authentication. Same as the value of the CGI variable REMOTE_USER.

Property	Description
contentLength	The length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known. For HTTP servlets, same as the value of the CGI variable CONTENT_LENGTH.
requestSessionId	The session ID specified by the client, or null if non specified.
scheme	Scheme being used, whether it be “http” or “https.”.
serverName	The host name of the server to which the request was sent. It is the value of the part before ":" in the “Host” header value, if any, or the resolved server name, or the server IP address.

By default, this gateway synchronously invokes the associated service and returns the service response payload as the HTTP response.

The HTTP Gateway always returns a synchronous response to a synchronous HTTP client, so it is never asynchronous in the absolute sense of the word. By default, the Gateway will synchronously invoke the action pipeline, returning the synchronous service response as the HTTP response from the gateway.

Asynchronous response behavior, from the point of view of this Gateway, simply means that the gateway returns a synchronous HTTP response after an asynchronous invocation of the action pipeline (that is, not a synchronous service invocation). Because it invokes the service asynchronously, it cannot return a service response as part of its synchronous HTTP response. Therefore, you need to configure the gateway telling it how to make the asynchronous response.

To configure the asynchronous behavior, add an <asyncHttpResponse> element to the <http-gateway>:

```
<listeners>
    <http-gateway name="Http" urlPattern="esb-cars/*">
        <asyncResponse />
    </http-gateway>
</listeners>
```

If configured as above, the gateway will return a zero length HTTP response payload, with a HTTP status of 200 (OK).

The asynchronous response HTTP status code can be configured (away from the default of 200) by simply setting the "statusCode" attribute on the <asyncResponse> element:

```
<listeners>
    <http-gateway name="Http" urlPattern="esb-cars/*">
        <asyncResponse statusCode="202" />
    </http-gateway>
</listeners>
```

As stated above, a zero length payload is returned (by default) for asynchronous responses. This can be overridden by specifying a <payload> element on the <asyncResponse> element:

```
<listeners>
    <http-gateway name="Http" urlPattern="esb-cars/*">
```

```
        <asyncResponse statusCode="202">
            <payload classpathResource="/202-static-response.xml"
                     content-type="text/xml"
                     characterEncoding="UTF-8" />
        <asyncResponse>
    </http-gateway>
</listeners>
```

Table 10.13.

Property	Description	Required
classpathResource	Specifies the path to a file on the classpath that contains the response payload.	Required
contentType	Specifies the content/mime type of the payload data specified by the classpathResource attribute.	Required
characterEncoding	The character encoding of the data specified by the classpathResource attribute.	Optional

Consistent with how the gateway creates a HttpRequest object instance for the associated service, the associated service can create a HttpResponse object for the gateway on a synchronous HTTP gateway invocation.

Use this code in your service's action to create and set a HttpResponse instance on the response message:

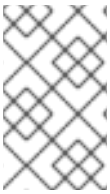
```
HttpResponse responseInfo = new HttpResponse(HttpServletResponse.SC_OK);

responseInfo.setContentType("text/xml");
// Set other response info ...

// Set the HttpResponse instance on the ESB response Message instance
responseInfo.setResponse(responseMessage);
```

The HttpResponse object can contain the following properties, which are mapped onto the outgoing HTTP gateway response:

Table 10.14.

Property	Description
responseCode	The HTTP Response/Status Code (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)to be set on the gateway response.
contentType	The response payload MIME Type.
encoding	The response payload content encoding.
length	The response payload content length.

Property	Description
headers	A java.util.List<HttpHeader> containing the request headers.



NOTE

Using the HttpResponse class is efficient because this class is also used by internal actions such as the HttpRouter, making it easy to perform proxying operations using this gateway.

NOTE

The response payload content encoding can also be set through the HttpResponse instance as is the HTTP response status code (ulink url="http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html" />).

By default, this gateway will wait for 30,000 ms (30 s) for the synchronous service invocation to complete, before raising a ResponseTimeoutException. To override the default timeout, configure the "synchronousTimeout" property:

```
<listeners>
    <http-gateway name="Http" urlPattern="esb-cars/*">
        <property name="synchronousTimeout" value="120000"/>
    </http-gateway>
</listeners>
```

You can map action pipeline exceptions to specific HTTP response codes through the ESB configuration. The mappings can be specified in the top level <http-provider> and can also be specified directly on the <http-gateway>, allowing per-listener override of the exception mappings defined "globally" on the <http-provider>. Here is an example of an exception mapping made directly on a <http-gateway> configuration:

```
<http-gateway name="http-gateway">
    <exception>
        <mapping class="com.acme.AcmeException" status="503" />
    </exception>
</http-gateway>
```

Configuring exception mappings at the <http-provider> level is exactly the same.

You can also configure a mapping file, which is a simple .properties format file containing "{exception-class}={http-status-code}" mappings. The file is looked up on the class path and should be bundled inside your .ESB deployment. It is configured as follows (this time on the <http-provider>):

```
<http-provider name="http">

    <!-- Global exception mappings file... -->
    <exception mappingsFile="/http-exception-mappings.properties" />
</http-provider>
```

Report a bug

10.18. SECURING THE HTTP GATEWAY

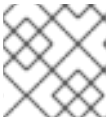
To configure security constraints, add your requisite values to the ESB Configuration's <http-provider> configuration section. (In other words, this can not be done directly on the <http-gateway> configuration.)

Report a bug

10.19. SECURE THE HTTP GATEWAY

Procedure 10.1. Task

1. Open the **jbossesb-properties.xml** file in your text editor: **vi jbossesb-properties.xml**
2. Specify a <http-bus> in the <http-provider> section of the file.
3. Reference the <http-bus> from the <http-gateway> using the "busrefid" attribute.



NOTE

See the "http-gateway" quick start for an example of a full configuration.

4. Save the file and exit.

Report a bug

10.20. FURTHER HTTP GATEWAY SECURITY

To force an end-point to use a log-in, utilise the <protected-methods> and <allowed-roles> sections of a <http-bus> configuration file:

```
<http-bus busid="secureSalesDeletes">
    <allowed-roles>
        <role name="friend" />
    </allowed-roles>
    <protected-methods>
        <method name="DELETE" />
    </protected-methods>
</http-bus>
```

The above configuration stipulates that a valid "friend" log-in is required for delete requests made on the "secureSalesDeletes" bus.

The following log-in matrix tries to illustrate which configurations will enforce a log-in, and when.

Table 10.15.

Methods Specified	Roles Specified	Log-in Required
No	No	No
No	Yes	For All Methods
Yes	Yes	For Specified Methods Only
Yes	No	No. Specified methods blocked to all.

Configure the authentication method and security domain from within the <war-security> configuration's <globals> element:

```
<http-provider name="http">
    <http-bus busid="secureFriends">
        <allowed-roles>
            <role name="friend" />
        </allowed-roles>
        <protected-methods>
            <method name="DELETE" />
        </protected-methods>
    </http-bus>

    <auth method="BASIC" domain="java:/jaas/JBossWS" />
</http-provider>
```

The method attribute can be one of "BASIC" (default), "CLIENT-CERT" or "DIGEST".

You can configure the HTTP Transport Guarantee on a per http-bus basis by specifying it on the bus using the "transportGuarantee" attribute.

```
<http-bus busid="secureFriends" transportGuarantee="CONFIDENTIAL">
    <!-- etc etc -->
</http-bus>
```

The valid values for transportGuarantee are "CONFIDENTIAL", "INTEGRAL" and "NONE".

Report a bug

10.21. APACHE CAMEL

Camel is an open source rules-based router developed by the Apache Project.

Report a bug

10.22. CAMEL GATEWAY

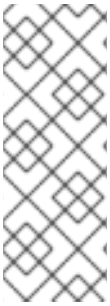
The Camel Gateway allows you to expose ESB-unaware Camel end-points. This gateway leverages Apache Camel's input capabilities, translates the Camel message into a JBoss Enterprise SOA Platform message, and invokes the associated service.

The most apparent difference between the Camel Gateway and the other Gateways provided within the JBoss Enterprise SOA Platform is that the Camel Gateway is not tied to any one type of transport.

Report a bug

10.23. CONFIGURING THE CAMEL GATEWAY

To see all the different transports Camel can handle, please visit the Camel Component list here: http://camel.apache.org/components.html



NOTE

Different Camel components have different library dependencies. The JBoss Enterprise SOA Platform only contains the **`camel-core.jar`**. You will have to add any other dependencies (including other camel-* jars or third party jars) you require into server/**`/deployers/esb.deployer/lib`** - not your ESB archive. For more information on using non-core Camel components, see http://community.jboss.org/wiki/CamelGatewayUsingNon-coreComponents/.

If you declare the use of the updated (**`jbossesb-1.3.0.xsd`**) schema, in your **`jboss-esb.xml`** file, you will see a new <camel-provider> section appear under Providers:

```
<camel-provider name="...">
<camel-bus busid="...">
 <from uri="..."/>
<from uri="..."/>
</camel-bus>
</camel-provider>
```

The most interesting part is that which is contained within the <camel-bus> element. An unbounded number of <from uri=""/> elements can be added here. Those familiar with the Camel XML configuration should be very comfortable with that element, as it does exactly what it does in the native Camel XML configuration. There is also a new <camel-gateway> element, which can reference the bus via the busidref attribute:

```
<camel-gateway name="..." busidref="..."/>
```

It is possible to define <from uri=""/> elements under the <camel-gateway> element, without using a <camel-provider> at all:

```
<camel-gateway name="...">
<from uri=""/>
<from uri=""/>
</camel-gateway>
```

There is also a short-hand mechanism, where you can specify one Camel "from" URI as an XML attribute at either the gateway level or at the bus level. Here is the gateway level:

```
<camel-gateway name="..." from-uri="..."/>
```

Now here is the bus level:

```
<camel-bus name="..." busid="..." from-uri="..."/>
```

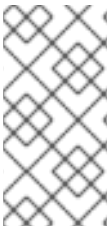
NOTE

It is important to understand that all Camel "from" URIs defined at both the <camel-bus> and <camel-gateway> level are cumulative, whether you use the element form and/or the short-hand form.

At this point, you might wonder where the <to uri=””/> elements are, because in Camel you need to define at least one destination. In the JBoss Enterprise Service Bus, every Camel "from" URI translates to one route (added to all other routes in that gateway+bus), with an implicit Camel “to” URI which invokes the associated Service where you are assigning the <camel-gateway>. Under-the-hood, all routes in that gateway+bus will end up invoking that service, and they are all run within the same CamelContext, whose lifecycle is tied to that CamelGateway's lifeycle.

NOTE

It is important to be aware of the fact that this gateway only supports Camel routes that can be defined in a single "from" URI. The basic stanza supported by the gateway is from(endpointUri).to(esbService). The gateway does not support routes that would require intermediate routing to other Camel components.



NOTE

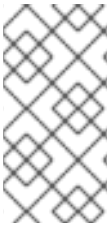
It is important to be aware of the fact that this gateway only supports Camel routes that can be defined in a single "from" URI. The basic stanza supported by the gateway is from(endpointUri).to(esbService). The gateway does not support routes that would require intermediate routing to other Camel components.

Some Camel components perform scheduled routing tasks, for example the HTTP component can be used to periodically poll a HTTP address, or the File component can poll a filesystem directory. Not all of these components support a URI option for configuring the poll frequency. The HTTP component is one such example. In these cases, you just need to prefix the component URI Scheme with "esbschedule: <frequency-in-millis>" (for example, . <from uri="esbschedule:5000:http://www.jboss.org" /> would poll jboss.org every five seconds.)

Finally, there are two other optional attributes that one can place at either the <camel-gateway> or <camel-bus> level (the gateway overriding the bus in these cases): async and timeout:

```
<camel-gateway name="..." from-uri="..." async="false" timeout="30000"/>
```

- The async attribute (defaults to “false”) says whether the underlying ServiceInvoker should invoke the associated Service synchronously or asynchronously.
- The timeout attribute (defaults to “30000”), defines how many milliseconds the ServiceInvoker should wait on a synchronous invocation before giving up.



NOTE

For more detailed information, please visit this wiki page:
http://community.jboss.org/wiki/CamelGateway

You can also study the camel_helloworld quick start.

Report a bug

10.24. TRANSITIONING FROM THE OLD CONFIGURATION MODEL TO THE NEW

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

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

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

1. 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 – all depending on how you were feeling on the day. This type of free form configuration on <listener> components is not supports on the XSD to ConfigTree mapping. In other words, 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.

Report a bug

10.25. CONFIGURING THE ENTERPRISE SERVICE BUS

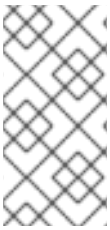
All components within the core of the product receive their configuration parameters as XML. How these parameters are provided to the system is hidden by the **`org.jboss.soa.esb.parameters.ParamRepositoryFactory`**:

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

This instruction 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 the JBoss Enterprise SOA Platform, there is only a single implementation (the **`org.jboss.soa.esb.parameters.ParamFileRepository`**) which expects to be able to load the parameters from a file. The implementation to use this may be over-ridden using the **`org.jboss.soa.esb.paramsRepository.class`** property.



