JBoss Enterprise SOA Platform 5

ESB Services Guide

Your guide to services available on the JBoss Enterprise SOA Platform

JBoss Enterprise SOA Platform 5 ESB Services Guide
Your guide to services available on the JBoss Enterprise SOA Platform
Edition 1.3

Copyright © 2010 Red Hat, Inc.. This material may only be distributed subject to the terms and conditions set forth in the Open Publication License, V1.0, (the latest version is presently available at *http://www.opencontent.org/openpub/*).

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

All other trademarks are the property of their respective owners.

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

This book contains details of the services available within the JBoss SOA Platform.

Preface **v**
1. Document Conventions v
1.1. Typographic Conventions v
1.2. Pull-quote Conventions vi
1.3. Notes and Warnings vii
2. We Need Feedback! vii

1. The Registry **1**
1.1. What is the Registry? 1
1.1.1. Introduction 1
1.1.2. Why Does One Need It? 1
1.1.3. How Does One Use It? 1
1.1.4. Registries versus Repositories 2
1.1.5. Service-Oriented Architecture Components 2
1.1.6. Universal Description, Discovery and Integration Registry 3
1.1.7. The Registry and the JBoss Service-Oriented Architecture Platform 3
1.2. Configuring the Registry 4
1.2.1. *new section* The Components Involved 6
1.2.2. The Registry Implementation Class 7
1.2.3. *updated* Using JAXR 7
1.2.4. Using jUDDI Transports 8
1.2.5. *new* Using Scout and jUDDI 9
1.3. Registry Configuration Examples 10
1.3.1. Embedding Components 10
1.3.2. Remote Method Invocation Using the jbossesb.sar File 11
1.3.3. Remote Method Invocation Using One's Own JNDI Registration of the RMI Service 13
1.3.4. SOAP 16
1.4. *Updated* Registry Troubleshooting 17
1.4.1. More Information 17

2. Rule Services **19**
2.1. *Updated* What is a Rule Service? 19
2.1.1. Introduction 19
2.2. *Updated* Rule Services Using JBoss Rules 20
2.2.1. Introduction 20
2.2.2. Rule-Set Creation 20
2.2.3. Rule Service Consumers 21
2.2.4. Configuration 22
2.2.5. Object Paths 25
2.2.6. Deploying and Packaging 26

3. Content-based Routing **29**
3.1. What is Content-Based Routing? 29
3.1.1. Introduction 29
3.1.2. Simple Example 29
3.1.3. Content-Based Routing using XPath 30
3.1.4. Content-Based Routing using Regex 32
3.2. Content-Based Routing Using JBoss Rules 33
3.2.1. Introduction 33
3.2.2. Three Different Routing Action Classes 34
3.2.3. Rule-Set Creation 34
3.2.4. XPath Domain Specific Language 35

4. *updated* jBPM Integration **43**
4.1. Integration Configuration 43
4.2. Configuring the jBPM 45
4.3. Creating and Deploying a Process Definition 46
4.4. From the Enterprise Service Bus to the jBPM 49
4.4.1. ESB to jBPM Exception Handling 52
4.5. jBPM-to-JBoss ESB 52
4.5.1. ESBNotifier 52
4.5.2. ESB Action Handler 54
4.5.3. jBPM-to-ESB Exception Handling 56
4.5.4. Scenerio One: Time-out 57
4.5.5. Scenerio Two: Exception Transition 58
4.5.6. Scenerio Three: Exception Decision 58

5. Service Orchestration **61**
5.1. Orchestrating Web Services 61
5.2. Orchestration Diagram 61
5.3. Process Deployment and "Instantiation" 68
5.4. Conclusion 70

6. Message Transformation **71**
6.1. Smooks 71
6.2. XSL Transformations 71

7. The Message Store **73**
7.1. Message Store Interface 73
7.2. Configuring the Message Store 74

8. *updated* Security **77**
8.1. Security Service Configuration 77
8.1.1. Configuring Security on Services 80
8.2. Authentication 81
8.2.1. Authentication Request 81
8.3. The JBoss Enterprise Service Bus Security Context 82
8.4. Security Context Propagation 82
8.5. Customising Security 82
8.6. Provided Log-in Modules 83
8.6.1. Certificate Log-in Module 83
8.6.2. Role Mapping 84
8.7. Password Encryption 84
8.7.1. Creating an Encrypted Password File 84
8.7.2. Security Service 85

A. Revision History **87**

Preface

1. Document Conventions

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

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

1.1. Typographic Conventions

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

`Mono-spaced Bold`

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

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

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

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

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

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

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

Proportional Bold

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

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

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

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

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

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

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

> To connect to a remote machine using ssh, type **`ssh`** ***`username@domain.name`*** at a shell prompt. If the remote machine is **`example.com`** and your username on that machine is john, type **`ssh john@example.com`**.
>
> The **`mount -o remount`** ***`file-system`*** command remounts the named file system. For example, to remount the **`/home`** file system, the command is **`mount -o remount /home`**.
>
> To see the version of a currently installed package, use the **`rpm -q`** ***`package`*** command. It will return a result as follows: ***`package-version-release`***.

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

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

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **`mono-spaced roman`** and presented thus:

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

Source-code listings are also set in **`mono-spaced roman`** but add syntax highlighting as follows:

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

```
import javax.naming.InitialContext;

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

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

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

1.3. Notes and Warnings

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

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

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

Warning

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

2. We Need Feedback!

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

When submitting a bug report, be sure to mention the manual's identifier: *ESB_Services_Guide*

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

The Registry

1.1. What is the Registry?

1.1.1. Introduction

Read this section to learn both some general theory about SOA Platform registries and also some specific information about JBoss' implementation.

In the context of a Service Oriented Architecture, a registry provides applications and businesses with a central point within which information about services can be stored. A registry is expected to provide both the same level of information and the same breadth of services as a conventional "marketplace." Ideally, a registry should also facilitate the automatic discovery and execution of electronic commerce to take place by providing a dynamic environment for business transactions. Therefore, a registry is more than a mere “e. business directory”. It is a fundamental component of a Service Oriented Architecture's infrastructure.

1.1.2. Why Does One Need It?

It is easy to discover and manage business partners and interface with them on a small scale using either manual or ad hoc techniques. However, this approach does not scale well when the number of services and frequency of interactions increase and the physical distribution of the environment expands. A registry provides a solution based upon agreed standards by providing a common, ubiquitous way to discover and "publish" services. It offers a central place in which one can query whether or not a partner has a service that is compatible with in-house technologies. It also allows one to find a list of companies that, for instance, support shipping services on the other side of the globe.

Hence, service registries are central to service-oriented architectures. At the time of execution, they act as contact points at which service requests can be correlated with actual behaviors. A service registry will hold meta-data entries for all of the *artifacts* within the Service Oriented Architecture that are used at both run-time and design time.

Items held within a service registry may include service description *artifacts* such as *Service Policy* descriptions, various *Extensible Mark-Up Language* (XML) schema used by services, artifacts representing different versions of services, governance and security artifacts (such as certificates and audit trail data) and so forth. During the design phase, business process architects may use the Registry to link calls to several different services together. In doing so, they create a work-flow or business process.

Note

Tip: replicate or federate the registry in order to improve performance and reliability. This will prevent it from being a single point of failure.

1.1.3. How Does One Use It?

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

1.1.4. Registries versus Repositories

The purpose of the registry to record services, discover meta-data and classify entities into pre-defined categories. Unlike a repository, it does not have the ability to store business process definitions, WSDLs or any other documents required for trade agreements. A registry is essentially a *catalogue* of items, whereas a repository is the storage area that actually contains those items.

1.1.5. Service-Oriented Architecture Components

"A Service Oriented Architecture is a specific type of distributed system in which the agents are 'services'."[1].

The key components of a Service-Oriented Architecture are:

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

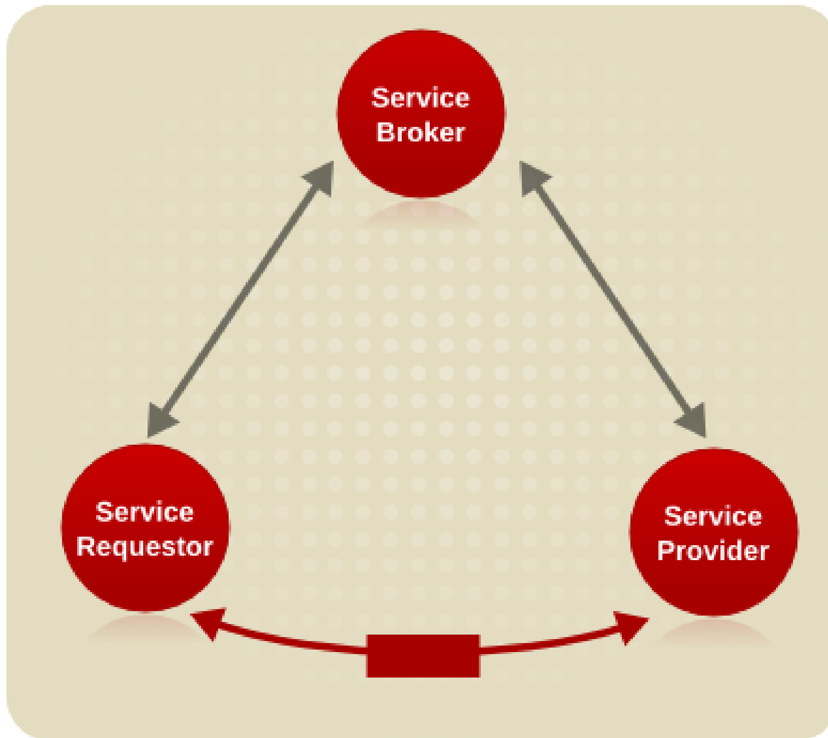
A "service" is essentially the messages exchanged between the system and its users. Within an Service Oriented Architecture, there are three critical roles: the service provider, the broker and the requester. Each shall now be defined in turn.

Service Provider
: A *service provider* facilitates access to services, creates descriptions of them and publishes them to the service broker.

Service Broker
: A *service broker* hosts a registry of service descriptions. It is responsible for linking a service requester to a service provider.

Service Requester
: A *service requester* is responsible for discovering a service. It does so by searching through the service descriptions given by the service broker. A requester is also responsible for binding to services obtained from the service provider.

Refer to the W3C Working Draft on *http://www.w3.org/TR/2003/WD-ws-arch-20030808/#id2617708* for a more detailed definition.



1.1.6. Universal Description, Discovery and Integration Registry

The *Universal Description, Discovery and Integration* (UDDI) Registry is a directory for *web services*. Use it to discover services through queries at design- or run-time. It also allows providers to publish descriptions of their services. The typical UDDI Registry will contain a *uniform resource locator* (URL) that points to both the WSDL document for the web services and the contact information for the service provider. Within the UDDI Registry, information is categorised in the following ways:

- *White Pages* contain general information, such as the name, address and other contact details for the company providing the service.
- *Yellow Pages* are used to categorize businesses based upon the industries to which they belong.
- *Green Pages* provide information that will enable a client to bind to the service that is being provided.

1.1.7. The Registry and the JBoss Service-Oriented Architecture Platform

The registry plays a central role within the **JBoss Enterprise Service-Oriented Architecture Platform**. It is used to store the *End Point References* (EPRs) for the services that have been deployed. It may either be updated dynamically (when services first start) or statically (by an external administrator.)

The registry cannot determine the status of those entities represented by the data it contains. Hence, an end-point reference might be in the Registry but there can be no guarantee that it is valid (as it may be malformed or it may represent a service that is no longer active.)

The **JBoss Enterprise SOA Platform** does not currently perform life-cycle monitoring of deployed services. The administrator must explicitly update or remove end-point references associated with

services that have been moved elsewhere or have failed, otherwise they will simply remain in the Registry.

Upon receipt of any warning or error messages from the Registry related to end-point references, one should inform those responsible for the services with which they are associated.

Important

ESB services create their own end-point references automatically. These end-points are internal implementations and, hence, modification of them is not supported.

1.2. Configuring the Registry

Read this section to learn how to configure the JBoss Enterprise SOA Platform Registry.

The default configuration uses Apache jUDDI v3 as its UDDI registry and Apache Scout (a JAXR implementation). *Figure 1.1, "Blueprint of the Registry Component Architecture"* shows an overview of all the components.

The `registry` is highly configurable. The **JBoss Enterprise Service Bus** directs all interaction with the registry through the Registry Interface. The default configuration of the Registry Interface uses the **Apache Scout** (a JAXR implementation) to communicate with the **jUDDI** registry.

Edit the registry section of the file ***$SOA_ROOT*/server/*$PROFILE*/deployers/esb.deployer/jbossesb-properties.xml** to configure the registry's properties.

