JBossESB 4.3 GA

Programmers Guide

JBESB-PG-5/20/08


JBoss®
a division of Red Hat

Legal Notices

The information contained in this documentation is subject to change without notice.

JBoss Inc. makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. JBoss Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Java™ and J2EE is a U.S. trademark of Sun Microsystems, Inc. Microsoft® and Windows NT® are registered trademarks of Microsoft Corporation. Oracle® is a registered U.S. trademark and Oracle9™, Oracle9 Server™ Oracle9 Enterprise Edition™ are trademarks of Oracle Corporation. Unix is used here as a generic term covering all versions of the UNIX® operating system. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Copyright

Software Version

JBossESB 4.3 GA

Restricted Rights Legend

Contents

Table of Contents

About This Guide

What This Guide Contains

The Programmers Guide contains information on how to use JBossESB 4.3 GA.

Audience

This guide is most relevant to engineers who are responsible for using JBossESB 4.3 GA installations and want to know how it relates to SOA and ESB principles.

Prerequisites

None.

Organization

This guide contains the following chapters:

- **Chapter 1, The ESB:** an overview of the ESB concept.
- **Chapter 2, JBossESB:** a description of the core components within JBossESB and how they are intended to be used.
- **Chapter 3, Services and Messages:** a discussion on the two core concepts within JBossESB.
- **Chapter 4, Building and Using Services:** How to use listeners and actions to develop services and consumers.
- **Chapter 5, Other Components:** An overview of the other services within JBossESB.
- **Chapter 6, Example:** A worked example using some of the principles examined so far.
- **Chapter 7, Advanced Topics:** Some advanced concepts available within JBossESB, such as automatic fail-over and scheduling.
- **Chapter 8, Fault-tolerance and Reliability:** A discussion of how failures may affect applications developed on an ESB and how JBossESB can help tolerate them.
- **Chapter 9, Configuration:** a description of the configuration options within JBossESB.

Documentation Conventions

The following conventions are used in this guide:

Convention	Description
Italic	In paragraph text, italic identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.
Bold	Emphasizes items of particular importance.
Code	Text that represents programming code.
Function | **Function**	A path to a function or dialog box within an interface. For example, "Select File | Open." indicates that you should select the Open function from the File menu.
( ) and |	Parentheses enclose optional items in command syntax. The vertical bar separates syntax items in a list of choices. For example, any of the following three items can be entered in this syntax: ***persistPolicy (Never | OnTimer | OnUpdate | NoMoreOftenThan)***
Note: **Caution**:	A note highlights important supplemental information. A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or

Table 1 Formatting Conventions

Additional Documentation

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

1. **JBossESB 4.3 GA** *Trailblazer Guide*: Provides guidance for using the trailblazer example.

2. **JBossESB 4.3 GA** *Getting Started Guide*: Provides a quick start reference to configuring and using the ESB.

3. **JBossESB 4.3 GA** *Administration Guide*: How to manage JBossESB.

4. **JBossESB 4.3 GA** *Release Notes*: Information on the differences between this release and previous releases.

5. **JBossESB 4.3 GA** *Services Guides*: Various documents related to the services available with the ESB.

Contacting Us

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

Chapter 1

The Enterprise Service Bus

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: Business Process Monitoring, Integrated Development Environment, Human Workflow User Interface, Business Process Management, Connectors, Transaction Manager, Security, Application Container, Messaging Service, Metadata Repository, Naming and Directory Service, Distributed Computing Architecture.

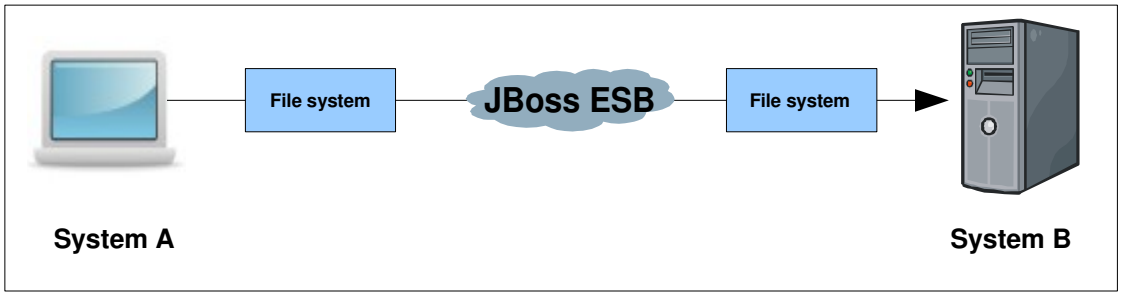
As with EAI systems, ESB is *not* about business logic – that is left to higher levels. It is about infrastructure logic. Although there are many different definitions of what constitutes an ESB, what everyone agrees on now is that an ESB is part of an SOA infrastructure. However, SOA is not simply a 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.

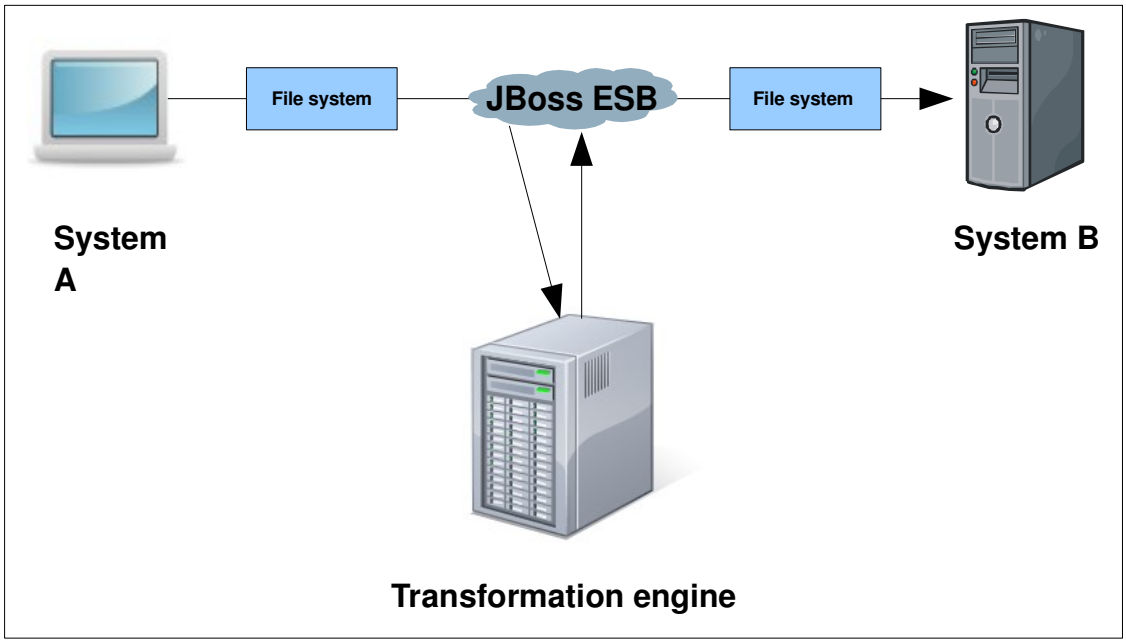
When would you use JBossESB?

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.

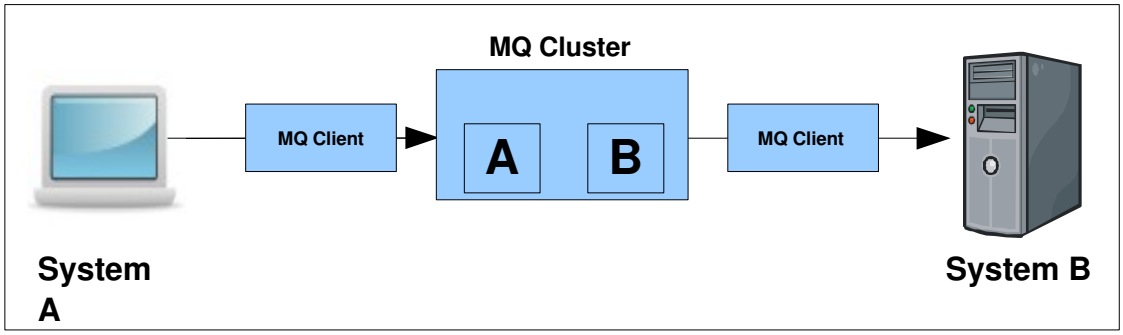
The first diagram shows simple file movement between two systems where messaging queuing is not involved.