NOTE

Red Hat recommends that you construct your ESB configuration file using **JBoss Developer Studio** or an XML editor of your choice. The JBossESB configuration information is supported by an annotated XSD, which should be of help if you are using a more basic editor.

Report a bug

CHAPTER 11. DATA DECODING: MIME DECODERS

11.1. MESSAGE COMPOSER

A Message Composer is a class which implements the **`MessageComposer`** interface and is responsible for constructing an ESB message instance and sending it to the associated service. The Message Coder is supported by every time of gateway.

Report a bug

11.2. MIME DECODER

A Mime Decoder is a class that implements the **`MimeDecoder`** interface. It can be used by a **`MessageComposer`** implementation to decode a binary array, transforming it into a specific type of Java Object. (The type of object is determined by the “mime type” of the binary-encoded data.)

Examples of gateways that use the MimeDecoder mechanism are the File and FTP Gateway Listeners. These gateways can be configured with ”mimeType” or “mimeDecoder” properties, which triggers automatic installation of the appropriate MimeDecoder implementation for the specified mime type.

Report a bug

11.3. IMPLEMENT A MIME DECODER

Procedure 11.1. Task

1. Create a class that activates the **`org.jboss.soa.esb.listeners.message.mime.MimeDecoder`** interface.
2. Add the @MimeType annotation to the class, specifying the mime type as the annotation value.
3. Define the newly created class in the **`META-INF/org/jboss/soa/esb/listeners/message/mime/decoders.lst`** file:

```
@MimeType("text/plain")
public class TextPlainMimeDecoder implements MimeDecoder,
Configurable {

    private Charset encodingCharset;

    public void setConfiguration(ConfigTree configTree) throws
ConfigurationException {
        AssertArgument.isNotNull(configTree, "configTree");

        String encoding = configTree.getAttribute("encoding", "UTF-
8");
        encodingCharset = Charset.forName(encoding);
    }

    public Object decode(byte[] bytes) throws MimeDecodeException {
        try {
```

```
            return new String(bytes, encodingCharset.name());
        } catch (UnsupportedEncodingException e) {
            throw new MimeDecodeException("Unexpected character
encoding error.", e);
        }
    }
}
```

NOTE

This file needs to be present on the classpath at runtime. If your module doesn't have this file, add it to your module source/resources and it will be located at runtime.

4. Optionally, if the MimeDecoder implementation needs access to the listener configuration (for additional configuration information), have the class implement the **`org.jboss.soa.esb.Configurable`** interface.

Report a bug

11.4. CONFIGTREE

The ConfigTree is the listener instance's configuration. This information is stored in the **`jboss-esb.xml`** file.

Report a bug

11.5. MIME DECODER IMPLEMENTATIONS AVAILABLE OUT-OF-THE-BOX

The JBoss Enterprise SOA Platform comes with the following MimeDecoder implementations:

text/plain

The TextPlainMimeDecoder handles “text/plain” data, decoding a byte[] to a String (default) or char[].

Table 11.1. Properties

Property	Description	Comments
encoding	Character encoding of the text/plain data encoded in the byte[].	Default “UTF-8”

Property	Description	Comments
decodeTo	How the text/plain data is to be decoded: • “STRING” (default): Decode the text/plain data to a java.lang.String. • “CHARS”: Decode the text/plain data to a char[].	Default “STRING”

Report a bug

11.6. USING MIME DECODERS IN GATEWAY IMPLEMENTATIONS

The easiest way to make a MessageComposer use the installed mime decoders is via the ConfigTree and the **`MimeDecoder.Factory`** class factory method:

```
this.mimeDecoder = MimeDecoder.Factory.getInstanceByConfigTree(config);
```

This relies on the listener configuration specifying either the “mimeType” or “mimeDecoder” configuration properties (as supported by the File and FTP listeners):

```
<fs-listener name="FileGateway1" busidref="fileChannel1" is-gateway="true"
                                 poll-frequency-seconds="10">
    <property name="mimeType" value="text/plain" />
    <property name="encoding" value="UTF-8" />
</fs-listener>
<fs-listener name="FileGateway2" busidref="fileChannel2" is-gateway="true"
                                 poll-frequency-seconds="10">
    <property name="mimeDecoder"
value="com.acme.mime.ImageJPEGMimeDecoder” />
</fs-listener>
```

To perform the actual decoding of the transport payload, the MessageComposer instance utilizes the decode method on its mimeDecoder instance:

```
Object decodedPayload = mimeDecoder.decode(payloadBytes);
```

It then sets the “decodedPayload” Object instance on the message instance being composed.

Report a bug

CHAPTER 12. WEB SERVICES SUPPORT

12.1. JBOSS WEB SERVICES

JBoss Web Services provides a JAX-WS web service stack for JBoss Enterprise Application Platform.

JAX-WS is the Java API for XML Web Services. It uses Java annotations to simplify the development of web service clients and endpoints.

Report a bug

12.2. JBOSS WEB SERVICES SUPPORT

The JBoss Enterprise SOA Platform contains several components that it uses to expose and invoke web service end-points.

1. SOAPProcessor: The SOAPProcessor action allows you to expose JBossWS 2.x and higher web service end-points through endpoints (listeners) running on the ESB (“SOAP onto the bus”). This allows you to use JBossESB to expose web service end-points (wrapper web services) for services that don't expose a web service Interface.

2. SOAPClient: The SOAPClient action allows you to make invocations on web service endpoints (“SOAP off the bus”).

Report a bug

CHAPTER 13. ACTIONS AVAILABLE FOR USE OUT OF THE BOX

13.1. OUT-OF-THE-BOX ACTIONS

Out-of-the-box actions are generic pieces of code for actions that come prepackaged with the JBoss Enterprise SOA Platform product. You can use them immediately in your services or customize them to suit your needs.

Report a bug

13.2. JBOSS ENTERPRISE SOA PLATFORM OUT-OF-THE-BOX ACTIONS

The out-of-the-box actions implemented in the SOA Platform are divided into the following functional groups:

Transformers and Converters

Use transformer and converter actions to change message data from one form to another.

Business Process Management

Use the business process management actions when integrating your software with the jBPM.

Scripting

Use scripting actions to automate tasks written in the supported scripting languages.

Services

Use service actions when integrating your code with Enterprise Java Beans.

Routing

Use routing actions when moving message data to destination services.

Notifier

Use notifier actions when sending data to ESB-unaware destinations.

Web Services/SOAP

Use web service actions when you need to support web services.

Report a bug

13.3. TRANSFORMER ACTIONS

13.3.1. Transformers

Transformers are a type of action processor that can transform a message payload from one type to another.

Report a bug

13.3.2. 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 13.1. ByteArrayToString Properties

Property	Description	Required
encoding	The binary data encoding on the message byte array. Defaults to **UTF-8**.	No

Example 13.1. Sample Configuration

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

Report a bug

13.3.3. LongToDateConverter

Input Type	**java.lang.Long**/**long**
Output Type	**java.util.Date**
Class	**org.jboss.soa.esb.actions.converters.LongToDateConverter**

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

Example 13.2. Sample Configuration

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.LongToDateConverter">
```

Report a bug

13.3.4. 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 13.2. 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.	No

Example 13.3. Sample Configuration

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

Report a bug

13.3.5. 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.

Table 13.3. ObjectToCSVString Properties

Property	Description	Required
bean-properties	List of Object bean property names used to get CSV values for the output CSV String. The Object should support a getter method for each of listed properties.	Yes
fail-on-missing-property	Flag indicating whether or not the action should fail if a property is missing from the Object, that is if the Object does not support a getter method for the property. Default value is **`false`**.	No

Example 13.4. Sample Configuration

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

Report a bug

13.3.6. 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 converts it into XML using the XStream processor.
.

Table 13.4. ObjectToXStream Properties

Property	Description	Required
class-alias	Class alias used in call to **`XStream.alias(String, Class)`** prior to serialization. Defaults to the input Object's class name.	No
exclude-package	Exclude the package name from the generated XML. Default is **`true`**. Not applicable if a class-alias is specified.	No
aliases	Specify additional aliases in order to help XStream to convert the XML elements into Objects.	No

Property	Description	Required
namespaces	Specify namespaces that should be added to the XML generated by XStream. Each namespace-uri is associated with a local-part which is the element on which this namespace should appear.	No
xstream-mode	Specify the XStream mode to use. Possible values are **XPATH_RELATIVE_REFERENCES** (the default), **XPATH_ABSOLUTE_REFERENCES**, **ID_REFERENCES** or **NO_REFERENCES**.	No
fieldAliases	Field aliases to be added to Xstream.	No
implicit-collections	Which will be registered with Xstream	No
converters	List of converters that will be registered with Xstream	No

Example 13.5. Sample Configuration

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.ObjectToXStream">
    <property name="class-alias" value="MyAlias" />
    <property name="exclude-package" value="true" />
    <property name="aliases">
        <alias name="alias1" class="com.acme.MyXXXClass1" />
        <alias name="alias2" class="com.acme.MyXXXClass2" />
        <alias name="xyz" class="com.acme.XyzValueObject"/>
        <alias name="x" class="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>
    <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-collection class="className" fieldName="fieldName"
            fieldType="fieldType" itemType="itemType"/>
        ...
    </property>
    <property name="converters">
        <converter class="className" fieldName="fieldName"
fieldType="fieldType"/>
```

```
            ...
        </property>
    </action>
```

Report a bug

13.3.7. XStreamToObject

Input Type	**`java.lang.String`**
Output Type	User Object (specified by "incoming-type" property)
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 13.5. XStreamToObject Properties

Property	Description	Required
class-alias	Class alias used during serialization. Defaults to the input Object's class name.	No
exclude-package	Flag indicating whether or not the XML includes a package name.	YES
incoming-type	Class type.	Yes
root-node	Specify a different root node than the actual root node in the XML. Takes an XPath expression.	No
aliases	Specify additional aliases to help XStream to convert the XML elements to Objects	No
attribute-aliases	Specify additional attribute aliases to help XStream to convert the XML attributes to Objects	No
fieldAliases	Field aliases to be added to Xstream.	No
implicit-collections	Which will be registered with Xstream	No
converters	Specify converters to help Xstream to convert the XML elements and attributes to Objects.	No

Example 13.6. Sample Configuration

```
<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>
```

Report a bug

13.3.8. XsltAction

This performs transformation on entire documents.

Table 13.6. `XsltAction` Properties

Property	Description	Required
get-payload-location	Message Body location containing the message payload. If unspecified the Default Payload Location is used.	NO
set-payload-location	Message Body location where result payload is to be placed. If unspecified the Default Payload Location is used.	No

Property	Description	Required
templateFile	Path to the XSL Template file. It can be defined with a file path within the deployed archive, or as a URL.	Yes
resultType	The type of Result to be set as the result Message payload. This property controls the output result of the transformation. The following values are currently available: • **`STRING`**: will produce a **`String`**. • **`BYTES`**: will produce a array of bytes, **`byte[]`**. • **`DOM`**: will produce a **`DOMResult`**. • **`SAX`**: will produce a **`SAXResult`**. • **`SOURCERESULT`** can be used to produce a customised result if the above do not suit your needs. When the message payload contains a **`SourceResult`** object (**`org.jboss.soa.esb.actions.transformation.xslt.SourceResult`**) this produces a result of the same type as the **`result`** attribute of the payload's **`SourceResult`** object. When the message payload is a **`SourceResult`** object and resultType is not set to **`SOURCERESULT`**, the result is returned as the type specified in resultType. The developer is responsible for ensuring the types are compatible.	No
failOnWarning	If **`true`** will cause a transformation warning to cause an exception to be thrown. If **`false`** the failure will be logged. Defaults to True.	No
uriResolver	Fully qualified class name of a class that implements **`URIResolver`**. This will be set on the tranformation factory.	No
factory.feature.*	Factory features that will be set for the tranformation factory. The feature name, which are fully qualified URIs, should be specified after the factory.feature. prefix. E.g. factory.feature.http://javax.xml.XMLConstants/feature/secure-processing	No
Factory.attribute.*	Factory attributes that will be set for the tranformation factory. The attribute name should be specified after the factory.attribute. prefix. E.g. factory.attribute.someVendorAttributename	NO
validation	If true will cause an invalid source document to cause an exception to be thrown. If false validation will not occur, although well-formed documents are enforced. . Default value is **`false`** .	No

Property	Description	Required
schemaFile	The input schema file (XSD) to use, located on the classpath. .	No
schemaLanguage	The input schema language to use.	No

Report a bug

13.3.9. Validating XsltActions

There are several different ways to configure the XsltAction validation. These are listed here with examples:

1. Disabled (the default)

   This can be explicitly configured to **`false`** or omitted to disable validation.

