Programmers Guide

Your guide to developing for the JBoss ESB

by JBoss ESB Development Team with Community Contributions

The Enterprise Service Bus

1.1. What is an ESB?

The ESB is seen as the next generation of EAI – better and without the vendor-lockin characteristics of old. As such, many of the capabilities of a good ESB mirror those of existing EAI offerings. Traditional EAI stacks consist of:

A traditional EAI stack consists of the following:

- Business Process Monitoring

- Integrated Development Environment

- Human Work-flow User Interface

- Business Process Management

- Connectors

- Transaction Manager

- Security

- Application Container

- Messaging Service

- Meta-data Repository

- Naming and Directory Service

- Distributed Computing Architecture

As with EAI systems, ESB is not about business logic – that is left to higher levels. It is about infrastructure logic. Although there are many different definitions of what constitutes an ESB, what everyone agrees on now is that an ESB is part of an SOA infrastructure. However, SOA is not simply a technology or a product: it's a style of design, with many aspects (such as architectural, methodological and organisational) unrelated to the actual technology. But obviously at some point it becomes necessary to map the abstract SOA to a concrete implementation and that's where the ESB comes in to play.

> **Note**
>
> You can learn more about SOA principles and ESB architectures in the SOA Background Concepts document.

1.2. When would you use JBossESB?

The figures below illustrate some concrete examples where JBossESB would be useful. Although these examples are specific to interactions between participants using non-interoperable JMS implementations, the principles are general and can be applied to other transports such as FTP and HTTP.

The first diagram shows simple file movement between two systems where messaging queuing is not involved.
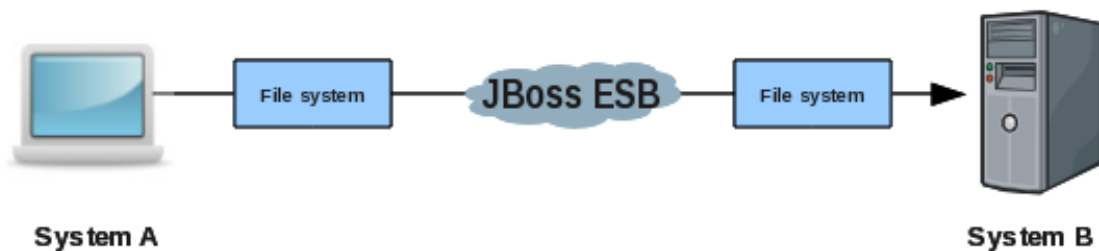
Figure 1.1. Simple File Movement

The next diagram illustrates how transformation can be injected into the same scenario using JBossESB.

In the next series of examples, we use a queuing system (e.g., a JMS implementation).
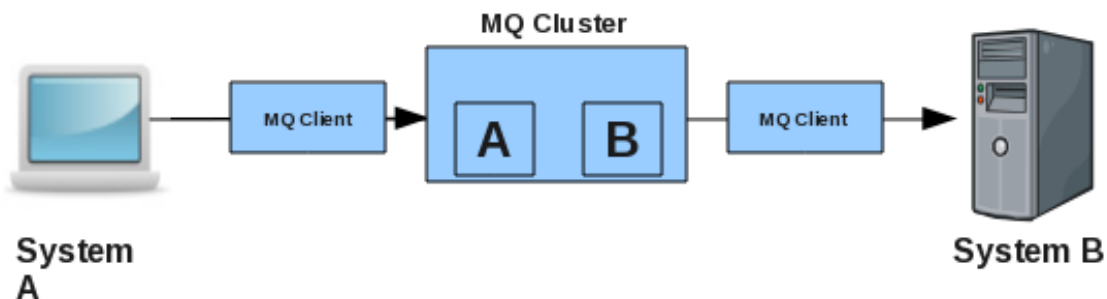
Figure 1.2. Simple File Movement with Transformation

The diagram below shows transformation and queuing in the same situation.
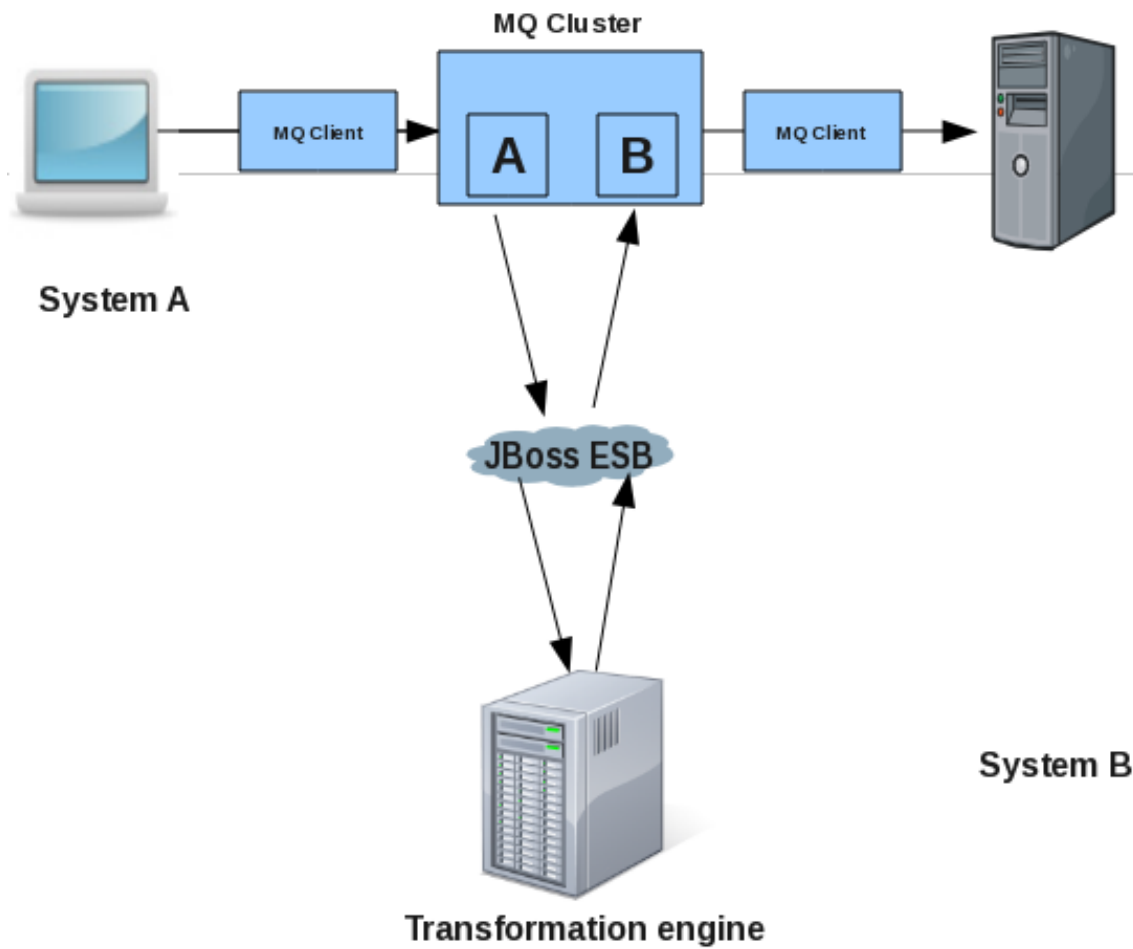
Figure 1.3. Simple File Movement with Transformation and Queuing

JBossESB can be used in more than multi-party scenarios. For example, the diagram below shows basic data transformation via the ESB using the file system.
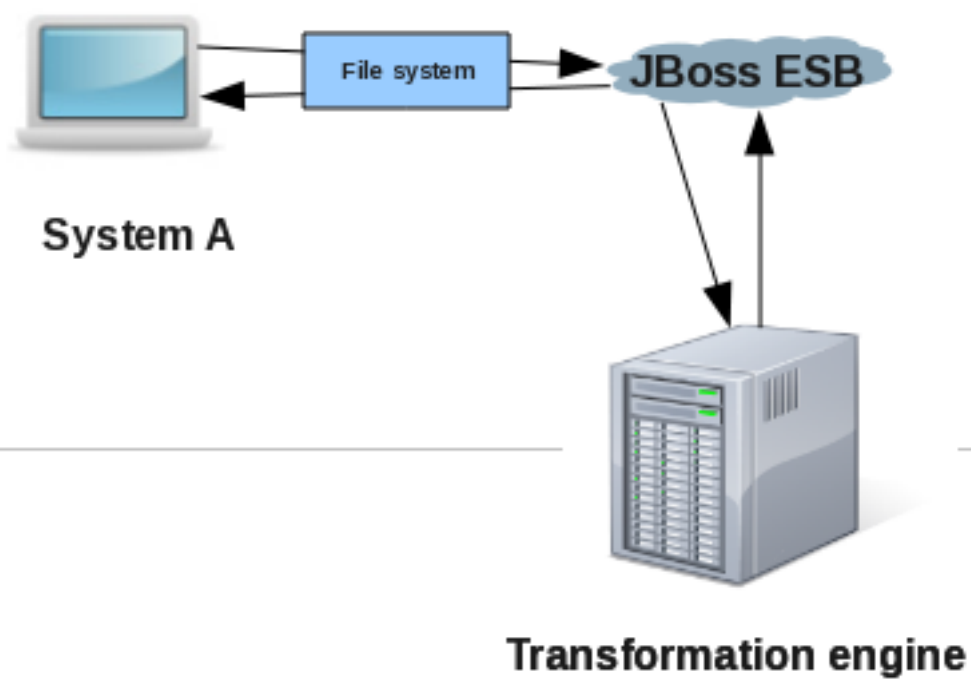
Figure 1.4. Basic Data Transformation

The final scenario is again a single party example using transformation and a queuing system.
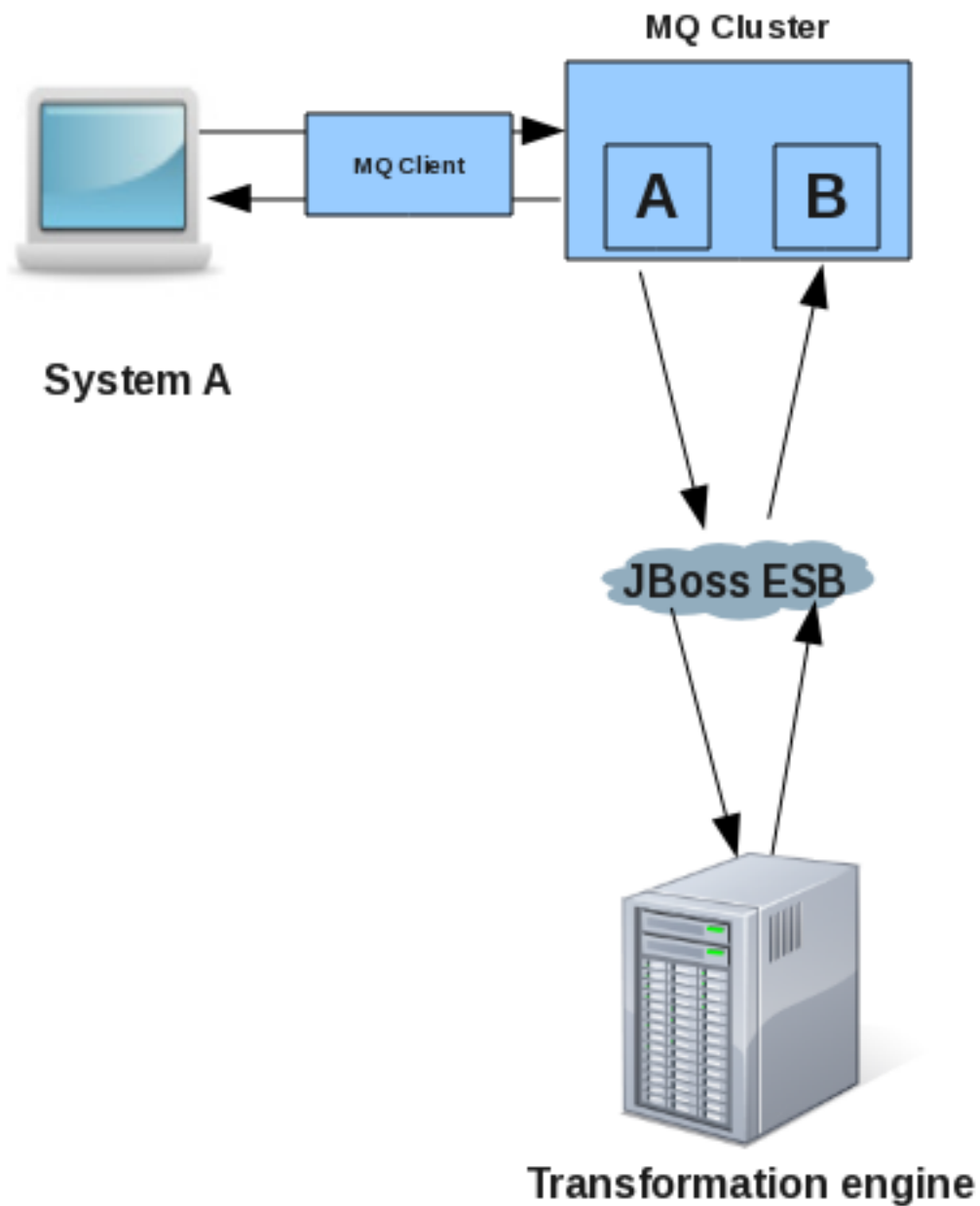
Figure 1.5. Transformation and Queuing

In the following chapters we shall look at the core concepts within JBossESB and how they can be used to develop SOA-based applications.

JBossESB

2.1. Rosetta

The core of JBossESB is Rosetta, an ESB that has been in commercial deployment at a mission critical site for over 3 years. The architectCommitted revision 28034. Figure of Rosetta is shown below.

> **Note**
>
> In the diagram, processor classes refer to the Action classes within the core that are responsible for processing on triggered events.
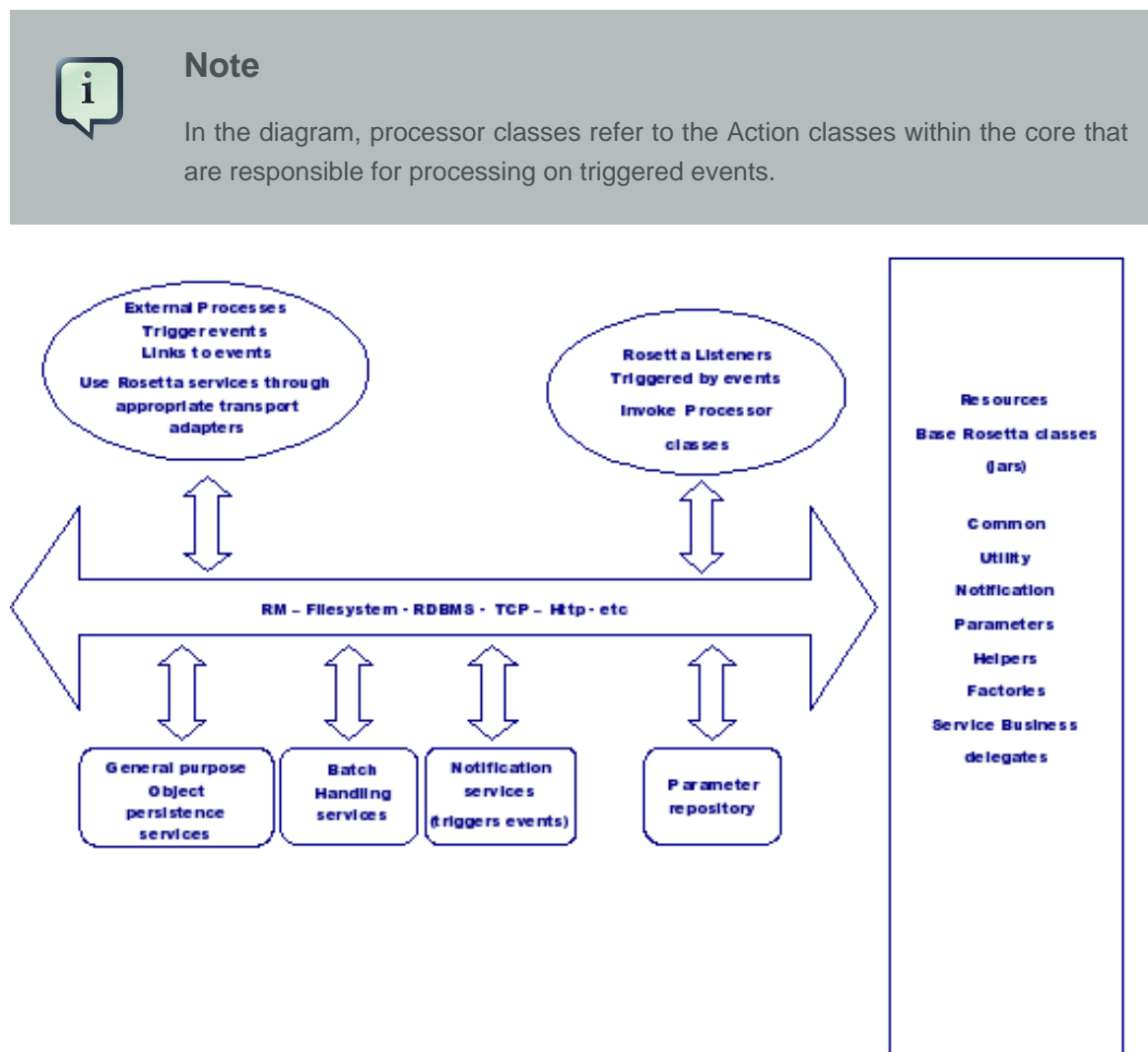
Figure 2.1. Rosetta

There are many reasons why users may want disparate applications, services and components to interoperate, e.g., leveraging legacy systems in new deployments. Furthermore, such interactions between these entities may occur both synchronously or asynchronously. As with most ESBs, Rosetta was developed to facilitate such deployments, but providing an infrastructure and set of tools that could:

• Be easily configured to work with a wide variety of transport mechanisms (e.g., email and JMS).

- Offer a general purpose object repository.

- Enable pluggable data transformation mechanisms.

- Support logging of interactions.

To date, Rosetta has been used in mission critical deployments using Oracle Financials. The multi platform environment included an IBM mainframe running z/OS, DB2 and Oracle databases hosted in the mainframe and in smaller servers, with additional Windows and Linux servers and a myriad of third party applications that offered dissimilar entry points for interoperation. It used JMS and MQSeries for asynchronous messaging and Postgress for object storage. Interoperation with third parties outside of the corporation's IT infrastructure was made possible using IBM MQSeries, FTP servers offering entry points to pick up and deposit files to/from the outside world and attachments in e-mail messages to 'well known' e-mail accounts.

As we shall see when examining the JBossESB core, which is based on Rosetta, the challenge was to provide a set of tools and a methodology that would make it simple to isolate business logic from transport and triggering mechanisms, to log business and processing events that flowed through the framework and to allow flexible plug ins of ad hoc business logic and data transformations. Emphasis was placed on ensuring that it is possible (and simple) for future users to replace/extend the standard base classes that come with the framework (and are used for the toolset), and to trigger their own 'action classes' that can be unaware of transport and triggering mechanisms.

> **Note**
>
> Within JBossESB source we have two trees: org.jboss.internal.soa.esb and org.jboss.soa.esb. You should limit your use of anything within the org.jboss.internal.soa.esb package because the contents are subject to change without notice. Alternatively anything within the org.jboss.soa.esb is covered by our deprecation policy.

2.2. The core of JBossESB in a nutshell

Rosetta is built on four core architectural components:

- Message Listener and Message Filtering code. Message Listeners act as "inbound" message routers that listen for messages (e.g. on a JMS Queue/Topic, or on the file system) and present the message to a message processing pipeline that filters the message and routes it ("outbound" router) to another message endpoint.

- Data transformation via the SmooksAction action processor. See the "Message Transformation" chapter in the ServicesGuide for further details.

- A Content Based Routing Service. See the "What is Content-Based Routing" chapter in the ServicesGuide for further information.

• A Message Repository, for saving messages/events exchanged within the ESB. See the "What is the Registry" chapter in the ServicesGuide for further details.

These capabilities are offered through a set of business classes, adapters and processors, which will be described in detail later. Interactions between clients and services are supported via a range of different approaches, including JMS, flat-file system and email.

A typical JBossESB deployment is shown below. We shall return to this diagram in subsequent sections.

> **Note**
>
> Some of the components in the diagram (e.g., LDAP server) are configuration choices and may not be provided out-of-the-box. Furthermore, the Processor and Action distinction shown in the above diagram is merely an illustrative convenience to show the concepts involved when an incoming event (message) triggers the underlying ESB to invoke higher-level services.
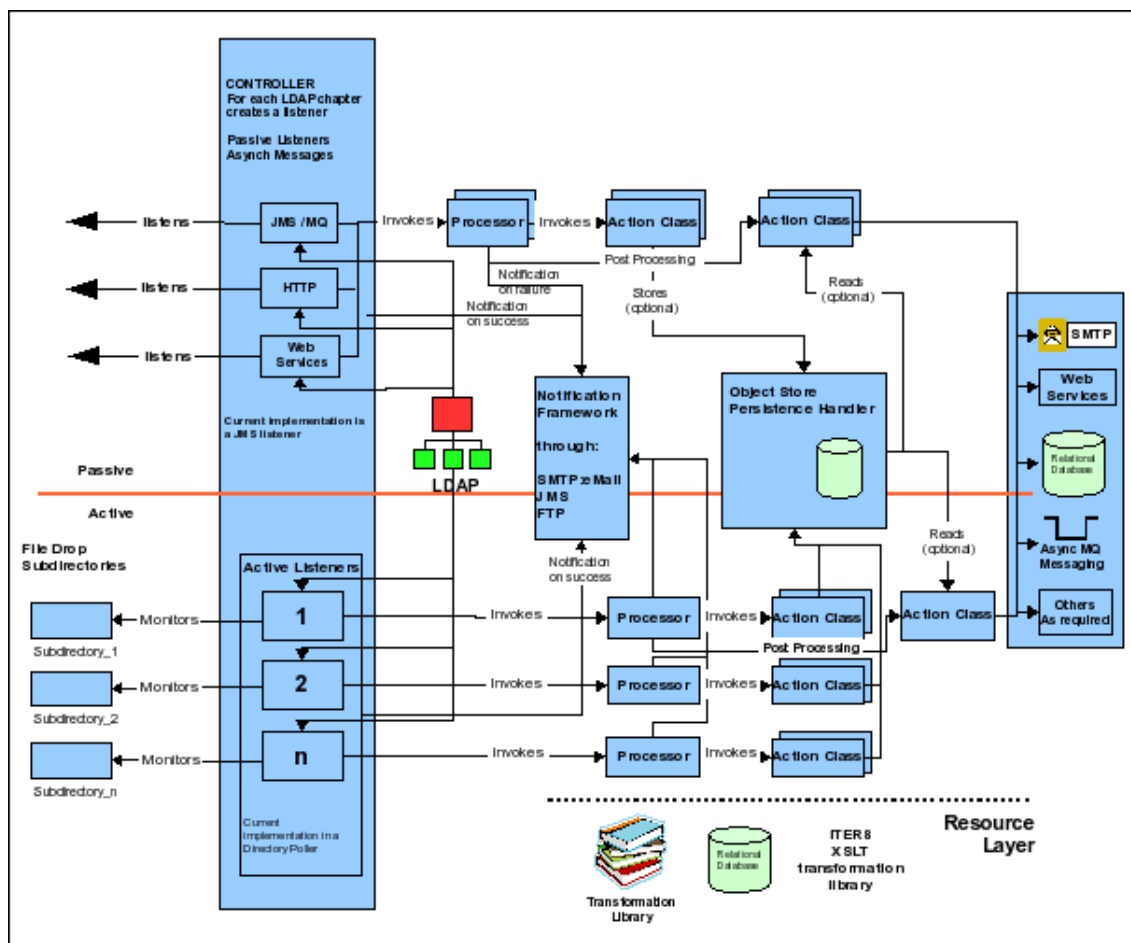
Figure 2.2. Typical Deployment

In the following chapters we shall look at the various components within JBossESB and show how they interact and can be used to develop SOA-based applications.

Services and Messages

In keeping with SOA principles, everything within JBossESB is considered to be either a service or a message. Services encapsulate the business logic or points of integration with legacy systems. Messages are the way in which clients and services communicate with each other.

In the following sections we shall look at how Services and Messages are supported within JBossESB.