```
<properties name="registry">
 <property name="org.jboss.soa.esb.registry.implementationClass"
 value="org.jboss.internal.soa.esb.services.registry.JAXRRegistryImpl"/>

 <property name="org.jboss.soa.esb.registry.factoryClass"
 value="org.apache.ws.scout.registry.ConnectionFactoryImpl"/>

    <property name="org.jboss.soa.esb.registry.queryManagerURI"
 value="org.apache.juddi.v3.client.transport.wrapper.UDDIInquiryService#inquire"/>

    <property name="org.jboss.soa.esb.registry.lifeCycleManagerURI"
 value="org.apache.juddi.v3.client.transport.wrapper.UDDIPublicationService#publish"/>
 <property name="org.jboss.soa.esb.registry.securityManagerURI"
 value="org.apache.juddi.v3.client.transport.wrapper.UDDISecurityService#secure"/>

    <property name="org.jboss.soa.esb.registry.user"  value="root"/>
    <property name="org.jboss.soa.esb.registry.password" value="root"/>

       <property name="org.jboss.soa.esb.scout.proxy.uddiVersion" value="3.0"/>
       <property name="org.jboss.soa.esb.scout.proxy.uddiNameSpace" value="urn:uddi-
org:api_v3"/>

    <property name="org.jboss.soa.esb.scout.proxy.transportClass"
 value="org.apache.ws.scout.transport.LocalTransport"/>
       <!-- specify the interceptors, in order -->
       <property name="org.jboss.soa.esb.registry.interceptors"
               value="org.jboss.internal.soa.esb.services.registry.InVMRegistryInterceptor,
 org.jboss.internal.soa.esb.services.registry.CachingRegistryInterceptor"/>
       <!-- The following properties modify the cache interceptor behaviour -->
       <property name="org.jboss.soa.esb.registry.cache.maxSize" value="100"/>
       <property name="org.jboss.soa.esb.registry.cache.validityPeriod" value="600000"/>
```

```
      <!-- Organization Category to be used by this deployment. -->
      <property name="org.jboss.soa.esb.registry.orgCategory"
              value="org.jboss.soa.esb.:category"/>
</properties>
```

These are the properties that can be configured:

Property	Description
`org.jboss.soa.esb.registry.implementationClass`	A class that implements the JBoss ESB Registry interface. One implementation, **`JAXRRegistryImpl`**, that uses the `JAXRRegistry` interface is included.
`org.jboss.soa.esb.registry.factoryClass`	The class name of the JAXR *ConnectionFactory* implementation.
`org.jboss.soa.esb.registry.queryManagerURI`	The URI that JAXR uses to query services.
`org.jboss.soa.esb.registry.lifeCycleManagerURI`	The URI that JAXR uses for editing.
`org.jboss.soa.esb.registry.user`	The user-name utilised for editing.
`org.jboss.soa.esb.registry.password`	The password for the specified user.
`org.jboss.soa.esb.scout.proxy.transportClass`	The class used by Apache Scout to transport from itself to the UDDI Registry.
`org.jboss.soa.esb.registry.interceptors`	The list of *interceptors* that are applied to the configured registry. The ESB provides two interceptors, one for handling InVM registrations and one that is used to apply a cache to the registry. The default interceptor list only contains one entry, the InVM interceptor.
`org.jboss.soa.esb.registry.cache.maxSize`	The maximum number of server entries allowed in the cache. If this value is exceeded, entries will be removed on a "Least Recently Used" basis. The default value is **`100`**.
`org.jboss.soa.esb.registry.cache.validityPeriod`	The period of validity for the caching interceptor. This is specified in milliseconds and defaults to **`600000`** (ten minutes).

Property	Description
	Set this value to **0** to have no cache expiry.
org.jboss.soa.esb.registry.orgCategory	This is the ESB instance's `organizational category` name. The default setting is **org.jboss.soa.esb.:category**.

Table 1.1. Registry Properties

1.2.1. *new section* The Components Involved

The `registry` can be configured in many ways. The image below is a blue-print of all of the `registry`'s components. From the top down, one can see that:

1. The **JBoss Enterprise Service Bus** funnels all interaction with the `registry` through the `registry interface`
2. It then calls into a JAXR implementation of this interface.
3. The JAXR API needs an implementation which, by default, is **Scout**.
4. The **Scout** JAXR implementation, in turn, calls into a **jUDDI** registry.

Note

Remember that these are just the defaults. There are many other configuration options.

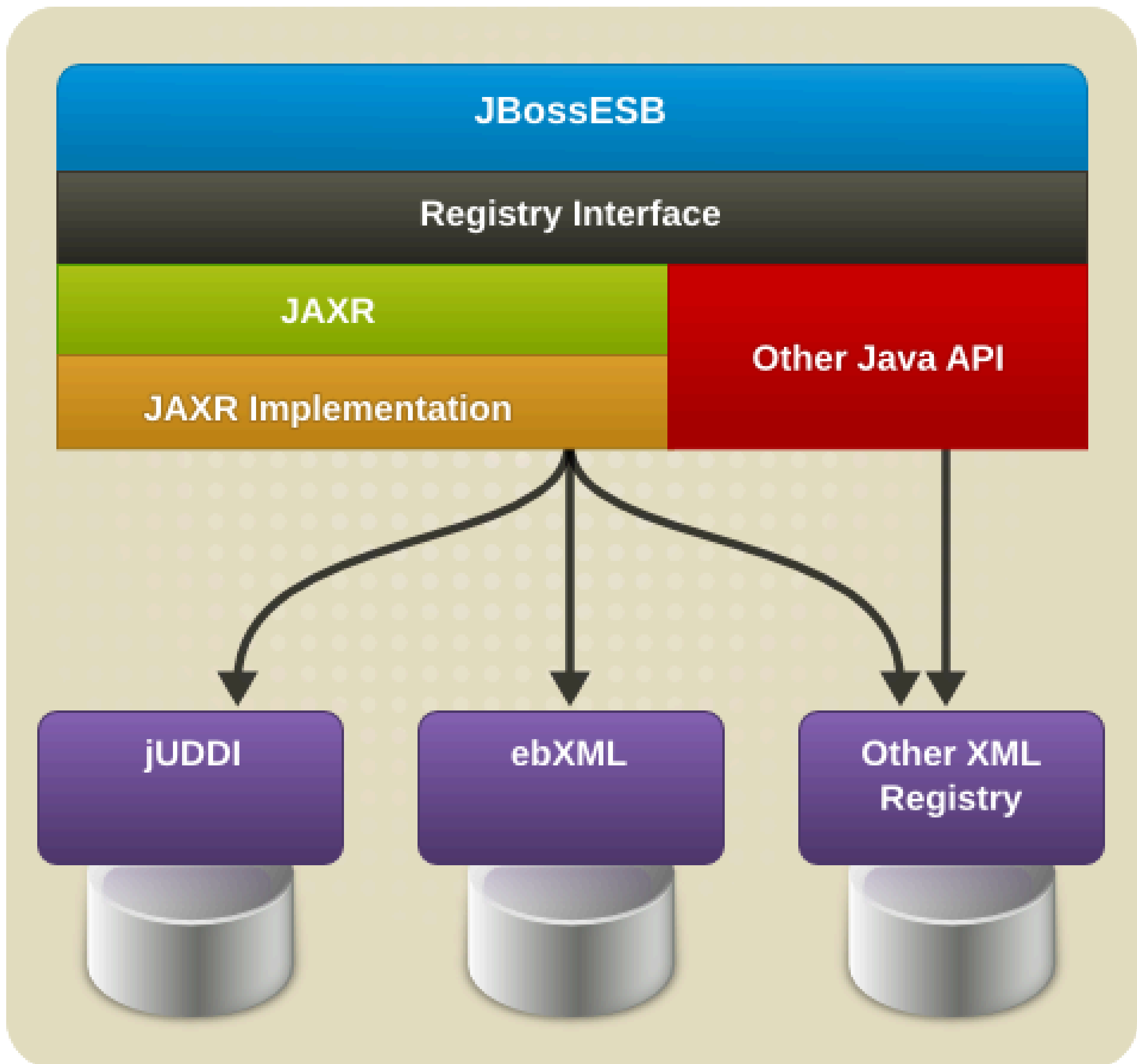


Figure 1.1. Blueprint of the Registry Component Architecture

1.2.2. The Registry Implementation Class

`org.jboss.soa.esb.registry.implementationClass`

By default, this class uses the JAXR application programming interface. This API is convenient since it allows one to connect any kind of XML-based registry or repository. However, if one wishes to use an alternative API, do so by writing a new **`SystinetRegistryImplemtation`** class and provide a reference to it within this property.

1.2.3. *updated* Using JAXR

`org.jboss.soa.esb.registry.factoryClass`

1. Firstly, choose a specific JAXR implementation. Then use this property to configure the class. The JBoss Enterprise SOA Platform uses **Scout** by default and, hence, as one would expect this property is set to the **Scout** factory class, namely **`org.apache.ws.scout.registry.ConnectionFactoryImpl`**.

2. Next, configure JAXR implementation with location of the registry that is to be used for querying and updating. This is done by editing the **org.jboss.soa.esb.registry.queryManagerURI**, **org.jboss.soa.esb.registry.lifeCycleManagerURI** and **org.jboss.soa.esb.registry.securityManagerURI**.

3. Set the user-name and password for the UDDI Registry by editing the **org.jboss.soa.esb.registry.user** and **org.jboss.soa.esb.registry.password** respectively.

1.2.4. Using jUDDI Transports

org.jboss.soa.esb.scout.proxy.transportClass

When using **Scout** with a UDDI implementation, one can set an additional parameter: the **transport** class that is to be used for communicating between **Scout** and the UDDI registry.

If one is using **Scout** to communicate with **jUDDI v. 3** leave the transport class as **LocalTransport** and configure the **esb.juddi.client.xml** file to make use **jUDDI**'s transports (InVM, RMI and WS). (**esb.juddi.client.xml** resides in the **server/config/deploy/jbossesb.sar/META-INF** directory.) This file defines the concept of a *node*, which is simply a **jUDDI** registry location. Use the node settings to select which transport to use:

```
<node>
     <!-- required 'default' node -->
     <name>default</name>
     <description>Main jUDDI node</description>
     <properties>
 <property name="serverName" value="localhost" />
  <property name="serverPort" value="8880" />
     </properties>
     <!-- JAX-WS Transport
     <proxyTransport>org.apache.juddi.v3.client.transport.JAXWSTransport</proxyTransport>
     <custodyTransferUrl>http://${serverName}:${serverPort}/juddiv3/services/custody-transfer?
wsdl</custodyTransferUrl>
    <inquiryUrl>http://${serverName}:${serverPort}/juddiv3/services/inquiry?wsdl</inquiryUrl>
    <publishUrl>http://${serverName}:${serverPort}/juddiv3/services/publish?wsdl</publishUrl>
    <securityUrl>http://${serverName}:${serverPort}/juddiv3/services/security?wsdl</
securityUrl>
    <subscriptionUrl>http://${serverName}:${serverPort}/juddiv3/services/subscription?wsdl</
subscriptionUrl>
    <subscriptionListenerUrl>http://${serverName}:${serverPort}/juddiv3/services/subscription-
listener?wsdl</subscriptionListenerUrl>
    <juddiApiUrl>http://${serverName}:${serverPort}/juddiv3/services/juddi-api?wsdl</
juddiApiUrl>
     -->
    <!-- In VM Transport Settings
    <proxyTransport>org.jboss.internal.soa.esb.registry.client.JuddiInVMTransport</
proxyTransport>
    <custodyTransferUrl>org.apache.juddi.api.impl.UDDICustodyTransferImpl</custodyTransferUrl>
     <inquiryUrl>org.apache.juddi.api.impl.UDDIInquiryImpl</inquiryUrl>
     <publishUrl>org.apache.juddi.api.impl.UDDIPublicationImpl</publishUrl>
    <securityUrl>org.apache.juddi.api.impl.UDDISecurityImpl</securityUrl>
    <subscriptionUrl>org.apache.juddi.api.impl.UDDISubscriptionImpl</subscriptionUrl>
    <subscriptionListenerUrl>org.apache.juddi.api.impl.UDDISubscriptionListenerImpl</
subscriptionListenerUrl>
    <juddiApiUrl>org.apache.juddi.api.impl.JUDDIApiImpl</juddiApiUrl>
  -->
 <!-- RMI Transport Settings -->
```

```
<proxyTransport>org.apache.juddi.v3.client.transport.RMITransport</proxyTransport>
 <custodyTransferUrl>/juddiv3/UDDICustodyTransferService</custodyTransferUrl>
 <inquiryUrl>/juddiv3/UDDIInquiryService</inquiryUrl>
 <publishUrl>/juddiv3/UDDIPublicationService</publishUrl>
<securityUrl>/juddiv3/UDDISecurityService</securityUrl>
<subscriptionUrl>/juddiv3/UDDISubscriptionService</subscriptionUrl>
<subscriptionListenerUrl>/juddiv3/UDDISubscriptionListenerService</subscriptionListenerUrl>
<juddiApiUrl>/juddiv3/JUDDIApiService</juddiApiUrl>
<javaNamingFactoryInitial>org.jnp.interfaces.NamingContextFactory</javaNamingFactoryInitial>
 <javaNamingFactoryUrlPkgs>org.jboss.naming</javaNamingFactoryUrlPkgs>
 <javaNamingProviderUrl>jnp://localhost:1099</javaNamingProviderUrl>
 </node>
```

A **`transport`** should specify:

- a **`proxyTransport`**
- a URL for all of the supported UDDI application programming interfaces (`inquiry`, `publish`, `security`, `subscription`, `subscription-listener` and `custodytransfer`)
- a jUDDI application programming interface URL.
- the RMI transport also includes JNDI settings

By default, the RMI settings are enabled. To switch transports, simply comment those ones out and enable whichever of the other transports is to be used.

1.2.5. *new* Using Scout and jUDDI

`org.jboss.soa.esb.scout.proxy.transportClass`

As noted above, when using **Scout** with **jUDDI**, once can set an additional parameter, this being the transport class that is to be used to communicate between the two pieces of software. Thus far, there are four implementations of this class, these being based upon SOAP, SAAJ, RMI and Embedded Java (Local) respectively. When communicating with**jUDDI**, leave the **`transportClass`** set to LocalTransport and use the **`uddi.xml`** file to utilise **jUDDI**'s transports (these being InVM, RMI and WS, respectively.)

However, when communicating with another UDDI registry, use **Scout**'s JAXR transports. There are four implementations of this class, these also being based on SOAP, SAAJ, RMI and Embedded Java (Local).

When changing the transport, always change the query and lifecycle URIs as well. Here is an example that shows how to do so:

```
SOAP
queryManagerURI http://localhost:8080/juddi/inquiry
lifeCycleManagerURI http://localhost:8080/juddi/publish
transportClass  org.apache.ws.scout.transport.AxisTransport

RMI
queryManagerURI  jnp://localhost:1099/InquiryService?
org.apache.juddi.registry.rmi.Inquiry#inquire
lifeCycleManagerURI  jnp://localhost:1099/PublishService?
org.apache.juddi.registry.rmi.Publish#publish
transportClass  org.apache.ws.scout.transport.RMITransport
```

```
Local
queryManagerURI org.apache.juddi.registry.local.InquiryService#inquire
lifeCycleManagerURI org.apache.juddi.registry.local.PublishService#publish
transportClass  org.apache.ws.scout.transport.LocalTransport
```

Two requirements must be fulfiled for **jUDDI**:

1. one must be able to access the **jUDDI** database. To achieve this, create a schema in the database and add the `jbossesb publisher`. (The **`product/install/jUDDI-registry`** directory contains **`database-create`** scripts for most common databases.)
2. **`esb.juddi.xml`** and **`esb.juddi.client.xml`** must exist. These contain the **jUDDI** configuration itself.

Note

The database can be generated automatically if the user account has been granted the right to create tables. **jUDDI** can create a database of any type for which there is an associated **Hibernate** dialect.