   ```
   <property name="validation" value="false"/>
   ```

2. DTD

   ```
   <property name="validation" value="true"/>
   <property name="schemaLanguage" value="http://www.w3.org/TR/REC-
   xml"/>
   ```

   Alernatively:

   ```
   <property name="validation" value="true"/>
   <property name="schemaLanguage" value=""/>
   ```

3. W3C XML Schema or RELAX NG

   ```
   <property name="validation" value="true"/>
   ```

   Alternatively:

   ```
   <property name="validation" value="true"/>
   <property name="schemaLanguage"
   value="http://www.w3.org/2001/XMLSchema"/>
   ```

   or

   ```
   <property name="validation" value="true"/>
   <property name="schemaLanguage"
   value="http://relaxng.org/ns/structure/1.0"/>
   ```

4. W3C XML Schema or RELAX NG with included schemaFile

```
<property name="validation" value="true"/>
<property name="schemaFile" value="/example.xsd"/>
```

or

```
<property name="validation" value="true"/>
<property name="schemaLanguage"
value="http://www.w3.org/2001/XMLSchema"/>
<property name="schemaFile" value="/example.xsd"/>
```

Aleternatively:

```
<property name="validation" value="true"/>
<property name="schemaLanguage"
value="http://relaxng.org/ns/structure/1.0"/>
<property name="schemaFile" value="/example.rng"/>
```

Depending on whether or not validation is enabled there are several different outcomes to an XsltAction:

1. If the XML is well-formed and valid:
   - The transformation will be performed.
   - The pipeline will continue.
2. If the XML is malformed:
   - an error will be logged
   - SAXParseException -> ActionProcessingException
   - pipeline stops
3. If the XML is well-formed but invalid:
   - If validation is not enabled:
     - The transformation may fail.
     - The pipeline will continue.
   - If validation is enabled:
     - an error will be logged
     - SAXParseException -> ActionProcessingException
     - pipeline stops

Report a bug

13.3.10. Smooks

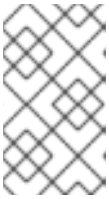
Smooks is a fragment-based data transformation and analysis tool. It is a general purpose processing

tool capable of interpreting fragments of a message. It uses visitor logic to accomplish this. It allows you implement your transformation logic in XSLT or Java and provides a management framework through which you can centrally manage the transformation logic for your message-set.

Report a bug

13.3.11. Using Smooks

- Use the **`SmooksAction`** component to "plug" Smooks into an ESB action pipeline.

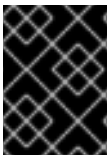


NOTE

You will find a number of quick-starts that demonstrate transformations in the **`samples/quick starts`** directory. (The name of each transformation of these quick starts is prefixed with the word **`transform_`**.)

Report a bug

13.3.12. SmooksTransformer



IMPORTANT

The SmooksTransformer action will be deprecated in a future release. Refer to SmooksAction for a more general purpose and more flexible Smooks action class.

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

The SmooksTransformer component supplies the JBoss Enterprise SOA Platform with message transformation functionality. This is an action component that allows the Smooks Data Transformation/Processing Framework to be plugged into an action pipeline.

A wide range of source and target data formats are supported by the SmooksTransformer component.

Table 13.7. SmooksTransformer Resource Configuration

Property	Description	Required
resource-config	The **`Smooks`** resource configuration file.	Yes

Table 13.8. SmooksTransformer Message Profile Properties (Optional)

Property	Description	Required
from	Message Exchange Participant name. Message Producer.	No
from-type	Message type/format produced by the "from" message exchange participant.	No

Property	Description	Required
to	Message Exchange Participant name. Message Consumer.	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 (via the **`Message.Properties`** class).

Example 13.7. Sample Configuration: 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 13.8. Sample Configuration: 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 13.9. Sample Configuration: Using Message Proiles

```
<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>
```

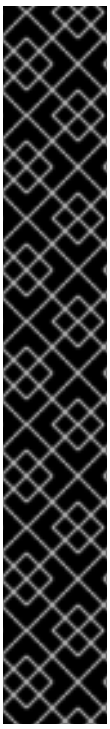
Java objects are bound to the **`Message.Body`** under their beanId.

Report a bug

13.3.13. SmooksAction

The **`org.jboss.soa.esb.smooks.SmooksAction`** class is the second generation action class for executing Smooks processes. (Note that it can do more than just transform messages).

The SmooksAction class can process a wider range of message payloads by using Smooks PayloadProcessor. This includes strings, byte arrays, InputStreams, readers, POJOs and more. As such, it can perform a wide range of transformations including Java-to-Java transformations. It can also perform other types of operations on a source messages stream, including content based payload Splitting and Routing (note that this is not the same as ESB message routing). The SmooksAction enables the full range of Smooks capabilities from within JBoss Enterprise SOA Platform.



IMPORTANT

Be aware that Smooks does not detect (and report errors on) certain types of configuration errors for resource configurations made through the base <resource-config>. If, for example, the resource (<resource>) is a Smooks Visitor implementation, and you misspell the name of the Visitor class, Smooks will not raise this as an error simply because it doesn't know that the misspelling was supposed to be a class. Remember, Smooks supports lots of different types of resource and not just Java Visitor implementations.

The easiest way to avoid this issue is to use the extended Smooks configuration namespaces for all out-of-the-box functionality. For example, instead of defining Java binding configurations by defining org.milyn.javabean.BeanPopulator <resource-config> configurations, use the http://www.milyn.org/xsd/smooks/javabean-1.2.xsd configuration namespace (that is, the <jb:bean> config, and so forth).

If you have implemented Smooks Visitor functionality, the easiest way to avoid this issue is to define an extended configuration name-space for this new resource type. This also has the advantage of making the new resource easier to configure as you can leverage the schema support built into JBoss Developer Studio.

The following illustrates the basic SmooksAction configuration:

Example 13.10. SmooksAction

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

Here are the optional configuration properties:

Table 13.9. 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

Property	Description	Default
mappedContextObjects	Comma separated list of Smooks ExecutionContext objects to be mapped into the EXECUTION_CONTEXT_ATTR_MAP_KEY Map on the ESB Message Body. Default is an empty list. Objects must be Serializable.	
resultType	The type of Result to be set as the result Message payload.	STRING
javaResultBeanId	Only relevant when **`resultType=JAVA`** The Smooks bean context beanId to be mapped as the result when the resultType is "JAVA". If not specified, the whole bean context bean Map is mapped as the JAVA result.	
reportPath	The path and file name for generating a Smooks Execution Report. This is a development aid and should not to be used in production.	

The SmooksAction uses the **`MessagePayloadProxy`** class to obtain and set the payload onto the message. Therefore, unless otherwise configured via the get-payload-location and set-payload-location action properties, the SmooksAction obtains and sets the Message payload on the default message location by using the Message.getBody().get() and Message.getBody().set(Object) methods.

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 and Java-to-Java transformations.

Report a bug

13.3.14. Use SmooksAction to Process XML, EDI, CSV and "Other Type" Message Payloads

Procedure 13.1. Task

1. Supply a source message. It must be one of the following:
   - String,
   - InputStream,
   - Reader, or
   - byte array
2. Configure Smooks (via the Smooks configuration file, not the Enterprise Service Bus configuration file) for processing the message type in question. For example, configure a parser if you are not dealing with an XML Source (such as EDI or CSV).

Report a bug

13.3.15. Specifying the SmooksAction Result Type

Because the Smooks Action can produce a number of different types of result, you need to be able to specify which one you want. The result you choose is then added back to the message payload.

The default ResultType is "STRING". You can change it to "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 message payload location.

Here is some sample code that binds the "order" bean from the Smooks bean context onto the message as its 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>
```

Report a bug

13.3.16. PersistAction

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

This is used to interact with the MessageStore when necessary.

Table 13.10. PersistAction Properties

Property	Description	Required
classification	This is used to classify where the Message will be stored. If the Message Property **`org.jboss.soa.esb.messagestore.classification`** is defined on the message, it 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 the Action is to be used to terminate a pipeline then this should be "true" (the default). If not, then set this to "false" and the input message will be returned from processing.	No

```
<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>
```

Report a bug

13.4. BUSINESS PROCESS MANAGEMENT ACTIONS

13.4.1. jBPM

The JBoss Business Process Manager (jBPM) is a workflow management tool that provides the user with control over business processes and languages. jBPM 3 is used as default.

Report a bug

13.4.2. JBPM Integration

The **JBoss Business Process Manager** is a work-flow and *business process management* engine.

Report a bug

13.4.3. 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`**

The JBoss Enterprise SOA Platform can make calls into the jBPM using the BpmProcessor action. The BpmProcessor action uses the jBPM command API to make calls into jBPM.

The following jBPM commands have been implemented:

- NewProcessInstanceCommand
- StartProcessCommand
- CancelProcessInstanceCommand
- GetProcessInstanceVariablesCommand

Table 13.11. BpmProcessor Properties

Property	Description	Required
command	The jBPM command being invoked. *Required* Allowable values: • NewProcessInstanceCommand • StartProcessInstanceCommand • SignalCommand • CancelProcessInstanceCommand	Yes
processdefinition	Required property for the New- and Start-ProcessInstanceCommands if the process-definition-id property is not used. The value of this property should reference a process definition that is already deployed to jBPM and of which you want to create a new instance. This property does not apply to the Signal- and CancelProcessInstance-Commands.	Depends
process-definition-id	Required property for the New- and Start-ProcessInstanceCommands if the processdefinition property is not used. The value of this property should reference a processdefintion id in jBPM of which you want to create a new instance. This property does not apply to the Signal- and CancelProcessInstance commands.	Depends
actor	Optional property to specify the jBPM actor id, which applies to the New- and StartProcessInstanceCommands only.	No
key	Optional property to specify the value of the jBPM key. For example one can pass a unique invoice id as the value for this key. On the jBPM side this key is as the “business” key id field. The key is a string based business key property on the process instance. The combination of business key and process definition must be unique if a business key is supplied. The key value can hold an MVEL expression to extract the desired value from the EsbMessage. For example if you have a named parameter called “businessKey” in the body of your message you would use “body.businessKey”. Note that this property is used for the New- and StartProcessInstanceCommands only	No
transition-name	Optional property. This property only applies to the StartProcessInstance- and Signal Commands, and is of use only if there are more then one transition out of the current node. If this property is not specified, the default transition out of the node is taken. The default transition is the first transition in the list of transition defined for that node in the jBPM processdefinition.xml.	No

Property	Description	Required
esbToBpmVars	Optional property for the New- and StartProcessInstanceCommands and the SignalCommand. This property defines a list of variables that need to be extracted from the EsbMessage and set into jBPM context for the particular process instance. The list consists of mapping elements. Each mapping element can have the following attributes: **esb** required attribute which can contain an MVEL expression to extract a value anywhere from the message. **bpm** optional attribute containing the name which be used on the jBPM side. If omitted the esb name is used. **default** optional attribute which can hold a default value if the esb MVEL expression does not find a value set in the EsbMessage.	No

Report a bug

13.5. SCRIPTING ACTIONS

13.5.1. Scripting Actions

Scripting actions are actions you can add to your service's pipeline. Once you add one, you can use scripting languages to define your action processing logic.

Report a bug

13.5.2. Groovy

Groovy is an agile and dynamic language for the Java Virtual Machine that builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk.

Refer to http://groovy.codehaus.org/ for more information.

Report a bug

13.5.3. GroovyActionProcessor

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

This action executes a Groovy action processing script, receiving the message, payloadProxy, action configuration and logger as variable input.

Table 13.12. GroovyActionProcessor Properties

Property	Description	Required
script	Path (on classpath) to Groovy script.	
supportMessageBasedScripting	Allow scripts within the message.	
cacheScript	Should the script be cached. Defaults to `true`.	No

Table 13.13. GroovyAction Processor Script Binding Variables

Variable	Description
message	The Message
payloadProxy	Utility for message payload (MessagePayloadProxy).
config	The action configuration (ConfigTree).
logger	The GroovyActionProcessor's static Log4J logger (Logger). The logging category is jbossesb.<esb_archive_name>.<category>.<service>

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

Report a bug

13.5.4. Bean Scripting Framework (BSF)

The Bean Scripting Framework is an open source scaffolding developed by the Apache Foundation. It lets you insert scripts into Java code.

Report a bug

13.5.5. ScriptingAction

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

Executes a script using the Bean Scripting Framework, receiving the message, payloadProxy, action configuration and logger as variable input.

1. The Bean Scripting Framework does not provide an API to precompile, cache and reuse scripts. Because of this, each execution of the ScriptingAction will go through the compile step again. Please keep this in mind while evaluating your performance requirements.

2. When including BeanShell scripts in your application, Red Hat advises you should use a .beanshell extension instead of .bsh, otherwise the JBoss BSHDeployer might pick it up.

Table 13.14. ScriptingAction Properties

Property	Description	Required
script	Path (on classpath) to script.	
supportMessageBasedScripting	Allow scripts within the message.	
language	Optional script language (overrides extension deduction).	No

Table 13.15. 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)

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

Report a bug

13.6. SERVICE ACTIONS

13.6.1. Service Actions

Service Actions are actions defined within the JBoss Enterprise SOA Platform's Enterprise Service Bus.

Report a bug

13.6.2. 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 13.16. EJBProcessor Properties

Property	Description	Required
ejb3	When calling 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 call.	
lazy-ejb-init	Whether EJBs should be lazily initialised at runtime rather than at deploy time. Default is false.	No
ejb-params	The list of parameters to use when calling the method and where in the input Message they reside.	
esb-out-var	The location of the output. Default value is DEFAULT_EJB_OUT.	No

Example 13.11. 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 13.12. 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>
```

Report a bug

13.7. ROUTING ACTIONS

13.7.1. Routing Actions

Routing actions are actions that you can add to your service's pipeline. They perform conditional routing of messages between two or more message exchange participants.