3.1. The Service

A "Service" in JBossESB is defined as a list of "Action" classes that process an ESB Message in a sequential manner (see below). This list of Action classes is referred to as an "Action Pipeline". A Service can define a list of "Listeners", which act as inbound routers for the Service, routing messages to the Action Pipeline.

The following is a very simple JBossESB configuration that defines a single Service that simply prints the contents of the ESB Message to the console.

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

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

</jbossesb>
```

As you can see from the above example, a Service has "category" and "name" attributes. When JBossESB deploys the Service, it uses these attributes to register the Service endpoints (listeners) in the Service Registry (see Registry Guide). Clients can invoke the Service using the ServiceInvoker as follows.

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

message.getBody().add(#Hi there!#);
```

```
invoker.deliverAsync(message);
```

The ServiceInvoker uses the Service Registry (see ServicesGuide) to lookup the available Endpoint addresses for the "Retail:ShoeStore" Service. It takes care of all the transport details of getting the message from the Client to one of the available Service Endpoints (JMS, FTP, HTTP etc), hiding all of the lower level details from the Client.

The Endpoint addresses made available to the ServiceInvoker will depend on the list of listeners configured on the Service such as JMS, FTP or HTTP. No listeners are configured on the Service in the above example, but its InVM listener has been enabled using invmScope="GLOBAL"1. (The InVM transport is a new feature since JBossESB 4.3 that provides communication between services running on the same JVM. The section on "InVM Transport" contains more information about this feature.) To add additional Endpoints for the Service, we need to explicitly add listener configurations on the Service. JBossESB supports two forms of listener configuration:

- Gateway Listeners

  These listener configurations configure a "Gateway" Endpoint. These Endpoint types can be used to get messages onto an ESB bus. It is responsible for "normalizing" the message payload by wrapping it into an ESB Message (see below) before shipping it to the Service's Action Pipeline.

- ESB-Aware Listeners

  These listener configurations configure an "ESB Aware" Endpoint. These Endpoint types are used to exchange ESB Messages (see below) between ESB Aware components e.g. exchanging messages on the bus.

The following is an example of how a JMS Gateway listener can be added to the above ShoeStore Service.

```xml
<?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>
```

In the above configuration, we added a bus <providers> section to the configuration. This is where we configure the transport level details for Endpoints. In this case we added a <jms-provider> section that defines 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 two Endpoints – the InVM Endpoint and the JMS Gateway Endpoint. The ServiceInvoker will always use a Service's local InVM Endpoint, if available, in preference to other Endpoint types.

3.2. The Message

All interactions between clients and services within JBossESB occur through the exchange of Messages. In order to encourage loose coupling we recommend a message-exchange pattern based on one-way messages, i.e., requests and responses are independent messages, correlated where necessary by the infrastructure or application. Applications constructed in this way are less brittle and can be more tolerant of failures, giving developers more flexibility in their deployment and message delivery requirements.

To ensure loose coupling of services and develop SOA applications, it is necessary to:

1. Use one-way message exchanges rather than request-response.

2. Keep the contract definition within the exchanged messages. Try not to define a service interface that exposed back-end implementation choices, because that will make changing the implementation more difficult later.

3. Use an extensible message structure for the message payload so that changes to it can be versioned over time, for backward compatibility.

4. Do not develop fine-grained services: this is not a distributed-object paradigm, which can lead to brittle applications.

In order to use a one-way message delivery pattern with requests and responses, it is obviously necessary to encode information about where responses should be sent. That information may be present in the message body (the payload) and hence dealt with solely by the application, or part of the initial request message and typically dealt with by the ESB infrastructure.

Therefore, central to the ESB is the notion of a message, whose structure is similar to that found in SOAP:

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

Pictorially the basic structure of the Message can be represented as shown below. In the rest of this section we shall examine each of these components in more detail.

In UML, the Message structure can be represented as:
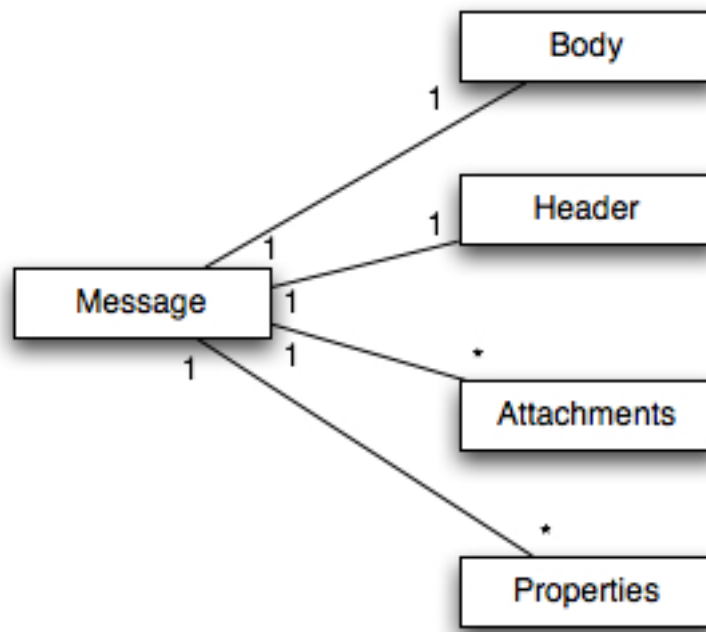
Figure 3.1. Basic Structure of a Message

Each message is an implementation of the org.jboss.soa.esb.message.Message interface. Within that package are interfaces for the various fields within the Message as shown below:

```java
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;
}
```

> ⚠️ **Warning**
>
> In JBossESB, Attachments and Properties are not treated differently from the Body. The general concepts they embody are currently being re-evaluated and may change significantly in future releases. As such, we recommend developers do not use Attachments.

The Header contains routing and addressing information for this message. As we saw earlier, JBossESB uses an addressing scheme based on the WS-Addressing standard from W3C. We shall discuss the org.jboss.soa.esb.addressing.Call class in the next section.

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

The Context contains session related information, such as transaction or security contexts.

> ℹ️ **Note**
>
> The 4.x release of JBossESB does not support user-enhanced Contexts. This will be a feature of the 5.0 release.

The Body typically contains the payload of the message. It may contain a list of Objects of arbitrary types. How these objects are serialized to/from the message body when it is transmitted is up to the specific Object type.

> **Note**
>
> You should be extremely careful about sending Serialized objects within the Body:
> not everything that can be Serialized will necessarily be meaningful at the receiver,
> e.g., database connections.

```java
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 ();
}
```

A Body can be used to convey arbitrary information types and arbitrary numbers of each type, i.e.,
it is not necessary to restrict yourself to sending and receiving single data items within a Body.

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

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

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

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

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

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

> **⚠ Warning**
>
> In JBossESB, Attachments and Properties are not treated differently from the Body. The general concepts they embody are currently being re-evaluated and may change significantly in future releases. As such, we recommend developers do not use Attachments or Properties.

A set of message properties, which can be used to define additional meta-data for 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();
}
```

> **ℹ Note**
>
> JBossESB does not implement Properties as java.util.Properties for the same reason Web Services stacks do not: it places restrictions on the types of clients and services that can used. If you need to send java.util.Properties then you can embed them within the current abstraction.

Messages may contain attachments that do not appear in the main payload body. For example, imagines, drawings, binary document formats, zip files etc. The Attachment interface supports both named and unnamed attachments.

```java
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();
}
```

Attachments may be used for a number of reasons (some of which have been outlined above). At a minimum, they may be used to more logically structure your message and improve performance of large messages, e.g., by streaming the attachments between endpoints.

> **Note**
>
> At present JBossESB does not support specifying other encoding mechanisms for the Message or attachment streaming. This will be added in later releases and where appropriate will be tied in to the SOAP-with-attachments delivery mechanism. Therefore, currently attachments are treated in the same way as named objects within the Body.

Given that there are attachments, properties, and named objects, you may be wondering where should you put your payload? The answer is fairly straightforward:

- As a service developer, you define the contract that clients use in order to interact with your service. As part of that contract, you will specify both functional and non-functional aspects of

the service, e.g., that it is an airline reservation service (functional) and that it is transactional (non-functional). You'll also define the operations (messages) that the service can understand. As part of the message definition, you stipulate the format (e.g., Java Serialized message versus XML) and the content (e.g., transaction context, seat number, customer name etc.) When defining the content, you can specify where in the Message your service will expect to find the payload. That can be in the form of attachments or specific named objects (even the default named object if you so wish). It is entirely up to the service developer to determine. The only restrictions are that objects and attachments must be globally uniquely named, or one service (or Action) may inadvertently pick up a partial payload meant for another if the same Message Body is forwarded across multiple hops.

• As a service users, you obtain the contract definition about the service (e.g., through UDDI or out-of-band communication) and this will define where in the message the payload must go. Information placed in other locations will likely be ignored and result in incorrect operation of the service.

> **Note**
>
> There is more information about how to define your Message payload in the Message Payload section of this document.

3.3. Getting and Setting Data on the Message Body

By default, all JBossESB 4.2.1GA+ components (Actions, Listeners, Gateways, Routers, Notifiers etc) get and set data on the message through the messages "Default Payload Location".

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

• get-payload-location: The location from which to retrieve the message payload.

• set-payload-location: The location on which to set the message payload.

> **Note**
>
> Prior to JBossESB 4.2.1GA there was no default message payload exchange pattern in place. JBossESB 4.2.1GA+ can be configured to exchange payload data according to the pre 4.2.1GA approach (i.e. is backward compatible with) by setting the "use.legacy.message.payload.exchange.patterns" property to "true" in the "core" section/module of the jbossesb-properties.xml file (found in the jbossesb.sar).

3.4. Extensions to Body

Although you can manipulate the contents of a Message Body directly in terms of bytes or name/value pairs, it is often more natural to use one of the following predefined Message structures, which are simply different views onto the data contained in the underlying Body.

As well as the basic Body interface, JBossESB supports the following interfaces, which are extensions on the basic Body interface:

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. The need for two different factories is explained in the section on Message Formats, which is described later in the document.

For each of the various Body types, you will find an associated create method (e.g., createTextBody) that allows you to create and initialize a Message of the specific type. Once created, the Message can be manipulated directly through the raw Body or via the specific interface. If the Message is transmitted to a recipient, then the Body structure will be maintained, e.g., it can be manipulated as a TextBody.

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

> **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.

3.5. The Message Header

As we saw above, the Header of a Message contains a reference to 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 properties below support both one way and request reply interaction patterns:

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

Property	Type	Required	Description
To	EPR	Yes	The address of the intended receiver of this message.
From	EPR	No	Reference of the endpoint where the message originated from.
ReplyTo	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.
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 (e.g., 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.

The relationship between the Header and the various EPRs can be illustrated as follows in UML:
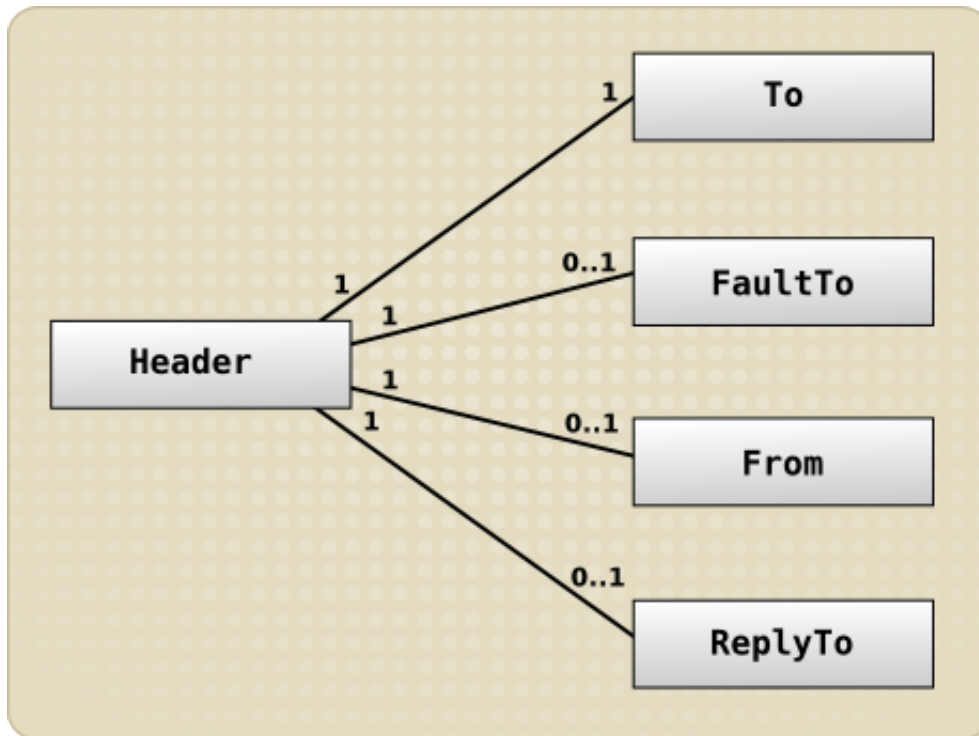
Figure 3.2. Relationship Between the Header and EPR in UML

When working with Messages, you should consider the role of the header when developing and using your clients and services. For example, if you require a synchronous interaction pattern based on request/response, you will be expected to set the ReplyTo field, or a default EPR will be used; even with request/response, the response need not go back to the original sender, if you so choose. Likewise, when sending one-way messages (no response), you should not set the ReplyTo field because it will be ignored.

> **Note**
>
> Please see details on the LogicalEPR.

> **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 by the creator and is immutable once transmitted between endpoints. Although the interfaces allow the recipient to modify the individual values, JBossESB will ignore such modifications. In future releases it is likely that such modifications will be disallowed by the API as well for improved clarity. These rules are laid down in the WS-Addressing standards.

3.6. LogicalEPR

A LogicalEPR is an EPR that simply specifies the name and category of an ESB Service/Endpoint. It contains no physical addressing information.

Clients setting the ReplyTo or FaultTo EPRs should always use the LogicalEPR, as opposed to one of the Physical EPRs (JMSEpr etc). The LogicalEPR is the preferred option because it makes no assumptions about the capabilities of the user of the EPR (typically the ESB itself, but not necessarily), or when the EPR will be used i.e. a physical EPR may no longer be valid by the time it gets used. By it's non-Physical nature, a LogicalEPR is also a lot easier to "handle" from a user perspective. The user of the LogicalEPR can use the Service name and category details supplied in the EPR to lookup the physical endpoint details for that Service/Endpoint at the point in time when they intend making the invocation i.e. they will get relevant addressing information. The user will also be able to select an endpoint type that suits it i.e. if it's easier for the user to make the invocation using a file based transport (Vs e.g. JMS), then they can select that type of transport.

3.7. Default FaultTo

When sending Messages, it is possible that errors will occur, either during the transmission or reception/processing of the Message. JBossESB will route any faults to the EPR mentioned in the FaultTo field of the incoming message. If this is not set, then it will use the ReplyTo field or, failing that, the From field. 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 about such faults, such as when sending a one-way message, you may wish to use the DeadLetter Queue Service EPR as your FaultTo. In this way, any faults that do occur will be saved for later processing.

> **Note**
>
> Please see details on the LogicalEPR.

3.8. Default ReplyTo

Because the recommended interaction pattern within JBossESB is based on one-way message exchange, responses to messages are not necessarily automatic: it is application dependent as

to whether or not a sender expects a response. As such, a reply address (EPR) is an optional part of the header routing information and applications should be setting this value if necessary. However, in the case where a response is required and the reply EPR (ReplyTo EPR) has not been set, JBossESB supports default values for each type of transport. Some of these ReplyTo defaults require system administrators to configure JBossESB correctly.

Table 3.2. 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	For both local and remote files, no administration changes are required: responses will be written into the same directory as the request but with a unique suffix to ensure that only the original sender will pick up the response.

Note

Please see details on the LogicalEPR.

3.9. The Message payload

From an application/service perspective the message payload is a combination of the Body and Attachments. In this section we shall give an overview of best practices when constructing and using the message payload.

Warning

In JBossESB, Attachments and Properties are not treated differently from the Body. The general concepts they embody are currently being re-evaluated and may change significantly in future releases. As such we shall not be considering the Attachments as part of the payload in the rest of this discussion.

The UML representation of the payload is shown below:

More complex content may be added through the add method, which supports named Objects. Using <name, Object> pairs allows for a finer granularity of data access. The type of Objects that can be added to the Body can be arbitrary: they do not need to be Java Serializable. However, in the case where non-Serializable Objects are added, it is necessary to provide JBossESB with the ability to marshal/unmarshal the Message when it flows across the network. See the section of Message Formats for more details.

If no name is supplied to set or get, then the default name defined by DEFAULT_LOCATION will be used.

> **Note**
>
> Be careful when using Serialized Java objects in messages because it constrains the service implementations.

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

> **Note**
>
> In general you will find it easier to work with the Message Body through the named Object approach. You can add, remove and inspect individual data items within the Message payload without having to decode the entire Body. Furthermore, you can combine named Objects within the payload with the byte array.

> **Note**
>
> In the current release of JBossESB only Java Serialized objects may be attachments. This restriction will be removed in a subsequent release.

3.10. The MessageFactory

Internally to an ESB component, the message is a collection of Java objects. However, messages need to be serialized for a number of reasons, e.g., transmitted between address spaces (processes) or saved to a persistent datastore for auditing or debugging purposes. The external representation of a message may be influenced by the environment in which the ESB is deployed. Therefore, JBossESB does not impose a specific normalized message format, but supports a range of them.

All implementations of the org.jboss.soa.esb.message.Message interface are obtained from the org.jboss.soa.esb.message.format.MessageFactory class:

Example 3.1. `org.jboss.soa.esb.message.format.MessageFactory`

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

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

Message serialization implementations are uniquely identified by a URI. The type of implementation required may be specified when requesting a new instance, or the configured default implementation may be used. Currently JBossESB provides two implementations, which are defined in the org.jboss.soa.esb.message.format.MessageType class:

- MessageType.JBOSS_XML: this uses an XML representation of the Message on the wire. The schema for the message is defined in the message.xsd within the schemas directory. The URI is urn:jboss/esb/message/type/JBOSS_XML.

- MessageType.JAVA_SERIALIZED: this implementation requires that all components of a Message are Serializable. It obviously requires that recipients of this type of Message have sufficient information (the Java classes) to be able to de-serialize the Message. The URI is urn:jboss/esb/message/type/JAVA_SERIALIZED.

> **Important**
>
> You should be wary about using the JAVA_SERIALIZED version of the Message format because it more easily ties your applications to specific service implementations, i.e., it breaks loose coupling.

Other Message implementations may be provided 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 using property names with the org.jboss.soa.esb.message.format.plugin extension.

> **Note**
>
> The default Message type is JBOSS_XML. However, this can be changed by setting the property org.jboss.soa.esb.message.default.uri to the desired URI.

3.11. Message Formats

As mentioned previously, JBossESB supports two serialized message formats: MessageType.JBOSS_XML and MessageType.JAVA_SERIALIZED. In the following sections we shall look at each of these formats in more detail.

`MessageType.JAVA_SERIALIZED`

This implementation requires that all contents are Java Serializable. Any attempt to add a non-Serializable object to the Message will result in a IllegalParameterException being thrown.

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

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

`MessageType.JBOSS_XML`

This implementation uses an XML representation of the Message on the wire. The schema for the message is defined in the message.xsd within the schemas directory. Arbitrary objects may be added to the Message, i.e., they do not have to be Serializable. Therefore, it may be necessary to provide a mechanism to marshal/unmarshal such objects to/from XML when the Message needs to be serialized. This support can be provided through the org.jboss.soa.esb.message.format.xml.marshal.MarshalUnmarshalPlugin:

```java
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 ();
}
```

> **Note**   29
>
> Java Serialized objects are supported by default.

Plug-ins can be registered with the system through the jbossesb-properties.xml configuration file. They should have attribute names that start with the MARSHAL_UNMARSHAL_PLUGIN. When packing objects in XML, JBossESB runs through the list of registered plug-ins until it finds one that can deal with the object type (or faults). When packing, the name (type) of the plug-in that packed the object is also attached to facilitate unpacking at the Message receiver.

Now that we have looked at the concepts behind services and Messages, we shall examine how to construct services using the framework provided by Rosetta in the following Chapter.

Building and Using Services

4.1. Listeners, Routers/Notifiers and Actions

4.1.1. Listeners

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

> **i** **Note**
>
> Various parameters can be configured for listeners, such as the number of active worker threads. See the chapter on Defining Service Configurations for a full range of these options.

4.1.2. Routers

Routers are the way in which either the ESB Message itself of its payload can be routed to an endpoint. 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 ESB Message as is and the receiving endpoint must be ESB-aware so that it can deal with the message.

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

Other routers like StaticWiretap, StaticRouter are only for routing to other services. There are also router that can be used for dynamic routing based on the message content. Please see the section "What is Content Based Routing" in the ServicesGuide for more formation on content based routers.

> **i** **Note**
>
> For information about the usage of different routers please see the routers section in the "Out-of-the-box Actions" chapter.

4.1.3. Notifiers

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

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

> **Note**
>
> JMSNotifier was sending the type of JMS message (TextMessage or ObjectMessage) depending upon the type of ESB Message (XML or Serializable, respectively). This was wrong, as the type of ESB Message should not affect the way in which the Notifier sends responses. As of JBossESB 4.2.1CP02, the message type to be used by the Notifier can be set as a property (org.jboss.soa.esb.message.transport.jms.nativeMessageType) on the ESB message. Possible values are NotifyJMS.NativeMessage.text or NotifyJMS.NativeMessage.object. For backward compatibility with previous releases, the default value depends upon the ESB Message type: object for Serializable and text for XML. However, we encourage you not to rely on defaults.

As outlined above, the responsibility of a listener is to act as a message delivery endpoint and to deliver messages to an "Action Processing Pipeline". Each listener configuration needs to supply information for:

- the Registry (see service-category, service-name, service-description and EPR-description tag names). If you set the optional remove-old-service tag name to true then the ESB will remove any existing service entry from the Registry prior to adding this new instance. However, this should be used with care, because the entire service will be removed, including all EPRs.

- instantiation of the listener class (see listenerClass tag name).

- the EPR that the listener will be servicing. This is transport specific. The following example corresponds to a JMS EPR (see connection-factory, destination-type, destination-name, jndi-type, jndi-URL and message-selector tag names).

- the "action processing pipeline". One or more <action> elements each that must contain at least the 'class' tagname that will determine which action class will be instantiated for that step in the processing chain.

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

```xml
<jbossesb xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/
jbossesb-1.0.1.xsd" parameterReloadSecs="5">