```
!-- <entry key="juddi.tablePrefix">JUDDI_</entry> -->
 <entry key="juddi.isCreateDatabase">true</entry>
 <entry key="juddi.databaseExistsSql">select * from ${prefix}BUSINESS_ENTITY
 </entry>
 <entry key="juddi.sqlFiles">
  juddi-sql/mysql/create_database.sql,
              juddi sql/mysql/insert_publishers.sql
 </entry>
```

Example 1.1. Configuring jUDDI to Generate a Database Automatically

The JBoss Enterprise SOA Platform includes a tool that automates **jUDDI** configuration. This tool is found in the ***`${SOA_ROOT}`*`/tools/schema/`** sub-directory. Directions for using it can be found in the "Switching Databases" section of the *Administration Guide*.

1.3. Registry Configuration Examples

Study the examples in this section to learn about different registry configuration options.

1.3.1. Embedding Components

All of those server components, (including the Enterprise Service Bus and Web Service) that have a relationship of any kind with the `registry` can share the latter between themselves. Indeed, multiple instances of the **JBoss Enterprise SOA Platform** can use the same `registry` via a shared database.

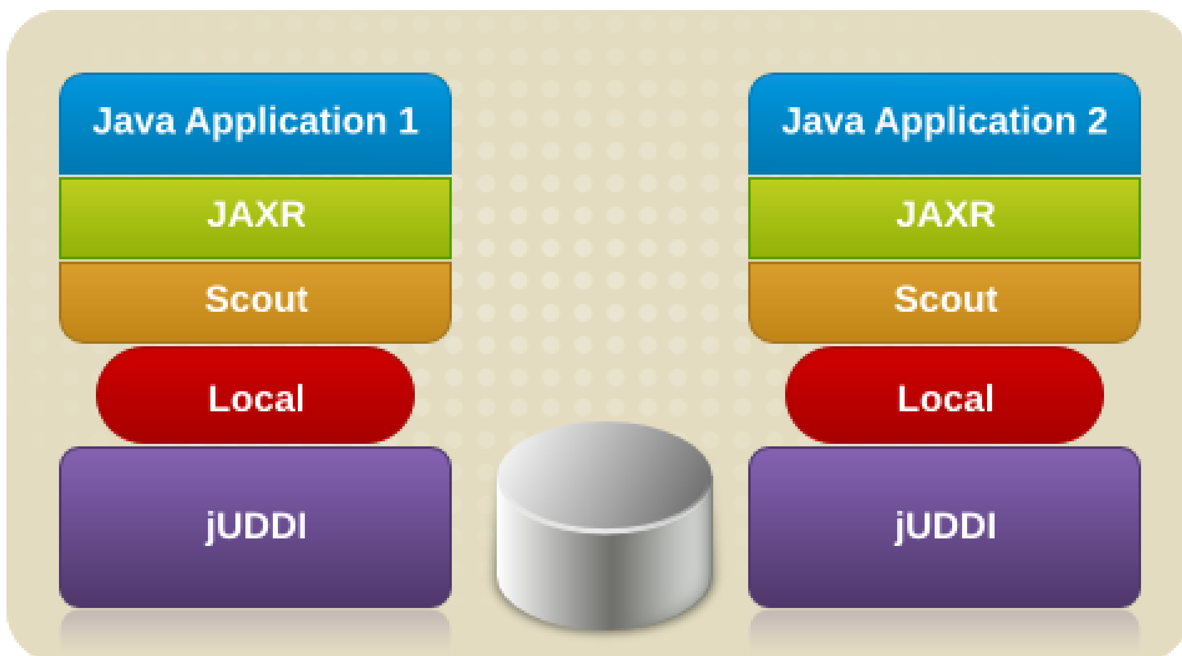


Figure 1.2. Embedded jUDDI

```
<properties name="registry">

    <property name="org.jboss.soa.esb.registry.implementationClass"
    value="org.jboss.internal.soa.esb.services.registry.JAXRRegistryImpl"/>

    <property name="org.jboss.soa.esb.registry.factoryClass"
    value="org.apache.ws.scout.registry.ConnectionFactoryImpl"/>

    <property name="org.jboss.soa.esb.registry.queryManagerURI"
    value="org.apache.juddi.registry.local.InquiryService#inquire"/>

    <property name="org.jboss.soa.esb.registry.lifeCycleManagerURI"
    value="org.apache.juddi.registry.local.PublishService#publish"/

    <property name="org.jboss.soa.esb.registry.securityManagerURI"
    value="org.apache.juddi.registry.local.SecurityService#secure"/>

    <property name="org.jboss.soa.esb.registry.user" value="jbossesb"/>
    <property name="org.jboss.soa.esb.registry.password" value="password"/>

    <property name="org.jboss.soa.esb.scout.proxy.transportClass"

    value="org.apache.ws.scout.transport.LocalTransport"/>

</properties>
```

Example 1.2. Properties for Embedded jUDDI

1.3.2. Remote Method Invocation Using the jbossesb.sar File

This is a straightfoward process. Simply deploy a version of the jUDDI Registry that brings up a Remote Method Invocation service. (The JBoss Enterprise Service Bus deploys the Remote Method Invocation service by default: it starts the registry within the **`jbossesb.sar`** archive. This same archive also registers an RMI service.)

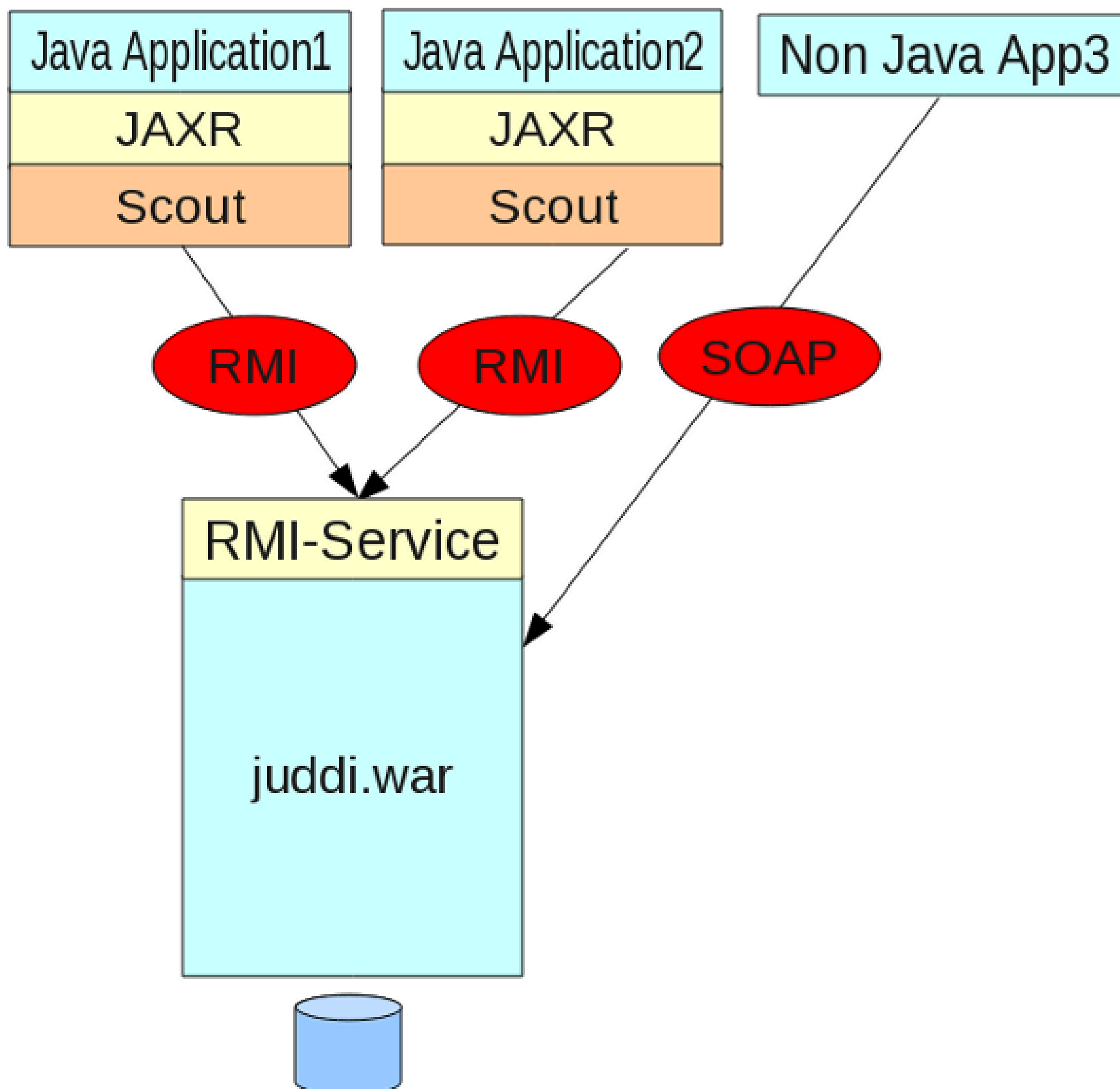


Figure 1.3. Remote Method Invocation

Here are the properties:

```
<properties name="registry">
 <property name="org.jboss.soa.esb.registry.implementationClass"
 value="org.jboss.internal.soa.esb.services.registry.JAXRRegistryImpl"/>