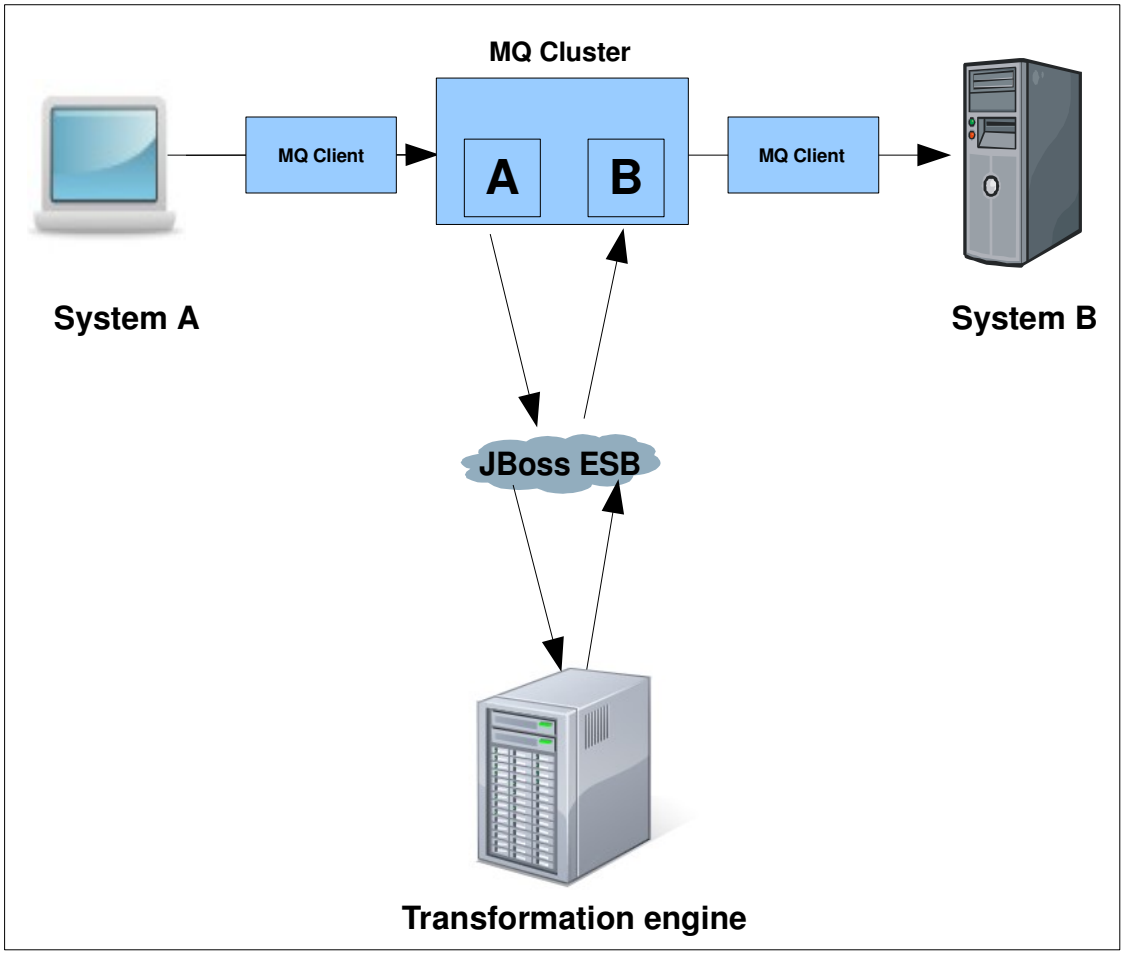
The next diagram illustrates how transformation can be injected into the same scenario using JBossESB.



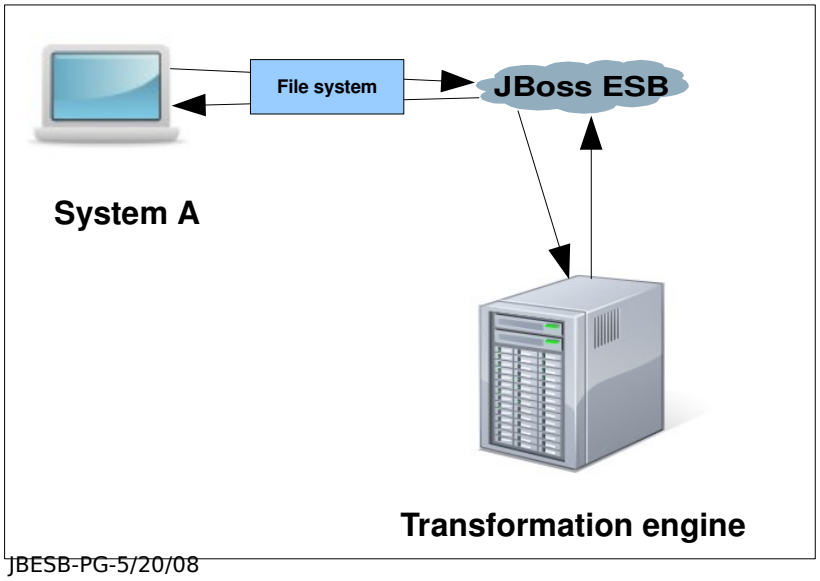
In the next series of examples, we use a queuing system (e.g., a JMS implementation).



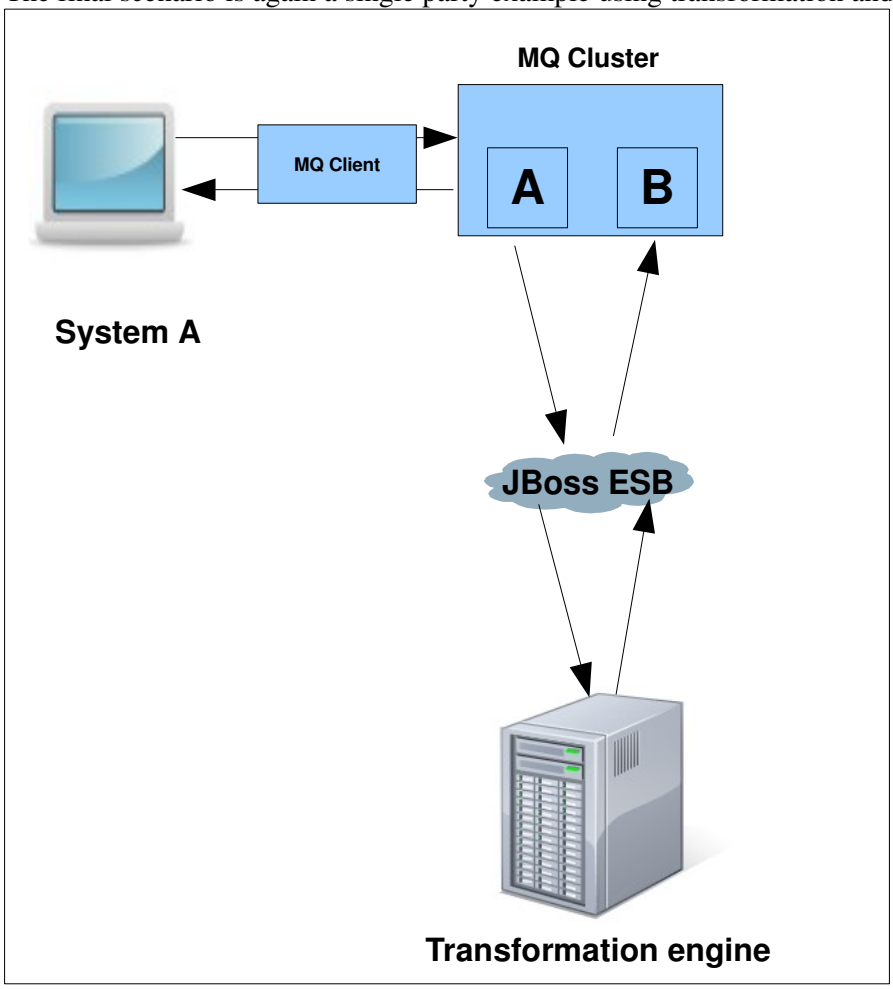
The diagram below shows transformation and queuing in the same situation.