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

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

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

```
        </NotificationList>
      </property>

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

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

action1
    MyJMSListenerAction (a trivial example follows)

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

The following trivial action class will prove useful for debugging your XML action configuration

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

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

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

- A public constructor that takes a single argument of type ConfigTree

- One or more public methods that take a Message argument, and return a Message result

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

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

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

2. Analyze contents of the 'process' attribute.

   Contents can be a comma separated list of public method names of the instantiated class (step 1), each of which must take a single argument of type Message, and return a Message object that will be passed to the next step in the pipeline

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

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

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

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

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

5. Callback method for success in each <action> element: If the list of methods in the 'process' attribute was executed successfully, the pipeline will analyze contents of the 'okMethod' attribute. If none is specified, processing will continue with the next <action> element. If a method name is provided in the 'okMethod' attribute, it will be invoked using the Message returned by the last method in step 3. If the pipeline succeeds then the okMethod notification will be called on all handlers from the last one back to the initial one.

6. Callback method for failure in each <action> element: If an Exception occurs then the exceptionMethod notification will be called on all handlers from the current (failing) handler back to the initial handler. At present time, if no exceptionMethod was specified, the only output will be the logged error. If an ActionProcessingFaultException is thrown from any process method then an error message will be returned as per the rules defined in the next section. The contents of the error message will either be whatever is returned from the getFaultMessage of the exception, or a default Fault containing the information within the original exception.

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

> **Note**
>
> In JBossESB 4.3 the name of the property used to enclose NotificationList elements in the <action> target is not validated.

4.1.4. Actions and Messages

Actions are triggered by the arrival of a Message. The specific Action implementation is expected to know where the data resides within a Message. Because a Service may be implemented using an arbitrary number of Actions, it is possible that a single input Message could contain information on behalf of more than one Action. In which case it is incumbent on the Action developer to choose one or more unique locations within the Message Body for its data and communicate this to the Service consumers.

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

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

If Actions share data within an input Message and each one modifies the information as it flows through the chain, by default we recommend retaining the original information so that Actions further down the chain still have access to it. Obviously there may be situations where this is either not possible or would be unwise. Within JBossESB, Actions that modify the input data can place this within the org.jboss.soa.esb.actions.post named Body location. This means that if there are N Actions in the chain, Action N can find the original data where it would normally look, or if Action N-1 modified the data then N will find it within the other specified location. To further facilitate Action chaining, Action N can see if Action N-2 modified the data by looking in the org.jboss.soa.esb.actions.pre named Body location.

> **Note**
>
> As mentioned earlier, you should use the default named Body location with care when chaining Actions in case chained Actions use it in a conflicting manner.

4.1.5. Handling Responses

There are two processing mechanisms supported for handling responses in the action pipeline, implicit processing (based on the response of the actions) and explicit processing.

If the processing is implicit then responses will be processed as follows:

- If any action in the pipeline returns a null message then no response will be sent.

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

If the processing is explicit then responses will be processed as follows:

- If the action pipeline is specified as 'OneWay' then the pipeline will never send a response

- If the pipeline is specific as 'RequestResponse' then a reply will be sent to the ReplyTo EPR of the request message or, if not set, to the From EPR of the request message. In the event that there is no EPR is specified then no error message will be logged by the system.

> **Note**
>
> We recommend that all action pipelines should use the explicit processing mechanism. This can be enabled by simply adding the 'mep' attribute to the 'actions' element in the jboss-esb.xml file. The value of this attribute should be either 'OneWay' or 'RequestResponse'.

4.1.6. Error handling when processing actions

When processing an action chain, it is possible that errors may occur. Such errors should be thrown as exceptions from the Action pipeline, thus terminating the processing of the pipeline. As mentioned earlier, a Fault Message may be returned within an ActionProcessingFaultException. If it is important for information about errors to be returned to the sender (or some intermediary) then the FaultTo EPR should be set. If this is not set, then JBossESB will attempt to deliver error messages based on the ReplyTo EPR and, if that is also not set, the From EPR. If none of these EPRs has been set, then error information will be logged locally.

Error messages of various types can be returned from the Action implementations. However, JBossESB supports the following "system" error messages, all of which may be identified by the mentioned URI in the message Fault, in the case that an exception is thrown and no application specific Fault Message is present:

urn:action/error/actionprocessingerror
    this means that an action in the chain threw an ActionProcessingFaultException but did not include a fault message to return. The exception details will be contained within the "reason" String of the Fault.

urn:action/error/unexpectederror
    an unexpected exception was caught during the processing. Details about the exception can be found in the "reason" String of the Fault.

urn:action/error/disabled
    action processing is disabled.

If an exception is thrown within your Action chain, then it will be propagated back to the client within a FaultMessageException, which is re-thrown from the Courier or ServiceInvoker classes. This exception, which is also thrown whenever a Fault message is received, will contain the Fault code and reason, as well as any propagated exception.

4.2. Meta-Data and Filters

As a message flows through the ESB it may be useful to attach meta-data to it, such as the time it entered the ESB and the time it left. Furthermore, it may be necessary to dynamically augment the message; for example, adding transaction or security information. Both of these capabilities are supported in JBossESB through the filter mechanism, for both gateway and ESB nodes.

> **Note**
>
> The filter property name, the package for the InputOutputFilter and its signature all changed in JBossESB 4.2 MR3 from earlier milestone releases.

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

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

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

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

> **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 all ESB instances can process the same meta-data.

JBossESB ships with:

- org.jboss.internal.soa.esb.message.filter.MetaDataFilter

- org.jboss.internal.soa.message.filter.GatewayFilter

- org.jboss.internal.soa.esb.message.filter.EntryExitTimeFilter

which add the following meta-data to the Message as Properties with the indicated property names and the returned String values.

Gateway-related Message Properties

`org.jboss.soa.esb.message.transport.type`
　　File, FTP, JMS, SQL, or Hibernate.

`org.jboss.soa.esb.message.source`
　　The name of the file from which the message was read.

`org.jboss.soa.esb.message.time.dob`
　　The time the message entered the ESB, e.g., the time it was sent, or the time it arrived at a gateway.

`org.jboss.soa.esb.mesage.time.dod`
　　The time the message left the ESB, e.g., the time it was died.

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

`org.jboss.soa.esb.gateway.original.queue.name`
　　If the message was received via a JMS gateway node, then this element will contain the name of the queue from which it was received.

`org.jboss.soa.esb.gateway.original.url`
　　If the message was received via a SQL gateway node, then this element will contain the original database URL.

> ### Note
>
> Although it is safe to deploy the GatewayFilter on all ESB nodes, it will only add information to a Message if it is deployed on a gateway node.

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

Gateway-Generated Message Parameters

`org.jboss.soa.esb.gateway.file`
> The File from which the Message was sourced. This will only be present if this gateway is file based.

`org.jboss.soa.esb.gateway.config`
> The ConfigTree that was used to initialize the gateway instance.

> **Note**
>
> Only file based, JMS and SQL gateways have support for the GatewayFilter in JBossESB 4.8.

4.3. What is a Service?

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

As such and as we have seen, JBossESB uses a message driven pattern for service definitions and structures: clients send Messages to services and the basic service interface is essentially a single process method that operates on the Message received. Internally a service is structured from one or more Actions, that can be chained together to process incoming the incoming Message. What an Action does is implementation dependent, e.g., update a database table entry, or call an EJB.

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

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

Clients can then use the service as long as they do so according to the published contract. How your service processes the Message and performs the work necessary, is an implementation choice. It could be done within a single Action, or within multiple Actions. There will be the usual trade-offs to make, e.g., manageability versus re-useability.

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

4.3.1. ServiceInvoker

From a clients perspective, the Courier interface and its various implementations can be used to interact with services. However, this is still a relatively low-level approach, requiring developer code to contact the registry and deal with failures. Furthermore, since JBossESB has fail-over capabilities for stateless services, this would again have to be managed by the application. See the Advanced chapter for more details on fail-over.

In JBossESB 4.2, the ServiceInvoker was introduced to help simplify the development effort. The ServiceInvoker hides much of the lower level details and opaquely works with the stateless service fail-over mechanisms. As such, ServiceInvoker is the recommended client-side interface for using services within JBossESB.

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

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

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

> **Note**
>
> From JBossESB 4.5 onwards the ResponseTimeoutException is thrown, which is derived from MessageDeliverException.

Failures to contact the Registry or to successfully look up the service are indicated by throwing a RegistryException from deliverSync. Timeout values may indicate that the service has failed, simply overloaded and cannot respond in the time or that the work requested takes longer than the timeout allowed. In some cases the problem will be transient and trying again later may be sufficient.

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

> **Note**
>
> When using the deliverAsync method all errors are thrown as MessageDeliverExcepion instances, with the actual exception embedded within it.

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

We will discuss the fail-over properties of JBossESB in the Advanced section and how ServiceInvoker can opaquely mask failures of individual service instances if multiple copies appear in the registry. However, in some cases it may be desired to prevent automatic fail-over and inform the application immediately that a failure occurs. This can be set at the global level by setting the org.jboss.soa.esb.exceptionOnDeliverFailure property to true in the JBossESB property file. Alternatively this can be configured on a per message basis by setting the same property in the specific Message property to true. In both cases the default is false.

4.3.2. Transactions

Some Couriers support transactional delivery semantics, e.g., InVM and JMS. The current delivery semantics for such Couriers is based on JMS transacted sessions, i.e., the message is placed on a queue within the scope of a transaction but not actually delivered until the enclosing transaction has committed; it is then pulled by the receiver in the scope of a separate transaction. Unfortunately for synchronous request/response interactions this can result in a timeout waiting for the response since the sender blocks waiting for the response before it can terminate the delivery transaction.

From JBossESB 4.5 onwards we attempt to detect these blocking situations and throw the IncompatibleTransactionScopeException. The application should catch this and act accordingly.

4.3.3. Services and the ServiceInvoker

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

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

4.3.4. InVM Transport

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

> **i**
>
> **Note**
>
> It is important to realize that 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 shutdown before the queue is emptied, those Messages will not be delivered. Further limitations are mentioned throughout this section within corresponding Notes. 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.

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

```xml
<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.

4.3.5. InVM Scope

InVM Service invocation scope can be controlled through the "invmScope" attribute on the <service> element. The ESB currently supports 2 scopes:

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

2. GLOBAL: (Default) The Service is invokable over the InVM transport from within the same Classloader scope.

A "LOCAL" scope will be added in a future release, which will restrict invocation to within the same .esb deployment.

Each service can specify their own InVM scope in the invmScope attribute on the <service> element of their services configuration:

```xml
<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>
```

The default InVM Scope for an ESB deployment can be set in the jbossesb-properties.xml file through the "core:jboss.esb.invm.scope.default" config property. If not defined, the default scope is "GLOBAL".

4.3.6. InVM Transacted

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.

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.

4.3.7. Transaction Semantics

The InVM transport in JBossESB 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 shutdown.

> **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, we recommend that you use one of the other transactional transports, such as JMS or database.

When using InVM within a transaction, the message will not appear on the receiver's queue until the transaction commits, although the sender will get an immediate acknowledgement 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.

4.3.8. Threading

To change the number of listener threads associated with an InVM transport,

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

4.3.9. Lock-step Delivery

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

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

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

Lock Step delivery is disabled by default, but can be easily enabled for a service using its <property> settings on the <service>:

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

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

```xml
<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 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.

4.3.10. Load Balancing

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.

4.3.11. Pass-by-Value/Pass-by-Reference

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":

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

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

4.4. Service Contract Definition

A contract definition can be specified on a service by the inclusion of XML schema definitions representing the incoming request, outgoing response and fault detail messages which are supported by the corresponding service. The schemas representing the request and response messages are used to define the format of the contents for the main body section of the message and can enforce validation of that content.

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

Table 4.1. Service Contract Attributes

Name	Description	Type
inXsd	The resource containing the schema for the request message, representing a single element.	xsd:string
outXsd	The resource containing the schema for the response message, representing a single element.	xsd:string
faultXsd	A comma separated list of schemas, each representing one or more fault elements.	xsd:string

Name	Description	Type
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

4.5. Message validation

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

Validation is disabled by default.

4.6. Exposing an ESB service as a webservice

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

Table 4.2. Web Service Attributes

Name	Description
false	No web service end-point will be published.
true	A web service end-point is published (default).

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

Table 4.3. 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 4.4. WS-Addressing Properties

Property	Description
`org.jboss.soa.esb.gateway.ebws.messageID`	The WS-Addressing message id.

Property	Description
`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.

```xml
<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'.

```xml
<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>
```

Other Components

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

5.1. The Message Store

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

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

5.2. Data Transformation

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

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

> **Note**
>
> For more details see the "Message Transformations" chapter in the ServicesGuide.

5.3. Content-Based Routing

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

5.4. The Registry

In the context of SOA, a registry provides applications and businesses a central point to store information about their services. It is expected to provide the same level of information and the same breadth of services to its clients as that of a conventional market place. Ideally a registry should also facilitate the automated discovery and execution of e-commerce transactions and enabling a dynamic environment for business transactions. Therefore, a registry is more than an "e-business directory". It is an inherent component of the SOA infrastructure.

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

Example

6.1. How to use the Message

The Message is a critical component in the SOA development approach. In contains application specific data sent from clients to services and vice versa. In some cases that data may be as simple as "turn on the light", or as complex as "search this start chart for any anomalous data that may indicate a planet." What goes into a Message is entirely application specific and represents an important aspect of the contract between a service and its clients. In this section we shall describe some best practices around the Message and how to use it.

Let's consider the following example which uses a Flight Reservation service. This service supports the following operations:

reserveSeat

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

querySeat

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

upgradeSeat

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

When developing this service, it will likely use technologies such as EJB3, Hibernate etc. to implement the business logic. In this example we shall ignore how the business logic is implemented and concentrate on the service.

The role of the service is to plug the logic into the bus. In order to do this, we must determine how the service is exposed on to the bus, i.e., what contract it defines for clients. In the current version of JBossESB, that contract takes the form of the Messages that clients and services can exchange. There is no formal specification for this contract within the ESB, i.e., at present it is something that the developer defines and must communicate to clients out-of-band from the ESB. This will be rectified in subsequent releases.

6.1.1. The Message Structure

From a service perspective, of all the components within a Message, the Body is probably the most important, since it is used to convey information specific to the business logic. In order to interact, both client and service must understand each other. This takes the form of agreeing on the transport (e.g., JMS or HTTP), as well as agreeing on the dialect (e.g., where in the Message data will appear and what format it will take).

If we take the simple case of a client sending a Message directly to our Flight Reservation service, then we need to determine how the service can determine which of the operations the Message concerns. In this case the developer decides that the opcode (operation code) will appear within

the Body as a String ("reserve", "query", "upgrade") at the location "org.example.flight.opcode". Any other String value (or the absence of any value) will be considered an illegal Message.

> **ℹ Note**
>
> It is important that all values within a Message are given unique names, to avoid clashes with other clients or 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 type. The other parameters necessary for carrying out the business logic associated with each operation would also be suitably encoded.

- "org.example.flight.seatnumber" for the seat number, which will be an integer.

- "org.example.flight.flightnumber" for the flight number, which will be a String.

- "org.example.flight.upgradenumber" for the upgraded seat number, which will be an integer.

Table 6.1. Operation Parameters

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

As we have mentioned, all of these operations return information to the client. Such information will likewise be encapsulated within a Message. The determination of the format of such response Messages will go through the same processes as we are currently describing. For simplification purposes we shall not consider the response Messages further.

From a JBossESB Action perspective, the service may be built using one or more Actions. For example, one Action may pre-process the incoming Message and transform the content in some way, before passing it on to the Action which is responsible for the main business logic. Each of these Actions may have been written in isolation (possibly by different groups within the same organization or by completely different organizations). In such an architecture it is important that each Action has its own unique view of where the Message data resides that is of interest only to that Action or it is entirely possible for chained Actions to overwrite or interfere with one another.

6.1.2. The Service

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

```
class AirlineReservationSystem
```

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

> **i** **Note**
>
> You could develop your business logic from POJOs, EJBs, Spring etc. JBossESB provides support for many of these approaches out of the box. You should examine the relevant documentation and examples.

The process method of the service Action (we'll assume no chaining of Actions) then becomes (ignoring error checking):

```
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;
}
```

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

6.1.3. Unpicking the payload

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

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

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

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

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

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

   // now deliver the response Message
}
```

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

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

6.1.4. The Client

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

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

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

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

Message response = null;

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

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

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

} while maximumRetriesNotExceeded;
```

> **Note**
>
> Much of what we have outlined above may seem similar to those who have
> worked with traditional client/server stub generators. In those systems, the low-
> level details, such as opcodes and parameters, would be hidden behind higher
> level stub abstractions. In future releases of JBossESB we intend to support such
> abstractions to easy the development approach. As such, working with the raw
> Message components, such as Body and Header, will be hidden from the majority
> of developers.

6.1.5. Configuration for a remote ServiceInvoker

Using the ServiceInvoker from within actions in the ESB does not require any additional
configuration and will work out of the box. But using the ServiceInvoker from a remote JVM, such
as from a stand-alone Java application, a servlet, an EJB etc, will require the following jars need
to be available:

```
jbossesb-rosetta.jar                    trove.jar
jbossesb-config-model-1.0.1.jar         juddi-client-3.0.0.aop.jar
jbossts-common.jar                      juddi-core-3.0.0.aop.jar
log4j.jar                               juddi-ws-3.0.0.jar
stax-ex.jar                             commons-configuration-1.5.jar
stax-api-1.0.1.jar                      commons-lang-2.3.jar
jbossall-client.jar                     jboss-messaging-client.jar
scout-1.2.aop.jar                       jboss-remoting.jar
xbean-2.2.0.jar                         commons-codec-1.3.jar
commons-logging.jar                     wstx-asl-3.2.0.jar
jboss-aop-jdk50.jar                     xercesImpl.jar
javassist.jar
```

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

- `jbossesb-properties.xml`

- `META-INF/uddi.xml`

6.1.6. Sample Client

The following Java program can be used to verify that the configuration of a remote client. This assumes that the helloworld quickstart has been deployed and that the ESB server is running.

```java
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();
    }
  }
}
```

6.1.7. Hints and Tips

You may find the following useful when developing your clients and services.

- When developing your Actions make sure that any payload information specific to an Action is maintained in unique locations within the Message Body.

- Try not to expose any back-end service implementation details within your Message. This will make it difficult to change the implementation without affecting clients. Message definitions (contents, formats etc.) which are implementation agnostic help to maintain loose coupling.

- For stateless services, use the ServiceInvoker as it will opaquely handle fail-over.

- When building request/response applications, use the correlation information (MessageID and RelatesTo) within the Message Header.

- Consider using the Header Action field for your 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 JBossESB provides more automatic support for service contract definitions and dissemination, consider maintaining a separate repository of these definitions that is available to developers and users.

Advanced Topics

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

7.1. Fail-over and load-balancing support

In mission critical systems it is important to design with redundancy in mind. JBossESB 4.2.GA is the first version with built-in fail-over, load balancing and delayed message redelivery to help you build a robust architecture. When you use SOA it is implied that the Service has become the building unit. JBossESB allows you to replicate identical services across many nodes. Where each node can be a virtual or physical machine running an instance of JBossESB. The collective of all these JBossESB instances is called "The Bus". Services within the bus use different delivery channels to exchange messages. In ESB terminology one such channel maybe JMS, FTP, HTTP, etc. These different "protocols" are provided by systems external to the ESB; the JMS-provider, the FTP server, etc. Services can be configured to listen to one or more protocols. For each protocol that it is configured to listen on, it creates an End Point Reference (EPR) in the Registry.

7.1.1. Services, EPRs, listeners and actions

As we have discussed previously, within the jboss-esb.xml each service element consists of one or more listeners and one or more actions. Let's take a look at the JBossESBHelloworld example. The configuration fragment below is loosely based on the configuration of the JBossESBHelloworld example. When the service initializes it registers the category, name and description to the UDDI registry. Also for each listener element it will register a ServiceBinding to UDDI, in which it stores an EPR. In this case it will register a JMSEPR for this service, as it is a jms-listener. The jms specific like queue name etc are not shown, but appeared at the top of the jboss-esb.xml where you can find the 'provider' section. In the jms-listener we can simply reference the "quickstartEsbChannel" in the busidref attribute.

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

Given the category and service name, another service can send a message to our Hello World Service by looking up the Service in the Registry. It will receive the JMSEPR and it can use that to send a message to. All this heavy lifting is done in the ServiceInvoker class. When our

HelloWorld Service receives a message over the quickstartEsbChannel, it will hand this message to the process method of the first action in the ActionPipeline, which is the SystemPrintln action.

> **Note**
>
> Because ServiceInvoker hides much of the fail-over complexity from users, it necessarily only works with native ESB Messages. Furthermore, in JBossESB 4.2.1 not all gateways have been modified to use the ServiceInvoker, so incoming ESB-unaware messages to those gateway implementations may not always be able to take advantage of service fail-over.

7.1.2. Replicated Services

In our example we have this service running on let's say Node1. What happens if we simply take the helloworld.esb and deploy it to Node2 as well (see figure 7-2)? Let's say we're using jUDDI for our Registry and we have configured all our nodes to access one central jUDDI database (it is recommended to use a clustered database for that). Node2 will find that the FirstServiceESB - SimpleListener Service is already registered! It will simply add a second ServiceBinding to this service. So now we have 2 ServiceBindings for this Service. We now have our first replicated Service! If Node1 goes down, Node2 will keep on working.
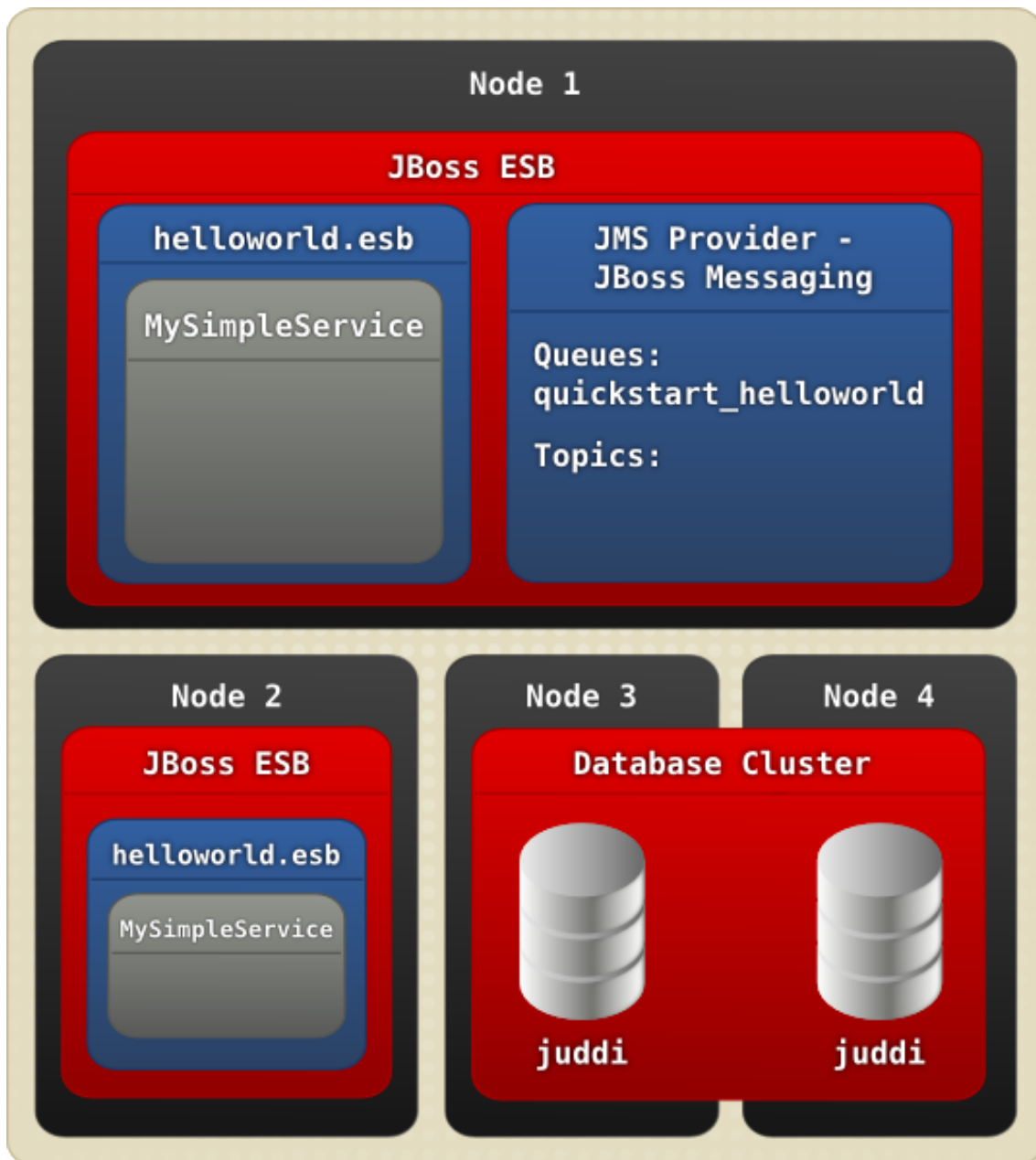
Figure 7.1. Two service instances each on a different node.

You will get load balancing as both service instances listen to the same queue. However this means that we still have a single point of failure in our setup. This is where Protocol Clustering maybe an option, which we shall describe in the next section.

This type of replication can be used to increase the availability of a service or to provide load balancing. To further illustrate, consider the diagram below which has a logical service (Application Service) that is actually comprised of 4 individual services, each of which provides the same capabilities and conforms to the same service contract. They differ only in that they do not need to share the same transport protocol. However, as far as the users of Application Service are concerned they see only a single service, which is identified by the service name and category.

The ServiceInvoker hides the fact that Application Service is actually composed of 4 other services from the clients. It masks failures of the individual services and will allow clients to make forward progress as long as at least one instance of the replicated service group remains available.