 <property name="org.jboss.soa.esb.registry.factoryClass"
 value="org.apache.ws.scout.registry.ConnectionFactoryImpl"/>

     <property name="org.jboss.soa.esb.registry.queryManagerURI"
value="jnp://localhost:1099/InquiryService?org.apache.juddi.registry.rmi.Inquiry#inquire"/>

     <property name="org.jboss.soa.esb.registry.lifeCycleManagerURI"
value="jnp://localhost:1099/PublishService?org.apache.juddi.registry.rmi.Publish#publish"/>

 <property name="org.jboss.soa.esb.registry.securityManagerURI"
value="jnp://localhost:1099/PublishService?org.apache.juddi.registry.rmi.Publish#publish"/>
>
```

```
    <property name="org.jboss.soa.esb.registry.user" value="jbossesb"/>
    <property name="org.jboss.soa.esb.registry.password" value="password"/>

    <property name="org.jboss.soa.esb.scout.proxy.transportClass"
 value="org.apache.ws.scout.transport.RMITransport"/>
</properties>
```

The **juddi.war** is configured to bring up a RMI Service, this being triggered by the following setting in the **web.xml** file:

```
<!--  uncomment if you want to enable making calls in juddi with rmi  -->
  <servlet>
   <servlet-name>RegisterServicesWithJNDI</servlet-name>
   <servlet-class>org.apache.juddi.registry.rmi.RegistrationService</servlet-class>
   <load-on-startup>1</load-on-startup>
  </servlet>
```

Include the following JNDI settings in **juddi.properties**:

```
# JNDI settings (used by RMITransport)
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming
```

Important

Remember to include **scout-client.jar** in the RMI client's class-path.

1.3.3. Remote Method Invocation Using One's Own JNDI Registration of the RMI Service

If, for some reason, one does not to deploy the **juddi.war**, simply configure one of the Enterprise Service components running in the same Java Virtual Machine as **jUDDI** to register the RMI service:

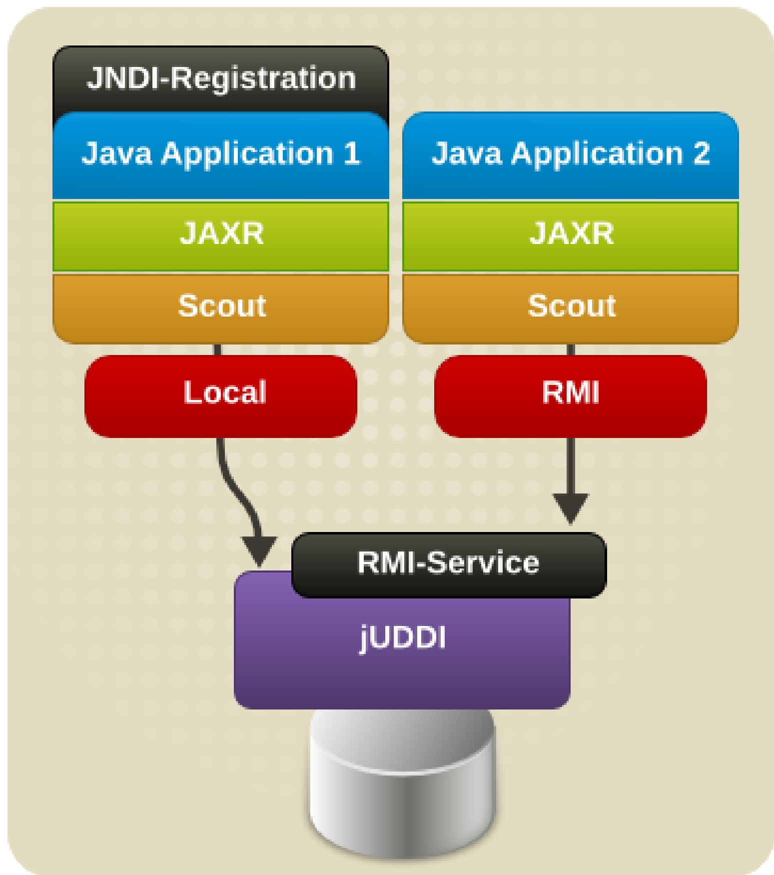


Figure 1.4. RMI Using One's Own JNDI Registration

For Application One, local settings are needed:

```
<properties name="registry">
 <property name="org.jboss.soa.esb.registry.implementationClass"
value="org.jboss.internal.soa.esb.services.registry.JAXRRegistryImpl"/>

 <property name="org.jboss.soa.esb.registry.factoryClass"
value="org.apache.ws.scout.registry.ConnectionFactoryImpl"/>

     <property name="org.jboss.soa.esb.registry.queryManagerURI"
value="org.apache.juddi.registry.local.InquiryService#inquire"/>

     <property name="org.jboss.soa.esb.registry.lifeCycleManagerURI"
value="org.apache.juddi.registry.local.PublishService#publish"/>

     <property name="org.jboss.soa.esb.registry.securityManagerURI"
value="org.apache.juddi.registry.local.SecurityService#secure"/>

     <property name="org.jboss.soa.esb.registry.user" value="jbossesb"/>
     <property name="org.jboss.soa.esb.registry.password" value="password"/>

     <property name="org.jboss.soa.esb.scout.proxy.transportClass"
value="org.apache.ws.scout.transport.LocalTransport"/>
</properties>
```

Example 1.3. Properties One

Application Two requires the Remote Method Invocation settings:

```
<properties name="registry">
 <property name="org.jboss.soa.esb.registry.implementationClass"
value="org.jboss.internal.soa.esb.services.registry.JAXRRegistryImpl"/>

 <property name="org.jboss.soa.esb.registry.factoryClass"
value="org.apache.ws.scout.registry.ConnectionFactoryImpl"/>

     <property name="org.jboss.soa.esb.registry.queryManagerURI"
value="jnp://localhost:1099/InquiryService?org.apache.juddi.registry.rmi.Inquiry#inquire"/>

     <property name="org.jboss.soa.esb.registry.lifeCycleManagerURI"
value="jnp://localhost:1099/PublishService?org.apache.juddi.registry.rmi.Publish#publish"/>

     <property name="org.jboss.soa.esb.registry.user" value="jbossesb"/>
     <property name="org.jboss.soa.esb.registry.password" value="password"/>

     <property name="org.jboss.soa.esb.scout.proxy.transportClass"
value="org.apache.ws.scout.transport.RMITransport"/>
</properties>
```

Example 1.4. Properties Two

Point the hostnames of the **queryManagerURI** and **lifeCycleManagerURI** classes to the host on which jUDDI is running (this is also where Application One is running.) Obviously, Application One needs to have access to a naming service. To register it, undertake the following process:

```
//Getting the JNDI setting from the config
Properties env = new Properties();
env.setProperty(RegistryEngine.PROPNAME_JAVA_NAMING_FACTORY_INITIAL,factoryInitial);
env.setProperty(RegistryEngine.PROPNAME_JAVA_NAMING_PROVIDER_URL, providerURL);
env.setProperty(RegistryEngine.PROPNAME_JAVA_NAMING_FACTORY_URL_PKGS, factoryURLPkgs);

InitialContext context = new InitialContext(env);
Inquiry inquiry = new InquiryService();
log.info("Setting " + INQUIRY_SERVICE + ", " + inquiry.getClass().getName());
mInquery = inquiry;
context.bind(INQUIRY_SERVICE, inquiry);
Publish publish = new PublishService();
log.info("Setting " + PUBLISH_SERVICE + ", " + publish.getClass().getName());
mPublish = publish;
context.bind(PUBLISH_SERVICE, publish);
```

Example 1.5. Registration Process

For example, make sure to include the following JNDI settings in the file **jbossesb-registry.sar/esb.juddi.xml**:

```
# JNDI settings (used by RMITransport)
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming
```

Example 1.6. JNDI Settings for **jbossesb-registry.sar/esb.juddi.xml**

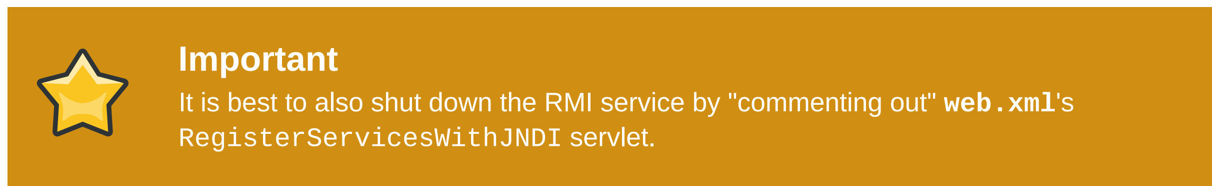
Important

Always include the **scout-client.jar** file in the classpath of the RMI clients.

1.3.4. SOAP

Read this section to learn how to configure **Apache Scout** to use SOAP to communicate with **jUDDI**.

Firstly, deploy the **`juddi.war`** and configure the data-source.



Important

It is best to also shut down the RMI service by "commenting out" **`web.xml`**'s `RegisterServicesWithJNDI` servlet.

Figure 1.5. SOAP-based Communications

```
<properties name="registry">
 <property name="org.jboss.soa.esb.registry.implementationClass"
 value="org.jboss.internal.soa.esb.services.registry.JAXRRegistryImpl"/>

 <property name="org.jboss.soa.esb.registry.factoryClass"
 value="org.apache.ws.scout.registry.ConnectionFactoryImpl"/>
```

```
    <property name="org.jboss.soa.esb.registry.queryManagerURI"
    value="http://localhost:8080/juddi/inquiry"/>