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.



The final scenario is again a single party example using transformation and a queuing system.



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

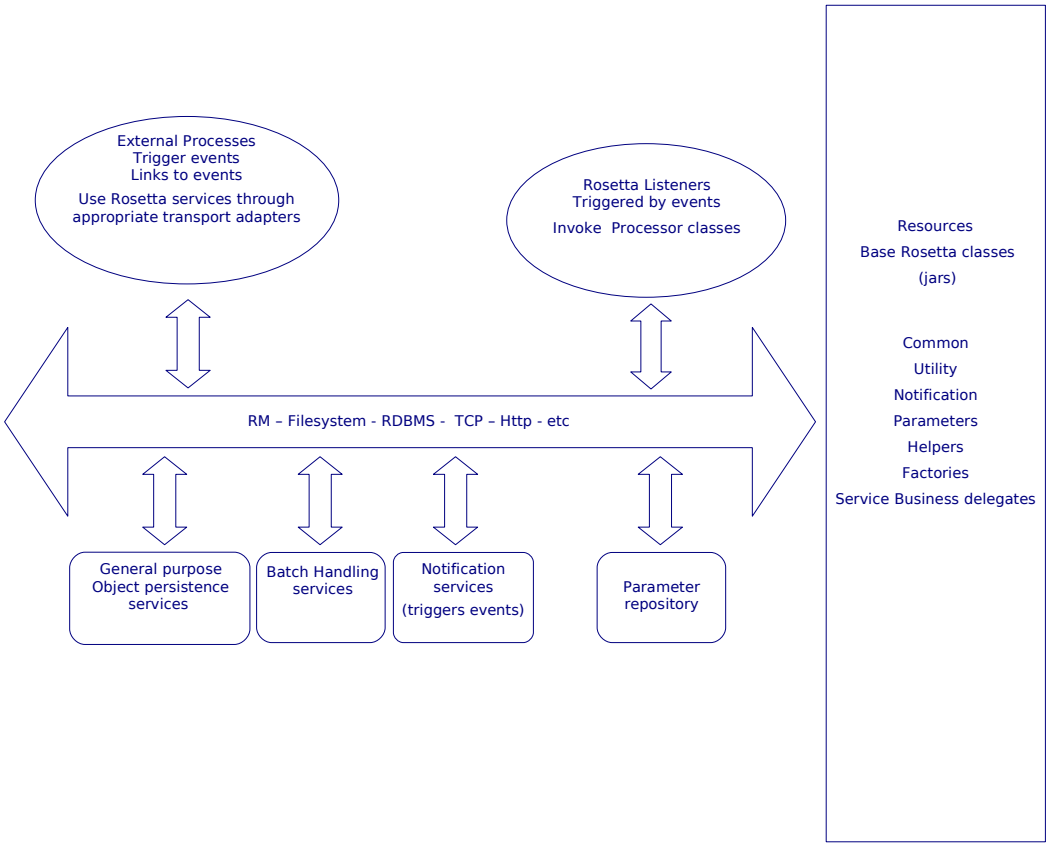
Chapter 2

JBossESB

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 architecture of Rosetta is shown below in Figure 1:

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



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 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.

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 filesystem) 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 Action Guide.
- A Content Based Routing Service. See the CBR Guide.
- A Message Repository, for saving messages/events exchanged within the ESB. See the Message Store Guide 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.

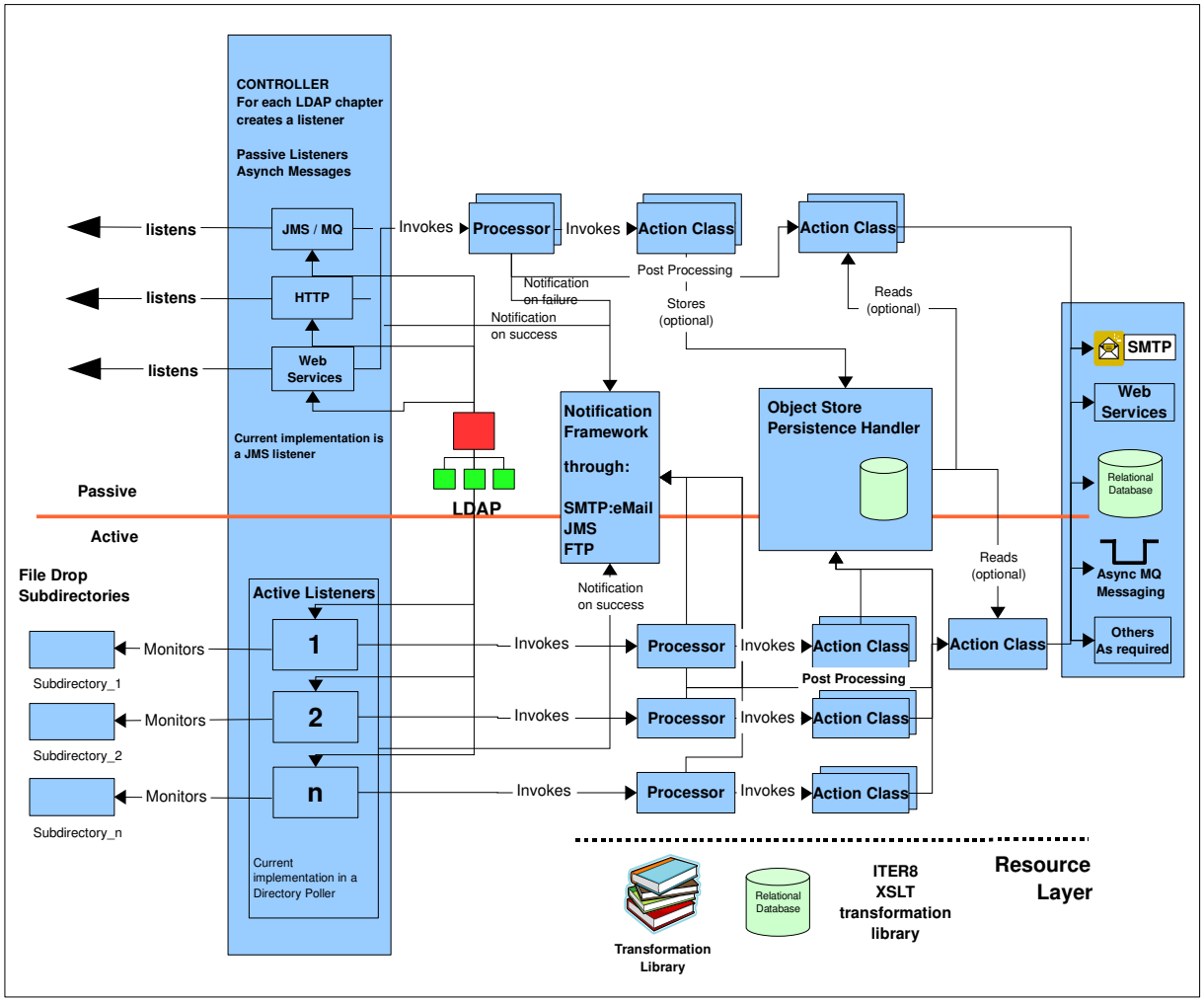


Figure 2: ESB Core components.

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.

Chapter 3

Services and Messages

Introduction

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.