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

Service D (SQL)

Service C (FTP)

Service A (JMS)

Service B (FTP)

Application Service

Registry

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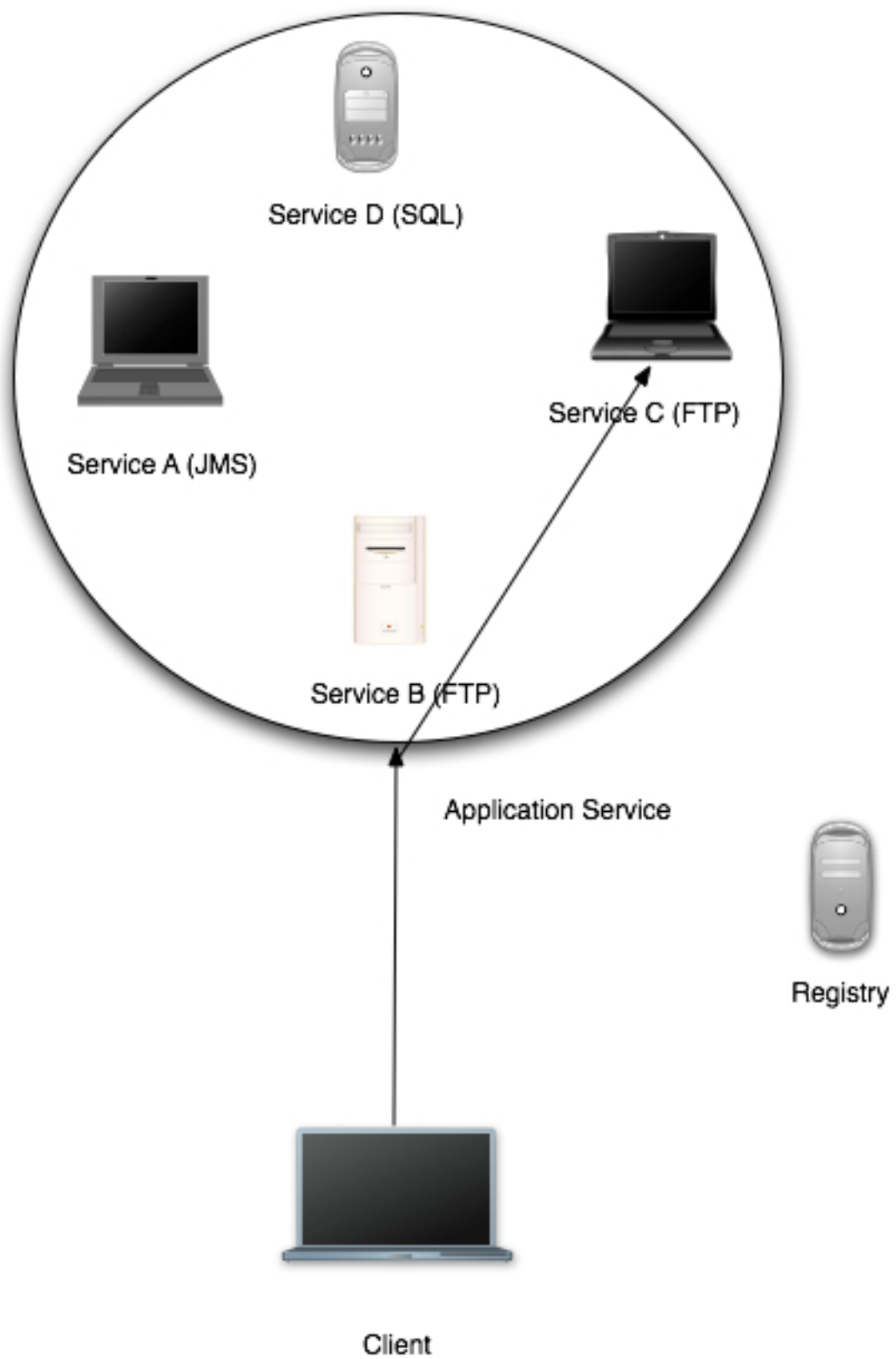
Replication of services may be defined by service providers outside of the control of service consumers. As such, there may be times when the sender of a message does not want to silently fail-over to using an alternative service if one is mentioned within the Registry. As such, if the Message property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

7.1.3. Protocol Clustering

Some JMS providers can be clustered. JBossMessaging (*http://community.jboss.org/wiki/ JBossMessaging*)is one of these providers, which is why we use this as our default JMS provider in JBossESB (*http://community.jboss.org/wiki/jbossesb*). When you cluster JMS you remove a single point of failure from your architecture.
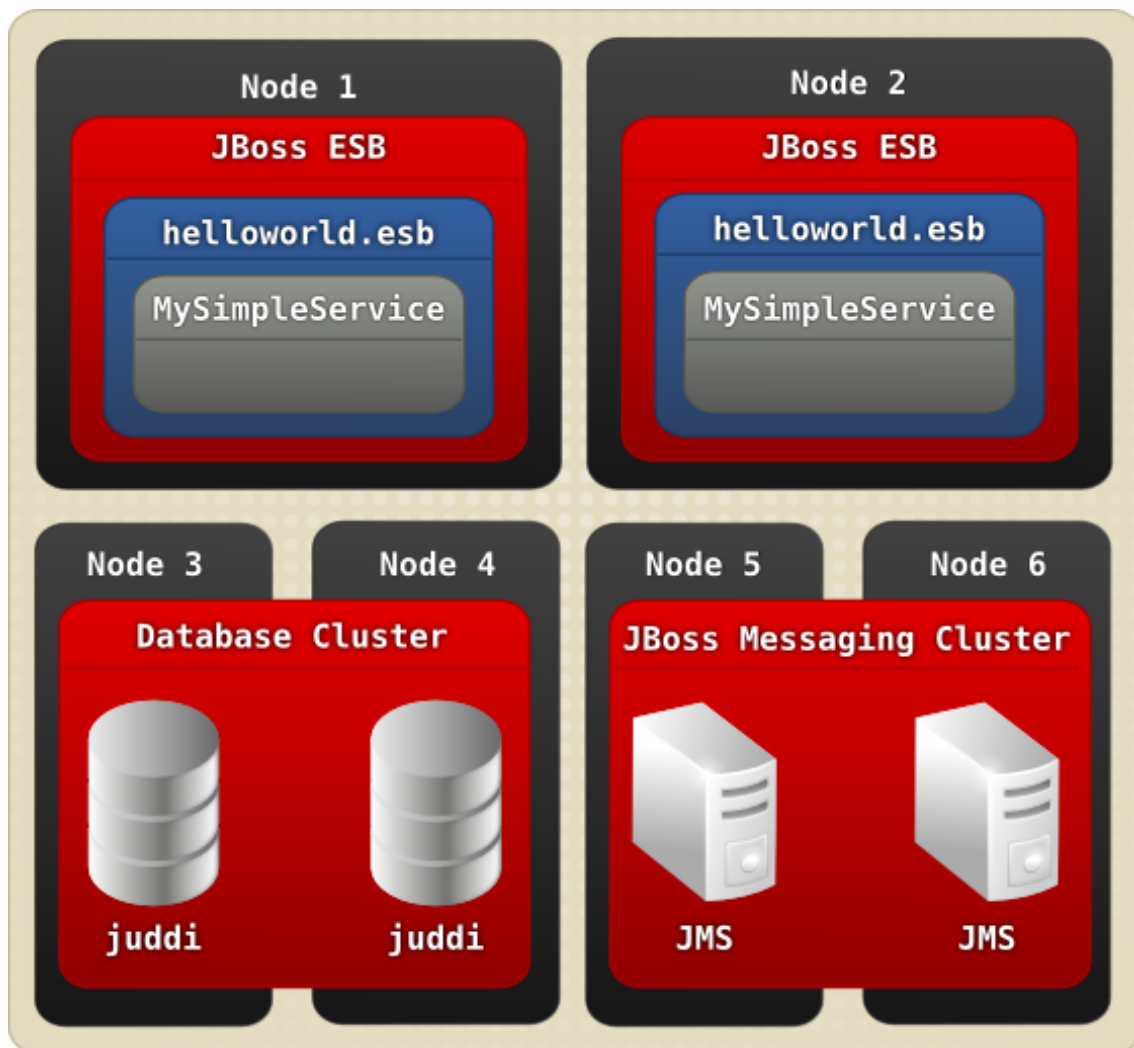
Figure 7.2. Example of Protocol Clustering Using JMS

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

Service D (SQL)

Service C (FTP)

Service A (JMS)

Service B (FTP)

Application Service

Registry

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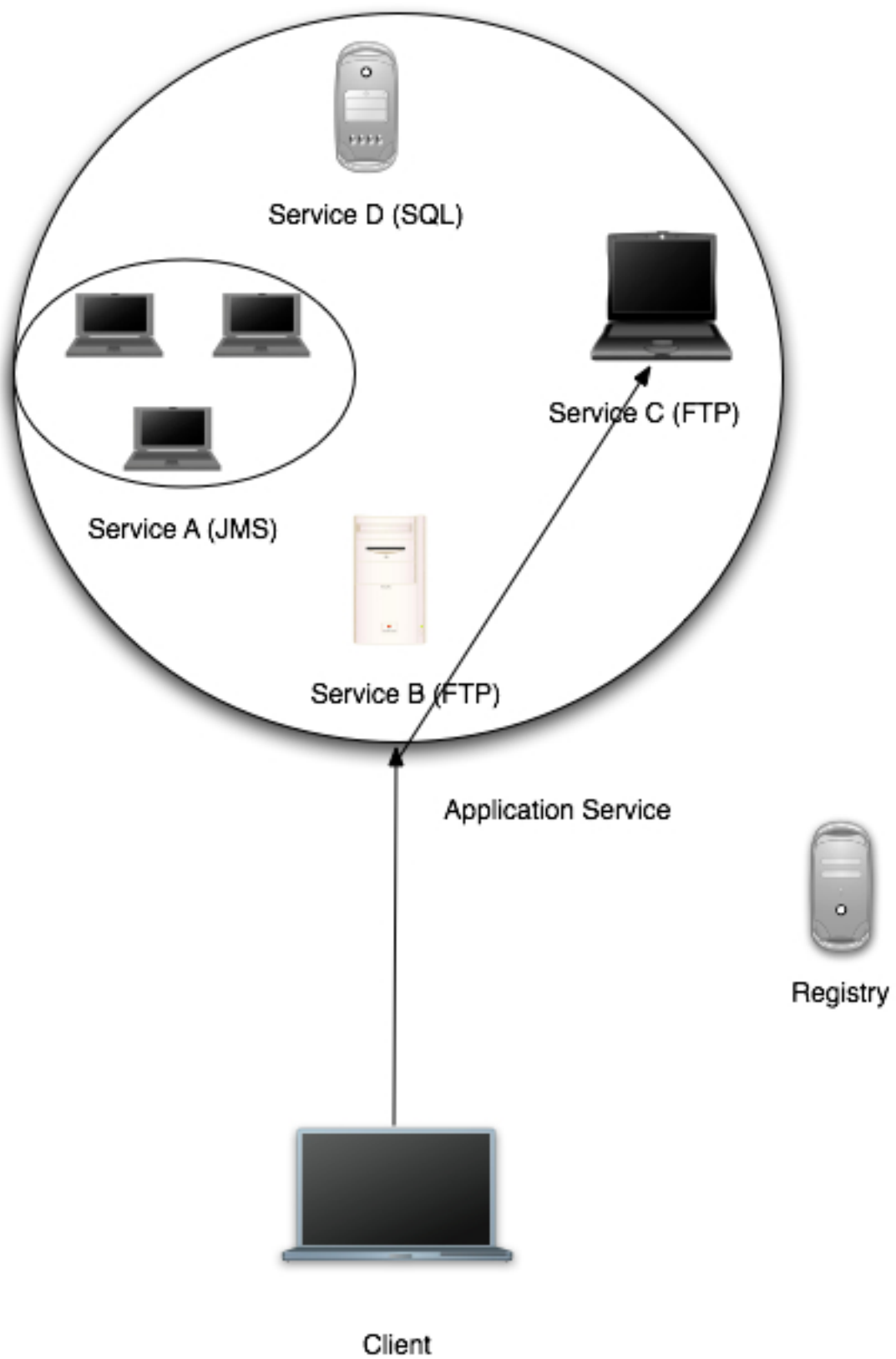
> **Note**
>
> If using JMS clustering in this way you will obviously need to ensure that your configuration is correctly configured. For instance, if you place all of your ESB services within a JMS cluster then you cannot expect to benefit from ESB replication.

Other examples of Protocol Clustering would be a NAS for the FileSystem protocol, but what if your provider simply cannot provide any clustering? Well in that case you can add multiple listeners to your service, and use multiple (JMS) providers. However this will require fail-over and load-balancing across providers which leads us to the next section.

7.1.4. Clustering

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

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

60 seconds is the default timeout.

7.1.5. Channel Fail-over and Load Balancing

Our HelloWorld Service can listen to more then 1 protocol. Here we have added 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>
...
```

Now our Service is simultaneously listening to two JMS queues. Now these queues can be provided by JMS providers on different physical boxes! So we now have a made a redundant JMS connection between two services. We can even mix protocols in this setup, so we can also add and ftp-listener to the mix.
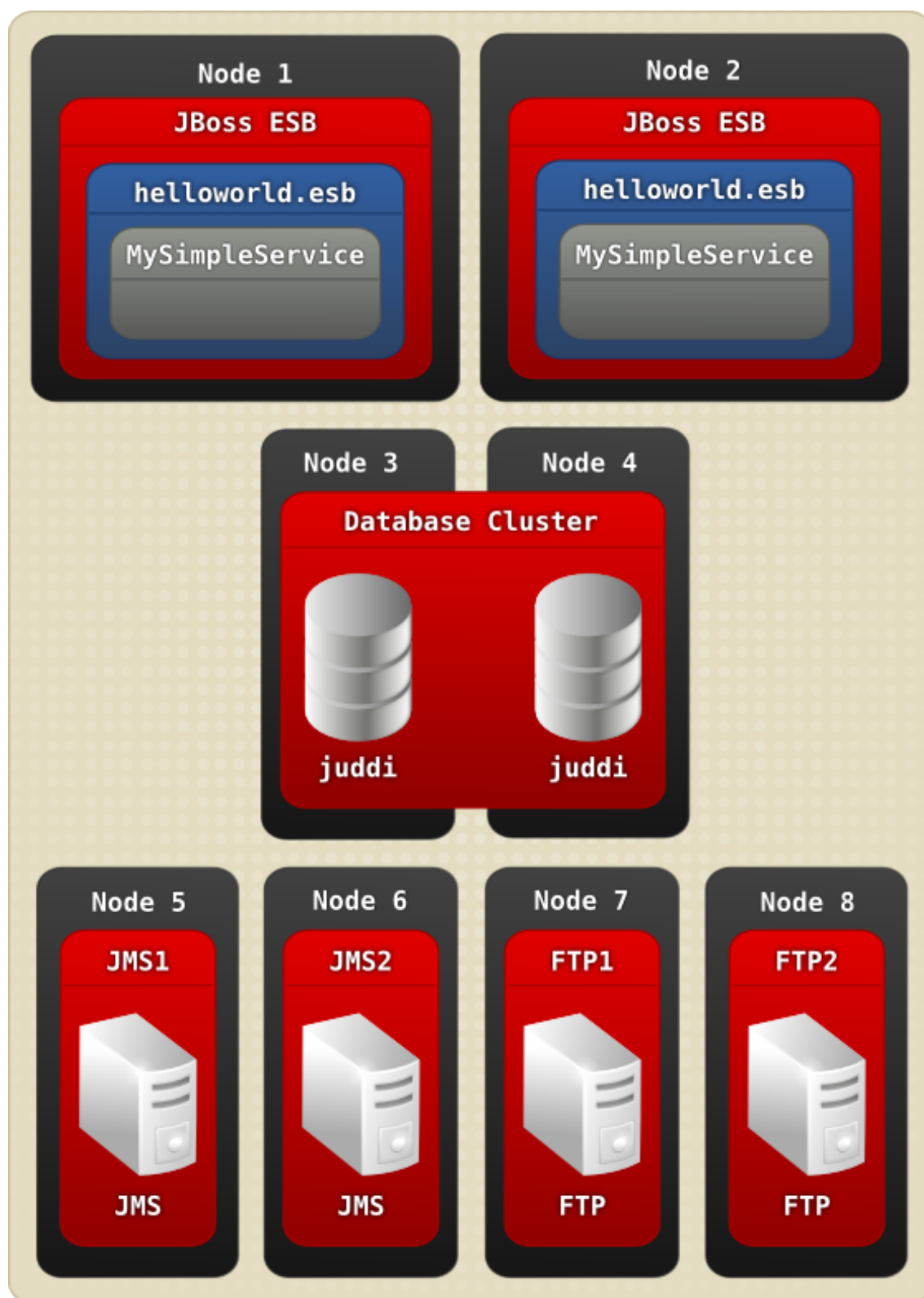
Figure 7.3. Adding two FTP servers to the mix.

...

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

When the ServiceInvoker tries to deliver a message to our Service it will get a choice of 8 EPRs now (4 EPRs from Node1 and 4 EPRs from Node2). How will it decide which one to use? For that you can configure a Policy. In the jbossesb-properties.xml you can set the 'org.jboss.soa.esb.loadbalancer.policy'. Right now three Policies are provided, or you can create your own.

- First Available. If a healthy ServiceBinding is found it will be used unless it dies, and it will move to the next EPR in the list. This Policy does not provide any load balancing between the two service instances.

- Round Robin. Typical Load Balance Policy where each EPR is hit in order of the list.

- Random Robin. Like the other Robin but then random.

The EPR list the Policy works with may get smaller over time as dead EPRs will be removed from the (cached) list. When the list is exhausted or the time-to-live of the list cache is exceeded, the ServiceInvoker will obtain a fresh list of EPRs from the Registry. The 'org.jboss.soa.esb.registry.cache.life' can be set in the jbossesb-properties file, and is defaulted to 60,000 milliseconds. What if none of the EPRs work at the moment? This is where we may use Message Redelivery Service.

7.1.6. Message Redelivery

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

- If you are trying to deliver the message synchronously it will send the message to the DeadLetterService, which by default will store to the DLQ MessageStore, and it will send a failure back to the caller. Processing will stop. Note that you can configure the DeadLetterService in the jbossesb.esb if for instance you want it to go to a JMS queue, or if you want to receive a notification.

- If you are trying to deliver the message asynchronously (recommended), it too will send the message to the DeadLetterService, but the message will get stored to the RDLVR

MessageStore. The Redeliver Service (jbossesb.esb) will retry sending the message until the maximum number of redelivery attempts is exceeded. In that case the message will get stored to the DLQ MessageStore and processing will stop.
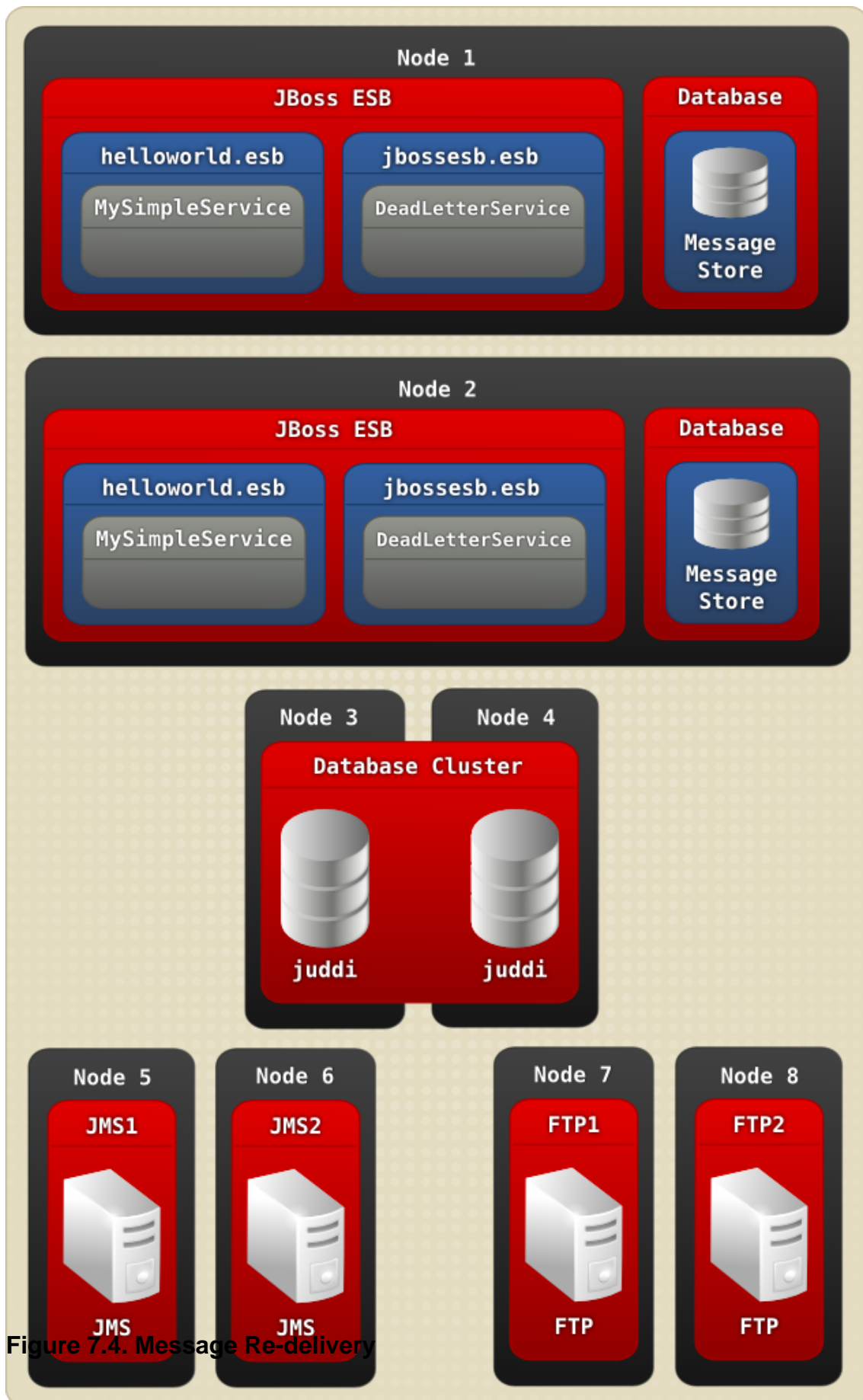
Figure 7.4. Message Re-delivery

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

7.2. Scheduling of Services

JBossESB 4.8 supports 2 types of providers:

1. Bus Providers, which supply messages to action processing pipelines via messaging protocols such as JMS and HTTP. This provider type is "triggered" by the underlying messaging provider.

2. Schedule Providers, which supply messages to action processing pipelines based on a schedule driven model i.e. where the underlying message delivery mechanism (e.g. the file system) offers no support for triggering the ESB when messages are available for processing, a scheduler periodically triggers the listener to check for new messages.

Scheduling is new to 4.2 of the ESB and not all of the listeners have been migrated over to this model yet.

JBossESB 4.8 offers a <schedule-listener> as well as 2 <schedule-provider> types - <simple-schedule> and <cron-schedule>. The <schedule-listener> is configured with a "composer" class, which is an implementation of the org.jboss.soa.esb.listeners.ScheduledEventMessageComposer interface..

7.2.1. Simple Schedule

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

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

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

execCount
    The number of times the schedule should be executed.

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

Example:

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

7.2.2. Cron Schedule

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

scheduleid

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

cronExpression

    CRON expression

startDate

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

Example:

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

7.2.3. Scheduled Listener

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

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

ScheduledEventListener

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

ScheduledEventMessageComposer

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

The attributes of this listener are:

1. name: The name of the listener instance

2. event-processor: The event processor class that is called on every schedule trigger. See above for implementation details.

3. One of:

    a. name: The name of the listener instance.

    b. scheduleidref: The scheduleid of the schedule to use for triggering this listener.

    c. schedule-frequency: Schedule frequency (in seconds). A convenient way of specifying a simple schedule directly on the listener.

7.2.4. Example Configurations

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

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

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

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

      <listeners>
```

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

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

</jbossesb>
```

7.2.5. Quartz Scheduler Property Configuration

The Scheduling functionality in JBossESB is built on top of the Quartz Scheduler (*http://www.quartz-scheduler.org/*). The default Quartz Scheduler instance configuration used by JBossESB is as follows:

```
org.quartz.scheduler.instanceName = DefaultQuartzScheduler
org.quartz.scheduler.rmi.export = false
org.quartz.scheduler.rmi.proxy = false
org.quartz.scheduler.wrapJobExecutionInUserTransaction = false

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 2
org.quartz.threadPool.threadPriority = 5
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true

org.quartz.jobStore.misfireThreshold = 60000

org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

Any of these Scheduler configurations can be overridden, or/and new ones can be added. You can do this by simply specifying the configuration directly on the <schedule-provider> configuration as a <property> element. For example, if you wish to increase the thread pool size to 5:

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

Fault-tolerance and Reliability

In this Chapter we shall look at the reliability characteristics of JBossESB. We shall examine what failure modes you should expect to be tolerated with this release and give advice on how to improve the fault tolerance of your applications. However, in order to proceed we need to define some important terms. If you wish to skip the following sections because you understand this topic already, you may go straight to the Reliability Guarantees section.