Report a bug

13.7.2. Aggregator

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

This is a message aggregation action. It is an implementation of the aggregator enterprise integration pattern. (see 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 the ContentBasedRouter and StaticRouter actions.

The data has the following format:

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

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

Table 13.17. Aggregator Properties

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

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

Report a bug

13.7.3. Streaming Aggregator

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

This action allows invocation of external (ESB-unaware) HTTP end-points from an ESB action pipeline. This action uses the Apache Commons HttpClient. An implementation of the Aggregator Enterprise Integration Pattern can be viewed here: http://www.enterpriseintegrationpatterns.com/Aggregator.html

The Streaming Aggregator is an improved version of the message aggregation action. Unlike the previous aggregator, the streaming aggregator does not require all messages to have complete aggregation details - messages must have the message order number and a unique aggregation id, but all messages do not need to specify how many messages will be aggregated in each message. The number of messages aggregated can be sent in a subsequent message, which is a performance improvement when dealing with extremely large files which need to be line counted or parse, or Smooks fragments which need to be split.

Data is set on the message as a property called "Aggregate.AggregateDetails" which sets should contain a **`org.jboss.soa.esb.actions.aggregator.AggregateDetails`** object.

The data has the following format:

```
[SeriesUUID] ":" [message-sequence] ":" [sequence-count]
```

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

Table 13.18. Aggregator Properties

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

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

Report a bug

13.7.4. EchoRouter

This action echoes the incoming message payload to the information log stream and returns the input message from the process method.

Report a bug

13.7.5. HttpRouter

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

This action allows invocation of external HTTP end-points from an action pipeline. This action uses Apache Commons HttpClient.

Table 13.19. Apache Commons HttpRouter

Property	Description	Required
unwrap	Setting this to **`true`** (the default) will extract the message payload from the Message object before sending. **`false`** will send the serialized Message as either XML or Base64 encoded JavaSerialized object, based on the MessageType.	No
endpointUrl	The endpoint to which the message will be forwarded.	Yes
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.	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

Property	Description	Required
headers	To be added to the request. Supports multiple <header name="test" value="testvalue" /> elements.	No
MappedHeaderList	A comma separated list of header names that should be propagated to the target endpoint. The value for the headers will be retrieved from those present on a request entering the enterprise service bus via the http-gateway or within the properties of the current message.	No

```
<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>
```

Report a bug

13.7.6. Java Message Service

A Java Message Service (JMS) is a Java API for sending messages between two clients. It allows the different components of a distributed application to communicate with each other and thereby allows them to be loosely coupled and asynchronous. There are many different Java Message Service providers available. Red Hat recommends using HornetQ.

Report a bug

13.7.7. JMSRouter

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

Routes the incoming message to the Java Message Service.

Table 13.20. JMSRouter

Property	Description	Required
unwrap	Setting this to **`true`** will extract the message payload from the Message object before sending. **`false`** (the default) will send the serialized Message object.	No

Property	Description	Required
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, **`true`** (the default) or **`false`**.	No
priority	The JMS priority to be used. Default is **`javax.jms.Message.DEFAULT_PRIORITY`**.	No
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	Properties to be set on the message are prefixed with **`message-prop`**.	No
jndi-prefixes	A comma separated String of of prefixes. Properties that have these prefixes will be added to the JNDI environment.	No

Report a bug

13.7.8. EmailRouter

Class	**`org.jboss.soa.esb.actions.routing.email.EmailRouter`**

Routes the incoming message to a configured email account.

Table 13.21. EmailRouter Properties

Property	Description	Required

Property	Description	Required
unwrap	true will extract the message payload from the Message object before sending. false (the default) will send the serialized Message object.	
host	The host name of the SMTP server. If not specified will default to the property 'org.jboss.soa.esb.mail.smtp.host' in jbossesb-properties.xml.	
port	The port for the SMTP server. If not specified will default to the property 'org.jboss.soa.esb.mail.smtp.port' in jbossesb-properties.xml.	
username	The username for the SMTP server. If not specified will default to the property 'org.jboss.soa.esb.mail.smtp.user' in jbossesb-properties.xml.	
password	The password for the above username on the SMTP server. If not specified will default to the property 'org.jboss.soa.esb.mail.smtp.password' in jbossesb-properties.xml.	
auth	If true will attempt to authenticate the user using the AUTH command. If not specified will default to the property 'org.jboss.soa.esb.mail.smtp.auth' in jbossesb-properties.xml	
from	The from email address.	
sendTo	The destination email account.	
subject	The subject of the email.	
messageAttachmentName	filename of an attachment containing the message payload (optional). If not specified the message payload will be included in the message body.	
message	a string to be prepended to the ESB message contents which make up the e-mail message (optional)	
ccTo	comma-separated list of email addresses (optional)	
attachment	Child elements that contain files that will be added as attachments to the email sent.	

```
<action name="send-email"
class="org.jboss.soa.esb.actions.routing.email.EmailRouter">
    <property name="unwrap" value="true" />
    <property name="host" value="smtpHost" />
    <property name="port" value="25" />
    <property name="username" value="smtpUser" />
    <property name="password" value="smtpPassword" />
```

```
    <property name="from" value="jbossesb@xyz.com" />
    <property name="sendTo" value="system2@xyz.com" />
    <property name="subject" value="Message Subject" />
</action>
```

Report a bug

13.7.9. Content-Based Router

Content-based routers send messages that do not have destination addresses to their correct end-points. Content-based routing works by applying a set of rules (which can be defined within XPath or Drools notation) to the body of the message. These rules ascertain which parties are interested in the message. This means the sending application does not have to supply a destination address.

A typical use case is to serve priority messages in a high priority queue. The advantage here is that the routing rules can be changed on-the-fly while the service runs if it is configured in that way. (However, this has significant performance drawbacks.)

Other situations in which a content-based router might be useful include when the original destination no longer exists, the service has moved or the application simply wants to have more control over where messages go based on its content of factors such as the time of day.

Report a bug

13.7.10. The RegexProvider

The *RegexProvider* utilises regular expressions. It allows you to define low-level rules specific to the format of selected data fields in the message being filtered. For example, it may be applied to a particular field to validate the syntax to make sure the right e-mail address is being used.

Report a bug

13.7.11. XPath Domain-Specific Language

NOTE

You may find it convenient to undertake an XPath-based evaluation of XML-based messages. Red Hat supports this by shipping a domain-specific language implementation. Use this implementation to add XPath expressions to the rule file.

1. First, define the expressions in the **`XPathLanguage.dsl`** file and use the following code to reference it in the rule set:

   ```
   expander XPathLanguage.dsl
   ```

2. The XPath Language makes sure the message is in **`JBOSS_XML`** and that the following items have been defined:

   1. **`xpathMatch`**<*element*> : this yields **`true`** if an element by this name is matched.

2. **`xpathEquals`***<element>* , *<value>* : this yields **`true`** if the element is found and its value equals the value.
3. **`xpathGreaterThan`***<element>* , *<value>* : this yields **`true`** if the element is found and its value is greater than the value.
4. **`xpathLessThan`***<element>* , *<value>* : this yields **`true`** if the element is found and its value is lower then the value.

NOTE

The **`fun_cbr`** quick-start demonstrates this use of XPath.

NOTE

It is possible to define a completely different domain-specific language.

Report a bug

13.7.12. ContentBasedRouter

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

Content based message routing action.

This action supports the following routing rule provider types:

- XPath: Simple XPath rules, defined inline on the action, or externally in a .properties format file.
- Drools: Drools rules files (DSL). Out of the box support for an XPath based DSL.

Table 13.22. ContentBasedRouter Properties

Property	Description	Required
cbrAlias	Content Based Routing Provider alias. Supported values are "Drools" (default), "Xpath" and "Regex".	
ruleSet	Externally defined rule file. It will be a Drools DSL file if the Drools rule provider is in use, or a .properties rule file if the XPath or Regex provider is in use.	
ruleLanguage	CBR evaluation Domain Specific Language (DSL) file. Only relevant for the Drools rule provider.	
ruleReload	Flag indicating whether or not the rules file should be reloaded each time. Default is “false”.	

Property	Description	Required
ruleAuditType	Optional property to have Drools perform audit logging. The log can be read into the Drools Eclipse plugin and inspected. Valid values are CONSOLE, FILE and THREADED_FILE. The default is that no audit logging will be performed.	
ruleAuditFile	Optional property to define the filepath for audit logging. Only applies to FILE or THREADED_FILE ruleAuditType. The default is "event". Note that JBoss Drools will append ".log" for you. The default location for this file is "." - the current working directory (which for JBoss is in its bin/ directory).	
ruleAuditInterval	Optional property to define how often to flush audit events to the audit log. Only applies to the THREADED_FILE ruleAuditType. The default is 1000 (milliseconds).	

destinations	Container property for the <route-to> configurations. If the rules are defined externally, this configuration will have the following format:
	`<route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/>`
	If the rules are defined inline in the configuration, this configuration will have the following format (not supported for the Drools provider):
	`<route-to service-category="ExpressShipping" service-name="ExpressShippingService" expression="/order[@statusCode='2']" />`	

namespaces	Container property for the <namespace> configurations where required (for example, for the XPath ruleprovider). The <namespace> configurations have the following format: `<namespace prefix="ord" uri="http://acme.com/order" />`	

Table 13.23. ContentBasedRouter "process" methods

Property	Description	Required
process	Do not append aggregation data to the message.	
split	Append aggregation data to the message.	

Example 13.13. Sample Configuration XPATH (inline)

```
<action process="split" name="ContentBasedRouter"
    class="org.jboss.soa.esb.actions.ContentBasedRouter">
```

```
    <property name="cbrAlias" value="XPath"/>
    <property name="destinations">
        <route-to service-category="ExpressShipping"
            service-name="ExpressShippingService"
expression="/order['status='1']" />
        <route-to service-category="NormalShipping"
            service-name="NormalShippingService"
expression="/order['status='2']" />
    </property>
</action>
```

Example 13.14. Sample Configuration XPATH (external)

```
<action process="split" name="ContentBasedRouter"
    class="org.jboss.soa.esb.actions.ContentBasedRouter">
    <property name="cbrAlias" value="XPath"/>
    <property name="ruleSet" value="xpath-rules.properties"/>
    <property name="ruleReload" value="true"/>
    <property name="destinations">
        <route-to destination-name="express" service-
category="ExpressShipping"
            service-name="ExpressShippingService"/>
        <route-to destination-name="normal" service-
category="NormalShipping"
            service-name="NormalShippingService"/>
    </property>
</action>
```

Regex is configured in exactly the same way as XPath. The only difference is the expressions are Regex expressions (instead of XPath expressions).

Report a bug

13.7.13. StaticRouter

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

Static message routing action. This is basically a simplified version of the content-based router, except it does not support content based routing rules.

Table 13.24. StaticRouter Properties

Property	Description

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

Table 13.25. StaticRouter Process Methods

method	Description
process	Do not append aggregation data to message.
split	Append aggregation data to message.

```
<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>
```

Report a bug

13.7.14. SyncServiceInvoker

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

Synchronous message routing action. This action makes a synchronous invocation on the configured service and passes the invocation response back into the action pipeline for processing by subsequent actions (if there are any), or as the response to if the service is a RequestResponse service.

Table 13.26. SyncServiceInvoker Properties

Property	Description	Required
service-category	Service Category.	Yes
service-name	Service Name.	Yes

Property	Description	Required
failOnException	Should the action fail on an exception from the target service invocation. If set to "false", the action will simply return the input message to the pipeline, allowing the service to continue processing. If you need to know the failure state, leave this parameter set to true and use the normal "faultTo" mechanism by allowing the pipeline to fail (default is "true").	No
suspendTransaction	This action will fail if executed in the presence of an active transaction. The transaction can be suspended if this property is set to "true". Default is "false".	No
ServiceInvokerTimeout	Invoker timeout in milliseconds. In the event of a timeout, an exception will occur, causing the action to behave according to the "failOnException" configuration. The default is 30000.	No

```
<action name="route" class="org.jboss.soa.esb.actions.SyncServiceInvoker">
    <property name="service-category" value="Services" />
    <property name="service-name" value="OM" />
</action>
```

Report a bug

13.7.15. StaticWireTap

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

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

Table 13.27. StaticWireTap Properties

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

Table 13.28. StaticWireTap Process Methods

method	Description
process	Do not append aggregation data to message.