The Service

A "Service" in JBossESB is defined 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 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">

    <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.

```
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 Registry Guide) 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 (JMS, FTP, HTTP etc). No listeners are configured on the Service in the above example. This is perfectly valid. Every Service is, by default, configured with an "InVM" listener, so the ServiceInvoker will always have access to InVM addresses for locally deployed Services (i.e. in the same VM). To add additional Endpoints for the Service, we need to explicitly add listener configurations on the Service. JBossESB supports two forms of listener configuration:

1. **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.

2. **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 i.e. 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 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">
             <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.

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:

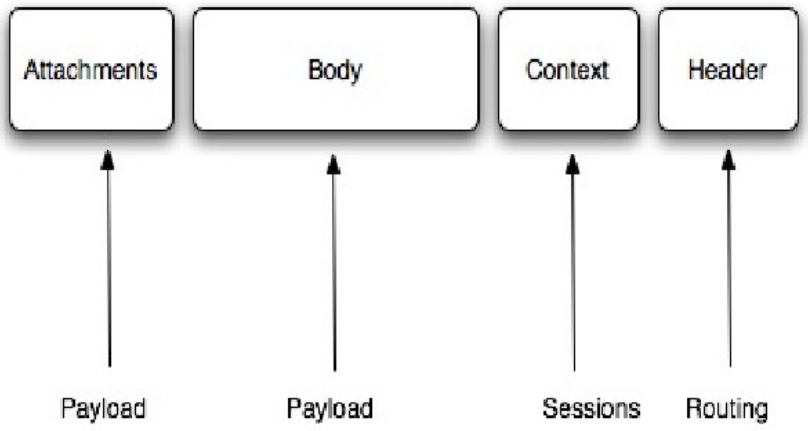
- Use one-way message exchanges rather than request-response.
- Keep the contract definition within the exchanged messages. Try not to define a service interface that exposed back-end implementation choices, because that will make changing the implementation more difficult later.
- Use an extensible message structure for the message payload so that changes to it can be versioned over time, for backward compatibility.
- 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.

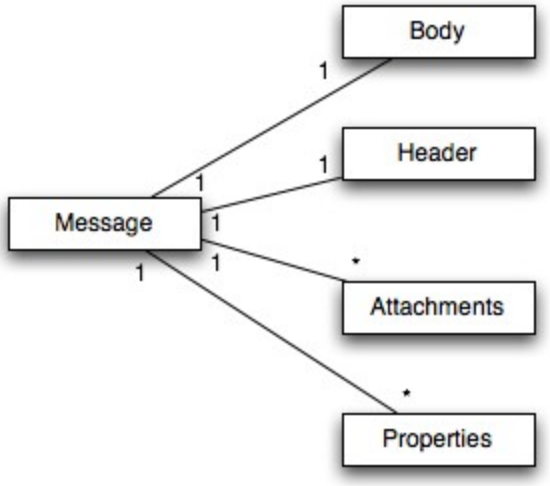
Therefore, central to the ESB is the notion of a *message*, whose structure is similar to that found in SOAP:

```
<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:



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:

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

> Note: 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.

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.

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

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

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`.

Note: 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);
}
```

> Note: 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();
}
```

> 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.

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

  Object remove(String name);

  String[] getNames();

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

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

  public int getNamedCount();
}
```

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 specific 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.

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

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:

1. "*get-payload-location*": The location from which to retrieve the message payload.
2. "*set-payload-location*": The location on which to set the message payload.

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).

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:

- `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.

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

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

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

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

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

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

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

  public void copy (Call from);
}
```

The properties below support both one way and request reply interaction patterns:

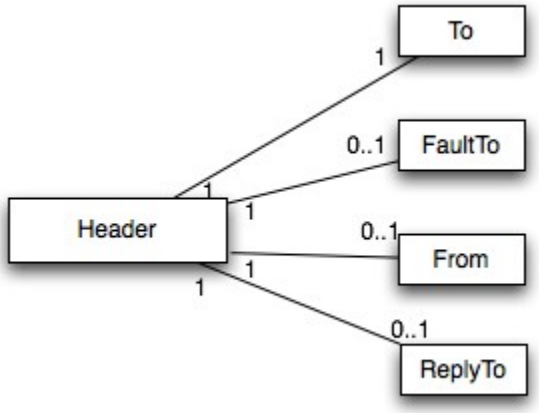
- **[To]** : EPR (mandatory). The address of the intended receiver of this message.
- **[From]** : endpoint reference (0..1). Reference of the endpoint where the message originated from.
- **[ReplyTo]** : endpoint reference (0..1). An endpoint reference that identifies the intended receiver for replies 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]** : endpoint reference (0..1). 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 (mandatory). An identifier that uniquely (and opaquely) identifies the semantics implied by this message.

- [**MessageID**] : URI (0..1). 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:



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**.

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.

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.

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**.

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.

- For JMS, it is assumed to be a queue with a name based on the one used to deliver the original request: <request queue name>_reply
- For JDBC, it is assumed to be 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.
- For files (both local and remote), 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.

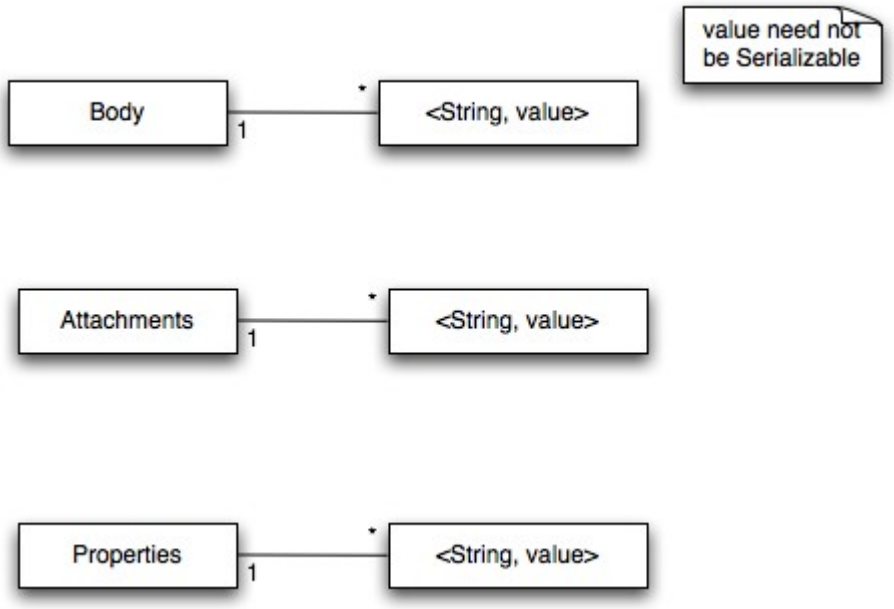
Note:Please see details on the **LogicalEPR**.

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.

> Note: 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`. Names must be unique on behalf of a given `Message` or an appropriate exception will be thrown. 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.

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.

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:

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

  public static MessageFactory getInstance ();
}
```

Message serialization implementations are uniquely identified by 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`.

Note: 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 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.

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.

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***:

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

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

  public Object unmarshal (Element doc) throws UnmarshalException;

  public URI type ();
}
```

Note: 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.

Chapter 4

Building and Using Services

Listeners, Notifiers/Routers and Actions

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.

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"?>
<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):

1. action1. `MyJMSListenerAction` (a trivial example follows)
2. notificationAction. An `org.jboss.soa.esb.actions.SystemPrintln`

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

```
public class MyJMSListenerAction
{
    ConfigTree _config;


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

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

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

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

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

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

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

2. Analyze contents of the 'process' attribute.

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

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

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

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

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

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

5. Callback method for success in each <action> element: If the list of methods in the 'process' attribute was executed successfully, the pipeline will analyze contents of the 'okMethod' attribute. If none is specified, processing will continue with the next <action> element. If a method name is provided in the 'okMethod' attribute, it will be invoked using the Message returned by the last

method in step 3. If the pipeline succeeds then the okMethod notification will be called on all handlers from the last one back to the initial one.

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

Action classes supplied by users to tailor 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.

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.

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.

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'.

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.

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 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` and `org.jboss.internal.soa.message.filter.GatewayFilter` which add the following meta-data to the `Message` as `Properties` with the indicated property names and the returned String values. See the Adapter Guide for more information about Gateways.

Message Property Name	Value

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

Name	Value
`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 4.3 GA.

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.

ServiceInvoker

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

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

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

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

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

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

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

> Note: It is possible that multiple EPRs may be present in the Registry with the same Service Name/Category and that some of these EPRs may not be ESB-aware, e.g., Gateways. If the ServiceInvoker receives ESB-unaware EPRs from the Registry then it will ignore them. You may see the warning: “Invalid EPR for service (probably ESB-unaware)”.

Services and ServiceInvoker

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

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

InVM Transport

Every ESB Service supports an InVM transport, which means that the Service can be invoked (via ServiceInvoker) from within that VM with minimal overhead i.e. without incurring any networking or message serialization overhead.

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

```
<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.