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

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

8.1. Failure classification

Given a (distributed) system, it would be useful if we were able to describe its behavior formally in a way that will help establish the correctness of the applications run on it. If this then imposes restrictions on the permissible behavior of the applications we will need to understand how these restrictions can be enforced and the implications in weakening or strengthening them. A useful method of building such a formal description with respect to fault-tolerance is to categorize the system components according to the types of faults they are assumed to exhibit.

Four possible classifications of failures are: omission, value, timing, and arbitrary. Associated with each component in the system will be a specification of its correct behavior for a given set of inputs. A non-faulty component will produce an output that is in accordance with this specification. The response from a faulty component need not be as specified, i.e., it can be anything. The response from a given component for a given input will be considered to be correct if not only the output value is correct but also that the output is produced on time, i.e., produced within a specified time limit.

The classifications are:

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

Value fault/failure

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

Timing fault/failure

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

Arbitrary fault/failure

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

An arbitrary fault causes any violation of a component's specified behavior. All other fault types preclude certain types of fault behavior, the omission fault type being the most restrictive. Thus the omission and arbitrary faults represent two ends of a fault classification spectrum, with the other fault types placed in between. The latter failure classifications thus subsume the characteristics of those classes before them, e.g., omission faults (failures) can be treated as a special case of value, and timing faults (failures). Such ordering can be represented as a hierarchy:
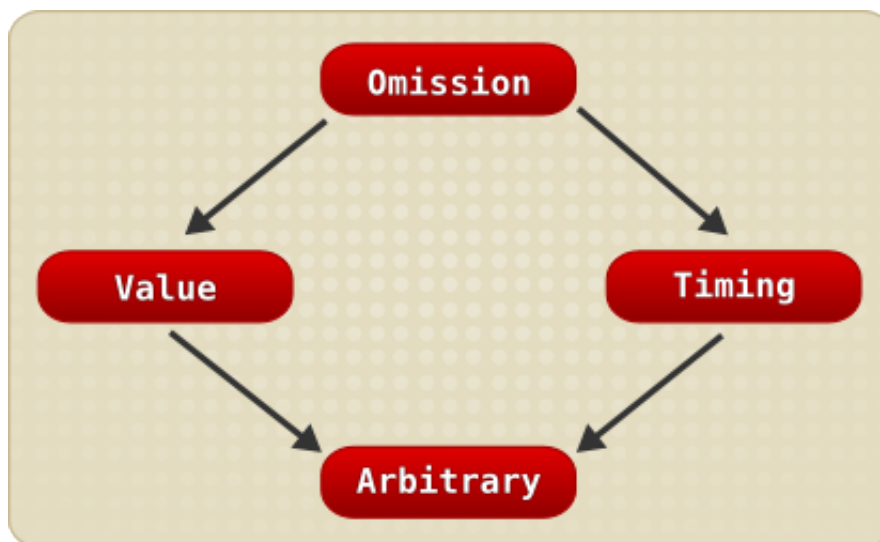
Figure 8.1. Fault classification hierarchy

8.1.1. JBossESB and the Fault Models

Within JBossESB there is nothing that will allow it to tolerate Byzantine/arbitrary failures. As you can probably imagine, these are extremely difficult failures to detect due to their nature.

Protocols do exist to allow systems to tolerate arbitrary failures, but they often require multi-phase coordination or digital signatures. Future releases of JBossESB may incorporate support for some of these approaches.

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

Tolerating Messages that are delayed is sometimes easier than tolerating those that arrive too early. However, from an application perspective, if an early Message is lost (e.g., by buffer overflow) it is not possible to distinguish it from one that is infinitely delayed. Therefore, if you construct your applications (consumers and services) to use a retry mechanism in the case of lost Messages, timing and omission failures should be tolerated, with the following exception: your consumer picks up an early response out of order and incorrectly processes it (which then becomes a value failure). Fortunately if you use RelatesTo and MessageID within the Message header, you can spot incorrect Message sequences without having to process the entire payload (which is obviously another option available to you).

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

> **ℹ Note**
>
> You should use caution when retransmitting Messages to services. JBossESB currently has no notion of retained results or detecting retransmissions within the service, so any duplicate Messages will be delivered to the service automatically. This may mean that your service receives the same Message multiple times (e.g., it was the initial service response that got lost rather than the initial request). As such, your service may attempt to perform the same work. If using retransmission (either explicitly or through the ServiceInvoker fail-over mechanisms), you will have to deal with multiple requests within your service to ensure it is idempotent.

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

8.1.2. Failure Detectors and Failure Suspectors

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

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

Given that guaranteed failure detection based upon timeouts is not possible, there has been much work on failure suspectors: a failure suspector works by realising that although guaranteed failure detection is impossible, enforcing a decision that a given entity may have failed on to other, active entities is possible. Therefore, if one entity assumes another has failed, a protocol is executed between the remaining entities to either agree that it will be assumed to have failed (in which case it is excluded from the system and no further work by it will be accepted) or that it has not failed: the fact that one entity thinks it has failed does not mean that all entities will reach the same decision. If the entity has not failed and is excluded then it must eventually execute another protocol to be recognised as being alive.

The advantage of the failure suspector is that all correctly functioning entities within the distributed environment will agree upon the liveness of another suspected failed entity. The disadvantage is that such failure suspection protocols are heavy-weight, typically requiring several rounds of agreement. In addition, since suspected failure is still based upon timeout values, it is possible for non-failed entities to be excluded, thus reducing (possibly critical) resource utilisation and availability.

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

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

8.2. Reliability Guarantees

As we have seen, there are a range of ways in which failures can happen within a distributed system. In this section we will translate those into concrete examples of how failures could affect JBossESB and applications deployed on it. In the section on Recommendations we shall cover ways in which you can configure JBossESB to better tolerate these faults, or how you should approach your application development.

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

It is never possible to guarantee 100% reliability and fault tolerance. The laws of physics (namely thermodynamics and the always increasing nature of entropy) mean that hardware degrades and human error is inevitable. All we can ever do is offer a probabilistic approach: with a high degree of probability, a system will tolerate failures and ensure data consistency/make forward progress. Furthermore, proving fault-tolerance techniques such as transactions or replication comes at a price: performance. This trade-off between performance and fault-tolerance is best achieved with application knowledge: any attempts at opaquely imposing a specific approach will inevitably lead to poorer performance in situations where it is imply not necessary. As such, you will find that many of the fault-tolerance techniques supported by JBossESB are disabled by default. You should enable them when it makes sense to do so.

8.2.1. Message Loss

We have previously discussed how message loss or delay may adversely affect applications. We have also shown some examples of how messages could be lost within JBossESB. In this section we shall discuss message loss in more detail.

Many distributed systems support reliable message delivery, either point-to-point (one consumer and one provider) or group based (many consumers and one provider). Typically the semantics imposed on reliability are that the message will be delivered or the sender will be able to know with certainty that it did not get to the receiver, even in the presence of failures. It is frequently the case that systems employing reliable messaging implementations distinguish between a message being delivered to the recipient and it being processed by the recipient: for instance, simply getting the message to a service does not mean much if a subsequent crash of the service occurs before it has time to work on the contents of the message.

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

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

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

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

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

8.2.2. Suspecting Endpoint Failures

We saw earlier how failure detection/suspicion is difficult to achieve. In fact until/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 (often referred to as split-brain syndrome).

8.2.3. Supported Crash Failure Modes

Unless using transactions or a reliable message delivery protocol such as JMS, JBossESB will only tolerate crash failures that are not catastrophic (i.e., the entire system does not fail) and result in the ability of JBossESB and/or the application to unambiguously reason about the liveness of the endpoints involved. If services crash or shutdown cleanly before receiving messages, then it is safe to use transports other than JMS.

8.2.4. Component Specifics

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

8.2.4.1. Gateways

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

8.2.4.2. ServiceInvoker

The ServiceInvoker will place undeliverable Messages to the Redelivery Queue if sent asynchronously. Synchronous Message delivery that fails will be indicated immediately to the sender. In order for the ServiceInvoker to function correctly the transport must indicate an unambiguous failure to deliver to the sender. A simultaneous failure of the sender and receiver may result in the Message being lost.

8.2.4.3. JMS Broker

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

8.2.4.4. Action Pipelining

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

8.3. Recommendations

Given the previous overview of failure models and the capabilities within JBossESB to tolerate them, we arrive at the following recommendations:

- 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 JMS implementation (clustered preferably).

- 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 (e.g., 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 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, e.g., 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 previously this is a best-effort protocol for improved robustness and does not use transactions or reliable message delivery. This means that certain failures may result in Messages being lost entirely (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.

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

Defining Service Configurations

9.1. Overview

JBossESB 4.8 configuration is based on the jbossesb-1.2.0 XSD (*http://anonsvn.jboss.org/repos/
labs/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.2.0.xsd*). This XSD is always the
definitive reference for the ESB configuration.

The model has 2 main sections:
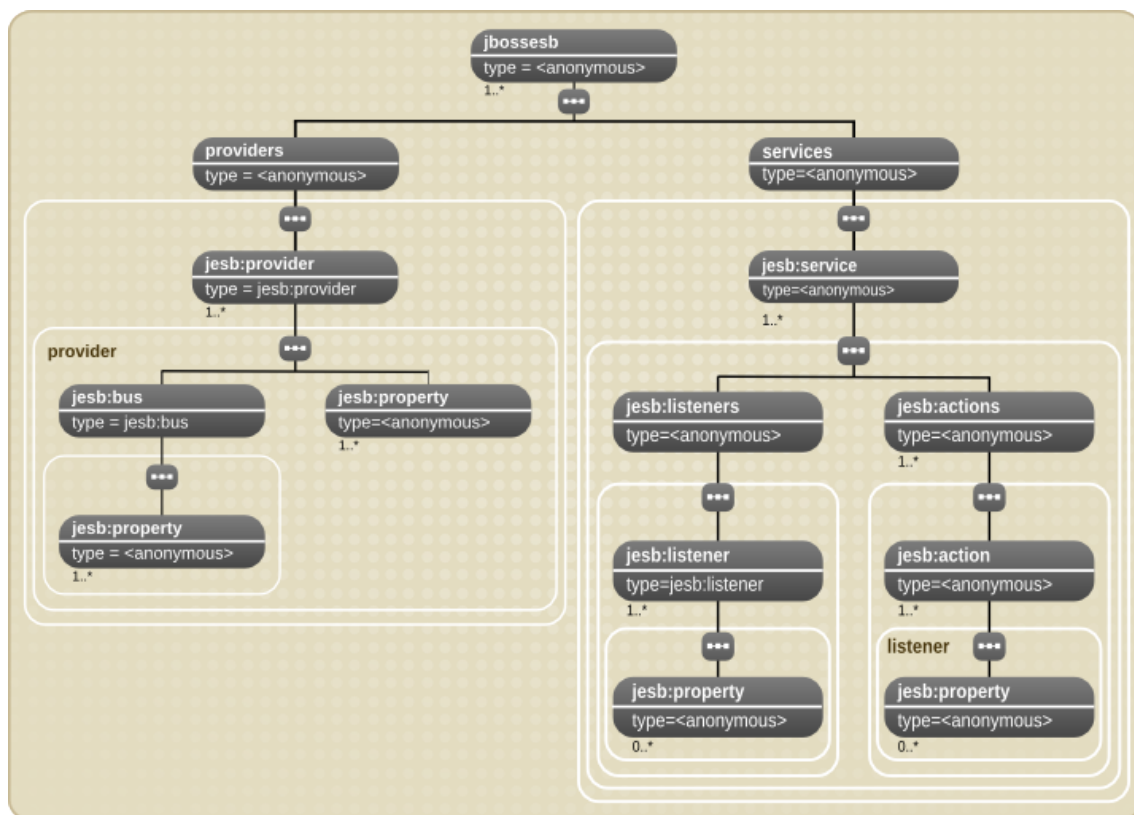
Figure 9.1. JBoss Enterprise Service Bus Configuration Model

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

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

By far the easiest way to create configurations based on this model is using JBoss Developer
Studio (*http://www.jboss.com/products/devstudio/*), but you can also to use an XSD aware XML

Editor such as the XML Editor in the Eclipse IDE. This provides the author with auto-completion features when editing the configuration. Right mouse-click on the file -> Open With -> XML Editor.

9.2. Providers
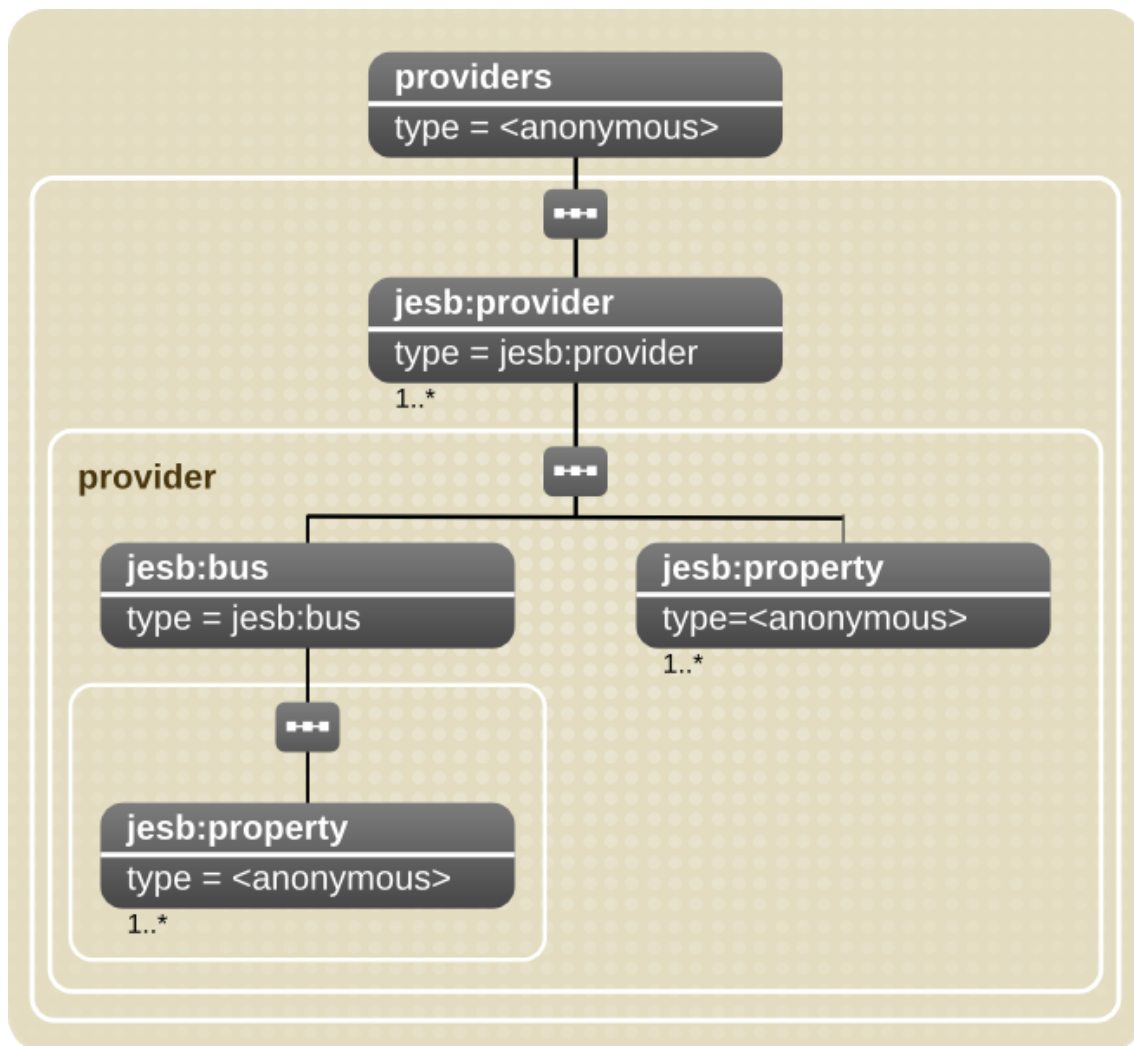
Figure 9.2. Providers Configuration Model

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

- Bus Providers: These specify provider details for "Event Driven" providers, that is, for listeners that are "pushed" messages. Examples of this provider type would be the <jms-provider>.

- Schedule Provider: Provider configurations for schedule-driven listeners (that is, listeners that "pull" messages.)

A Bus Provider (e.g. <jms-provider>) can contain multiple <bus> definitions. The <provider> can also be decorated with <property> instances relating to provider specific properties

that are common across all <bus> instances defined on that <provider> (e.g. for JMS - "connection-factory", "jndi-context-factory" etc). Likewise, each <bus> instance can be decorated with <property> instances specific to that <bus> instance (e.g. for JMS - "destination-type", "destination-name" etc).

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

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


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

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

9.3. Services
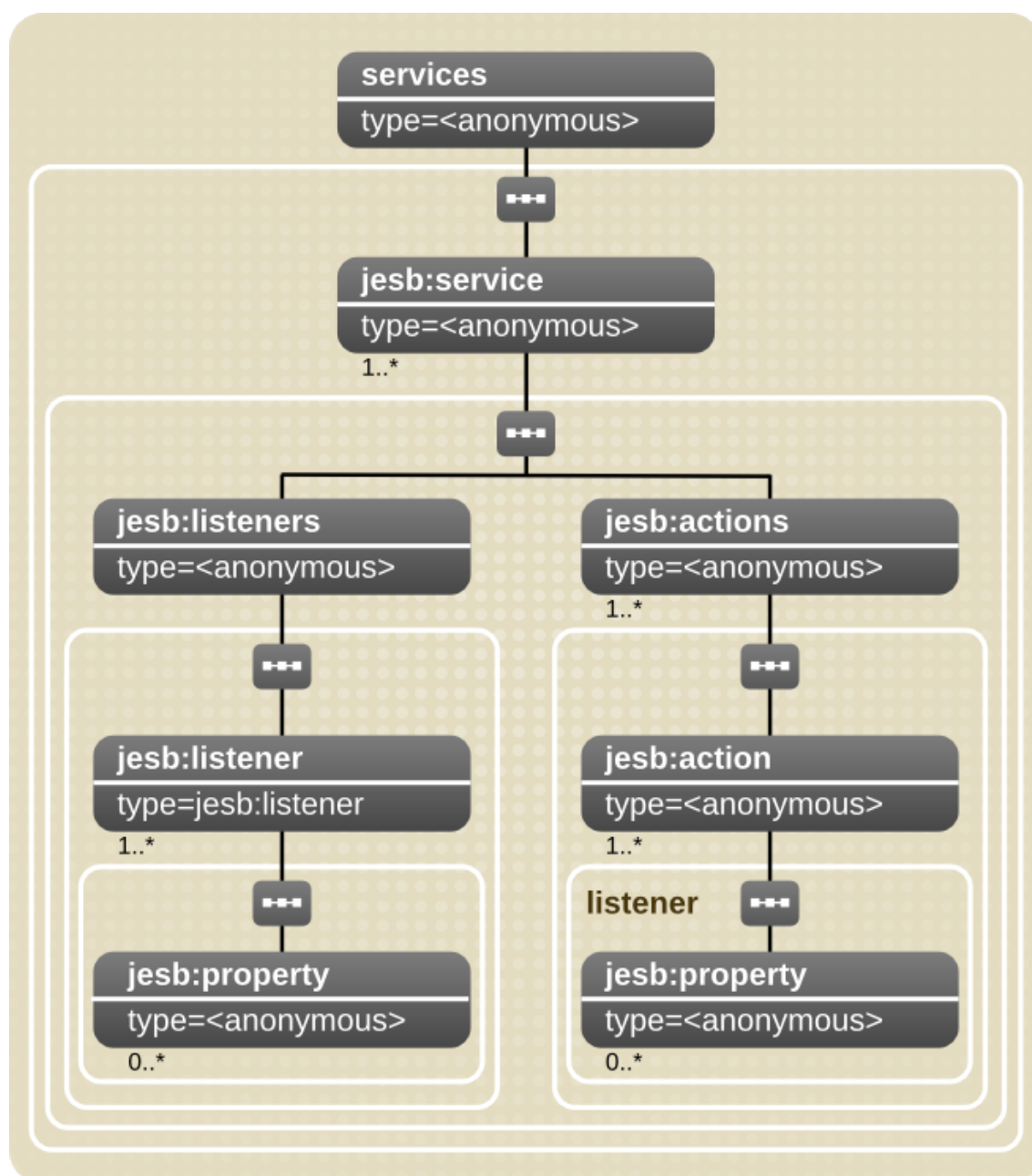
Figure 9.3. Services Configuration Model

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

Table 9.1. Service Attributes

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

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

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

Table 9.2. Listener Attributes

Name	Description	Type	Required
name	The name of the listener. This attribute is required primarily for logging purposes.	xsd:string	true
busrefid	Reference to the busid of the <bus> through which the listener instance receives messages.	xsd:string	true
maxThreads	The max number of concurrent message processing threads that the listener can have active.	xsd:int	True
is-gateway	Whether or not the listener instance is a "Gateway" or "Message Aware" Listener. See footnote. [a]	xsd:boolean	true

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

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

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

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

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

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

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

Table 9.3. Action Attributes

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

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

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

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

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

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

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

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

9.4. Transport Specific Type Implementations

The JBoss ESB configuration model defines transport specific specializations of the "base" types <provider>, <bus> and <listener> (JMS, SQL etc). This allows us to have stronger validation on the configuration, as well as making configuration easier for those that use an XSD aware XML Editor (e.g. the Eclipse XML Editor). These specializations explicitly define the configuration requirements for each of the transports supported by JBoss ESB out of the box. It is recommended to use these specialized types over the "base" types when creating JBoss ESB configurations,

the only alternative being where a new transport is being supported outside an official JBoss ESB release.

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

1. Define the provider configuration e.g. <jms-provider>.

2. Add the bus configurations to the new provider (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 release are:

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.

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

> **Note**
>
> In order to list and describe the attributes for each transport specific type, you can use the jbossesb-1.2.0 XSD (*http://anonsvn.labs.jboss.com/labs/jbossesb/ trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd*), which is fully annotated with descriptions of each of the attributes. Using an XSD aware XML Editor such as the Eclipse XML Editor makes working with these types far easier.

Table 9.4. JMS Message Filter Configuration

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

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 9.5. 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 e.g. 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.