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

Report a bug

13.7.16. E.-Mail WireTap

Class	**org.jboss.soa.esb.actions.routing.email.EmailWiretap**

Table 13.29. E.-Mail WireTap Properties

Property	Description
host	The host name of the SMTP server. If not specified, will default to the property 'org.jboss.soa.esb.mail.smtp.host' in jbossesb-properties.xml.
port	The port for the SMTP server. If not specified, will default to the property 'org.jboss.soa.esb.mail.smtp.port' in jbossesb-properties.xml.
username	The username for the SMTP server. If not specified, will default to the property 'org.jboss.soa.esb.mail.smtp.user' in jbossesb-properties.xml.
password	The password for the above username on the SMTP server. If not specified, will default to the property 'org.jboss.soa.esb.mail.smtp.password' in jbossesb-properties.xml.
auth	If true will attempt to authenticate the user using the AUTH command. If not specified, will default to the property 'org.jboss.soa.esb.mail.smtp.auth' in jbossesb-properties.xml.
from	The "from" email address.
sendTo	The destination email account.
subject	The subject of the email.
messageAttachmentName	The filename of an attachment containing the message payload (optional). If not specified the message payload will be included in the message body.

Property	Description
message	A string to be prepended to the ESB message contents which make up the e-mail message (optional).
ccTo	Comma-separated list of email addresses (optional).
attachment	Child elements that contain file that will be added as attachments to the email sent.

```
<action name="send-email"
class="org.jboss.soa.esb.actions.routing.email.EmailWiretap">
    <property name="host" value="smtpHost" />
    <property name="port" value="25" />
    <property name="username" value="smtpUser" />
    <property name="password" value="smtpPassword" />
    <property name="from" value="jbossesb@xyz.com" />
    <property name="sendTo" value="systemX@xyz.com" />
    <property name="subject" value="Important message" />
</action>
```

Report a bug

13.8. NOTIFIER ACTIONS

13.8.1. Notifier Action

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

The Notifier is an action which does no processing of the message during the first stage of the action pipeline's processing. Rather, it sends the specified notifications during the second stage.

Report a bug

13.8.2. Notifier

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

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

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

To be notified 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.

```
<action name="notify" 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>
```

Report a bug

13.8.3. NotifyConsole

Class	**NotifyConsole**

Performs a notification by printing out the contents of the ESB message on the console.

Example 13.15. NotifyConsole

```
<target class="NotifyConsole" />
```

Report a bug

13.8.4. NotifyFiles

Class	**NotifyFiles**
Purpose	Performs a notification by writing the contents of the ESB message to a specified set of files.
Attributes	none
Child	file

Child Attributes	• **append** – if value is true, append the notification to an existing file • **URI** – any valid URI specifying a file

```
<target class="NotifyFiles" >
<file append="true" URI="anyValidURI"/>
<file URI="anotherValidURI"/>
</target>
```

Report a bug

13.8.5. NotifySqlTable

Class	**NotifySqlTable**
Purpose	Performs a notification by inserting a record into an existing database table. The database record contains the ESB message contents and, optionally, other values specified using nested <column> elements.
Attributes	• driver-class • connection-url • user-name • password • table - table in which notification record is stored. • dataColumn - name of table column in which ESB message contents are stored.
Child	column
Child Attributes	• **name** – name of table column in which to store additional value • **value** – value to be stored

```
<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>
```

Report a bug

13.8.6. NotifyQueues

Class	**NotifyQueues**
Purpose	Performs a notification by translating the ESB message (including its attached properties) into a JMS message and sending the JMS message to a list of Queues. Additional properties may be attached using the <messageProp> element.
Attributes	none
Child	queue
Child Attributes	• **jndiName** – the JNDI name of the Queue. **Required**. • **jndi-URL** – the JNDI provider URL **Optional**. • **jndi-context-factory** - the JNDI initial context factory **Optional**. • **jndi-pkg-prefix** – the JNDI package prefixes **Optional**. • **connection-factory** - the JNDI name of the JMS connection factory. **Optional**, defaults to **ConnectionFactory**.
Child	messageProp
Child Attributes	• **name** - name of the new property to be added • **value** - value of the new property

```
<target class="NotifyQueues" >
<messageProp name="aNewProperty" value="theValue"/>
<queue jndiName="queue/quickstarts_notifications_queue" />
</target>
```

Report a bug

13.8.7. NotifyTopics

Class	**NotifyTopics**
Purpose	Performs a notification by translating the ESB message (including its attached properties) into a JMS message and publishing the JMS message to a list of Topics. Additional properties may be attached using the <messageProp> element.

Attributes	none
Child	topic
Child Attributes	• **jndiName** – the JNDI name of the Queue. **Required** • **jndi-URL** – the JNDI provider URL. **Optional** • **jndi-context-factory** - the JNDI initial context factory. **Optional** • **jndi-pkg-prefix** – the JNDI package prefixes. **Optional** • **connection-factory** - the JNDI name of the JMS connection factory. **Optional**, default is **ConnectionFactory**
Child	messageProp
Child Attributes	• **name** - name of the new property to be added • **value** - value of the new property

```
<target class="NotifyTopics" >
<messageProp name="aNewProperty" value="theValue"/>
<queue jndiName="queue/quickstarts_notifications_topic" />
</target>
```

Report a bug

13.8.8. NotifyEmail

Class	**NotifyEmail**
Purpose	Sends a notification e-mail containing the ESB message content and, optionally, any file attachments.
Attributes	None.
Child	Topic.

Child Attributes	• **`from`** – email address (**`javax.email.InternetAddress`**). **`Required`** • **`sendTo`** – comma-separated list of email addresses. **`required`** • **`ccTo`** - comma-separated list of email addresses. **`Optional`** • **`subject`** – email subject. **`Required`** • **`message`** - a string to be prepended to the ESB message contents which make up the e-mail message. **`Optional`**
Child	Attachment. **`Optional`**
Child Text	The name of the file to be attached.

```
<target class="NotifyEmail" from="person@somewhere.com"
sendTo="person@elsewhere.com"
subject="theSubject">
<attachment>attachThisFile.txt</attachment>
</target>
```

Report a bug

13.8.9. NotifyFTP

Class	**`NotifyFTP`**
Purpose	Performs a notification by creating a file containing the ESB message content and transferring it via FTP to a remote file system.
Attributes	None.
Child	FTP
Child Attributes	• **`URL`** – a valid FTP URL • **`filename`** – the name of the file to contain the ESB message content on the remote system

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

Report a bug

13.8.10. NotifyFTPList

Class	**`NotifyFTPList`**
Purpose	NotifyFTPList extends NotifyFTP and adds the ability to take a single file name or list of file names located in the ESB Message object. The file(s) in the Message payload should contain a list of files (full paths). This list will be iterated over and every file in the list will be sent to the configured destination FTP server directory if the "listFiles" property is false. If "listFiles" is true, the file(s) are read line by line, with each line containing the name of a file to be transferred. So, you can supply: 1. A single file to be transferred. (single String payload with listFiles = false) 2. A list of files to be transferred. (List<String> payload with listFiles = false) 3. A single list file of files to be transferred. (single String payload with listFiles = true) 4. A list of list files of files to be transferred. (List<String> payload with listFiles = true)
Attributes	None.
Child	FTP.
Child Attributes	• **`URL`** – a valid FTP URL • **`filename`** – the name of the file to contain the ESB message content on the remote system • **`listFiles`** – true if the file(s) named in the message payload is/are list file(s), otherwise false. Default is false. • **`deleteListFile`** – true if the list file is to be deleted, otherwise false. Default is false.

```
<target class="NotifyFTPList">
    <ftp URL="ftp://username:password@localhost/outputdir"
        filename="{org.jboss.soa.esb.gateway.file}">
        listFiles="true"
        deletelistFile="true"
</target>
```

Report a bug

13.8.11. NotifyTCP

Class	**NotifyTCP**
Purpose	Send message via TCP. Each connection is maintained only for the duration of the notification. Only supports sending of string data payloads explicitly (as a String, or encoded as a byte array (byte[]).
Attributes	None.
Child	Destination (supports multiple destinations).
Child Attributes	• **URI** – The TCP address to which the data is to be written. Default port is 9090.

```
<target class="NotifyTcp" >
  <destination URI="tcp://myhost1.net:8899" />
  <destination URI="tcp://myhost2.net:9988" />
</target>
```

Report a bug

13.9. SOAP CLIENT ACTIONS

13.9.1. Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) is a lightweight protocol that enables the user to define the content of a message and to provide hints as to how recipients should process that message. SOAP is an XML-based communication protocol.

Report a bug

13.9.2. SOAPProcessor

The SOAPProcessor is an action that allows you to invoke a JBossWS-hosted web service end-point through any JBossESB-hosted listener. Via this action, the enterprise service bus can expose web service end-points for services that do not already expose them. You can then invoke services over any transport channel supported by the enterprise service bus, such as HTTP, FTP or JMS.

The SOAPProcessor supports both the JBossWS-Native and JBossWS-CXF stacks.

Report a bug

13.9.3. SOAPProcessor Action Configuration

The SOAPProcessor action action requires only one mandatory property value, which is the "jbossws-endpoint" property. This property names the JBossWS endpoint that the SOAPProcessor is exposing (invoking).

```
<action name="JBossWSAdapter"
class="org.jboss.soa.esb.actions.soap.SOAPProcessor">
    <property name="jbossws-endpoint" value="ABI_OrderManager" />
    <property name="jbossws-context" value="ABIV1OrderManager_war" />
    <property name="rewrite-endpoint-url" value="true" />
</action>
```

jbossws-endpoint

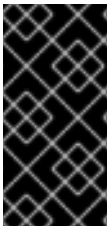
This is the JBossWS endpoint that the SOAPProcessor is exposing. Mandatory.

jbossws-context

This optional property is the context name of the Webservice's deployment and can be used to uniquely identify the JBossWS endpoint.

rewrite-endpoint-url

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 these situations. It has no effect for non-HTTP protocols. Default is true.



IMPORTANT

Any JBossWS Webservice endpoint can be exposed by ESB listeners using this action. However, this means the action can only be used when your .esb deployment is installed on the JBoss Application Server as it is not supported on the JBoss Enterprise SOA Platform Server.

Report a bug

13.9.4. Use the SOAPProcessor Action

Prerequisites

- The soap.esb service. This is available in the product's **lib** directory.

Procedure 13.2. Task

- Write a thin service wrapper web service (for example, a JSR 181 implementation) that wraps calls to a target service that does not have a web service end-point.

  Expose the target service via end-points running on the enterprise service bus.

Report a bug

13.9.5. SOAPClient

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

```
<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"/>
    <property name="wise-config">
        <wise-property name="wise.tmpDir" value="/tmp" />
        <wise-property name="wise.jaxb.bindings"
value="some.xjb,additional.xml" />
    </property>
</action>
```

Table 13.30. SOAPClient Optional Properties

Property	Description
wsdl	The WSDL to be used.
operationName	The name of the operation as specified in the webservice WSDL.
SOAPAction	The endpoint operation, now superseded by operationName.
EndPointName	The endpoint invoked. Webservices can have multiple endpoints. If it's not specified, the first name 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.
smooks-handler-config	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 supports 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 a fully qualified name and be separated by semi-columns.

Property	Description
LoggingMessages	It's useful for debugging purposes to view the SOAP messages sent and the response received. Wise achieves this goal using a JAX-WS handler which prints all messages exchanged on System.out. Boolean value.
wise-config	A list of wise-property elements that can be used to configure Wise client. The wise-property element consists of name and value attributes. A special ESB property wise.jaxb.bindings can be used to specify JAXB customizations. It accepts a comma -eparated list of resource names included in the ESB archive.

IMPORTANT

If there is a SOAP fault and HTTP 500 error in the web service being called, the JBoss Enterprise SOA Platform's SOAP user interface client will do the following:

1. Print "WARN [SOAPClient] Received status code '500' on HTTP SOAP (POST) request to..."
2. Ignore the fault and just continue to the next action.

A number of quick starts demonstrating how to use this action are available in the JBoss Enterprise SOA Platform's **`samples/quickstarts`** directory.

Use the soapUI Client Service to construct and populate a message for the target service. This action then routes that message to that service. See http://www.soapui.org/.

The SOAP operation parameters are supplied in either of these two ways:

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

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

1. With a set of objects that are accessed (for SOAP message parameters) using the OGNL framework.
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.

Report a bug

13.9.6. Object Graph Navigation Library (OGNL)

The Object Graph Navigation Library is an open source language developed by the Apache Foundation. It is an expression language for obtaining Java object properties. The JBoss Enterprise SOA Platform uses it as the mechanism used for selecting the SOAP parameter values to be injected into the SOAP message from the supplied parameter map. The OGNL expression for a specific parameter within the SOAP message depends on the position of that parameter within the SOAP body.

Report a bug

13.9.7. Using the Object Graph Navigation Library

In the following message, the OGNL expression representing the customerNumber parameter is "customerOrder.header.customerNumber".

```
<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>
```

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. If this doesn't yield a value, this parameter location within the SOAP message will remain blank.

Taking the sample message above and populating the "customerNumber" requires an object instance (for instance, 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 (for example. a "Header" object instance). The object instance behind the "header" property (for example, a "Header" object instance) should have a "customerNumber" property.