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.

The following is an example of how to configure a Service to be un-invokable over the InVM transport:

```
<service category="ServiceCat" name="ServiceName" description="Test Service" invmScope="NONE">
    <listeners>
        .....
    </listeners>
    <actions>
        .....
    </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".

Lock-step Delivery

Because the InVM Transport delivers messages with low overhead (to an in-memory message queue), it can result in situations where message delivery is happening to fast for the Service consuming the messages. For situations such as this, the InVM Transport supports a "Lock Step" delivery mechanism, whereby message delivery blocks until the receiving Service picks up the message. This ensures that message delivery is in "Lock Step" with consumption and so the receiving in-memory message queue is never overwhelmed.

Lock Step delivery should not be confused with a synchronous message delivery mechanism. The Lock Step delivery mechanism only blocks until the message is "picked up" i.e. it doesn't wait for a respose, or for the receiving Service to process the message.

Lock Step delivery is off by default, but can be easily configured on through addition of simple <property> settings on the <service>:

```
<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>
```

Chapter 5

Other Components

Introduction

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.

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.

Data Transformation

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

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

For more details see the Message Transformation Guide.

Content-based Routing

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

route `Messages` based on arbitrarily complex rules, which can be defined within XPath or Jboss Rules notation.

The Registry

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

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

Chapter 6

Example

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.

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.

> 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.

Operation	org.example.flight.opcode	org.example.flight.seatnumber	org.example.flight.flightnumber	org.example.flight.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.

The Service

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

```
class AirlineReservationSystem
{
```

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

Note: 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;
}
```

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.