    <property name="org.jboss.soa.esb.registry.lifeCycleManagerURI"
    value="http://localhost:8080/juddi/publish"/>

    <property name="org.jboss.soa.esb.registry.user" value="jbossesb"/>
    <property name="org.jboss.soa.esb.registry.password" value="password"/>

    <property name="org.jboss.soa.esb.scout.proxy.transportClass"
    value="org.apache.ws.scout.transport.AxisTransport"/>
</properties>
```

Example 1.7. Sample Properties

1.4. *Updated* Registry Troubleshooting

- If using RMI, be sure to obtain the **juddi-client.jar**, (found in the **jUDDI** distribution.)
- Ensure that the **jbossesb-properties.xml** file is on the class-path and being read correctly. If not, the `registry` will try to instantiate classes using "null" as the name.
  - Make sure that **META-INF/esb.juddi.client.xml** specifies a valid transport.
  - Make sure that the **persistence.xml** file's settings are valid and that the chosen **Hibernate** dialect matches that for the database in use.
- Ensure that the **esb.juddi.xml** file is on the class-path. The **jUDDI** registry requires this so that it can configure itself.
- In the event that a service fails or otherwise fails to shut down cleanly, old entries may possibly "persist" in the registry. Remove these manually.

1.4.1. More Information

Learn more about troubleshooting the registry at these locations:

- The JBoss jUDDI Wiki: *http://www.jboss.org/community/docs/DOC-11217*
- The JBoss ESB User Forum: *http://www.jboss.com/index.html?module=bb&op=viewforum&f=246*.

Rule Services

2.1. Updated What is a Rule Service?

Study this section to learn about *rule services* and ways in which to utilize them. An understanding of the **JBoss Business Rules Management System** (BRMS) will aid the reader in understanding these types of services.

2.1.1. Introduction

As its name implies, the **JBoss Enterprise SOA Platform**'s rule service allows one to deploy *rules* that have been created in **JBoss Rules** as *services* on the ESB. This has two major benefits:

1. The amount of client code required to integrate the rules into one's application environment is dramatically reduced.
2. Rules can be accessed either as part of an *action chain* or from within an *orchestrated business process*.

Note

The **JBoss Business Rules Management System** is supported but one can also use a `rule engine` if need be.

Rule Services are supported by the **`BusinessRuleProcessor`** and the **`DroolsRuleService`** action classes, the latter of which implements the `RuleService` interface.

The **`BusinessRuleProcessor`** supports rules loaded from the class-path. These rules are defined in **`.drl`** and **`.dsl`** files, and also in decision tables (which use **`.xls`** files.) However, there is no way to specify multiple rule files for a single **`BusinessRuleProcessor`** action. (One can, in general, have multiple rule files, though.) These file-based rules exist primarily for the purpose of testing prototypes and very simple rule services. More complex rule services need to use the **JBoss Rules** `RuleAgent`, because there is no way to specify multiple rule files in **`jboss-esb.xml`**.

The `RuleService` uses the `RuleAgent` to access rule packages from either the **Business Rules Management System** or the local file system. These rule packages can contain thousands of rules, originating in different ways. These files can originate from the following sources:

- the **BRMS**
- imported **`DRL`** files
- domain-specific language files
- decision tables

Important

Red Hat recommends using the **JBoss Rules** `RuleAgent` approach on production systems.

The **BusinessRuleProcessor** action supports both of **JBoss Rules**' *execution models*, namely the *stateless* and *stateful* models.

Most rule services will be "stateless." In the stateless model, a message is sent to the `rule service`. Every fact to be inserted into the `rule engine` is included in the body of the message. The rules execute and update either the message or the facts.

"Stateful" execution takes place over time. In this case, several messages are sent to the `rule service`. Each time the rules are executed, either the message or the facts are updated. This continues until the service receives a final message that tells it to dispose of the stateful session.

Note

This configuration model is currently limited, in the sense that there can only be a single stateful `rule service` in the message flow.

2.2. *Updated* Rule Services Using JBoss Rules

2.2.1. Introduction

JBoss Rules is the name of the engine that provides the **SOA Platform** with *rule service* support. Read this section to learn more about it.

JBoss Rules is integrated through the following components:

- the **BusinessRulesProcessor** action class
- rules written in any one of **JBoss Rules**, DRL, DSL, decision tables or the **Business Rule Editor**.
- the Enterprise Service Bus Message. (This is inserted into the Rules Engine's working memory.)

When a message is sent to the **BusinessRulesProcessor**, a *rule set* executes over the objects contained therein. The rule set will update either one of those objects or the message itself.

2.2.2. Rule-Set Creation

Create a rule-set by using **JBoss Developer Studio**. Since the message is added as a global, there is a needs to add the **jbossesb-rosetta.jar** file to the **JBoss Rules** project.

Note

For a detailed study of rule creation and the JBoss Rules language itself, please refer to the included *JBoss Rules Reference Guide*.

One must adhere to three requirements when writing rules for deployment as services on the **JBoss Enterprise SOA Platform**:

1. all rules deployed as `rule services` must define the ESB Message as a global.

   (Most rule services will want to communicate results to other services in the flow. They do so by way of updating the message, so the **BusinessRulesProcessor** or **DroolsRuleService** will always set the ESB Message as a global.)

```
#declare any global variables here
global org.jboss.soa.esb.message.Message;
```

Example 2.1. Defining an ESB Message as a Global

2. Set any other globals that are required in a rule with a higher *salience* than that for the Enterprise Service Bus Message.

   Note that the **`BusinessRulesProcessor`** and **`DroolsRuleService`** do not currently provide any means to set globals in the **`jboss-esb.xml`** file. However, this functionality may be added in the future.

   ```
   rule "Set a global"
    salience 100
    when
    then
     drools.setGlobal("foo", new Foo());
   end
   ```

   Example 2.2. Declaring a Global in a Rule with Higher Salience

3. The **`DroolsRuleService`** does not provide a means by which to start a *RuleFlow* from the `rule service` itself. (This functionality may be added in the future.) For now, this can be achieved in a rule with higher salience, as per this sample code:

   ```
   rule "Start a ruleflow"
    salience 100
    when
    then
     drools.startProcess("processId");
   end
   ```

   Example 2.3. Declaring a Global in a Higher Salience Rule

2.2.3. Rule Service Consumers

A *rule service consumer* has little to do. There is no need for it to create *rule-bases* or *working memories*, to insert facts or to execute the rules. It only has to add facts and, on occassions, properties, to the message.

In some cases, the client is *ESB-aware* meaning that it can add objects directly to the message, as per this example:

```
MessageFactory factory = MessageFactory.getInstance();
message = factory.getMessage(MessageType.JAVA_SERIALIZED);
order = new Order();
order.setOrderId(0);
order.setQuantity(20);
order.setUnitPrice(new Float("20.0"));
message.getBody().add("Order", order);
```

Example 2.4. Adding Objects to a Directly to a Message

In other cases, the data may be in an Extensible Mark-Up Language (XML) message. If so, a *transformation service* will be added to the message flow. As its name implies, its purpose is to

transform the XML into *Plain Old Java Object* (POJO) files prior to the invocation of the `rule service`.

Stateful Rule Execution

One must add a few properties to the message so that stateful rule execution can occur.

For the first message:

```
message.getProperties().setProperty("dispose", false);
message.getProperties().setProperty("continue", false);
```

For all the subsequent messages bar the final one:

```
message.getProperties().setProperty("dispose", false);
message.getProperties().setProperty("continue", true);
```

For the final message:

```
message.getProperties().setProperty("dispose", true);
message.getProperties().setProperty("continue", true);
```

Important

An ESB-aware client can add these directly. However a client that is not ESB-aware will have to communicate the position of the message (whether it is first, ongoing or last) within the data. Also, an action class must be added to the pipeline so that the properties of the Enterprise Service Bus message are included.

Note that the **`quickstarts/business_ruleservice_stateful`** file is an example of this type of service.

Note

In the releases up to, and including, **Enterprise Service Bus 4.6**, the `continue` functionality for the stateful rule execution did not dispose of the working memory if the value of the property was either **`false`** or completely absent. This has now been fixed through the work for **`JBESB-2900`**.

If there is a need to re-enable the previous behaviour, do so by changing the value of the configuration property called org.jboss.soa.esb.services.rules.continueState to **`true`**. This property is found in the **`jbossesb-properties.xml`** file.

2.2.4. Configuration

Configure a `rule service` via its `jboss-esb` action element.

To do this, the name and action class are both required. The name is user-defined:

```
<action class="org.jboss.soa.esb.actions.BusinessRulesProcessor"
```

```
  name="OrderDiscountRuleService">
```

One of the following is also required:

- a **DRL** file

```
<property name="ruleSet" value="drl/OrderDiscount.drl" />
```

- **DSL** and **DSLR** (*Domain Specific Language*) files

```
<property name="ruleSet" value="dsl/approval.dslr" />
<property name="ruleLanguage" value="dsl/acme.dsl" />
```

- a **decisionTable** on the classpath

```
<property name="decisionTable" value="PolicyPricing.xls" />
```

- a "properties" file on the classpath, the purpose of which is to tell the `rule agent` how to find the rules package. It can do so by specifying either an URL or a local file.