9.5. FTP Provider Configuration

Table 9.6. FTP Provider Configuration

Property	Description	Required
hostname	Can be a combination of <host:port> of just <host> which will use port 21.	Yes

Property	Description	Required
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

9.6. FTP Listener Configuration

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

9.6.1. Read-only FTP Listener

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

The read-only implementation uses JBoss TreeCache to hold a list of the 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 9.7. Read-only FTP Listener Configuration

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

Example configuration:

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

</ftp-listener>
```

Example snippet from JBoss cache configuration:

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

Table 9.8. Configuration

Property	Description	Comments
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

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

9.7. UDP Gateway

Is a gateway implementation for receiving ESB unaware messages sent using the UDP protocol. The payload will be passed along to the action chain in the default ESB Message object location.

Actions can call esbMessage.getBody().get() to retrieve the byte array payload from within their actions.

Table 9.9. 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.

Example configuration:

```
<udp-listener
  name="udp-listener"
  host="localhost"
  port="9999"
  handlerClass="org.jboss.soa.esb.listeners.gateway.mina.DefaultMessageHandler"
  is-gateway="true"
<udp-listener/>
```

9.8. JBoss Remoting (JBR) Configuration

The JBoss Remoting Gateway hooks JBoss Remoting (*http://www.jboss.org/jbossremoting/*) into JBoss ESB as a transport option. It leverages support for HTTP(S) and Socket (+SSL) via JBR.

The basic configuration of the JBR Provider is as flows:

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

So the basic <jbr-provider> and <jbr-bus> configuration is very simple. The <jbr-bus> can then be referenced from a <service> configuration through the <jbr-listener>:

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

The <jbr-listener> is only supported as a gateway i.e. setting is-gateway to false will result in a Service deployment error.

The JBR Gateway supports a number of configuration properties that can be set on either the <jbr-provider>, <jbr-bus> or <jbr-listener> elements (as <property> elements).

Table 9.10. Configuration

Name	Description	Default
synchronous	Is the target Service to be invoked Synchronously.	True
serviceInvokerTimeout	Asynchronous invocation timeout.	20000
asyncResponse	Asynchronous response.	"<ack/>

Name	Description	Default
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 the JBR Gateway supports setting of JBR specific configuration properties. This can be done by prefixing the property name with "jbr-". Consult the JBoss Remoting docs (*http://www.jboss.org/jbossremoting/*)for the JBR specific settings relevant to the configured protocol. The following is an example of a configuration that specifies JBR specific settings for configuring a Keystore and Client authentication mode for HTTPS:

```xml
<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>
```

The JBR Gateway expects all response headers to be located in the Message.Properties as instances of org.jboss.soa.esb.message.ResponseHeader class. So if you require the JBR Gateway to set specific response headers, the ESB Message provided to the Gateway response decompose (e.g. after a synchronous invocation of the target service) must contains instances of the ResponseHeader class, set on the Message.Properties.

9.9. HTTP Gateway

As it's name suggests, this gateway allows you to expose Message-Unaware HTTP endpoints on JBoss ESB.

This gateway uses the JBoss ESB/App Server HTTP Container for exposing HTTP endpoints, so many of the configurations are managed at the container level e.g. bind/port address, SSL etc.

9.9.1. Basic Configuration

The easiest way to configure the <http-gateway> on a Service is as follows (no provider configuration 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 since it doesn't define a busrefid attribute. It uses the Service name to construct the HTTP endpoint address as follows:

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

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

9.9.2. URL Patterns

The <http-gateway> also supports a urlPattern as follows:

```
<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 would expose a HTTP endpoint for the service, capturing all HTTP requests under the following address:

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

9.9.3. Port Restrictions

The property allowedPorts can be used to confine certain services to one or a set of HTTP ports. This is specified by listing the comma separated list of ports as shown below:

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

This would expose a HTTP endpoint for the service, capturing all HTTP requests under the following ports only and all other port's request would receive HTTP Status code 404 – Not Found.

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

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

9.9.4. Request Handling

The <http-gateway> is typically able to decode a HTTP Request payload based on the request MIME type. It uses the "core:org.jboss.soa.esb.mime.text.types" configuration property from the jbossesb-properties.xml file to decide whether or not the payload is to be decoded (for the Service) as a String, or simply 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 (note wildcard support):

- 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".

9.9.5. Request Information

The HTTP Request obviously contains a lot of information (aside from a data payload) that may be required by the Service i.e. not just a request payload (e.g. in the case of POST). This information is stored, by the gateway, in a HttpRequest object instance on the Message. Actions can access it as follows:

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

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

Table 9.11. 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.
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.

Property	Description
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 user name is sent with each subsequent request depends on the client and type of authentication. Same as the value of the CGI variable REMOTE_USER.
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.

9.9.6. Response Handling

9.9.6.1. Asynchronous Response Handling

This 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, this gateway will synchronously invoke the service 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 (i.e. not a synchronous service invocation). Because it invokes the service asynchronously, it cannot return a service response as part of it's synchronous HTTP response. Therefore, you need to configure the gateway, telling it how to make the asynchronous response.

Asynchronous behavior is configured by adding an <asyncResponse> element to the <http-gateway>, as follows:

<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 9.12.

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

9.9.6.2. Synchronous Response Handling

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

9.9.6.3. Response Information

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.

Services (Actions) can create and set a HttpResponse instance on their response message as follows:

```
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 get mapped onto the outgoing HTTP gateway response:

Table 9.13.

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

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

9.9.6.4. Payload Encoding

The response payload content encoding can be set through the HttpResponse instance (see above).

9.9.6.5. Response Status

The HTTP response status code (*http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html*)is set through the HttpResponse instance (see above).

9.9.6.6. Response Timeout

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, you need to configure the "synchronousTimeout" property:

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

9.9.6.7. Exception Handling

Service exceptions (Action pipeline exceptions) can be mapped 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>. The following 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 classpath, so 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>
```

9.9.7. Security Handling

To configure security constraints, one must add configurations to the <http-provider> section of the ESB configuration i.e. this can not be done directly on the <http-gateway> configuration.

So the process of securing a <http-gateway> is as follows:

1. Specify a <http-bus> in the <http-provider> section of the ESB configuration.

2. Specify the constraints on the <http-bus>.

3. Refence the <http-bus> from the <http-gateway> using the "busrefid" attribute.

> **Note**
>
> See the "http-gateway" Quickstart for a full config example.

9.9.8. Protected Methods and Allowed User Roles

Logins can be enforced for an endpoint using the <protected-methods> and <allowed-roles> sections of a <http-bus> configuration as follows:

```
<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" login is required for DELETE requests made on the "secureSalesDeletes" bus. The following login matrix tries to illustrate which configurations will enforce a login, and when.

Table 9.14.

Methods Specified	Roles Specified	Log-in Required
No	No	No

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

9.9.9. Authentication Method and Security Domain

The authentication method and security domain can be configured in the <war-security> configuration inside the <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" and "DIGEST".

> **Note**
>
> See the JBoss Application Server documentation for details on configuring application Security Domains.

9.9.10. Transport Guarantee

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

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

Allowed values for transportGuarantee are "CONFIDENTIAL", "INTEGRAL" and "NONE".

9.10. Camel Gateway

As it's name suggests, this gateway allows you to expose Message-Unaware Camel endpoints on JBossESB. This gateway leverages Apache Camel's input capabilities, translates the Camel Message to an ESB Message, and invokes the associated ESB Service.

The most apparent difference betweeen the CamelGateway and the other Gateways provided within JBossESB is that the CamelGateway is not specific to any one type of transport. This is because we want to leverage all the different transports that Camel supports, so the role of the CamelGateway is really all about integration. 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. JBossESB only contains the camel-core.jar. You will have to add any other dependencies (including other camel-* jars or third party jars) you require.

Within the <providers> section of JBoss ESB, a new <camel-provider> section is available if you declare the use of the updated schema (jbossesb-1.3.0.xsd) in your jboss-esb.xml file.

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

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

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

You can also define <from uri=""/> elements under the <camel-gateway> element, without requiring 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:

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

or at 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 JBossESB, 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 lifecycle.

> **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 e.g. 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>" e.g. <from uri="esbschedule:5000:http://www.jboss.org" /> would poll jboss.org every 5 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*
>
> There is also a quickstart available in the distribution: .../samples/quickstarts/camel_helloworld/
>
> For more information on Apache Camel, please visit this web page: *http://camel.apache.org/*

9.11. Transitioning from the Old Configuration Model

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

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

Developers that were used to using the old model now need to keep the following in mind:

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

2. The only location where free-form markup is supported in the new configuration is on the <property> element/type. This type is allowed on <provider>, <bus> and <listener> types (and sub-types). However, the only location in which <property> based free form markup is mapped into the ConfigTree configurations is where the <property> exists on an <action>. In this case,

the <property> content is mapped into the target ConfigTree <action>. Note however, if you have 1+ <property> elements with free form child content on an <action>, all this content will be concatenated together on the target ConfigTree <action>.

3. When developing new Listener/Action components, you must ensure that the ConfigTree based configuration these components depend on can be mapped from the new XSD based configurations. An example of this is how in the ConfigTree configuration model, you could decide to supply the configuration to a listener component via attributes on the listener node, or you could decide to do it based on child nodes within the listener configuration – 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 i.e. the child content in the above example would not be mapped from the XSD configuration to the ConfigTree style configuration. In fact, the XSD configuration simply would not accept the arbitrary content, unless it was in a <property> and even in that case (on a <listener>), it would simply be ignored by the mapping code.

9.12. Configuration

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

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

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

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

Within this version of the JBossESB, there is only a single implementation, the org.jboss.soa.esb.parameters.ParamFileRepository, which expects to be able to load the parameters from a file. The implementation to use may be overridden using the org.jboss.soa.esb.paramsRepository.class property.

Note

We recommend that you construct your ESB configuration file using Eclipse or some other XML editor. The JBossESB configuration information is supported by an annotated XSD which should help if using a basic editor.

Data Decoding – Mime Decoders

All JBossESB Gateways support the concept of a "Message Composer", which is a class that implements the MessageComposer interface and is responsible for constructing an ESB Message instance and sending it to the associated ESB Service.

A Mime Decoder is a class that implements the MimeDecoder interface. It can be used by a MessageComposer implementation for "decoding" a binary array to a specific Java Object type, based on the "mime type" of the binary encoded data.

Examples of Gateways using this MimeDecoder mechanism are the File and FTP Gateway Listeners. These gateways can be configured with "mimeType" or "mimeDecoder" properties, which trigger installation of the appropriate MimeDecoder implementation for the specified Mime Type.

10.1. Implementing a MimeDecoder

The process is very easy:

1. Create a class that implements the org.jboss.soa.esb.listeners.message.mime.MimeDecoder interface.

2. Annotate the newly created class with the @MimeType annotation, 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. This file needs to be present on the classpath at runtime. If your module doesn't have this file, simply add the file to your module source/resources and it will be located at runtime.

4. (Optional) If the MimeDecoder implementation needs access to the listener configuration (for additional configuration info), simply have the class implement the org.jboss.soa.esb.Configurable interface.

Example (real):

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

   private Charset encodingCharset;

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

```java
        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);
        }
    }
}
```

10.2. Using MimeDecoders in a Gateway Implementations

The easiest way of a MessageComposer using the installed Mime Decoders is via the Listener instances configuration (ConfigTree) and the MimeDecoder.Factory class factory method:

```java
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):

```xml
<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 simply calls the decode method on its mimeDecoder instance:

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

And then sets the "decodedPayload" Object instance on the ESB Message instance being composed.

10.3. Out-of-the-Box MimeDecoder Implementations

JBossESB 4.8 has the following MimeDecoder implementations. If you need a MimeDecoder that's not yet implemented, just follow the instructions provided earlier in this section on "Implementing a MimeDecoder".

10.3.1. text/plain

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

Table 10.1. Properties

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

Web Services Support

11.1. JBossWS

JBossESB supports both JBossWS-Native and JBossWS-CXF stacks. JBossESB has a number of Webservice based components for exposing and invoking Webservice endpoints (i.e. SOAP onto the bus and SOAP off the bus):

1. SOAPProcessor: The SOAPProcessor action allows you to expose JBossWS 2.x and higher Webservice Endpoints through endpoints (listeners) running on the ESB ("SOAP onto the bus"). This allows you to use JBossESB to expose Webservice Endpoints (wrapper Webservices) for services that don't expose a Webservice Interface. JBossWS Webservice Endpoints exposed via this JBossESB action are "ESB Message Aware" and can be used to invoke Webservice Endpoints over any transport channel supported by the ESB.

> **Caution**
>
> JBossWS-CXF Endpoints are not "ESB Message Aware".

2. SOAPClient: The SOAPClient action allows you to make invocations on Webservice endpoints ("SOAP off the bus").

> **Note**
>
> For more details on these components and how to configure and use them, see the ServicesGuide.
>
> You can also find more information within the wiki pages shipped with the JBossESB documentation.

Out-of-the-box Actions

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

12.1. Transformers and Converters

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

Unless stated otherwise, all of these Actions use the `MessagePayloadProxy` for getting and setting the message payload.

12.1.1. ByteArrayToString

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

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

Table 12.1. ByteArrayToString Properties

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

Example 12.1. Sample Configuration

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

12.1.2. 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 12.2. Sample Configuration

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

12.1.3. 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 12.2. 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 12.3. 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>
```

12.1.4. 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 12.3. 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 i.e., if the Object does not support a getter method for the property. Default value is `false`.	No

Example 12.4. Sample Configuration

```xml
<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>
```

12.1.5. 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. See *http://xstream.codehaus.org/*.

Table 12.4. 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
namespaces	Specify namespaces that should be added to the XML generated by XStream. Each namespace-uri is associated	No

Property	Description	Required
	with a local-part which is the element on which this namespace should appear.	
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 12.5. Sample Configuration

```xml
<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>
```

12.1.6. 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. See *http://xstream.codehaus.org/*.

Table 12.5. 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. Refer to *http://xstream.codehaus.org/converters.html* for information about converters.	No

Example 12.6. Sample Configuration

```xml
<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-collection class="className" fieldName="fieldName" fieldType="fieldType"
            itemType="itemType"/>
        ...
    </property>
    <property name="converters">
        <converter class="className" fieldName="fieldName" fieldType="fieldType"/>
        ...
    </property>
</action>
```

12.1.7. XsltAction

This performs transformation on entire documents. If you need per-fragment transformations,

Table 12.6. `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
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:	No

Property	Description	Required
	- 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 that the types are compatible.	
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 will 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. Refer to *Section 12.1.7.1, "XsltAction Validation"*. Default value is false .	No

Property	Description	Required
schemaFile	The input schema file (XSD) to use, located on the classpath. Refer to *Section 12.1.7.1, "XsltAction Validation"*.	No
schemaLanguage	The input schema language to use. Refer to *Section 12.1.7.1, "XsltAction Validation"*.	No

12.1.7.1. XsltAction Validation

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"/>
   ```

   or

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

3. W3C XML Schema or RELAX NG

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

   or

   ```
   <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"/>
```

or

```
<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 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

12.1.8. 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`

Message Transformation on JBossESB is supported by the SmooksTransformer component. This is an ESB Action component that allows the Smooks Data Transformation/Processing Framework to be plugged into an ESB Action Processing Pipeline.

A wide range of source (XML, CSV, EDI etc.) and target (XML, Java, CSV, EDI etc.) data formats are supported by the SmooksTransformer component. A wide range of Transformation Technologies are also supported, all within a single framework.

Refer to the Smooks website at *http://www.smooks.org* for more information.

Table 12.7. SmooksTransformer Resource Configuration

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

Table 12.8. 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
to	Message Exchange Participant name. Message Consumer.	No
to	Message Exchange Participant name. Message Consumer.	No

Property	Description	Required
to-type	Message type/format consumed by the "to" message exchange participant.	No

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

Example 12.7. 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 12.8. 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 12.9. 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. Refer to *http://community.jboss.org/wiki/MessageTransformation* for more details.

12.1.9. SmooksAction

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

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

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

> **Important**
>
> Users should 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 (in JBoss ESB 4.5.GA+) is to use the extended Smooks configuration namespaces for all out of the box functionality. For example, instead of defining Java binding configurations by defining org.milyn.javabean.BeanPopulator <resource-config> configurations, you should use the http://www.milyn.org/xsd/smooks/javabean-1.2.xsd configuration namespace (i.e. the <jb:bean> config etc).
>
> If you've implemented new Smooks Visitor functionality, the easiest way to avoid this issue is to define an extended configuration namespace for this new resource type. This also has the advantage of making this new resource easier to configure, because you're able to leverage the schema support built into your IDE.
>
> See the Smooks v1.1+ documentation for examples of these extended namespace configurations.

12.1.9.1. SmooksAction Configuration

The following illustrates the basic SmooksAction configuration:

Example 12.10. SmooksAction

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

The optional configuration properties are:

Table 12.9. 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
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. See *Section 12.1.9.5, "Specifying the Result type"*.	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.	

For more information on for Smooks Execution Report refer to *http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Checking_the_Smooks_Execution_Process*.

12.1.9.2. Message Input Payload

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

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

12.1.9.3. XML, EDI, CSV etc Input Payloads

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

1. String,

2. InputStream,

3. Reader, or

4. byte array

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

12.1.9.4. Java Input Payload

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

12.1.9.5. Specifying the Result type

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

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

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

Example 12.11. Specifying the Result Type

```
<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>
```

12.1.10. 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 12.10. 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 then that will be used instead. Otherwise, a default may be provided at instantiation time.	Yes
message-store-class	The implementation of the `MessageStore`.	Yes
terminal	If 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

Example 12.12. PersistAction

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

12.2. Business Process Management

12.2.1. jBPM - BpmProcessor

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

JBossESB can make calls into jBPM using the BpmProcessor action. Please also read the "jBPM Integration" chapter in the ServicesGuide to learn how to call JBossESB from jBPM. The BpmProcessor action uses the jBPM command API to make calls into jBPM.

The following jBPM commands have been implemented:

- NewProcessInstanceCommand

- StartProcessCommand

- CancelProcessInstanceCommand

- GetProcessInstanceVariablesCommand

Table 12.11. 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 CancelProcessInstanceCommands.	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 + process definition must be unique if	No

Property	Description	Required
	a business key is supplied. The key value can hold an MVEL expression to extract the desired value from the EsbMessage. For example if you have a named parameter called "businessKey" in the body of your message you would use "body.businessKey". Note that this property is used for the New- and StartProcessInstanceCommands only	
transition-name	Optional property. This property only applies to the StartProcessInstance- and Signal Commands, and is of use only if there are more then one transition out of the current node. If this property is not specified the default transition out of the node is taken. The default transition is the first transition in the list of transition defined for that node in the jBPM processdefinition.xml.	No
esbToBpmVars	Optional property for the New- and StartProcessInstanceCommands and the SignalCommand. This property defines a list of variables that need to be extracted from the EsbMessage and set into jBPM context for the particular process instance. The list consists of mapping elements. Each mapping element can have the following attributes: • `esb`: required attribute which can contain an MVEL expression to extract a value anywhere from the EsbMessage. • `bpm`: optional attribute containing the name which be used on the jBPM side. If omitted the esb name is used. • `default`: optional attribute which can hold a default value if the esb MVEL expression does not find a value set in the EsbMessage.	No

12.3. Scripting

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

12.3.1. GroovyActionProcessor

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

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

Table 12.12. 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 12.13. 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).

Example 12.13. Sample Configuration

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

12.3.2. ScriptingAction

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

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

1. JBoss ESB 4.8 contains BSF 2.3.0, which has less language support than BSF 2.4.0 (for example: no Groovy, and non-functioning Rhino). A future version will contain BSF 2.4.0, which will support Groovy and Rhino.

2. BSF 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.

3. When including BeanShell scripts in your application, it is advised to use a .beanshell extension instead of .bsh, otherwise the JBoss BSHDeployer might pick it up.

Table 12.14. ScriptingAction Properties

Property	Description	Required
script	Path (on classpath) to script.	

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

Table 12.15. 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)

Example 12.14. ScriptingAction

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

12.4. Services

Actions defined within the ESB Services.

12.4.1. EJBProcessor

Input Type	EJB method name and parameters
Output Type	EJB specific object
Class	`org.jboss.soa.esb.actions.EJBProcessor`

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

Table 12.16. EJBProcessor Properties

Property	Description	Required
ejb3	if this is a call 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.	

Property	Description	Required
method	EJB method name to call.	
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 12.15. 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 12.16. 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>
```

12.5. Routing

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

12.5.1. Aggregator

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

Message aggregation action. 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 12.17. Aggregator Properties

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

Example 12.17. Sample Configuration

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

12.5.2. EchoRouter

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

12.5.3. HttpRouter

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

This action allows invocation of external (ESB unaware) Http endpoints from an ESB action pipeline. This action uses Apache Commons HttpClient under the covers.

Table 12.18. 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. Refer to *Section 12.9, "HttpClient Configuration"* for more information about the factory properties.	No
method	Currently only supports GET and POST.	Yes
responseType	Specifies in what form the response should be sent back. Either STRING or BYTES. Default value is STRING.	No
headers	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 ESB via the http-gateway or within the properties of the current ESB message.	No

Example 12.18. Sample Configuration

```xml
<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>
```

12.5.4. JMSRouter

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

Routes the incoming message to JMS.

Table 12.19. 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
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

12.5.5. EmailRouter

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

Routes the incoming message to a configured email account.

Table 12.20. EmailRouter Properties

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 file that will be added as attachments to the email sent.	

Example 12.19. Sample Configuration

```
<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>
```

12.5.6. 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 12.21. ContentBasedRouter Properties

Property	Description	Required
cbrAlias	Content Based Routing Provider alias. Supported values are "Drools" (default), "Xpath" "Regex".	
ruleSet	Externally defined rule file. Will be a Drools DSL file if the Drools rule provider is in use. Will be 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".	
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).	

Property	Description	Required
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 e.g. for the XPath ruleprovider. The <namespace> configurations have the following format:	
	<namespace **prefix**="ord" **uri**="http://acme.com/order" />	

Table 12.22. ContentBasedRouter "process" methods

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

Section 12.5.1, "Aggregator".

Example 12.20. 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 12.21. 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 being that the expressions are regex expressions (instead of XPath expressions).

Example 12.22. Sample Configuration Drools

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

See the "What is Content-Based Routing?" chapter in the *Services Guide* for more information on content-based routing.

12.5.7. 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 12.23. StaticRouter Properties

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

Table 12.24. StaticRouter Process Methods

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

See *Section 12.5.1, "Aggregator"*.

Example 12.23. Sample Configuration - StaticRouter

```xml
<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>
```

12.5.8. 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 12.25. SyncServiceInvoker Properties

Property	Description	Required
service-category	Service Category.	Yes
service-name	Service Name.	Yes
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, then you should leave this parameter set to true and use the normal "faultTo" mechanism by allowing the pipeline to fail. (default = "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" confguration. Default is 30000.	No

Example 12.24. Sample Configuration - SyncServiceInvoker

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

12.5.9. StaticWireTap

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

The StaticWiretap action differs from the StaticRouter in that the StaticWiretap "listens in" on the action chain and allows actions below it to be executed, 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 12.26. StaticWireTap Properties

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

Property	Description	Required
	service-name="ExpressShippingService"/>	

Table 12.27. StaticWireTap Process Methods

method	Description
process	Do not append aggregation data to message.

See *Section 12.5.1, "Aggregator"*.

Example 12.25. Sample Configuration - StaticWireTap

```
<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>
```

12.5.10. Email WireTap

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

Table 12.28. Email 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.

Property	Description
messageAttachmentName	The 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 file that will be added as attachments to the email sent.

Example 12.26. Sample Configuration - Email Wiretap

```
<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>
```

12.6. Notifier

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

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

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

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

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

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

Example 12.27. Sample Configuration - Notifier

```
<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>
```

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

12.6.1. NotifyConsole

Class	NotifyConsole

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

Example 12.28. NotifyConsole

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

12.6.2. 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

Example 12.29. NotifyFiles

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

12.6.3. 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

Example 12.30. NotifySqlTable

```
<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>
```

12.6.4. 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

Example 12.31. NotifyQueues

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

12.6.5. 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

Example 12.32. NotifyTopics

```
<target class="NotifyTopics" >
<messageProp name="aNewProperty" value="theValue"/>
<queue jndiName="queue/quickstarts_notifications_topic" />
</target>
```

12.6.6. NotifyEmail

Class	`NotifyEmail`
Purpose	Performs a notification by sending an email 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

Example 12.33. NotifyEmail

```
<target class="NotifyEmail" from="person@somewhere.com"
sendTo="person@elsewhere.com"
subject="theSubject">
<attachment>attachThisFile.txt</attachment>
</target>
```

12.6.7. 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

Example 12.34. NotifyFTP

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

12.6.8. 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.

Example 12.35. NotifyFTP

```
<target class="NotifyFTPList">
  <ftp URL="ftp://username:password@localhost/outputdir"
    filename="{org.jboss.soa.esb.gateway.file}">
    listFiles="true"
    deletelistFile="true"
</target>
```

12.6.9. 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 i.e. 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.

Example 12.36. NotifyFTP

```
<target class="NotifyTcp" >
 <destination URI="tcp://myhost1.net:8899" />
 <destination URI="tcp://myhost2.net:9988" />
</target>
```

12.7. Webservices/SOAP

12.7.1. SOAPProcessor

JBoss Webservices SOAP Processor

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

> **ℹ Note**
>
> SOAPProcessor supports both JBossWS-Native and JBossWS-CXF stacks.

12.7.1.1. SOAPProcessor Action Configuration

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

```
<action name="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 situations such as this. It has no effect for non-HTTP protocols. Default it true.

12.7.1.2. Dependencies

1. JBoss Application Server 4.2.3.GA.

2. The soap.esb Service. This is available in the lib folder of the distribution.

12.7.1.3. "ESB Message Aware" Webservice Endpoints

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

> **Important**
>
> This feature has been deprecated and may no longer be available in future releases. Users are not encouraged to rely on this feature.

12.7.1.4. Webservice Endpoint Deployment

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

12.7.1.4.1. Endpoint Publishing

See the "Contract Publishing" section of the Administration Guide.