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`):

```
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().ge
```

```
tCall().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.

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.

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.

Chapter 7

Advanced Topics

Introduction

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

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.

Services, EPRs, listeners and actions

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

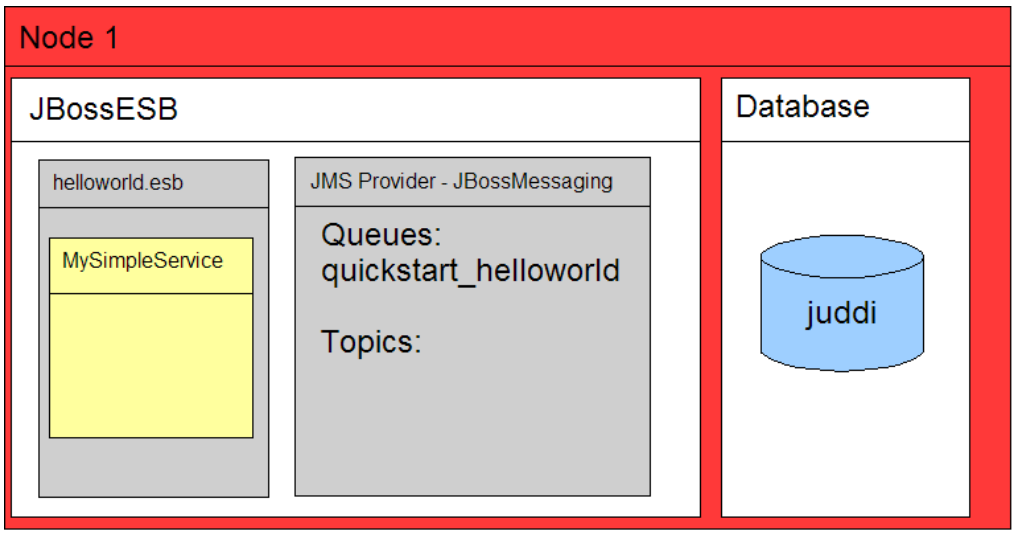


Figure 7-1: Hello World configuration fragment, one service instance on one node.

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

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.

Replicated Services

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

ServiceBinding to this service. So now we have 2 ServiceBindings for this Service. We now have our first replicated Service! If Node1 goes down, Node2 will keep on working.

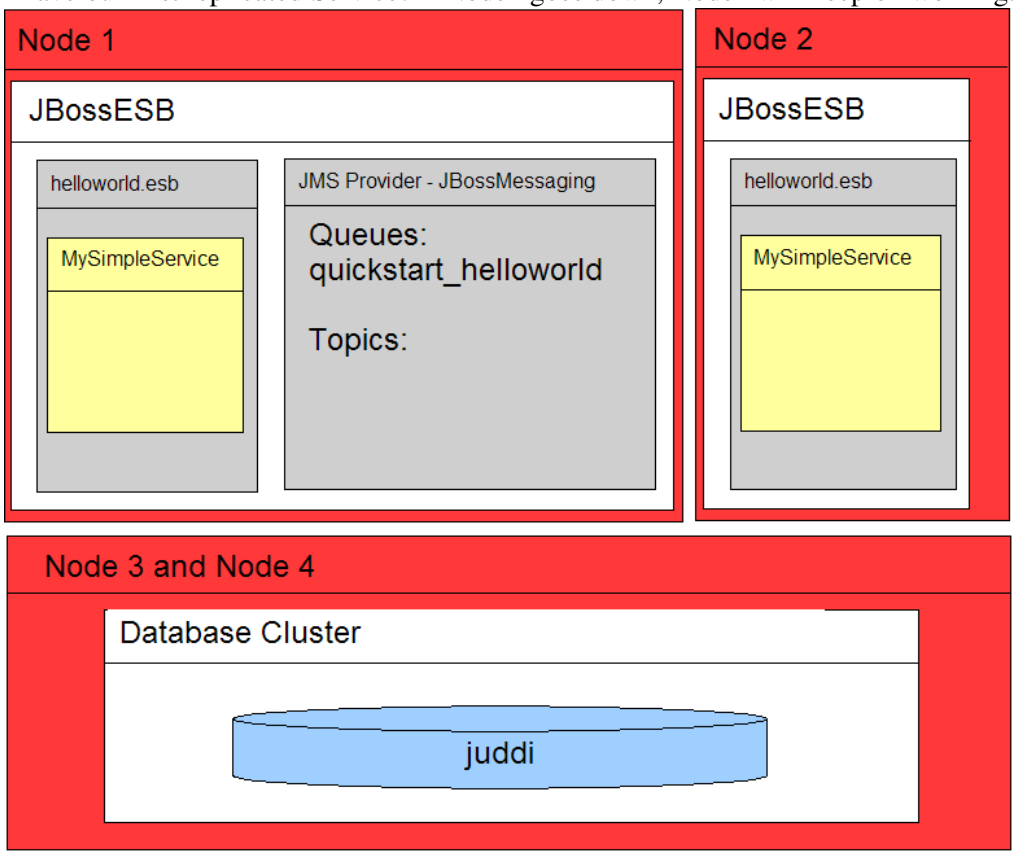
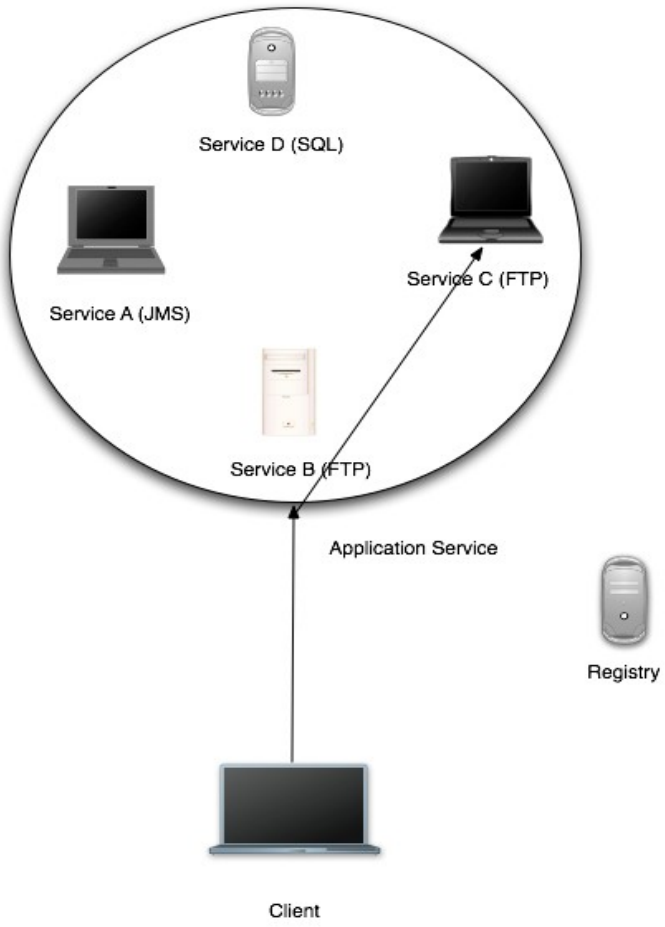


Figure 7-2: Two service instance each on a different node.

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.



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.

Protocol Clustering

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

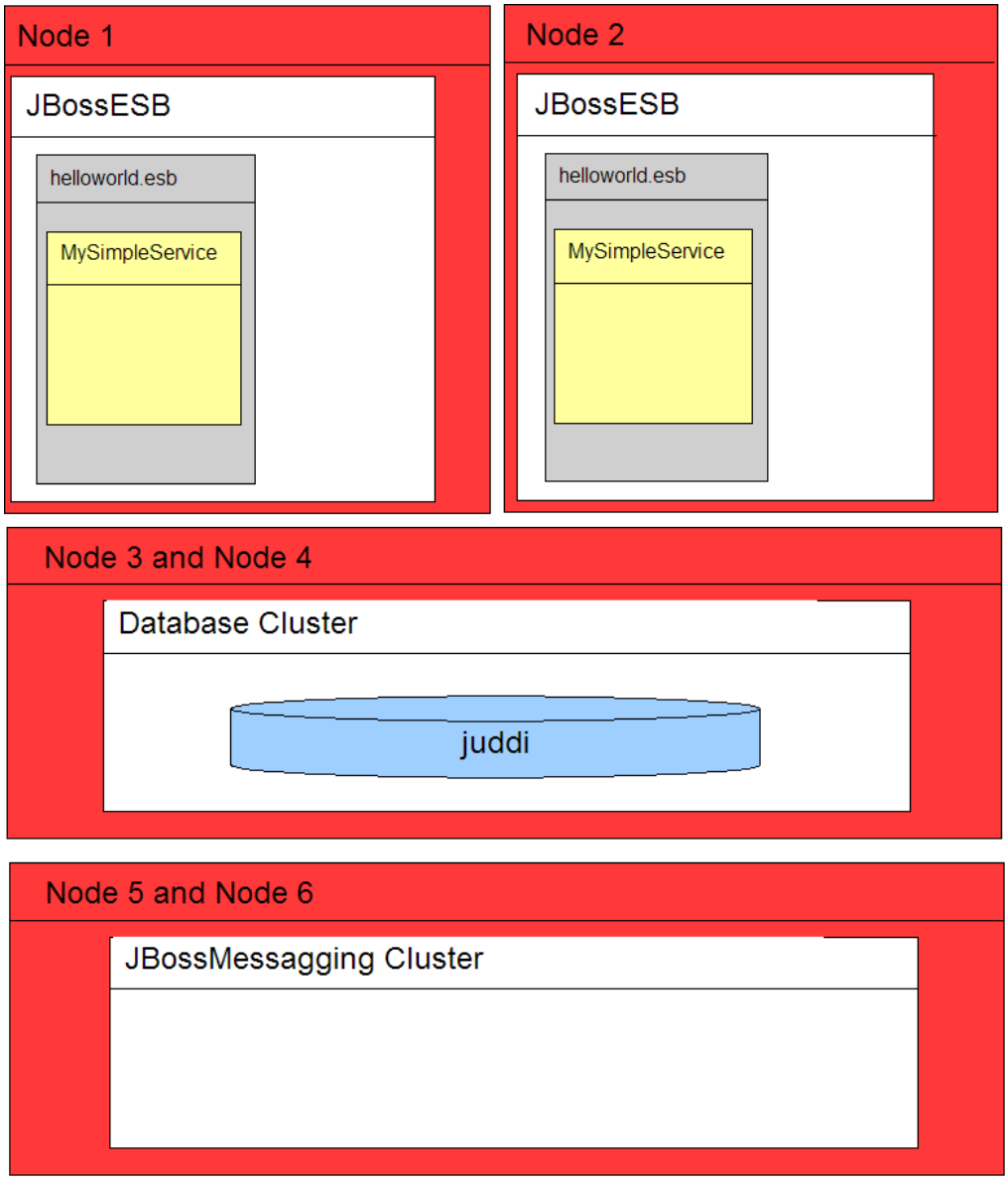
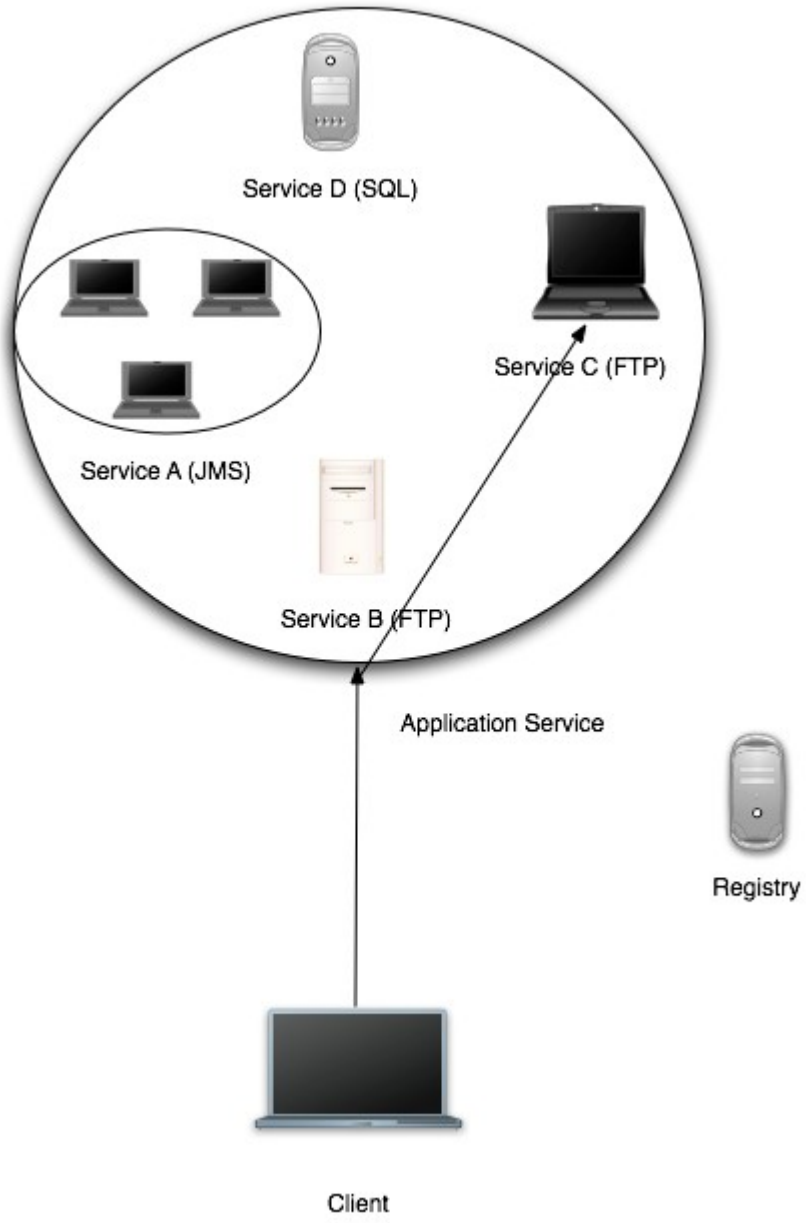


Figure 7-3: Protocol clustering: Here we cluster JMS.

Please read the documentation on Clustering for JBossMessaging if you want to enable JMS clustering. Both JBossESB replication and JMS clustering can be used together, as illustrated in the following figure. In this example, Service A is identified in the registry by a single JMSEpr. However, opaquely to the client, that JMSEpr points to a clustered JMS queue, which has been separately configured (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.



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.

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.

Channel Fail-over and Load Balancing

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

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

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

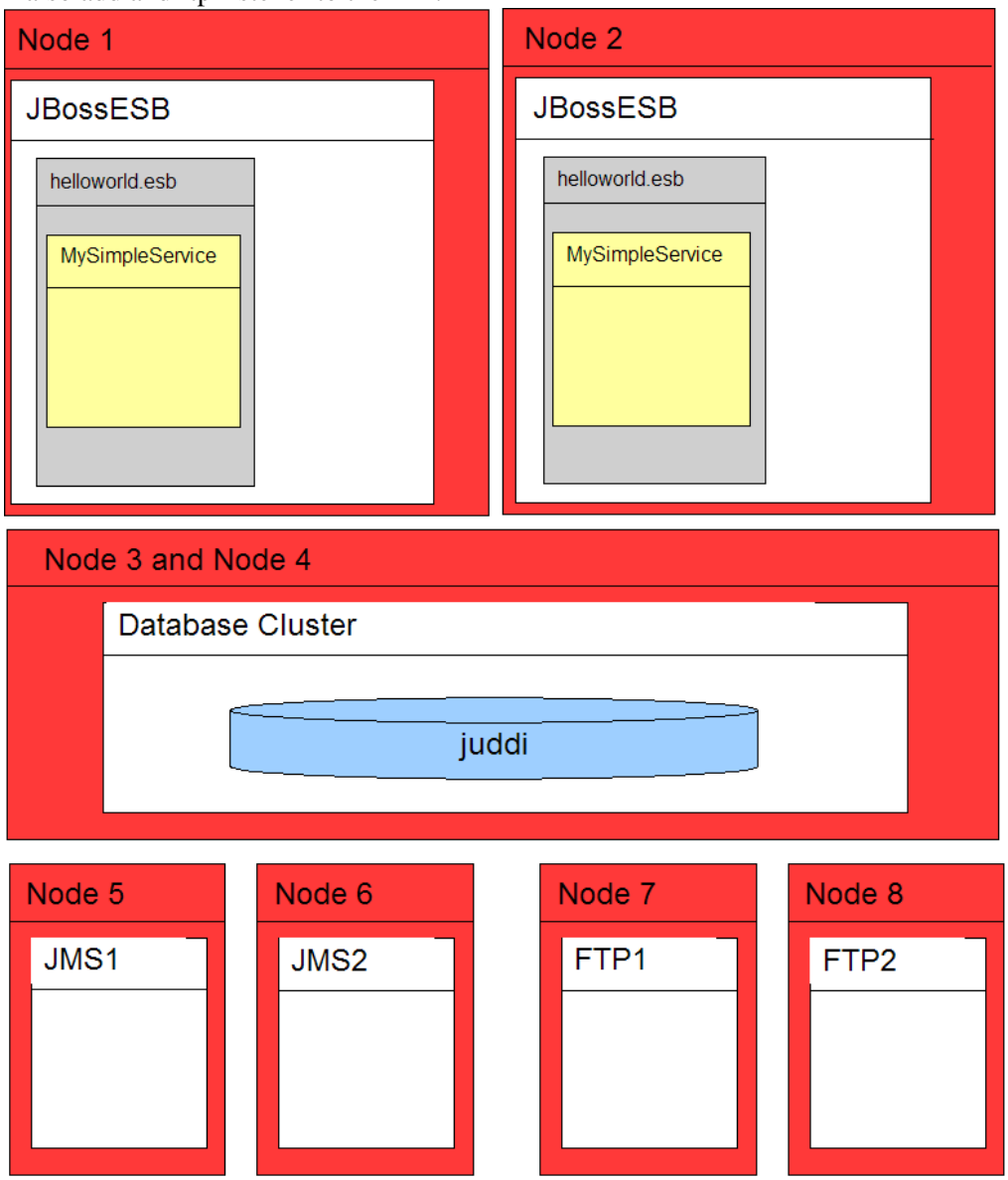


Figure 7-5: Adding an 2 FTP servers to the mix.