OGNL expressions associated with collections are constructed in a slightly different way:

```
<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 , this could be represented by an order object instance (keyed on the map as "customerOrder") containing an "items" list or array with the list entries being "OrderItem" instances. This in turn contains "partNumber" and any other properties.

To see the SOAP message template as it's being constructed and populated, add the "dumpSOAP" parameter to the parameter map.

WARNING

This can be a very useful developer aid, but should not be left on outside of development.

Report a bug

13.9.8. SOAP Operation Parameters

The SOAP operation parameters are supplied in either of these two ways:

- as a map instance set on the default body location **`(Message.getBody().add(Map))`**
- 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 two 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 message in one of the following ways:

- On the default body location (Message.getBody().add(Map))
- 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 either one of two 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 quick starts:

- webservice_consumer_wise, shows basic usage.
- webservice_consumer_wise2, shows how to use'SmooksRequestMapper' and 'SmooksResponseMapper'.
- webservice_consumer_wise3, shows how to use 'smooks-handler-config'.
- webservice_consomer_wise4, shows usage of 'custom-handlers'.

Report a bug

13.9.9. Specify an End-Point Operation for the SOAPClient Action

Procedure 13.3. Task

- Specify the "wsdl" and "operation" properties on the SOAPClient action:

```
<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>
```

Report a bug

13.9.10. Dealing with SOAP Response Messages

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

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

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

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

1. as an Object Graph created and populated by the XStream toolkit. We also plan to add support for unmarshaling the response using JAXB and JAXB Annotation Introductions.

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

3. if Options one or two are not specified in the action configuration, the raw SOAP response message (String) is attached to the message.

Report a bug

13.9.11. Use XStream to Populate an Object Graph

Procedure 13.4. Task

1. Configure XStream on an action:

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

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

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

namespace="http://schemas.acme.com/services/CustomerOrder.xsd" />
    </property>
</action>
```

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

2. Specify any field name mappings and XStream annotated classes:

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

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

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

Report a bug

13.9.12. Extract SOAP response data to an OGNL Keyed Map

Procedure 13.5. Task

- Replace the "responseXStreamConfig" property with the "responseAsOgnlMap" property:

```
<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>
```

NOTE

To return the raw SOAP message as a string, simply omit both the "responseXStreamConfig" and "responseAsOgnlMap" properties.

Report a bug

13.9.13. HttpClient

HttpClient is an open source library created by the Apache Foundation. It aims to implement an HTTP client that conforms with the latest standards and recommendations. The JBoss Enterprise SOA Platform's SOAPClient uses HttpClient to execute SOAP requests. The SOAPClient uses the HttpClientFactory to create and configure the HttpClient instance.

Report a bug

13.9.14. Configuring the HttpClient

You configure the HttpClient by specifying a set of properties. Here is an example configuration file:

- EasySSLProtocolSocketFactory can be used to create SSL connections that allow the target server to authenticate with a self-signed certificate.

- StrictSSLProtocolSocketFactory can be used to create SSL connections that can optionally perform host name verification in order to help preventing man-in-the-middle type of attacks.
- AuthSSLProtocolSocketFactory can be used to optionally enforce mutual client/server authentication. This is the most flexible implementation of a protocol socket factory. It allows for customization of most, if not all, aspects of the SSL authentication.

The only property that the HttpClientFactory requires is configurators, which specifies a comma-separated list of configurator implementations. Each configurator implementation configures different aspects of the HttpClient instance, extending the **`org.jboss.soa.esb.http.Configurator`** class and providing a **`configure(HttpClient, Properties)`** method.

Table 13.31. Out-of-the-box implementations

Configurator	Description	Required
HttpProtocol	Configure the HttpClient host, port and protocol information, including the socket factory and SSL keystore information.	Yes
AuthBasic	Configure HTTP Basic authentication for the HttpClient.	No
AuthNTLM	Configure NTLM authentication for the HttpClient.	No

Additional configurators can be created and configured by appending their class names to the list specified in the configurators property.

Configuration of the HTTP transport properties:

Table 13.32. Properties

Property	Description	Required
HttpProtocol	Configure the HttpClient host, port and protocol information, including the socket factory and SSL keystore information.	Yes
target-host-url	Target URL for http/https endpoint	Yes
https.proxyHost	Proxy Host for https connections	No
https.proxyPort	Proxy Port for https connections, defaulting to port 443	No
http.proxyHost	Proxy Host for http connections	No
http.proxyPort	Proxy Port for http connections, defaulting to port 80	No

Property	Description	Required
protocol-socket-factory	Override socket factory, implementing the **`ProtocolSocketFactory`** or **`ProtocolSocketFactoryBuilder`** interface. The default value for http is the httpclient **`DefaultProtocolSocketFactory`** whereas the default value for https is the contributed **`StrictSSLProtocolSocketFactory`**. There are two implementations of **`ProtocolSocketFactoryBuilder`** provided in the ESB codebase, **`AuthSSLProtocolSocketFactoryBuilder`** and **`SelfSignedSSLProtocolSocketFactoryBuilder`**, for configuring the **`AuthSSLProtocolSocketFactory`** factory and self signed **`SSLContext`** respectively.	No
keystore	KeyStore location	No
keystore-passw	KeyStore password or encrypted file	No
keystore-type	KeyStore type, defaulting to jks	No
truststore	TrustStore location	No
truststore-passw	TrustStore password or encrypted file	No
truststore-type	TrustStore type, defaulting to jks	No

Configuration of the HTTP Basic Authentication properties:

Table 13.33. Properties

Property	Description	Required
auth-username	Authentication Username	Yes
auth-password	Authentication Password	Yes
authscope-host	Authentication Scope Host	Yes
authscope-port	Authentication Scope Port	Yes
authscope-domain	Authentication Scope Domain	Yes

Configuration of the HTTP Basic Authentication NTLM properties:

Table 13.34. Properties

Property	Description	Required
ntauth-username	Authentication Username	Yes
ntauth-password	Authentication Password	Yes
ntauthscope-host	Authentication Scope Host	Yes
ntauthscope-port	Authentication Scope Port	Yes
ntauthscope-domain	Authentication Scope Domain	Yes
ntauthscope-realm	Authentication Scope Realm	No

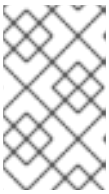
Report a bug

13.9.15. Specify the HttpClientFactory Configuration on the SOAPClient

Procedure 13.6. Task

- Add an additional property to the "wsdl" property:

```
<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>
```



NOTE

The "file" property value will be evaluated as a filesystem, classpath or URI based resource (in that order). This resource contains the HttpClient configuration in the standard Java properties format.

Report a bug

13.9.16. Configure the HttpClient Directly in the Action Configuration

Procedure 13.7. Task

- Set the properties 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>
```

Report a bug

13.9.17. SOAPProxy

The SOAPProxy is an action that "consumes" external web service end-points. It also allows you to re-publish a web service end-point via the Enterprise Service Bus. Sitting between the external services and the ESB, the purpose of this intermediary is to provide an abstraction layer it provides the following functionality:

- it facilitates loose coupling between the client and service (since they are both completely unaware of each other.)
- it means the client no longer has a direct connection to the remote service's hostname/IP address.
- the client will see modified WSDL that changes the inbound/outbound parameters. At a minimum, the WSDL must be tweaked so that the client is pointed to the ESB's exposed end-point instead of the original, now proxied endpoint.
- it allows you to introduce a transformation of the SOAP envelope/body via the action pipeline both for the inbound request and outbound response.
- it makes service versioning possible since clients can connect to two or more proxy end-points on the enterprise service bus, each with its own WSDL and/or transformations and routing requirements, and the ESB will send the appropriate message to the appropriate endpoint and provide an ultimate response.
- it allows for complex context-based routing via ContentBasedRouter.

Report a bug

13.9.18. Using the SOAPProxy Action

A SOAPProxy action is:

- both a producer and consumer of web services.
- all that is required is a property pointing to the external wsdl.
- a way to allow the WSDL to be automatically transformed (via the optional wsdlTransform property.)
- a way to ensure that SOAP is not tied to HTTP. The WSDL is read, and if an HTTP transport is defined, that will be used.
- a way to optionally apply HttpRouter properties as overrides if you are using HTTP.

If the WSDL specifies an HTTP transport, then any of the HttpRouter properties can be applied.

Table 13.35.

Property	Description	Required
wsdlTransform	A <smooks-resource-list> xml config file allowing for flexible wsdl transformation.	No
wsdlCharset	The character set the original wsdl (and imported resources) is encoded in UTF-8. It will be transformed to UTF-8 if it is a supported encoding by the underlying platform.	No
endpointUrl	Example of an HttpRouter property, useful when domain name matching is important for SSL certs.	No
file	Apache Commons HTTPClient properties file, useful when proxying to a web service via SSL.	No
clientCredentialsRequired	Whether the Basic Auth credentials are required to come from the end client, or if the credentials specified inside file can be used instead. Default is "true".	No
wsdl	The original wsdl url whose WS endpoint will get re-written and exposed as new wsdl from the ESB. Depending upon the **`<definitions><service><port> <soap:address`** location attribute's protocol (for example "http"), a protocol-specific SOAPProxyTransport implementation is used. The value can reference a location based on five different schemes: • **`http://`** When you want to pull wsdl from an external web server. Example: http://host/foo/HelloWorldWS?wsdl • **`https://`** When you want to pull wsdl from an external web server over SSL. Example: https://host/foo/HelloWorldWS?wsdl • **`file://`** When your wsdl is located on disk, accessible by the ESB JVM. Example: file:///tmp/HelloWorldWS.wsdl Note: Three slashes in the example above. This is so we can specify an absolute versus relative file path. • **`classpath://`** When you want to package your wsdl inside your ESB archive. Example: classpath:///META-INF/HelloWorldWS.wsdl	Yes

Property	Description	Required
	Note the three slashes in the example above. This is so we can specify an absolute versus relative classloader resource path. • `internal://` When the wsdl is being provided by a JBossWS web service inside the same JVM as this ESB deployment. Example: internal://jboss.ws:context=foo,endpoint=HelloWorldWS  **NOTE** This scheme should be used instead of http or https in the usage described above. This is because on server restart, Tomcat may not yet be accepting incoming http/s requests, and thus cannot serve the wsdl.	

Example 13.16. Sample Configuration: Basic scenario

```
<action name="proxy"
class="org.jboss.soa.esb.actions.soap.proxy.SOAPProxy">
    <property name="wsdl" value="http://host/foo/HelloWorldWS?wsdl"/>
</action>
```

Example 13.17. Sample Configuration: Basic Authentication and SSL

```
<action name="proxy"
class="org.jboss.soa.esb.actions.soap.proxy.SOAPProxy">
    <property name="wsdl" value="https://host/foo/HelloWorldWS?wsdl"/>
    <property name="endpointUrl"
value="https://host/foo/HelloWorldWS"/>
    <property name="file" value="/META-INF/httpclient-
8443.properties"/>
    <property name="clientCredentialsRequired" value="true"/>
</action>
```

Report a bug

13.10. MISCELLANEOUS ACTIONS

13.10.1. SystemPrintln

SystemPrintln is a simple action for printing out the contents of a message (like **`System.out.println`**). It formats the message contents as XML.

Report a bug

13.10.2. Using SystemPrintln

Input Type	**`java.lang.String`**
Class	**`org.jboss.soa.esb.actions.SystemPrintln`**
Properties	• message - A message prefix. **`Required`** • printfull - If true then the entire message is printed, otherwise just the byte array and attachments. • outputstream - if true then System.out is used, otherwise System.err.

```
<action name="action2"
class="org.jboss.soa.esb.actions.ServiceLoggerAction">
   <property name="text" value="Message arrived"/>
   <property name="log-payload-location" value="true"/>
</action>
```

Report a bug

13.10.3. SchemaValidationAction

SchemaValidationAction is a simple action for performing schema-based validation of XML messages.

Report a bug

13.10.4. Using SchemaValidationAction

Input Type	**`java.lang.String`**
Class	**`org.jboss.soa.esb.actions.validation.SchemaValidationAction`**
Properties	• schema - The classpath path of the validation schema file (for example, .xsd). • schemaLanguage - (Optional.) The schema type/language. Default: "http://www.w3.org/2001/XMLSchema" i.e. XSD.