12.7.2. SOAPClient

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

Example configuration:

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

Table 12.29. 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 superceded by operationName.
EndPointName	The EndPoint invoked. Webservices can have multiple endpoint. If it's not specified the first specified in wsdl will be used.
SmooksRequestMapper	Specifies a smooks config file to define the java-to-java mapping defined for the request.
SmooksResponseMapper	Specifies a smooks config file to define the java-to-java mapping defined for the response
serviceName	A symbolic service name used by wise to cache object generation and/or use already generated object. If it isn't provided wise uses the servlet name of wsdl.
username	Username used if the webservice is protected by BASIC Authentication HTTP.
password	Password used if the webservice is protected by BASIC Authentication HTTP.
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 support JAXWS Soap Handler, both custom or a predefined one based on smooks. Transformation of the SOAP request (before sending) is supported by configuring the SOAPClient action with a Smooks transformation configuration property.

Property	Description
custom-handlers	It's also possible to provide a set of custom standard JAXWS Soap Handler. The parameter accept a list of classes implementing SoapHandler interface. Classes have to provide full qualified name and be separated by semi-columns.
LoggingMessages	It's useful for debug purpose to view soap Message sent and response received. Wise achieve this goal using a JAX-WS handler printing all messages exchanged on System.out. Boolean value.

12.7.2.1. SOAP Operation Parameters

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

1. With a set of Objects of any type. In this case a Smooks config has to be specified in action attribute SmooksRequestMapper and Smooks is used to make the java-to-java conversion

2. With a set of String based key-value pairs(<String, Object>), where the key is the name of the SOAP parameter as specified in wsdls (or in generated class) to be populated with the key's value. SOAP Response Message Consumption

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

- 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 one of 2 ways:

1. With a set of Objects of any type. In this case a smooks config have to be specified in action attribute SmooksResponseMapper and smooks is used to make the java-to-java conversion

2. With a set of String based key-value pairs(<String, Object>), where the key is the name of the SOAP answer as specified in wsdls (or in generated class) to be populated with the key's value. JAX-WS Handler for the SOAP Request/Response

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

- 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'.

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

12.7.2.1.1. JAXB Annotation Introductions

The native JBossWS SOAP stack uses JAXB to bind to and from SOAP. This means that an unannotated typeset cannot be used to build a JBossWS endpoint. To overcome this we provide a JBossESB and JBossWS feature called "JAXB Annotation Introductions" which basically means you can define an XML configuration to "Introduce" the JAXB Annotations. For details on how to enable this feature in JBossWS 2.0.0, see the *Appendix A, Appendix A: Writing JAXB Annotation Introduction Configurations* .

This XML configuration must be packaged in a file called "jaxb-intros.xml" in the "META-INF" directory of the endpoint deployment.

For details on how to write a JAXB Annotation Introductions configuration, see *Appendix A, Appendix A: Writing JAXB Annotation Introduction Configurations* .

12.7.2.2. Quickstarts

A number of quickstarts demonstrating how to use this action are available in the JBossESB distribution (samples/quickstarts). See the "webservice_bpel" quickstart.

12.7.3. SOAPClient

SOAP Client action processor.

Uses the soapUI Client Service to construct and populate a message for the target service. This action then routes that message to that service. See *http://www.soapui.org/*.

12.7.3.1. Endpoint Operation Specification

Specifying the endpoint operation is a straightforward task. Simply specify the "wsdl" and "operation" properties on the SOAPClient action as follows:

```
<action name="soapui-client-action" class="org.jboss.soa.esb.actions.soap.SOAPClient">
  <property name="wsdl" value="http://localhost:18080/acme/services/RetailerCallback?wsdl"/>
   <property name="operation" value="SendSalesOrderNotification"/>
```

```
</action>
```

12.7.3.2. SOAP Request Message Construction

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

1. As a Map instance set on the default body location (Message.getBody().add(Map))

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

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

1. With a set of Objects that are accessed (for SOAP message parameters) using the OGNL framework. More on the use of OGNL below.

2. With a set of String based key-value pairs(<String, Object>), where the key is an OGNL expression identifying the SOAP parameter to be populated with the key's value. More on the use of OGNL below.

As stated above, OGNL is the mechanism we use for selecting the SOAP parameter values to be injected into the SOAP message from the supplied parameter Map. The OGNL expression for a specific parameter within the SOAP message depends on the position of that parameter within the SOAP body. 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 (Option 2 above). If this doesn't yield a value, this parameter location within the SOAP message will remain blank.

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

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

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:cus="http://schemas.active-endpoints.com/sample/customerorder/2006/04/CustomerOrder.xsd"
          xmlns:stan="http://schemas.active-endpoints.com/sample/standardtypes/2006/04/StandardTypes.xsd">

  <soapenv:Header/>
  <soapenv:Body>
    <cus:customerOrder>
      <cus:items>
        <cus:item>
          <cus:partNumber>FLT16100</cus:partNumber>
          <cus:description>Flat 16 feet 100 count</cus:description>
          <cus:quantity>50</cus:quantity>
          <cus:price>490.00</cus:price>
          <cus:extensionAmount>24500.00</cus:extensionAmount>
        </cus:item>
        <cus:item>
          <cus:partNumber>RND08065</cus:partNumber>
          <cus:description>Round 8 feet 65 count</cus:description>
          <cus:quantity>9</cus:quantity>
          <cus:price>178.00</cus:price>
          <cus:extensionAmount>7852.00</cus:extensionAmount>
        </cus:item>
      </cus:items>
    </cus:customerOrder>
  </soapenv:Body>

</soapenv:Envelope>
```

The above order message contains a collection of order "items". Each entry in the collection is represented by an "item" element. The OGNL expressions for the order item "partNumber" is

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

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

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

12.7.3.3. SOAP Response Message Consumption

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

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

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

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

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

Using XStream as a mechanism for populating an Object Graph (Option 1 above) is straightforward and works well, as long as the XML and Java object models are in line with each other.

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

```
<action name="soapui-client-action" class="org.jboss.soa.esb.actions.soap.SOAPClient">
```

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

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

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

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

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

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

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

12.7.3.4. HttpClient Configuration

The SOAPClient uses Apache Commons HttpClient to execute SOAP requests. It uses the HttpClientFactory to create and configure the HttpClient instance. Specifying the HttpClientFactory configuration on the SOAPClient is very easy. Just add an additional property to the "wsdl" property as follows:

```
<property name="wsdl" value="https://localhost:18443/active-bpel/services/RetailerCallback?wsdl">
   <http-client-property name="file" value="/localhost-https-18443.properties" >

   </http-client-property>
</property>
```

The "file" property value will be evaluated as a filesystem, classpath or URI based resource (in that order). This resource contains the HttpClient configuration in the standard java properties format.

These properties may also be set directly in the action configuration like so:

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

Refer to *Section 12.9, "HttpClient Configuration"* for more information about the HtptClient configuration options.

12.7.4. SOAPProxy

A SOAPProxy focuses on the consumption of an external WS endpoint (e.g. hosted on .NET, another external Java-based AS, LAMP) and re-publication of a WS endpoint via the ESB. The ESB sits between the ultimate consumer/client (e.g. .NET WinForm application) and the ultimate producer (e.g. RoR-hosted WS). The purpose of this intermediary is to provide an abstraction layer that solves the following problems:

- Provides for more loose coupling between the client and service; they are both completely unaware of each other.

- 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 endpoint instead of the original, now proxied endpoint.

- A transformation of the SOAP envelope/body can be introduced via the ESB action chain both for the inbound request and outbound response. See *Section 12.1.7, "XsltAction"* or *Section 12.1.9, "SmooksAction"*

- Service versioning is possible since clients can connect to 2 or more proxy endpoints on the ESB, 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.

- Complex context-based routing via ContentBasedRouter.

Other mechanisms of doing this are inappropriate or inadequate:

- SOAPClient is used to invoke external web services, not mirror them.

- SOAPProducer only executes internally-deployed JBoss WS services.

- HttpRouter requires too much by-hand configuration for easy WS proxying.

- EBWS strips out the SOAP Envelope and only passes along the body.

With a SOAPProxy action:

- It is both a producer and consumer of web services.

- All that is required is a property pointing to the external wsdl.

- The wsdl can be automatically transformed via the optional wsdlTransform property.

- It is understood that SOAP is not tied to http. The wsdl is read, and if an http transport is defined, that will be used. Other transports (jms) will need future consideration.

- If using http, any of the HttpRouter properties can also optionally be applied to as overrides.

If the wsdl specifies an http transport then any of the HttpRouter properties can be applied.

For other possible configuration properties, see the specific SOAPProxyTransport implementations themselves.

Table 12.30.

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, if not 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, but useful when domain name matching is important for SSL certs.	No

Property	Description	Required
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: 3 slashes in the example above. This is so we can specify an absolute vs. relative file path. • `classpath://` When you want to package your wsdl inside your ESB archive.	Yes

Property	Description	Required
	Example: classpath:///META-INF/HelloWorldWS.wsdl Note the three slashes in the example above. This is so we can specify an absolute vs. 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 12.37. Sample Configuration - Basic scenario

```xml
<action name="proxy" class="org.jboss.soa.esb.actions.soap.proxy.SOAPProxy">
  <property name="wsdl" value="http://host/foo/HelloWorldWS?wsdl"/>
</action>
```

Example 12.38. Sample Configuration - Basic Authentication and SSL

```xml
<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>
```

12.8. Miscellaneous

Miscellaneous Action Processors.

12.8.1. SystemPrintln

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

Will attempt to format the message contents as XML.

Input Type	`java.lang.String`
Class	`org.jboss.soa.esb.actions.SystemPrintln`
Properties	- message - A message prefix. `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.

Example 12.39. SystemPrintln

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

12.8.2. SchemaValidationAction

This is a simple action for performing Schema (e.g. XSD or RELAX NG) based validation of XML messages.

Input Type	`java.lang.String`
Class	`org.jboss.soa.esb.actions.validation.SchemaValidationAction`
Properties	- schema - The classpath path of the validation schema file (e.g. .xsd). - schemaLanguage - (Optional) The schema type/language. Default: "http://www.w3.org/2001/XMLSchema" i.e. XSD.

Example 12.40. SystemPrintln

```xml
<action name="val" class="org.jboss.soa.esb.actions.validation.SchemaValidationAction">
   <property name="schema" value="/com/acme/validation/order.xsd"/>
</action>
```

12.9. HttpClient Configuration

The HttpClient configuration is achieved through the specification of a set of properties. The following is an example of these properties when specified as a java property file:

```
# Configurators
configurators=HttpProtocol,AuthBASIC

# HttpProtocol config...
protocol-socket-
factory=org.apache.commons.httpclient.contrib.ssl.EasySSLProtocolSocketFactory
keystore=/packages/jakarta-tomcat-5.0.28/conf/chap8.keystore
keystore-passw=xxxxxx
https.proxyHost=localhost
https.proxyPort=443

# AuthBASIC config...
auth-username=tomcat
auth-password=tomcat
authscope-host=localhost
authscope-port=18443
authscope-realm=ActiveBPEL security realm
```

12.9.1. Configurators

The only property that the HttpClientFactory requires is the configurators property, 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 12.31. 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,

12.9.1.1. HttpProtocol

Configuration of the HTTP transport properties

Table 12.32. 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
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

12.9.1.2. AuthBASIC

Configuration of the HTTP Basic Authentication properties

Table 12.33. Properties

Property	Description	Required
auth-username	Authentication Username	Yes
auth-password	Authentication Password	Yes
authscope-host	Authentication Scope Host	Yes

Property	Description	Required
authscope-port	Authentication Scope Port	Yes
authscope-domain	Authentication Scope Domain	Yes

12.9.1.3. AuthNTLM

Configuration of the HTTP Basic Authentication properties

Table 12.34. 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

Developing Custom Actions

JBoss ESB allows custom actions to be developed in a number of ways, each of which has their own advantages. These can be split into

- 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 lifecycle events

- Whether the action methods are invoked directly or via reflection

13.1. Configuring Actions Using Properties

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 would look something 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.

13.2. Lifecycle Actions

Lifecycle actions are those which are derived from the lifecycle interfaces, `org.jboss.soa.esb.actions.ActionLifecycle` and `org.jboss.soa.esb.actions.ActionPipelineProcessor`. ActionLifecycle implements the pipeline lifecycle methods, `initialise` and `destroy`, with ActionPipelineProcessor extending

this interface to include the message processing methods, `process`, `processSuccess` and `processException`.

This interface supports the implementation of *managed life-cycle* stateless actions. A single instance of a class implementing either of these interfaces 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, e.g. caching resources needed in the `initialise` method, and 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` and can be used as follows:

```java
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...
  }
}
```

13.3. Java Bean Actions

One may take another approach to setting properties by adding "setters" on the action that will correspond to the property names. This will thereby allow the

framework to populate them automatically. The action class must implement the `org.jboss.soa.esb.actions.BeanConfiguredAction` marker interface in order to make the action Bean populate automatically, for example.

```java
public class PrintMessage implements BeanConfiguredAction {
 private String information;
 private Integer repeatCount;
 public setInformation(String information) {
  this.information = information;
 }
 public setRepeatCount(Integer repeatCount) {
  this.repeatCount = repeatCount;
 }
 public Message process(Message message) {
  for (int i=0; i < repeatCount; i++) {
    System.out.println(information);
  }
 }
}
```

> **Note**
>
> The Integer parameter in the `setRepeatCount()` method is automatically converted from the String representation specified in the XML.

The `BeanConfiguredAction` method of loading properties is a good choice for actions that take simple arguments, while the `ConfigTree` method is a better option in situations when one needs to deal with the XML representation directly.

These actions *do not* support the lifecycle methods, will be instantiated for *every* message passing through the pipeline and will always have their `process` methods invoked using reflection.

13.4. Annotated Action Classes

The **JBoss Enterprise Service Bus** has a set of *action annotations* that make it easier to create clean `action` implementations. This hides the complexity associated with implementing interfaces, abstract classes and dealing with the `ConfigTree` type (the configuration information provided in the `jboss-esb.xml` file). 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.

These are the annotations:

- @Process

- @ConfigProperty

- @Initialize

- @Destroy

- @BodyParam

- @PropertyParam

- @AttachmentParam

- @OnSuccess

- @OnException

13.4.1. @Process

The simplest implementation involves creating an action with a a basic *plain old Java object* (POJO) with a single method, annotated with @Process:

```java
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):

```xml
<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:

```xml
<service .....>
      <actions>
         <action name="logger" class="com.acme.actions.MyLogAction"
                        process="log" />
      </actions>
    </service>
```

13.4.2. @Process Method Return Values

@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 an ESB message instance. This becomes the active/current instance on the `action pipeline`.

- some other type. If the method does not return an ESB message instance, the object instance that is returned will be set on the current ESB message instance on the `action pipeline`. As to where it is set on the message depends on the set-payload-location <action> configuration property, which default according to the normal `MessagePayloadProxy` rules.

13.4.3. @Process Method Parameters

Use @Process methods to specify parameters in a range of different ways. One can:

1. specify the ESB 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:

```java
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:

```java
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:

```xml
<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 one to specify a named location (in the message body, properties or attachments) for a specific parameter:

```java
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.

13.4.4. @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:

```xml
<action name="logger" class="com.acme.actions.MyLogAction">
   <property name="logFile" value="logs/my-log.log" />
   <property name="logLevel" value="DEBUG" />
</action>
```

In order to utilize these properties, one must 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.

Whilst the tasks above are generally not difficult to undertake, they can be laborious, error-prone and lead to inconsistencies across actions with regard to how configuration mistakes are handled. One may also be required to add quite a lot of code, with a net result of less overall clarity.

The annotated action addresses these problems via @ConfigProperty. Expand the MyLogActions implementation, which has two mandatory configuration properties: logFile and logLevel:

```java
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...
    }
}
```

> **i Note**
>
> One can also define the @ConfigProperty annotation on "setter" methods (as opposed to 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:

```java
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...
        }
    }
```

13.4.5. Decoding Property Values

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.

13.4.6. @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.

To illustrate, 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 want to 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:

```java
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 customized 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 (one does 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**
>
> One can optionally specify a `ConfigTree` parameter on @Initialize methods. Do this to have access to the actions which underlie that `ConfigTree` instance.

13.4.7. @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:

```java
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. One has the choice of supplying none, some or all of the parameters shown above. The Enterprise Service Bus' framework resolves the relevant parameters in each case.

13.5. Legacy Actions

Any actions which do not fit into the above categories are treated by the pipeline as legacy, inheriting the behavior present in the original Rosetta codebase. In practice this implies the following

- 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 do done via reflection

Connectors and Adapters

14.1. Introduction

Not all clients and services of JBossESB will be able to understand the protocols and Message formats it uses natively. As such there is a need to be able to bridge between ESB-aware endpoints (those that understand JBossESB) and ESB-unaware endpoints (those that do not understand JBossESB). Such bridging technologies have existed for many years in a variety of distributed systems and are often referred to as Connectors, Gateways or Adapters.

One of the aims of JBossESB is to allow a wide variety of clients and services to interact. JBossESB does not require that all such clients and services be written using JBossESB or any ESB for that matter. There is an abstract notion of an Interoperability Bus within JBossESB, such that endpoints that may not be JBossESB-aware can still be "plugged in to" the bus.

> **i** **Note**
>
> In what follows, the terms "within the ESB" or "inside the ESB" refer to ESB-aware endpoints.

All JBossESB-aware clients and services communicate with one another using Messages, (these will be described later.) A Message is simply a standardized format for information exchange, containing a header, body (payload), attachments and other data. Additionally, all JBossESB-aware services are identified using Endpoint References (EPRs).

It is important for legacy interoperability scenarios that a SOA infrastructure such as JBossESB allow ESB-unaware clients to use ESB-aware services, or ESB-aware clients to use ESB-unaware services. The concept that JBossESB uses to facilitate this interoperability is through Gateways. A gateway is a service that can bridge between the ESB-aware and ESB-unaware worlds and translate to/from Message formats and to/from EPRs.

JBossESB currently supports Gateways and Connectors. In the following sections we shall examine both concepts and illustrate how they can be used.

14.2. The Gateway

Not all users of JBossESB will be ESB-aware. In order to facilitate those users interacting with services provided by the ESB, JBossESB has the concept of a "Gateway," which is a specialized server that can accept messages from non-ESB clients and services and route them to the required destination.

A Gateway is a specialized listener process, that behaves very similarly to an ESB-aware listener. There are some important differences however:

- Gateway classes can pick up arbitrary objects contained in files, JMS messages, SQL tables and so forth (each 'gateway class' is specialized for a specific transport), whereas JBossESB listeners can only process JBossESB normalized Messages as described in "The Message" section of this document. However, those Messages can contain arbitrary data.

- Only one action class is invoked to perform the 'message composing' action. ESB listeners are able to execute an action processing pipeline.

- Objects that are 'picked up' will be used to invoke a single 'composer class' (the action) that will return an ESB Message object. This will then be delivered to a target service that must be ESB-aware. The target service defined at configuration time, will be translated at runtime into an EPR (or a list of EPRs) by the Registry. The underlying concept is that the EPR returned by the Registry is analogous to the 'toEPR' contained in the header of ESB Messages but because incoming objects are 'ESB-unaware' and there is, thus, no dynamic way to determine the toEPR, this value is provided to the gateway at configuration time and included in all outgoing messages.

There are a few off the shelf composer classes: the default 'file' composer class will just package the file contents into the Message body; same idea for JMS messages. Default message composing class for a SQL table row is to package contents of all columns specified in configuration, into a java.util.Map.

Although these default composer classes will be adequate for most uses, it is relatively straightforward for users to provide their own. The only requirements are that, firstly, they must have a constructor that takes a single ConfigTree argument, and, secondly, they must provide a 'Message composing' method (whereby the default name is 'process' but this can be configured differently in the 'process' attribute of the <action> element within the ConfigTree provided at constructor time.) The processing method must take a single argument of type Object, and return a Message value.

From JBossESB 4.5 onwards, the FileGateway accepts the file-filter-class configuration attribute which allows you to define a FileFilter implementation that may be used to select the files used by the gateway in question. Initialisation of user defined FileFilter instances is performed by the gateway if the instance is also of type org.jboss.soa.esb.listeners.gateway.FileGatewayListener.FileFilterInit, in which case the init method will be called and passed the gateway ConfigTree instance.

By default the following FileFilter implementations are defined and used by the FileGateway: if an input suffix is defined in the configuration then files matching that suffix will be returned; alternatively if there is no input suffix then any file is accepted as long as it does not match the work suffix, error suffix and post suffix.

14.2.1. Gateway Data Mappings

When a non-JBossESB message is received by a Gateway, it must be converted to a Message. How this is done and where in the Message the received data resides, depends upon the type of

Gateway. How this conversion occurs depends upon the type of Gateway; the default conversion approach is described below:

- JMS Gateway: if the input message is a JMS TextMessage, then the associated String will be placed in the default named Body location; if it is an ObjectMessage or a BytesMessage then the contents are placed within the BytesBody.BYTES_LOCATION named Body location.

- Local File Gateway: the contents are placed within the BytesBody.BYTES_LOCATION named Body location.

- Hibernate Gateway: the contents are placed within the ListenerTagNames.HIBERNATE_OBJECT_DATA_TAG named Body location.

- Remote File Gateway: the contents are placed within the BytesBody.BYTES_LOCATION named Body location.

> **i** **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.

14.2.2. How to change the Gateway Data Mappings

If you want to change how this mapping occurs then it will depend upon the type of Gateway:

- File Gateways: instances of the org.jboss.soa.esb.listeners.message.MessageComposer interface are responsible for performing the conversion. To change the default behavior, provide an appropriate implementation that defines your own compose and decompose methods. The new MessageComposer implementation should be provided in the configuration file using the composer-class attribute name.

- JMS and Hibernate Gateways: these implementations use a reflective approach for defining composition classes. Provide your own Message composer class and use the composer-class attribute name in the configuration file to inform the Gateway which instance to use. You can use the composer-process attribute to inform the Gateway which operation of the class to call when it needs a Message; this method must take an Object and return a Message. If not specified, a default name of process is assumed.

> **i** **Note**
>
> Whichever of the methods you use to redefine the Message composition, it is worth noting that you have complete control over what is in the Message and not just

> the Body. For example, if you want to define ReplyTo or FaultTo EPRs for the newly created Message, based on the original content, sender etc., then you should consider modifying the header too.

14.3. Connecting via JCA

You can use JCA Message Inflow as an ESB Gateway. This integration does not use MDBs, but rather ESB's lightweight inflow integration. To enable a gateway for a service, you must first implement an endpoint class. This class is a Java class that must implement the org.jboss.soa.esb.listeners.jca.InflowGateway class:

```java
public interface InflowGateway
{
  public void setServiceInvoker(ServiceInvoker invoker);
}
```

The endpoint class must either have a default constructor, or a constructor that takes a ConfigTree parameter. This Java class must also implement the messaging type of the JCA adapter you are binding to. Here's a simple endpoint class example that hooks up to a JMS adapter:

```java
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 an MDB that is pooled. Only one instance of the class will service each and every incoming message, so you must write thread safe code.

At configuration time, the ESB creates a ServiceInvoker and invokes the setServiceInvoker method on the endpoint class. The ESB then activates the JCA endpoint and the endpoint class instance is ready to receive messages. In the JmsEndpoint example, the instance receives a JMS message and converts it to an ESB message type. Then it uses the ServiceInvoker instance to invoke on the target service.

> **Note**
>
> The JMS Endpoint class is provided for you with the ESB distribution under org.jboss.soa.esb.listeners.jca.JmsEndpoint It is quite possible that this class would be used over and over again with any JMS JCA inflow adapters.

14.3.1. Configuration

A JCA inflow gateway is configured in a jboss-esb.xml file. Here's an example:

```xml
<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>
```

```xml
<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 14.1. 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 filename of the RAR you deployed, e.g., 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.

14.3.2. Mapping Standard activation properties

A number of ESB properties are automatically mapped onto the activation configuration using an ActivationMapper. The properties, their location and their purpose are described in the following table

Table 14.2. 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
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.

The mapping of these properties onto an activation specification can be overridden by specifying a class which implements the ActivationMapper interface and can be declared globally or within each ESB deployment configuration.

Specifying the ActivationMapper globally is done through the jbossesb-properties.xml file and defines the default mapper used for the specified JCA adapter The name of the property to be configured is "org.jboss.soa.esb.jca.activation.mapper.<adapter name>" and the value is the class name of the ActivationMapper.

The following snippet the configuration of the default ActivationMapper used to map the properties on the the activation specification for the JBoss JCA adapter, jms-ra.rar.