```
...
<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.

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.

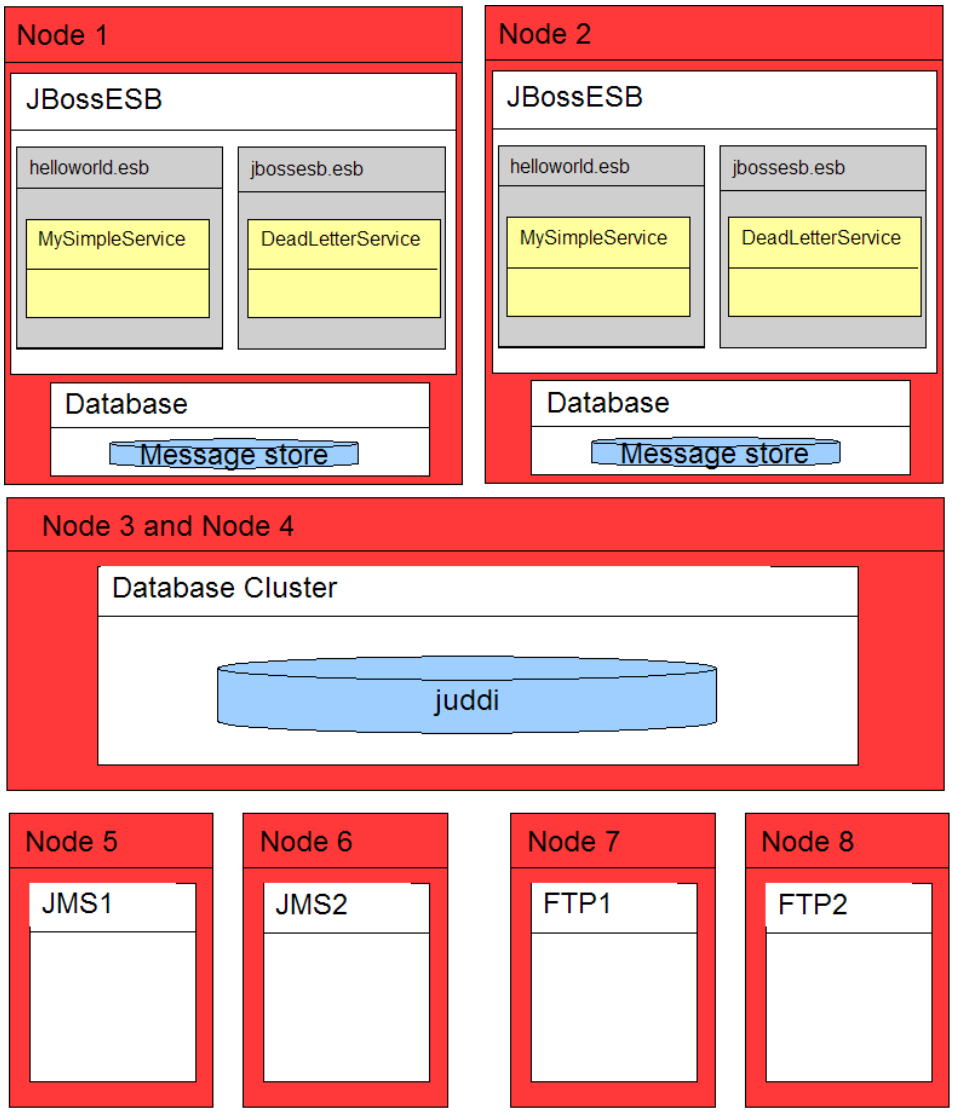


Figure 7-6. If all the EPRs are bad at a given moment, async requests can be store in the MessageStore for redelivery at a later time.

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.

Scheduling of Services

JBossESB 4.3 GA 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 4.3 GA 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.