```
<action name="val"
```

```
class="org.jboss.soa.esb.actions.validation.SchemaValidationAction">
    <property name="schema" value="/com/acme/validation/order.xsd"/>
</action>
```

Report a bug

13.10.5. ServiceLoggerAction

The ServiceLoggerAction is a simple action that logs custom text and possibly the message to a logger using the "<service-category>.<service-name<" appender. This action differs from LogAction in that it allows you to set LogLevelAction on a per service basis, rather than on a per-action basis, and that it allows you to log custom text.

Report a bug

13.10.6. Using the ServiceLoggerAction

- Using the Debug level setting will result in the message being output.
- Using the trace level setting will result in the output of the message payload

Input Type	**`java.lang.String`**
Class	**`org.jboss.soa.esb.actions.ServiceLoggerAction`**
Properties	• text - A message prefix. **`Required`** • get-payload-location - True or False value which specifies whether the payload location should be logged in Trace

```
<action name="servicelogger"
class="org.jboss.soa.esb.actions.ServiceLoggerAction">
    <property name="text" value="Reached here"/><br/>
    <property name="get-payload-location" value="true"/>
</action>
```

NOTE

The <property> text is not required. If omitted, <category>.<service> will be printed instead.

Report a bug

CHAPTER 14. DEVELOPING YOUR OWN ACTIONS

14.1. DEVELOPING CUSTOM ACTIONS

You can develop your own actions in any of the following ways, each of which has its own advantages and disadvantages:

- Lifecycle actions, implementing **`org.jboss.soa.esb.actions.ActionLifecycle`** or **`org.jboss.soa.esb.actions.ActionPipelineProcessor`**
- Java bean actions, implementing **`org.jboss.soa.esb.actions.BeanConfiguredAction`**
- Annotated actions
- Legacy actions

In order to understand the differences between each implementation it is necessary to understand:

- How the actions are configured
- When the actions are instantiated and the implications on thread safety
- Whether they have visibility of life-cycle events
- Whether the action methods are invoked directly or via reflection

Report a bug

14.2. CONFIGURING AN ACTION BY SETTING PROPERTIES FOR IT

Actions generally act as templates, requiring 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 should look like this:

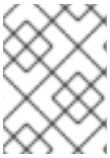
```
<action name="PrintAMessage" class="test.PrintMessage">
 <property name="information" value="Hello World!" />
 <property name="repeatCount" value="5" />
</action>
```

How this configuration is then mapped on to the action instance will depend on the type of action.

Report a bug

14.3. REFLECTION

When working with Java, you can use the *reflection* method to inspect and modify objects within the runtime. It can be added to a piece of code to assist the user in finding terms within the script, adding and modifying parts of it, and invoking applications. It can be used to call services and applications during runtime. It is useful in instances where you need to find the name of a field that you cannot remember or need to modify something efficiently.



NOTE

Using the reflection method can result in slower loading times and it is sometimes better to simply perform these tasks manually.

Report a bug

14.4. MANAGED LIFECYCLE

This term refers to being able to control all aspects of a process from start to finish. There are various ways to attain this sort of control. Management beans and services can both be used to manage the lifecycle of a process. This allows you to make modifications at various stages of an object's lifecycle.

Report a bug

14.5. LIFE-CYCLE ACTION

Lifecycle actions are those which are derived from the lifecycle interfaces, **`org.jboss.soa.esb.actions.ActionLifecycle`** and **`org.jboss.soa.esb.actions.ActionPipelineProcessor`**.

Report a bug

14.6. ACTIONLIFECYCLE

ActionLifecycle is an interface that implements the **`initialise`** and **`destroy`** action pipeline life-cycle methods,

Report a bug

14.7. ACTIONPIPELINEPROCESSOR

ActionPipelineProcessor is an interface that extends the ActionLifecycle interface so that it can utilise the **`process`**, **`processSuccess`** and **`processException`** message processing methods.

This interface supports the implementation of managed life-cycle stateless actions.

Report a bug

14.8. IMPLEMENTING ACTIONLIFECYCLE AND ACTIONPIPELINEPROCESSOR

A single instance of a class implementing either the ActionLifecycle or ActionPipelineProcessor interface is instantiated on a "per-pipeline" basis (in other words, per-action configuration) and must be thread safe. The **`initialise`** and **`destroy`** methods can be overridden to allow the action to perform

resource management for the lifetime of the pipeline. (For example, caching resources needed in the **`initialise`** method, then cleaning them up in the **`destroy`** method.)

These actions must define a constructor which takes a single **`ConfigTree`** instance as a parameter, representing the configuration of the specific action within the pipeline.

The pipeline will invoke each method directly provided that the action has implemented the appropriate interface and that the method names are not overridden in the configuration. Any method invocations which are not implemented via the interfaces, or which have been overridden in the action configuration, will be invoked using reflection.

To simplify development there are two abstract base classes provided in the codebase, each implementing the appropriate interface and providing empty stub methods for all but the **`process`** method. These are **`org.jboss.soa.esb.actions.AbstractActionPipelineProcessor`** and **`org.jboss.soa.esb.actions.AbstractActionLifecycle`**. Use them like this:

```
public class ActionXXXProcessor extends AbstractActionPipelineProcessor {
  public ActionXXXProcessor(final ConfigTree config) {
    // extract configuration
  }

  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...
  }
}
```

Report a bug

14.9. JAVA BEAN ACTION

A Java Bean Action is an action for which properties are configured by means of using setters. These setters correspond to the property names. The framework populates them automatically.

Report a bug

14.10. CONFIGURING A JAVA BEAN ACTION

The Java Bean Action class must implement the **`org.jboss.soa.esb.actions.BeanConfiguredAction`** marker interface in order to make the action Bean populate automatically. Here is some sample code:

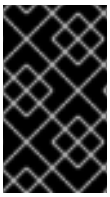
```
import org.jboss.soa.esb.message.Message;
import org.jboss.soa.esb.actions.BeanConfiguredAction;
```

```
public class PrintMessage implements BeanConfiguredAction {
  private String information;
  private Integer repeatCount;
  public void setInformation(String information) {
    this.information = information;
  }
  public void setRepeatCount(Integer repeatCount) {
    this.repeatCount = repeatCount;
  }
  public Message process(Message message) {
    for (int i=0; i < repeatCount; i++) {
      System.out.println(information);
    }
    return message;
  }
}
```

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.



IMPORTANT

These actions *do not* support the lifecycle methods. Rather they will be instantiated for *every* message passing through the action pipeline and will always have their **`process`** methods invoked using reflection.

Report a bug

14.11. ANNOTATED ACTION

Annotated Action Classes are action classes that have annotations that make it easier to create clean implementations. They hide the complexity associated with implementing interfaces, abstract classes and dealing with the **`ConfigTree`**. A single instance of annotated actions will be instantiated on a "per-pipeline" basis (in other words, per-action configuration) and must be thread safe. The pipeline will always invoke the action methods using reflection.

Report a bug

14.12. USING ANNOTATIONS

These are the annotations that you can add to your action classes:

@Process

The simplest implementation involves creating an action with a a basic *plain old Java object* (POJO) with a single method, annotated with @Process:

```
public class MyLogAction {

                @Process
       public void log(Message message) {
          // log the message...
       }
   }
```

The @Process annotation serves to identify the class as a valid ESB **`action`**. In cases in which there are multiple methods in the class, it also identifies the method which is to be used for processing the message instance (or some part of the message. This is explained in more depth when the @BodyParam, @PropertyParam and @AttachmentParam annotations are discussed.)

To configure an instance of this **`action`** on a **`pipeline`**, use the same process as that for low/base level **`action`** implementations (these being those that extend **`AbstractActionPipelineProcessor`** or implement **`ActionLifecycle`** or one of its other sub-types or abstract implementations):

```
<service .....>
      <actions>
          <action name="logger" class="com.acme.actions.MyLogAction" />
      </actions>
  </service>
```

In cases in which multiple methods annotated with @Process are associated with the **`action`** implementation, use the process attribute to specify which of them is to be used for processing the message instance:

```
<service .....>
      <actions>
          <action name="logger" class="com.acme.actions.MyLogAction"
                                          process="log" />
      </actions>
  </service>
```

@Process methods can be implemented to return:

- void. This means there will be no return value, as with the logger action implementation above.
- message: This is a message instance. This becomes the active/current instance on the action pipeline.
- another type. If the method does not return a message instance, the object instance that is returned will be set on the current message instance on the action pipeline.. Where you should set it on the message depends on the set-payload-location<action> configuration property, which default according to the normal **`MessagePayloadProxy`** rules.

Use @Process methods to specify parameters in a range of different ways. You can:

1. specify the message instance as a method parameter.

2. specify one or more arbitrary parameter types. The Enterprise Service Bus framework will search for data of that type in the active/current pipeline message instance. Firstly, it will search the message body, then properties, then attachments and pass this data as the values for those parameters (or **`null`** if not found).

An example of the first option was depicted above in the logger action. Here is an example of the second option:

```
public class OrderPersister {

      @Process
      public OrderAck storeOrder(OrderHeader orderHeader,
        OrderItems orderItems) {
          // process the order parameters and return an ack...
      }
  }
```

In this example, the @Process method is relying on a previous action in the **`pipeline`** to create the **`OrderHeader`** and **`OrderItem`** object instances and attach them to the current message. (Perhaps a more realistic implementation would have a generic action implementation that decodes an XML or EDI payload to an order instance, which it would then returns. The **`OrderPersister`** would then take an order instance as its sole parameter.) Here is an example:

```
public class OrderDecoder {

      @Process
      public Order decodeOrder(String orderXML) {
          // decode the order XML to an ORder instance...
      }
  }

  public class OrderPersister {

      @Process
      public OrderAck storeOrder(Order order) {
          // persist the order and return an ack...
      }
  }
```

Chain the two actions together in the service configuration:

```
<actions>
    <action name="decode" class="com.acme.orders.OrderDecoder" />
    <action name="persist" class="com.acme.orders.OrderPersister" />
</actions>
```

The code is easier to read in Option #2 because there are less annotations, but it carries a risk because the process of run-time "hunting" through the message for the appropriate parameter values is not completely *deterministic*. Due to this, Red Hat supports the @BodyParam, @PropertyParam and @AttachmentParam annotations.

Use these @Process method parameter annotations to explicitly define from where in the message an individual parameter value for the @Process method is to be retrieved. As their names suggest, each of these annotations allow you to specify a named location (in the message body, properties or attachments) for a specific parameter:

```
public class OrderPersister {

        @Process
      public OrderAck storeOrder(
                    @BodyParam("order-header") OrderHeader
orderHeader,
                    @BodyParam("order-items") OrderItems orderItems)
{

          // process the order parameters and return an ack...
      }
  }
```

If the message location specified does not contain a value, then null will be passed for this parameter (the @Process method instance can decide how to handle this). If, on the other hand, the specified location contains a value of the wrong type, a **`MessageDeliverException`** will be thrown.

@ConfigProperty

Most actions require some degree of custom configuration. In the ESB action configuration, the properties are supplied as <property> sub-elements of the <action> element:

```
<action name="logger" class="com.acme.actions.MyLogAction">
    <property name="logFile" value="logs/my-log.log" />
    <property name="logLevel" value="DEBUG" />
</action>
```

To utilise these properties, use the low/base level action implementations (do so by extending **`AbstractActionPipelineProcessor`** or by implementing **`ActionLifecycle`**). This involves working with the **`ConfigTree`** class, (which is supplied to the action via its constructor). In order to implement an action, follow these steps:

1. Define a constructor on the action class that supplies the **`ConfigTree`** instance.
2. Obtain all of the relevant action configuration properties from the **`ConfigTree`** instance.
3. Check for mandatory action properties and raise exceptions in those places where they are not specified on the <action> configuration.
4. Decode all property values from strings (as supplied on the **`ConfigTree`**) to their appropriate types as used by the action implementation. For example, decide **`java.lang.String`** to **`java.io.File`**, **`java.lang.String`** to Boolean, **`java.lang.String`** to long and so forth.
5. Raise exceptions at those places where the configured value cannot be decoded to the target property type.
6. Implement *unit tests* on all the different configuration possibilities to ensure that the tasks listed above were completed properly.

While the tasks above are generally straightforward, they can be laborious, error-prone and lead to inconsistencies across actions with regard to how configuration mistakes are handled. You may also be required to add a lot of code resulting in additional confusion.

The annotated action addresses these problems via @ConfigProperty. Expand the MyLogActions implementation, which has two mandatory configuration properties: logFile and logLevel:

```
public class MyLogAction {

      @ConfigProperty
      private File logFile;

      @ConfigProperty
      private LogLevel logLevel;

      public static enum LogLevel {
          DEBUG,
          INFO,
          WARN
      }

      @Process
      public void log(Message message) {
          // log the message at the configured log level...
      }
  }
```

NOTE

You can also define the @ConfigProperty annotation on "setter" methods (instead of on the field).

That is all that needs to be done. When the Enterprise Service Bus deploys the action, it examines both the implementation and the maps found within the decoded value (including any support for enums, as with the LogLevel enum above). It finds the action fields possessing the @ConfigProperty annotation. The developer is not required to deal with the **`ConfigTree`** class at all or develop any extra code.

By default, every class field possessing the @ConfigProperty annotation is mandatory. Non-mandatory fields are handled in one of these two ways:

1. by specifying **`use = Use.OPTIONAL`** on the field's @ConfigProperty annotation.
2. by specifying a defaultVal on the field's @ConfigProperty annotation. (This is optional.)

To make the log action's properties optional only, implement the action like this:

```
public class MyLogAction {

      @ConfigProperty(defaultVal = "logs/my-log.log")
      private File logFile;

      @ConfigProperty(use = Use.OPTIONAL)
      private LogLevel logLevel;

      public static enum LogLevel {
          DEBUG,
          INFO,
          WARN
      }
```