```
<properties name="jca">
    <property name="org.jboss.soa.esb.jca.activation.mapper.jms-ra.rar"
        value="org.jboss.soa.esb.listeners.jca.JBossActivationMapper"/>
  </properties>
```

Specifying the ActivationMapper within the deployment will override any global setting. The mapper can be specified within the listener, the bus or the provider with the precedence being the same order.

The following snippet shows an example specifying the mapper configuration within the listener configuration.

```
<jms-listener name="listener" busidref="bus" maxThreads="100">
    <property name="jcaActivationMapper" value="TestActivationMapper"/>
```

```
    </jms-listener>
```

The following snippet shows an example specifying the mapper configuration within the bus configuration:

```
<jms-bus busid="bus">
    <property name="jcaActivationMapper" value="TestActivationMapper"/>
    <jms-message-filter dest-type="TOPIC" dest-name="DestName"/>
  </jms-bus>
```

The following snippet shows an example specifying the mapper configuration within the provider configuration.

```
<jms-jca-provider name="provider" connection-factory="ConnectionFactory">
    <property name="jcaActivationMapper" value="TestActivationMapper"/>
    <jms-bus busid="bus">
      <jms-message-filter dest-type="TOPIC" dest-name="DestName"/>
    </jms-bus>
  </jms-jca-provider>
```

Appendix A. Appendix A: Writing JAXB Annotation Introduction Configurations

JAXB Annotation Introduction configurations are very easy to write. If you're already familiar with the JAXB Annotations, you'll have no problem writing a JAXB Annotation Introduction configuration.

The XSD for the configuration is available online 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.

Only 3 annotations are currently supported:

1. @XmlType (*https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlType.html*): On the "Class" element.

2. @XmlElement (*https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlElement.html*): On the "Field" and "Method" elements.

3. @XmlAttribute (*https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlAttribute.html*): On the "Field" and "Method" elements.

The basic structure of the configuration file follows the basic structure of a Java class i.e. a "Class" containing "Fields" and "Methods". The <Class>, <Field> and <Method> elements all require a "name" attribute for the name of the Class, Field or Method. The value of this name attribute supports regular expressions. This allows a single Annotation Introduction configuration to be targeted at more than one Class, Field or Member e.g. setting the namespace for a fields in a Class, or for all Classes in a package etc.

The Annotation Introduction configurations match exactly with the Annotation definitions themselves, with each annotation "element-value pair" represented by an attribute on the annotations introduction configuration. Use the XSD and your IDE to editing the configuration.

So here is an example:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jaxb-intros xmlns="http://www.jboss.org/xsd/jaxb/intros">

  <!--
  The type namespaces on the customerOrder are
  different from the rest of the message...
```

```
-->

<Class name="com.activebpel.ordermanagement.CustomerOrder">
 <XmlType propOrder="orderDate,name,address,items" />
  <Field name="orderDate">
   <XmlAttribute name="date" required="true" />
  </Field>
  <Method name="getXYZ">
   <XmlElement
    namespace="http://org.jboss.esb/quickstarts/bpel/ABI_OrderManager"
    nillable="true" />
  </Method>
</Class>

<!-- More general namespace config for the rest of the message... -->
<Class name="com.activebpel.ordermanagement.*">
 <Method name="get.*">
  <XmlElement namespace="http://ordermanagement.activebpel.com/jaws" />
 </Method>
</Class>

</jaxb-intros>
```

Appendix B. Appendix B: Service Orientated Architecture Overview

JBossESB is a Service Oriented Architecture (SOA) infrastructure. SOA represents a popular architectural paradigm [1] for applications, with Web Services as probably the most visible way of achieving an SOA. [2] Web Services implement capabilities that are available to other applications (or even other Web Services) via industry standard network and application interfaces and protocols. SOA advocates an approach in which a software component provides its functionality as a service that can be leveraged by other software components. Components (or services) represent reusable software building blocks.

SOA allows the integration of existing systems, applications and users into a flexible architecture that can easily accommodate changing needs. Integrated design, reuse of existing IT investments and above all, industry standards are the elements needed to create a robust SOA.

As enterprises slowly emerge from the mad rush of cost reduction into a more stable period of cost management, many of them find themselves in unfamiliar territory. Prior to the economic slow down, most firms understood the options they had for IT investment. Many embarked on major package implementations (e.g., Siebel, Peoplesoft and so on), while others built on the legacy systems they have trusted for years. Either way, most firms recognized the return promised and made the investment. Today, the appetite for such large investment is gone.

However, enterprises still need to make forward progress and keep ahead of the competition. SOA (and typically Web Services as a concrete implementation of those principles) make this possible. The result is dramatic improvements in collaboration between users, applications and technology components, generating significant value for any business creating competitive advantage.

Imagine a company that has existing software from a variety of different vendors, e.g., SAP, PeopleSoft. Some of these software packages may be useful to conduct business with other companies (customers, suppliers, etc.) and therefore what the company would like to do is to take those existing systems and make them available to other companies, by exposing them as services. A service here is some software component with a stable, published interface that can be invoked by clients (other software components). So, requesting and executing services involves software components owned by one company talking to components owned by another company, i.e., business-to-business (B2B) transactions.

Conventional distributed system infrastructures (middleware) are not sufficient for these cross-organizational exchanges. For instance

- You would need agreement between the parties involved on the middleware platform.

---

[1] The principles behind SOA have been around for many years, but Web Services have popularised

[2] It is possible to build non-SOA applications using Web Services.

- There is an implicit (and sometimes explicit) lack of trust between the parties involved.

- Business data is confidential and should only to be seen by the intended recipient.

- Many assumptions of conventional middleware are invalid in cross-organizational interactions. Transactions, for instance, last longer, possibly for hours or days, so conventional transaction protocols such as "two phase commit" are not applicable.

So, in B2B exchanges, the lack of standardization across middleware platforms makes point-to-point solutions costly to realize in practice. The Internet alleviated some of these problems by providing standard interaction protocols (HTTP) and data formats (XML) but, by themselves, these standards are not enough to support application integration. They do not define interface definition languages, name and directory services, transaction protocols and so forth. It is the gap between that which the Web provides and what application integration requires that Web services are trying to fill.

However, whilst the ultimate goal of SOA is inter-company interactions, services do not need to be accessed using the Internet. They can be easily made available to clients residing on a local network. It is common for web services to be used in this context to provide integration between systems operating within a single company.

An example of how Web services can connect applications both intra-company and inter-company can be understood by considering a stand-alone inventory system. If you don't connect it to anything else, it's not as valuable as it could be. The system can track inventory, but not much more. Inventory information may have to be entered separately in the accounting and customer relationship management systems. The inventory system may be unable to automatically place orders to suppliers. The benefits of such an inventory system are diminished by high overhead costs.

However, if you connect your inventory system to your accounting system with XML, it gets more interesting. Now, whenever you buy or sell something, the implications for your inventory and your cash flow can be tracked in one step. If you go further, and connect your warehouse management system, customer ordering system, supplier ordering systems, and your shipping company with XML, suddenly that inventory management system is worth a lot. You can do end-to-end management of your business while dealing with each transaction only once, instead of once for every system it affects. A lot less work and a lot less opportunity for errors. These connections can be made easily using Web services.

Businesses are waking up to the benefits of SOA. These include:

- opening the door to new business opportunities by making it easy to connect with partners;

- saving time and money by cutting software development time and consuming a service created by others;

- increasing revenue streams by easily making your own services available.

B.1. Why SOA?

The problem space can be categorized by past IT investments in the area of eProcurement, eSourcing, Supply Chain Management, Customer Relationship Management (CRM) and, indeed, Internet computing in general. All of these investments were made in a "silo." The decisions made in this space, (along with the incremental growth of these systems to meet short-term tactical requirements), hurt the long-term viability of the applications and infrastructure.

The three key drivers for implementing an SOA approach are:

1. Cost Reduction: Achieved by the ways services talk to each other. The direct cost effect is delivered through enhanced operations productivity, effective sourcing options, and a significantly enhanced ability to shift ongoing costs to a variable model.

2. Delivering IT solutions faster and smarter: A standards based approach will allow organizations to connect and share information and business processes much faster and easier than before. IT delivery productivity is markedly improved through simplification of the developer's role by providing standard frameworks and interfaces. Delivery timescales have been drastically reduced by easing the integration load of individual functionality, and applying accelerated delivery techniques within the environment.

3. Maximizing return on investment: Web Services opens the way for new business opportunities by enabling new business models. Web Services present the ability to measure value and discrete return much differently than traditional functional-benefit methods. Typical Total Cost of Ownership (TCO) models do not take into account the lifetime value generated by historical investment. This cost centric view destroys many opportunities to exploit these past investments and most enterprises end up building redundancy into their architecture, not out of necessity, but of perceived need. These same organizations focus the value proposition of their IT investment on a portfolio of applications, balanced by the overhead of infrastructure. An approach based on Web Services takes into account the lifetime contribution of legacy IT investment and promotes an evolution of these investments rather than a planned replacement.

SOA/Web Services fundamentally changes the way enterprise software is developed and deployed. SOA has evolved to the point where new applications will not be developed using monolithic approaches but, instead, become a virtualized on-demand execution model that breaks the current economic and technological bottleneck that has been caused by traditional approaches.

Software-as-a-service has become a pervasive model for forward-looking enterprises wishing to streamline operations, as it leads to lower cost of ownership and provides competitive differentiation in the marketplace. Using Web Services gives enterprises a viable opportunity to drive significant costs out of software acquisitions, react to rapidly changing market conditions and conduct transactions with business partners at will. Loosely coupled, standards-based architectures are one approach to distributed computing. They allow software resources available on the network to be leveraged. Applications that provide separate business processes, presentation rules, business rules and data access in separate, loosely-coupled layers will not only assist in the construction of better software but also make it more adaptable to future change.

SOA will allow for combining existing functions with new development efforts, allowing the creation of composite applications. Leveraging what works lowers the risks in software development projects. By reusing existing functions, it leads to faster deliverables and better delivery quality.

Loose coupling helps preserve the future by allowing parts to change at their own pace without the risks linked to costly migrations using monolithic approaches. SOA allows business users to focus on business problems at hand without worrying about technical constraints. For the individuals who develop solutions, SOA helps in the following manner:

- Business analysts focus on higher order responsibilities in the development lifecycle while increasing their own knowledge of the business domain.

- Separating functionality into component-based services that can be tackled by multiple teams enables parallel development.

- Quality assurance and unit testing become more efficient; errors can be detected earlier in the development lifecycle

- Development teams can deviate from initial requirements without incurring additional risk

- Components within architecture can aid in becoming reusable assets in order to avoid reinventing the wheel

- Functional decomposition of services and their underlying components with respect to the business process helps preserve the flexibility, future maintainability and eases integration efforts

- Security rules are implemented at the service level and can solve many security considerations within the enterprise

B.2. Basics of SOA

Traditional distributed computing environments have been tightly coupled in that they do not deal with a changing environment well. For instance, if an application is interacting with another application, how do they handle data types or data encoding if data types in one system change? How are incompatible data-types handled?

The service-oriented architecture (SOA) consists of three roles: requester, provider, and broker.

Service Provider

A service provider allows access to services, creates a description of a service and publishes it to the service broker.

Service Requestor

A service requester is responsible for discovering a service by searching through the service descriptions given by the service broker. A requester is also responsible for binding to services provided by the service provider.

Service Broker

> A service broker hosts a registry of service descriptions. It is responsible for linking a requestor to a service provider.

B.3. Advantages of SOA

SOA provide several significant benefits for distributed enterprise systems. Some of the most notable benefits include: interoperability, efficiency, and standardization. We will briefly explore each of these in this section.

B.3.1. Interoperability

Interoperability is the ability of software on different systems to communicate by sharing data and functionality. SOA/Web Services are as much about interoperability as they are about the Web and Internet scale computing. Most companies will have numerous business partners throughout the life of the company. Instead of writing a new addition to your applications every time you gain a new partner, you can write one interface using Web service technologies like SOAP. So now your partners can dynamically find the services they need using UDDI and bind to them using SOAP. You can also extend the interoperability of your systems by implementing Web services within your corporate intranet. With the addition of Web services to your intranet systems and to your extranet, you can reduce the cost integration, increase communication and increase your customer base.

It is also important to note that the industry has even established the Web Services Interoperability Organization.

"The Web Services Interoperability Organization is an open industry effort chartered to promote Web Services interoperability across platforms, applications, and programming languages. The organization brings together a diverse community of Web services leaders to respond to customer needs by providing guidance, recommended practices, and supporting resources for developing interoperable Web services." (*http://www.ws-i.org*)

The WS-I will actually determine whether a Web service conforms to WS-I standards as well as industry standards. In order to establish integrity and acceptance, companies will seek to build their Web services in compliance with the WS-I standards.

B.3.2. Efficiency

SOA will enable you to reuse your existing applications. Instead of creating totally new applications, you can create them using various combinations of services exposed by your existing applications. Developers can be more efficient because they can focus on learning industry standard technology. They will not have to spend a lot of time learning every new technology that arises. For a manager this means a reduction in the cost of buying new software and having to hire new developers with new skill sets. This approach will allow developers to meet changing business requirements and reduce the length of development cycles for projects. Overall, SOA provides for an increase in efficiency by allowing applications to be reused, decreasing the learning curve for developers and speeding up the total development process.

B.3.3. Standardization

For something to be a true standard, it must be accepted and used by the majority of the industry. One vendor or small group of vendors must not control the evolution of the technology or specification. Most if not all of the industry leaders are involved in the development of Web service specifications. Almost all businesses use the Internet and World Wide Web in one form or another. The underlying protocol for the WWW is of course HTTP. The foundation of Web services is built upon HTTP and XML. Although SOA does not mandate a particular implementation framework, interoperability is important and SOAP is one of the few protocols that all good SOA implementations can agree on.

B.3.4. Stateful and Stateless Services

Most proponents of Web Services agree that it is important that its architecture is as scalable and flexible as the Web. As a result, the current interaction pattern for Web Services is based on coarse-grained services or components. The architecture is deliberately not prescriptive about what happens behind service endpoints: Web Services are ultimately only concerned with the transfer of structured data between parties, plus any meta-level information to safeguard such transfers (e.g., by encrypting or digitally signing messages). This gives flexibility of implementation, allowing systems to adapt to changes in requirements, technology etc. without directly affecting users. Furthermore, most businesses will not want to expose their back-end implementation decisions and strategies to users for a variety of reasons.

In distributed systems such as CORBA, J2EE and DCOM, interactions are typically between stateful objects that reside within containers. In these architectures, objects are exposed as individually referenced entities, tied to specific containers and therefore often to specific machines. Because most web services applications are written using object-oriented languages, it is natural to think about extending that architecture to Web Services. Therefore a service exposes web services resources that represent specific states. The result is that such architectures produce tight coupling between clients and services, making it difficult for them to scale to the level of the World Wide Web.

Right now, there are two primary models for the session concept that are being defined by companies participating in defining Web services: the WS-Addressing EndpointReferences with ReferenceProperties/ReferenceParameters and the WS-Context explicit context structure. Both of these models are supported within JBossESB. The WS-Addressing session model provides coupling between the web service endpoint information and the session data, which is analogous to object references in distributed object systems.

WS-Context provides a session model that is an evolution of the session models found in HTTP servers, transaction, and MOM systems. On the other hand, WS-Context allows a service client to more naturally bind the relationship to the service dynamically and temporarily. The client's communication channel to the service is not impacted by a specific session relationship.

This has important implications as we consider scaling Web services from internal deployments to general services offered on the Internet. The current interaction pattern for web services is based

on "coarse-grained" services or components. The architecture is deliberately not prescriptive about what happens behind service endpoints: web services are ultimately only concerned with the transfer of structured data between parties, plus any meta-level information to safeguard such transfers (e.g., by encrypting or digitally signing messages). This gives flexibility of implementation, allowing systems to adapt to changes in requirements, technology etc. without directly affecting users. It also means that issues such as whether or not a service maintains state on behalf of users or their (temporally bounded) interactions, has been an implementation choice not typically exposed to users.

If a session-like model based on WS-Addressing were to be used when interacting with stateful services, then the tight coupling between state and service would impact on clients. As in other distribution environments where this model is used (e.g., CORBA or J2EE), the remote reference (address) that the client has to the service endpoint must be remembered by the client for subsequent invocations. If the client application interacts with multiple services within the same logical session, then it is often the case that the state of a service has relevance to the client only when used in conjunction with the associated states of the other services. This necessarily means that the client must remember each service reference and somehow associate them with a specific interaction; multiple interactions will obviously result in different reference sets that may be combined to represent each sessions.

For example, if there are N services used within the same application session, each maintaining m different states, the client application will have to maintain N*m reference endpoints. It is worth remembering that the initial service endpoint references will often be obtained from some bootstrap process such as UDDI. But in this model, these references are stateless and of no use beyond starting the application interactions. Subsequent visits to these sites that require access to specific states must use different references in the WS-Addressing model.

This obviously does not scale to an environment the size of the Web. However, an alternative approach is to use WS-Context and continue to embrace the inherently loosely-coupled nature of Web Services. As we have shown, each interaction with a set of services can be modeled as a session, and this in turn can be modeled as a WS-Context activity with an associated context. Whenever a client application interacts with a set of services within the same session, the context is passed on to the services and they map this context to the necessary states that the client interaction requires.

How this mapping occurs is an implementation specific choice that need not be exposed to the client. Furthermore, since each service within a specific session gets the same context, upon later revisiting these services and providing the same context again, the client application can be sure to return to a consistent set of states. So for the N services and m states in our previous example, the client need only maintain N endpoint references and as we mentioned earlier, typically these will be obtained from the bootstrap process anyway. Thus, this model scales much better.

B.4. JBossESB and its Relationship with SOA

SOA is more than technology: it does not come in a shrink-wrapped box and requires changes to the way in which people work and interact as much as assistance from underlying infrastructures,

such as JBossESB. With JBossESB 4.8, Red Hat is providing a base SOA infrastructure upon which SOA applications can be developed. With the 4.2.1 release, most of the necessary hooks for SOA development are in place and Red Hat is working with its partners to ensure that their higher level platforms leverage these hooks appropriately. However, the baseline platform (JBossESB) will continue to evolve, with out-of-the-box improvements around tooling, runtime management, service life-cycle etc. In JBossESB 4.8, it may be necessary for developers to leverage these hooks themselves, using low-level API and patterns.

Appendix C. References

The following references are recommended by the JBoss Enterprise SOA Platform team for guidance when building SOA applications.

Recommended Reading

Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. by Gregor Hohpe and Bobby Woolf. 0321200683. Addison-Wesley Professional.

Enterprise Service Oriented Architectures. by James McGovern, Oliver Sims, Ashish Jain, and Mark Little. 140203704X. Springer.

Open Source SOA. by Jeff Davis. 1933988541. Manning Publications Co.

SOA Design Patterns. by Thomas Erl. 0136135161. Prentice Hall PTR.

Glossary

Access Control List	A mean of determining the appropriate access rights to a given object depending on certain aspects of the process that is making the request.
Action Classes	A component that is responsible for doing a certain type of work after a receipt of a message inside the ESB.
Bus	A subsystem that transfers data between computer components inside a computer or between computers. Unlike a point-to-point connection, a bus can logically connect several components over the same structure.
Content Based Router	An interchangable service inside the ESB that provides capabilities for message routing based on the content of the message.
CORBA	Common Object Request Broker Architecture. A standard defined by the Object Management Group that enables software components written in multiple computer languages and running on multiple computers to interoperate.
CORBA IDL	CORBA Interface Definition Language. A computer language used to describe a software component's interface. It describes an interface in a language-neutral way, enabling communication between software components written in different languages.
EAI	Enterprise Application Integration. A practice that makes use of software and computer systems architectural principles to integrate a set of different enterprise computer applications.
Endpoint Reference (EPR)	A standard XML structure used to identify and address services inside the ESB. This includes the destination address of the message, any additional parameters (reference properties) necessary to route the message to the destination, and optional metadata (reference parameters) about the service.
ESB	Enterprise Service Bus. An abstraction layer on top of an implementation of an enterprise messaging system that provides the features with which Service Oriented Architectures may be implemented.
Fault	A type of message that express an error condition inside a Web Service. Similar to the Exception object in some programming languages.

Gateway	A specialized ESB listener process that can accept messages from non-ESB clients and services and route them to the required destination inside the ESB, taking care of the appropriate bridging of message types and EPRs.
J2EE/JEE	Java Platform Enterprise Edition (formerly known as Java 2 Platform Enterprise Edition). A programming platform, based on the Java language, for developing and running distributed multi-tier Java applications. It is based largely on modular software components running on an application server.
JBI	Java Business Integration. An API that provides a standard pluggable architecture to build integration systems that hosts service producers and consumers components. Components interoperate through mediated normalized message exchanges.
JMS	Java Message Service. An API for sending messages between two or more systems.
JTA	Java Transaction API. An API that allows distributed transactions to be done across multiple XA resources
Listener Classes	A component that encapsulates the endpoints for message reception on the ESB.
Message	A data item that is sent (usually asynchronously) to a communication endpoint. This concept is the higher-level version of a datagram except that messages can be larger than a packet and can optionally be made reliable, durable, secure, and/or transacted.
Message Factory	A service inside the ESB that can build specific types of messages according to their serialization capabilities.
Message Store	A interchangeable service inside the ESB that persists messages for auditing and tracking purposes.
MOM	Message Oriented Middleware. A software component that makes possible inter-application communication relying on asynchronous message-passing.
Quality of Service	A term that refers to control mechanisms that can provide different priority to different users or data flows, or guarantee a certain level of performance to a data flow in accordance with requests from the application program.
RPC	Remote Procedure Call. A protocol that allows a computer program running on one computer to call a subroutine on another

computer without the programmer explicitly coding the details for this interaction.

SCA

Service Component Architecture. A set of specifications that describe a model for building applications and systems using Service-Oriented Architecture. It encourages an SOA organization of applications based on components that offer their capabilities through service-oriented interfaces and which consume functions offered by other components through service-oriented interfaces, called service references.

Service Registry

A persistent repository of Service information. Used by ESB components to publish, discover and consume services.

SOA

Service Oriented Architecture. A perspective of software architecture that defines the use of loosely coupled software services to support the requirements of the business processes and software users. In an SOA environment, resources on a network are made available as independent services that can be accessed without knowledge of their underlying platform implementation.

SOAP

A protocol for exchanging XML-based messages over computer network, normally using HTTP. SOAP forms the foundation layer of the Web services stack, providing the basic messaging framework.

Transformation Service

A interchangeable service inside the ESB that provides capabilities for transforming messages from one data format to another.

UDDI

Universal Description, Discovery, and Integration. A platform-independent, XML-based registry and core Web Services standard. It is designed to be interrogated by SOAP messages and to provide access to Web Services Description Language documents describing the protocol bindings and message formats required to interact with the web services listed in its directory.

WS-Addressing

A Web Service specification for addressing web services and messages in a transport-neutral manner. This specification enables messaging systems to support message transmission through networks that include processing nodes such as endpoint managers, firewalls, and gateways.

WS-BPEL

Web Services Business Process Execution Language. A choreography language for the formal specification of business

	processes and business interaction protocols using Web Services. The BPEL messaging facilities depend on the use of Web Services Description Language (WSDL) 1.1 to describe incoming and outgoing messages.
WS-Context	A Web Service specification that provides a definition, a structuring mechanism, and a software service definition for organizing and sharing context across multiple Web Services endpoints. The context contains information (such as a unique identifier) that allows a series of operations to share a common outcome.
WSDL	Web Services Description Language. An XML format for describing the public interface to a Web services based on how to communicate using the web service; namely, the protocol bindings and message formats required to interact with it.
WS-Policy	A Web Service specification that allows web services to advertise their policies (on security, Quality of Service, etc.) and for web service consumers to specify their policy requirements.
WS-Security	A Web Service specification that provides a set of mechanisms to secure SOAP message exchanges. Specifically, it describes enhancements to provide quality of protection through the application of message integrity, message confidentiality, and single message authentication to SOAP messages.
WS-Trust	A Web Service specification that uses the secure messaging mechanisms of WS-Security to define additional primitives and extensions for the issuance, exchange and validation of security tokens.
XA	An X/Open specification for distributed transaction processing. It describes the interface between the global transaction manager and the local resource manager to support a two-phase commit protocol.
XML	Extensible Markup Language. A general-purpose markup language that supports a wide variety of applications. Its primary purpose is to facilitate the sharing of data across different information systems.

Appendix D. Revision History

Revision History

Revision 1	Fri Jul 16 2010	DavidLe Sage<dlesage@redhat.com>, DarrinMison<dmison@redhat.com>

Initial conversion from OpenOffice ODT files.