Simple Schedule

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

1. "**scheduleid**": A unique identifier string for the schedule. Used to reference a schedule from a listener.
2. "**frequency**": The frequency (in seconds) with which all schedule listeners should be triggered.
3. "**execCount**": The number of times the schedule should be executed.
4. "**startDate**": The schedule start date and time. The format of this attribute value is that of the XML Schema type "dateTime". See dateTime.
5. "**endDate**": The schedule end date and time. The format of this attribute value is that of the XML Schema type "dateTime". See dateTime.

Example:

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

Cron Schedule

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

1. "**scheduleid**": A unique identifier string for the schedule. Used to reference a schedule from a listener.
2. "**cronExpression**": CRON expression.
3. "**startDate**": The schedule start date and time. The format of this attribute value is that of the XML Schema type "dateTime". See dateTime.
4. "**endDate**": The schedule end date and time. The format of this attribute value is that of the XML Schema type "dateTime". See dateTime.

Example:

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

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

Scheduled Listener

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

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

- ***ScheduledEventListener**:* Event Processors that implement this interface are simply triggered through the "*onSchedule*" method. No action processing pipeline is executed.
- ***ScheduledEventMessageComposer**:* Event Processors that implement this interface are capable of "composing" a message for the action processing pipeline associated with the listener.

The attributes of this listener are:

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

Example Configurations

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

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

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

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

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

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

```
</jbossesb>
```

Quartz Scheduler Property Configuration

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

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

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

org.quartz.jobStore.misfireThreshold = 60000

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

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

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

Chapter 8

Fault-tolerance and Reliability

Introduction

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.

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:

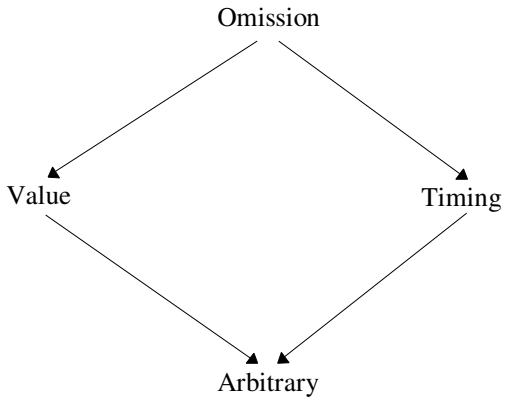
- *Omission fault/failure*: a component that does not respond to an input from another component, and thereby fails by not producing the expected output is exhibiting an *omission fault* and the corresponding failure an *omission failure*. A communication link which occasionally loses messages is an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



Fault classification hierarchy.

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.

> 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.

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.

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.

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.

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*).

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.

Component Specifics

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

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.

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.

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.

Action Pipelining

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

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.

Chapter 9

Configuration

Overview

JBossESB 4.3 GA configuration is based on the jbossesb-1.0.1 XSD. This XSD is always the definitive reference for the ESB configuration

The model has 2 main sections:

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 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.

Providers

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

- **Bus Providers**: These specify provider details for "Event Driven" providers i.e. for lsiteners that are "pushed" messages. Examples of this provider type would be the <jms-provider>.
- **Schedule Provider**: Provider configurations for schedule driven listeners i.e. listeners that "pull" messages.

A Bus Provider (e.g. <jms-provider>) can contain multiple <bus> definitions. The <provider> can also be decorated with <property> instances relating to provider specific properties that are common across all <bus> instances defined on that <provider> (e.g. for JMS - "connection-factory", "jndi-context-factory" etc). Likewise, each <bus> instance can be decorated with <property> instances specific to that <bus> instance (e.g. for JMS - "destination-type", "destination-name" etc).

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

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


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

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

Services

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

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

Service Attributes (<service>)

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

The "base" <listener> defines the following attribute. These attribute definitions are inherited by all <listener> extensions.

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 #5.*	xsd:boolean	true

Listener Attributes (<listener>)

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

> Note: For each gateway listener defined in a service, an ESB aware listener (or "native") listener must also be defined as gateway listeners do not define bidirectional endpoints, but rather "startpoints" into the ESB. From within the ESB you cannot send a message to a Gateway. Also, note that since a gateway is not an endpoints, 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>. The actions are effectively the "meat" of the Service. <action>s typically contain the logic for processing the payload of the messages received by the service (through it's listeners). Alternatively, it may contain the transformation or routing logic for messages to be consumed by an external Service/entity.

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

Name	Description	Type	Required
name	The name of the action. This attribute is required primarily for logging purposes.	xsd:string	true
class	The *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 (their "process" method is called) in the order in which the <action> instances appear in the <actions> set. The message returned from each <action> is used as the input message to the next <action> in the list.

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

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

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

It is 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>
```

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-provder>.

2. Add the bus configurations to the new provider (e.g. <jms-bus>), assigning a unique **busid** attribute value.

3. Define your <services> as normal, adding transport specific listener configurations (e.g. <jms-listener> that reference (using **busrefid**) the new bus configurations you just made e.g. <jms-listener> referencing a <jms-bus>.

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

So the transport specific implementations that are in place in this release are:

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 <ftp-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> selements. 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 intercepted.

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 <sql-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, encompassing actions within a JTA transaction.

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

> Note: In order to list and describe the attributes for each transport specific type, you can use the 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.

JMS Message filter configuration

Property Name	Description	Comments
dest-type	The type of destination, either QUEUE or TOPIC	Mandatory.
dest-name	The name of the Queue or Topic	Mandatory.
selector	Allows multiple listeners to register with the same queue/topic, but they will filter on this message selector.	Optional.
persistent	Indicates if the delivery mode for JMS	Optional. Default is true

	should be persistent or not. True or false	
acknowledge-mode	The JMS Session acknowledge mode. Can be one of AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE	Optional. Default is AUTO_ACKNOWLEDGE
jms-security-principal	JMS destination user name. Will be used when creating a connection to the destination.	Optional.
jms-security-credential	JMS destination password. Will be used when creating a connection to the destination.	Optional.

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>
```

FTP configuration

Property Name	Description	Comments
hostname	Can be a combination of <host:port> of just <host> which will use port 21.	Mandatory.
username	Username that will be used for the ftp connection.	Mandatory.
password	Password for the above user	Mandatory.
directory	The ftp directory that is monitored for incoming new files	Mandatory.
input-suffix	The file suffix used to filter files targeted for comsumption 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.	Mandatory.
work-suffix	The file suffix used while the file is being process, so that another thread or process won't pick it up too.	Optional. Defaults to .esbInProcess.
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.	Optional. Defaults to true.

post-directory	The ftp directory to which the file will be moved after it is processed by the ESB	Optional. Defaults to the value of directory above.
post-suffix	The file suffix which will be added to the file name after it is processed.	Optional. Defaults to .esbDone.
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.	Optional. Defaults to true.
error-directory	The ftp directory to which the file will be moved after when an error occurs during processing.	Optional. Defaults to the value of directory above.
error-suffix	The file suffix which will be added to the file name after an error occurs during processing.	Optional. Defaults to .esbError.
protocol	The protocol, can be on of: ● sftp (SSH File Transfer Protocol) ● ftps (FTP over SLL) ● ftp (default).	Optional. Defaults to ftp.
passive	Indicates that the ftp connection is in passive. Setting this to true means the ftp client will establish two connection to the ftpserver client.	Optional. Defaults to false, meaning that the client will tell the ftpserver which port the ftpserver should connect to . The ftpserver then estabilshes the connection to the client.
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.	Optional. Defaults to false. See section “Read-only FTP Listener for more information

FTP Listener configuration

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

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.

Please note that 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.

Read-only FTP Listener Configuration

Property Name	Description	Comments
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.`	Optional. Defaults to `org.jboss.soa.esb.listeners.gateway.remotestrategies.ReadOnlyRemoteFileSystemStrategy`
remoteFilesystemStrategy-configFile	Specifiy a JBoss TreeCache configuration file on the local file system or one that exists on the classpath.	Optional. 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.	Optional. Default is no TreeCacheListener.

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.DeleteOnEvictTreeCach
eListener"/>

</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>
```

Property Name	Description	Comments
maxNodes	The maximum number of files that will be	0 denotes no limit

	stored in the cache.	
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).

Transitioning From The Old Configuration Model

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

The old configuration model used a free form (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.

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.

Index