```
        @Process
        public void log(Message message) {
            // log the message...
        }
    }
```

The @ConfigProperty annotation supports two additional fields:

1. name: use this to explicitly specify the name of the action configuration property to be used to populate the field of that name on the action instance.

2. choice: use this field to constrain the configuration values allowed for itself. This can also be achieved using an enumeration type (as with the LogLevel).

The name field might be used in various situations such as when migrating an old action (that uses the low/base level implementation type) to the newer annotation-based implementation, only to find that the old configuration name for a property (which cannot be changed for backward-compatibility reasons) does not map to a valid Java field name. Taking the log action as an example, imagine that this was the old configuration for the log action:

```
<action ...>
      <property name="log-file" value="logs/my-log.log" />
      <property name="log-level" value="DEBUG" />
  </action>
```

The property names here do not map to valid Java field names, so specify the name on the @ConfigProperty annotation:

```
public class MyLogAction {

      @ConfigProperty(name = "log-file")
      private File logFile;

      @ConfigProperty(name = "log-level")
      private LogLevel logLevel;

      public static enum LogLevel {
          DEBUG,
          INFO,
          WARN
      }

      @Process
      public void log(Message message) {
          // log the message...
      }
  }
```

The bean configuration's property values are decoded from their string values. To then match them against the appropriate POJO bean property type, these simple rules are used:

1. If the property type has a single-argument string constructor, use that.

2. If it is a "primitive," use its object type's single-argument string constructor. For example, if it is an int, use the Integer object.

3. If it is an enum, use **`Enum.valueOf`** to convert the configuration string to its enumeration value.

@Initialize and @Destroy

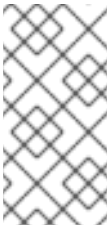
Sometimes action implementations need to perform initialization tasks at deployment time. They may also need to perform a clean-up whilst being undeployed. For these reasons, there are @Initialize and @Destroy method annotations.

Here are some examples. At the time of deployment, the logging action may need to perform some checks (that, for example, files and directories exist). It may also perform some initialization tasks (such as opening the log file for writing). When it is undeployed, the action may need to perform some clean-up tasks (such as closing the file). Here is the code to perform these tasks:

```
public class MyLogAction {

      @ConfigProperty
      private File logFile;

      @ConfigProperty
      private LogLevel logLevel;

      public static enum LogLevel {
          DEBUG,
          INFO,
          WARN
      }

      @Initialize
      public void initializeLogger() {
   // Check if file already exists… check if parent folder
   // exists etc...
   // Open the file for writing...
      }

      @Destroy
      public void cleanupLogger() {
          // Close the file...
      }

      @Process
      public void log(Message message) {
          // log the message...
      }
  }
```

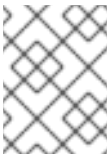
NOTE

All of the @ConfigProperty annotations will have been processed by the time the ESB deployer invokes the @Initialize methods. Therefore, the @Initialize methods can rely on these fields being ready before they execute the customised initialization.



NOTE

There is no need to always use both of these annotations to specify methods. Only specify them if there is a need; in other words, if a method only needs initialization, only use the @Initialize annotation (You do not have to supply a "matching" method annotated with the @Destroy annotation).



NOTE

It is possible to specify a single method and annotate it with both @Initialize and @Destroy.

NOTE

You can optionally specify a **`ConfigTree`** parameter on @Initialize methods. Do this to have access to the actions which underlie the **`ConfigTree`** instance.

@OnSuccess and @OnException

Use these annotations to specify those methods to be executed on a successful or failed execution, respectively, of that **`pipeline`** in which the action is configured:

```
public class OrderPersister {

      @Process
      public OrderAck storeOrder(Order order) {
          // persist the order and return an ack...
      }

      @OnSuccess
      public void logOrderPersisted(Message message) {
          // log it...
      }

      @OnException
      public void manualRollback(Message message,
                                          Throwable theError) {
          // manually rollback...
      }
  }
```

In the cases of both of these annotations, the parameters passed to the methods are optional. You can supply none, some or all of the parameters shown above. The enterprise service bus' framework resolves the relevant parameters in each case.

Report a bug

14.13. LEGACY ACTION

Any action that is not a lifecycle action, Java Bean action or annotated action, is regarded as a legacy action. A legacy action inherits the behaviour present in the Enterprise Service Bus codebase.

Report a bug

14.14. BEHAVIOUR AND ATTRIBUTES OF A LEGACY ACTION

Legacy actions have these characteristics:

- Configuration of the action is through the provision of a constructor with a single **`ConfigTree`** parameter.
- The action will be instantiated for *every* message passing through the pipeline.
- The action has no knowledge of any of the lifecycle methods.
- The invocation of the **`process`** methods will always be done via reflection.

Report a bug

CHAPTER 15. GATEWAYS AND CONNECTORS

15.1. INTRODUCTION TO GATEWAYS AND CONNECTORS

Introduction

Not all clients interacting with the JBoss Enterprise SOA Platform will be able to understand its native protocols and message format. As such there is a need to be able to bridge between ESB-aware end-points (those that understand JBossESB) and ESB-unaware end-points (those that do not understand JBossESB). Such bridging technologies have existed for many years in a variety of distributed systems and are referred to as gateways and connectors.

It is important for legacy interoperability scenarios that a SOA infrastructure allow ESB-unaware clients to use ESB-aware services, or ESB-aware clients to use ESB-unaware services. One of the features of the JBoss Enterprise SOA Platform is the ability to allow a wide variety of clients and services to interact, including those that were not written using JBossESB or, indeed, any ESB. There is an abstract notion of an interoperability bus in JBossESB so that end-points which are not ESB-aware can still be plugged into the bus.

Report a bug

15.2. GATEWAYS

15.2.1. Gateway Listener

A gateway listener is used to bridge the ESB-aware and ESB-unaware worlds. It is a specialized listener process that is designed to listen to a queue for ESB-unaware messages that have arrived through an external (ESB-unaware) end-point. The gateway listener receives the messages as they land in the queue. When a gateway listener "hears" incoming data arriving, it converts that data (the non-ESB messages) into the **`org.jboss.soa.esb.message.Message`** format. This conversion happens in a variety of different ways, depending on the gateway type. Once the conversion has occurred, the gateway listener routes the data to its correct destination.

Report a bug

15.2.2. Differences Between a Gateway Listener and a Normal Listener

A gateway behaves similarly to an ESB-aware listener. However, there are some important differences:

- Gateway classes can pick up arbitrary objects contained in files, JMS messages, SQL tables and so forth (each 'gateway class' is specialized for a specific transport), whereas JBossESB listeners can only process JBossESB normalized Messages as described in “The Message” section of this document. However, those Messages can contain arbitrary data.
- Only one action class is invoked to perform the 'message composing' action. ESB listeners are able to execute an action processing pipeline.
- Objects that are 'picked up' are used to invoke a single 'composer class' (the action) that will return an ESB message object. This will then be delivered to a target service that must be ESB-aware. The target service defined at configuration time will be translated at runtime into an EPR (or a list of EPRs) by the registry. The EPR returned by the registry is similar to the 'toEPR'

contained in the header of ESB Messages. However, because incoming objects are 'ESB-unaware' with 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. The same process occurs for JMS messages. The default message composing class for an SQL table row is to package contents of all columns specified in the configuration, into a java.util.Map.

Although these default composer classes will be adequate for most uses, you can also provide your own. The only requirements are that they must have a constructor that takes a single ConfigTree argument, and that they must provide a 'Message composing' method (whereby the default name is 'process' but this can be configured differently in the 'process' attribute of the <action> element within the ConfigTree provided at constructor time). The processing method must take a single argument of type object, and return a message value.

The FileGateway accepts the **`file-filter-class`** configuration attribute which allows you to define a FileFilter implementation that may be used to select the files used by the gateway in question. Initialization of user-defined FileFilter instances is performed by the gateway. If the instance is also of type **`org.jboss.soa.esb.listeners.gateway.FileGatewayListener.FileFilterInit`**, the **`init`** method will be called and passed to the gateway ConfigTree instance.

By default, the FileFilter implementations are defined and used by the FileGateway. If an input suffix is defined in the configuration, files matching that suffix will be returned. Alternatively, if there is no input suffix, any file is accepted as long as it does not match the work suffix, error suffix and post suffix.

Report a bug

15.2.3. Gateway Data Mappings

When a non-ESB message is received by a gateway listener, it is converted into an ESB Message. How this is done depends upon the type of gateway listener involved. Here are the default approaches for each kind of gateway::

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

This one places the contents 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.

Report a bug

15.2.4. Changing Gateway Data Mappings

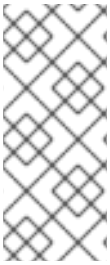
Mappings are changed in different ways for different gateway types:

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. Make sure that you provide the MessageComposer implementation in the configuration file by 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 end-point references for the newly created message, based on the original content, sender and so forth, then you should consider modifying the header too.

Report a bug

15.3. CONNECTORS

15.3.1. Java Connector Architecture (JCA)

The Java EE Connector Architecture (JCA) defines a standard architecture for Java EE systems to external heterogeneous Enterprise Information Systems (EIS). Examples of EISs include Enterprise Resource Planning (ERP) systems, mainframe transaction processing (TP), databases and messaging systems.

JCA 1.6 provides features for managing:

- connections
- transactions

- security
- life-cycle
- work instances
- transaction inflow
- message inflow

JCA 1.6 was developed under the Java Community Process as JSR-322, http://jcp.org/en/jsr/detail?id=313.

Report a bug

15.3.2. Connecting via JCA

You can use JCA Message Inflow as an ESB Gateway. To enable a gateway for a service, you must first implement an end-point class that implements the **`org.jboss.soa.esb.listeners.jca.InflowGateway`** class:

```
public interface InflowGateway
{
  public void setServiceInvoker(ServiceInvoker invoker);
}
```

The end-point 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.deliverAsync(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 a message-driven bean 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 end-point class. The ESB then activates the JCA end-point and the end-point class instance is ready to receive messages. In the JmsEndpoint example, the instance receives a JMS message and converts into an ESB message. It then uses the ServiceInvoker instance to invoke on the target service.

NOTE

The JMS end-point class is provided under **`org.jboss.soa.esb.listeners.jca.JmsEndpoint`**. You can use this class over and over again with any JMS JCA inflow adapters.

Report a bug

15.3.3. Configuring a JCA Inflow Gateway

Configure a JCA inflow gateway by changing the settings in a **`jboss-esb.xml`** file:

```
<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>
```

```
<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 15.1. 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, for example `jms-ra.rar`
endpointClass	yes	The name of your end point class.
messagingType	no	The message interface for the adapter. If you do not specify one, ESB will guess based on the endpoint class.
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.

Report a bug

15.3.4. Mapping Standard Activation Properties

A number of ESB properties are automatically mapped to the activation configuration using an ActivationMapper. The properties, their location and their purpose are described in the following table

Table 15.2. Mapping Standard 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.

Attribute	Location	Description
selector	jms-message-filter	The JMS message selector.
providerAdapterJNDI	jms-jca-provider	The JNDI location of a Provider Adapter which can be used by the JCA inflow to access a remote JMS provider. This is a JBoss-specific interface, supported by the default JCA inflow adapter and may be used, if necessary, by other in-flow adapters.

You can over-ride this behaviour by specifying a class which implements the **`ActivationMapper`**. interface. You can declare this class globally or within each individual deployment configuration.

Specifying it globally via the **`jbossesb-properties.xml`** file as it defines the default mapper used for the specified JCA adapter The name of the property to be configured is org.jboss.soa.esb.jca.activation.mapper.<adapter name> and the value is the class name of the ActivationMapper.

The following example shows the configuration of the default ActivationMapper used to map the properties on 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 with the precedence being the same order.

The following 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>
```

Here is how you specify 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>
```

Here is how you specify 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>
```

Report a bug

CHAPTER 16. JAXB ANNOTATION INTRODUCTIONS

16.1. JAXB ANNOTATION INTRODUCTIONS

JAXB Annotation Introductions is a feature of the JBoss Enterprise SOA Platform that enables you to define an XML configuration that allows for you to "introduce" JAXB annotation. This feature was added because the native JBossWS SOAP stack uses JAXB to bind to and from SOAP which meant that an unannotated typeset could be used to build a JBossWS endpoint.

Report a bug

16.2. USING JAXB ANNOTATION INTRODUCTIONS

The XML configuration must be saved in your end-point deployment's **`META-INF/jaxb-intros.xml`** file.

Report a bug

16.3. WRITING JAXB ANNOTATION INTRODUCTION CONFIGURATIONS

The 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. (In your IDE, register this XSD against the http://www.jboss.org/xsd/jaxb/intros namespace.)

@XmlType

https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlType.html: This goes on the “Class” element.

@XmlElement

https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlElement.html: This goes on the “Field” and “Method” elements.

@XmlAttribute

https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlAttribute.html: This goes on the “Field” and “Method” elements.

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. Use the XSD and your IDE to editing the configuration.

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>
```

Report a bug

APPENDIX A. REVISION HISTORY

Revision 5.3.1-78.400 **2013-10-31** **Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 5.3.1-78 **Thu Feb 07 2013** **CS Builder Robot**

Built from Content Specification: 10767, Revision: 372112