```
<property name="ruleAgentProperties"
          value="brmsdeployedrules.properties" />
```

Several example configurations follow:

```
<action class="org.jboss.soa.esb.actions.BusinessRulesProcessor"
  name="OrderDiscountRuleService">
    <property name="ruleSet" value="drl/OrderDiscount.drl" />
    <property name="ruleReload" value="true" />
    <property name="object-paths">
        <object-path esb="body.Order" />
    </property>
</action>
```

Example 2.5. Rules are in a **DRL** and Execution is Stateless

```
<action class="org.jboss.soa.esb.actions.BusinessRulesProcessor"
  name="OrderDiscountMultipleRuleServiceStateful">
    <property name="ruleSet
              value="drl/OrderDiscountOnMultipleOrders.drl" />
    <property name="ruleReload" value="false" />
    <property name="stateful" value="true" >
    <property name="object-paths">
        <object-path esb="body.Customer" />
        <object-path esb="body.Order" />
    </property>
</action>
```

Example 2.6. >Rules are in a **DRL** and Execution is Stateful

In this scenario, the client can, over time, send multiple messages to the rule service. The first message might contain a customer object, with the subsequent ones each containing orders for that customer. Every time a message is received, the rules will be "fired." The client can add a property to the final message that tells the rule service to dispose of the contents of the `working memory`.

```
action class="org.jboss.soa.esb.actions.BusinessRulesProcessor"
 name="PolicyApprovalRuleService">
   <property name="ruleSet" value="dsl/approval.dslr" />
   <property name="ruleLanguage" value="dsl/acme.dsl" />
   <property name="ruleReload" value="true" />
   <property name="object-paths">
       <object-path esb="body.Driver" />
       <object-path esb="body.Policy" />
   </property>
</action>
```

Example 2.7. Rules in a Domain Specific Language with Stateless Execution

```
<action class="org.jboss.soa.esb.actions.BusinessRulesProcessor"
  name="PolicyPricingRuleService">
   <property name="decisionTable"
             value="decisionTable/PolicyPricing.xls" />
   <property name="ruleReload" value="true" />
   <property name="object-paths">
       <object-path esb="body.Driver" />
       <object-path esb="body.Policy" />
   </property>
</action>
```

Example 2.8. Rules in a Decision Table with Stateless Execution

```
<action class="org.jboss.soa.esb.actions.BusinessRulesProcessor"
  name="RuleAgentPolicyService">
   <property name="ruleAgentProperties"
             value="ruleAgent/brmsdeployedrules.properties" />
   <property name="object-paths">
       <object-path esb="body.Driver" />
       <object-path esb="body.Policy" />
   </property>
</action>
```

Example 2.9. Rules in the BRMS with Stateless Execution

The *action configuration* attributes are found on the `action` tag. They specify the action to use and the name it is to be given.

Also use the `action configuration` attributes also specify the set of rules (the `ruleSet`) to employ in conjunction with this action.

`BusinessRulesProcessor` Action Configuration Attributes

Attribute	Description
Class	Action class
Name	Custom action name

`BusinessRulesProcessor` Action Configuration Properties

Property	Description
`ruleSet`	This is an optional reference to a file containing the `ruleSet`, which is the set of rules used to evaluate the content. Only one `ruleSet` can be given for each `rule service` instance.

Property	Description
`ruleLanguage`	This is an optional reference to a file containing the definition of a Domain Specific Language. This definition can be used for evaluating the rule set. If it is used, ensure that the file in the `ruleSet` is a **`dslr`**.
`ruleReload`	Set this optional property to **`true`** in order to enable the *hot redeployment* of `rule sets`. (However, enabling this feature will increase the overhead on the rules processing.) Note that rules will also reload if the **`.esb`** archive in which they live is redeployed.
`decisionTable`	This is an optional reference to a file containing the definition of a rule-specification spreadsheet.
`ruleAgentProperties`	This is an optional reference to a properties file containing the location (either a URL or file path) of the compiled rule packages. Note there is no need to specify `ruleReload` with a `ruleAgent`, as it is controlled through the properties file.
stateful	Set this optional property to **`true`** to specify that the `rule service` will be receiving multiple messages over time. (The new facts will be added to the `rule engine`'s `working memory` and the rules will be re-executed each time.)
object-paths	Use this optional property to pass message objects into **JBoss Rules**' `working memory`.

2.2.5. Object Paths

Note that **JBoss Rules** treats objects as though they are *shallow*. This is in order to achieve highly-optimized performance. Use the optional object-paths property to evaluate an object residing in a location that is "deeper" down than the `object tree`. (Set this property to extract those objects with an **`ESB Message Object Path`**.)

The *MVFLEX Expression Language* (MVEL) is used to extract the object. The path to be used must abide by the following syntax:

```
location.objectname.[beanname].[beanname]...
```

Understand that, in the above sample:

location
: is one of either the message body, header, properties or attachment;

objectname
: is the name of the object. (Attachments can be either named or numbered, so a number is a perfectly valid value to insert here);

beannames
: are optional. Use them in order to "traverse" a *bean graph*.

Example MVEL Expressions

Expression	Result
`properties.Order`	Use this to obtain the property object named `Order`

Expression	Result
`attachment.1`	obtains the first attachment object
`attachment.AttachmentOne`	obtains the attachment named **`AttachmentOne`**
`attachment.1.Order`	obtains `getOrder()` *return object* on the attached object.
`body.Order1.lineitem`	obtains the object named **`Order1`** from the body of the message. Next, it will call `getLineitem()` on this object. More elements can be added to the query in order to traverse the bean graph.

Important

Remember to add the **`java import`** statements to any objects that one imports into one's `rule set`.

The *Object Mapper* cannot "flatten out" entire collections. If one has a requirement to do that, run a "transformation" on the message first. (This will "unroll" the collection.)

2.2.6. Deploying and Packaging

Red Hat recommends that one packages one's code into "units of functionality." Use **`.esb`** packages to do so. Conceptually, the aim is to package routing rules alongside the `rule services` that use `the rule sets`. The figure below shows the layout of the **`business_rules_service`** Quick Start and, in doing so, depicts a "typical" package.

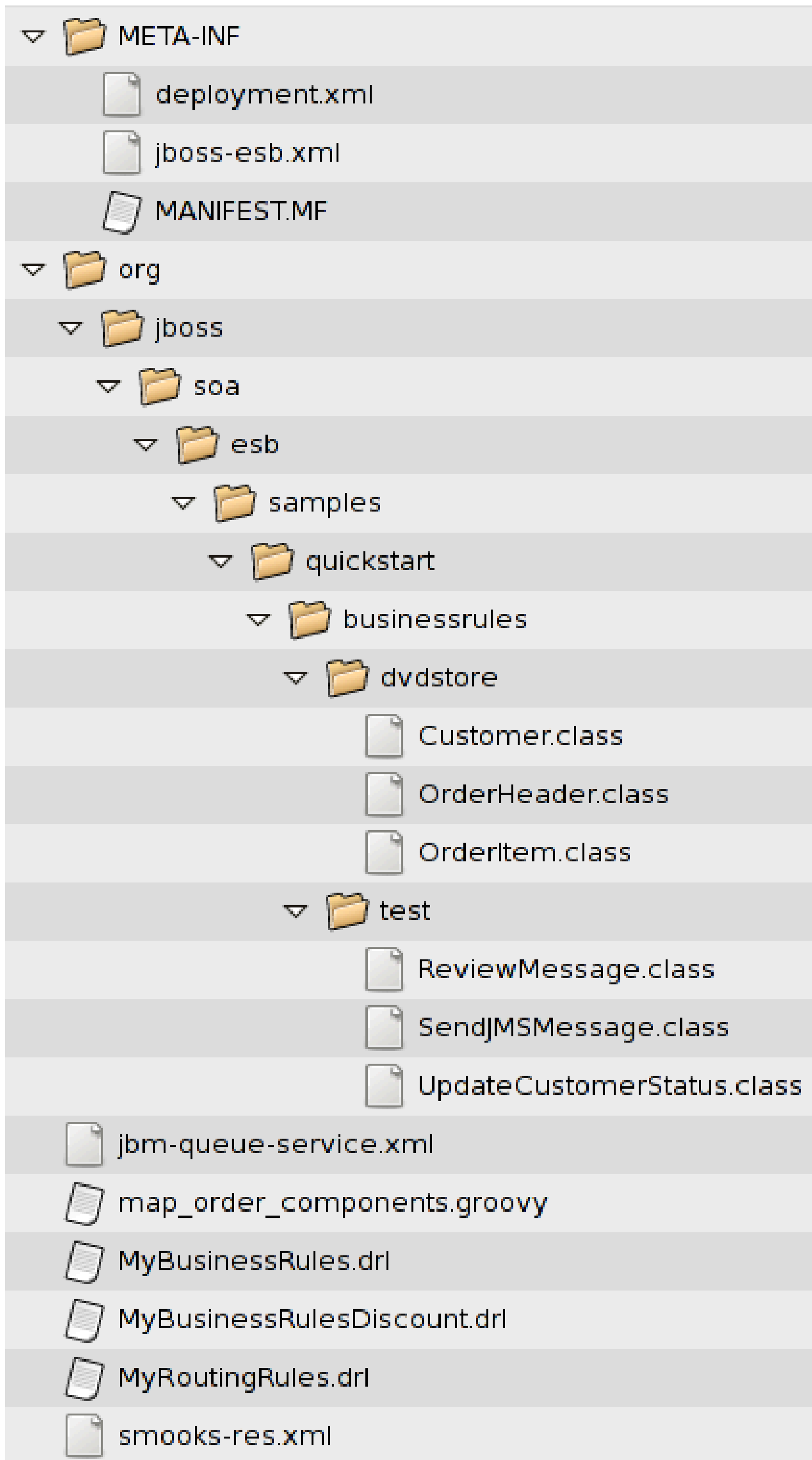


Figure 2.1. Typical **.esb** Archive which Uses JBoss Rules.

Finally, deploy and reference the **`jbrules.esb`** archive in the **`deployment.xml`** file. Here is an example of how to do this:

```
<jbossesb-deployment>
 <depends>jboss.esb:deployment=jbrules.esb</depends>
</jbossesb-deployment>
```

Content-based Routing

3.1. What is Content-Based Routing?

3.1.1. Introduction

3.1.1.1. Some Questions

In normal situations, information within the Enterprise Service Bus is conveniently packaged, transferred and stored all in the form of a *message*. Messages are addressed to *End Point References* (which are either services or clients.) An EPR's role is to identify the machine or process or object that will ultimately deal with the content of the message. However, what happen will if the specified address is no longer valid? Situations that may lead to this scenario include those in which the service has failed or been removed. It is also possible that the service no longer deals with messages of that particular type, in which case presumably some other service will still deal with the original function, but that still leaves the question of "How should the message be handled?" What if other services besides that which is the intended recipient are interested in the message's contents? What if no destination is specified?

3.1.1.2. Introducing Content-Based Routing

One possible answer to all of these problems is *Content-Based Routing* (CBR). Content-Based Routing seeks to route messages, not by a specified destination, but by the actual content of the message itself. In a typical application, a message is routed by being opened and then having a set of rules applied to its content. These rules are used to ascertain which parties are interested in it.

The Enterprise Service Bus can determine the destination of a given message based upon its content. This relieves the sending application of the onus of needing to know where the message should go.

Content-based routing and filtering networks are both extremely flexible and very powerful. When built upon established technologies such as MOM (*Message Oriented Middleware*), JMS (*Java Message Services*), and XML (Extensible Markup Language), they are also reasonably easy to implement.

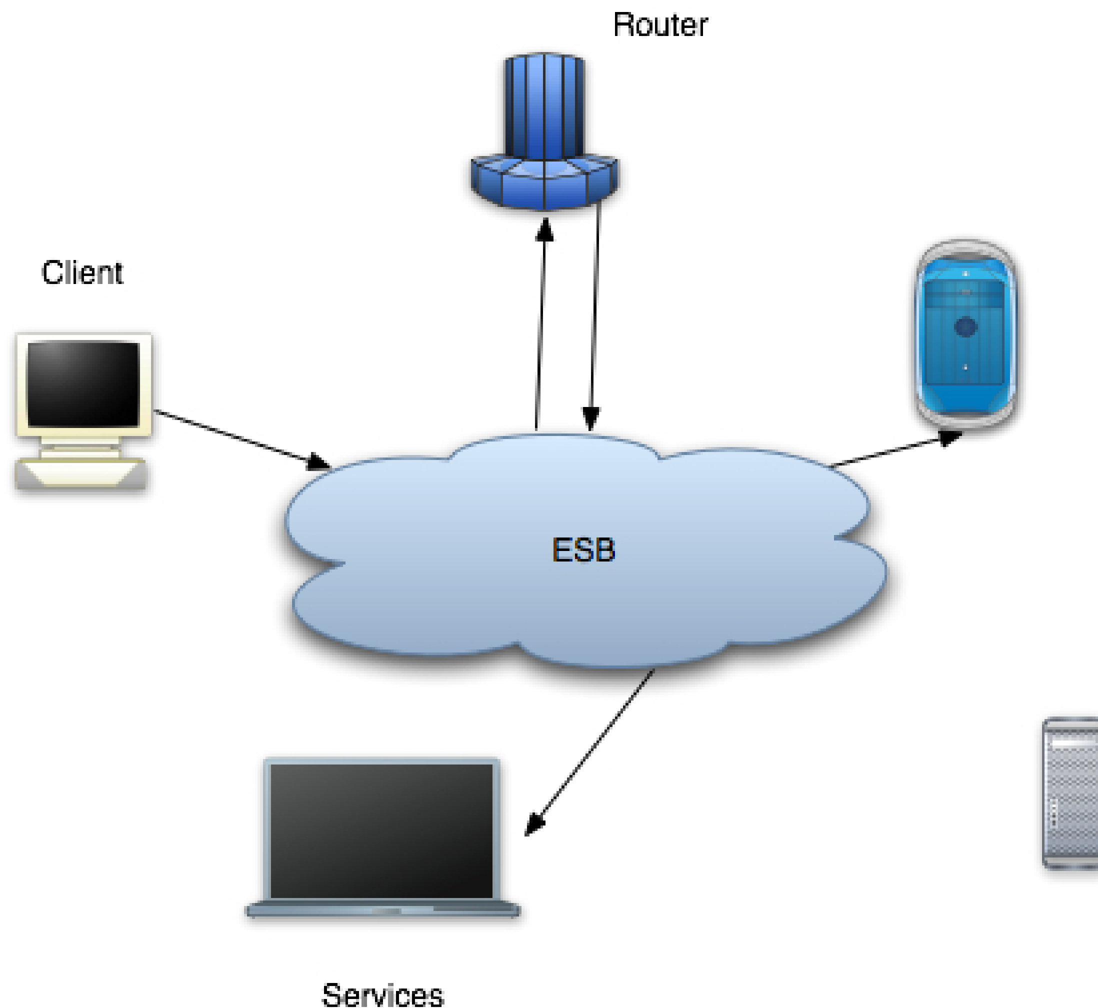
3.1.2. Simple Example

Content-based routing systems are typically built around two types of entities: routers (of which there may be only one) and services (of which there is usually more than one). Services are the ultimate consumers of messages. How services publish their interest in specific types of messages with the routers is implementation dependent, but some mapping must exist between message type (or some aspect of the message content) and services in order for the router to direct the flow of incoming messages.

Routers, as their name suggests, "route" messages. They examine the content of messages as they receive them, apply rules to that content and then forward the messages as the rules dictate.

In addition to routers and services, some systems may also include *harvesters*. These tools specialise in finding interesting information, packaging it up in the guise of a formatted message and then sending it to a router. Harvesters "mine" many sources of information, including *mail transfer agent* message stores, news servers, databases and other legacy systems.

The diagram below depicts a typical Content-Based Routing architecture that is using an Enterprise Service Bus. At the heart of the system, represented by the cloud, is the ESB. Messages are sent into it from the client, and it then directs them onwards to the router. The router is then responsible for sending the messages to their ultimate destination(s).



3.1.3. Content-Based Routing using XPath

An easy way of performing content based routing in the JBoss Enterprise Service Bus is via the *XPath Rules Provider* on the **`ContentBasedRouter`** action. This provider is very easy to use and supports both "inline" and external rule definitions.

3.1.3.1. Inline Rule Definitions

It is very simple to define inline routing rules using XPath. One merely needs to set the cbrAlias property to **`XPath`** and then define the routing rules in the **`route-to`** configurations that are found in the container destinations property.

```
<action class="org.jboss.soa.esb.actions.ContentBasedRouter" name="ContentBasedRouter">
   <property name="cbrAlias" value="XPath"/>
   <property name="destinations">
```

```
       <route-to service-category="BlueTeam"  service-name="GoBlue"  expression="/
Order[@statusCode='0']" />
       <route-to service-category="RedTeam"   service-name="GoRed"   expression="/
Order[@statusCode='1']" />
       <route-to service-category="GreenTeam" service-name="GoGreen" expression="/
Order[@statusCode='2']" />
   </property>
</action>
```

Example 3.1. Inline Rule Definition Example

3.1.3.2. External Rule Definitions

It is also very straightforward to define external XPath routing rules. Again, one must set the cbrAlias property to **XPath** and then:

- define the routing expressions in a .properties file, in which the property keys are equal to the destination names and the property values are the XPath expressions for routing to the destination in question.
- define the routing rules in the **route-to** configurations via the container destinations property, whereby the destination-name attribute will refer to the XPath rule key as defined in the external .properties file.

```
<action class="org.jboss.soa.esb.actions.ContentBasedRouter" name="ContentBasedRouter">
   <property name="cbrAlias" value="XPath"/>
   <property name="ruleSet" value="/rules/xpath-rules.properties"/>
   <property name="ruleReload" value="true"/>
   <property name="destinations">
       <route-to destination-name="blue"  service-category="BlueTeam"  service-name="GoBlue" /
>
       <route-to destination-name="red"   service-category="RedTeam"   service-name="GoRed" />
       <route-to destination-name="green" service-category="GreenTeam" service-
name="GoGreen" />
   </property>
</action>
```

Example 3.2. External Rule Definition Example

The XPath rules are in a **.properties** file. They are represented in this simple format:

```
blue=/Order[@statusCode='0']
red=/Order[@statusCode='1']
green=/Order[@statusCode='2']
```

Example 3.3. XPath Rules File

3.1.3.3. Namespaces

XML name-space prefix-to-URI mappings are defined in the namespace elements. These are contained within the **namespaces** container property. Name-space prefix-to-URI mappings are defined in exactly the same way for both inline and external rule definitions.

Here is an example, from a quick start, of how to define a name-space:

$ pwd

```
        /opt/local/50_ER7_Jan14/jboss-soa-p.5.0.0/jboss-as/samples/quickstarts
```

$ grep -ir "use namespaces" *

```
          fun_cbr/FunCBRRules-XPath.drl: xpathEquals expr "/order:Order/@statusCode", "0" use
namespaces "order=http://org.jboss.soa.esb/Order"
          fun_cbr/FunCBRRules-XPath.drl: xpathEquals expr "/order:Order/@statusCode", "1" use
namespaces "order=http://org.jboss.soa.esb/Order"
          fun_cbr/FunCBRRules-XPath.drl: xpathEquals expr "/order:Order/@statusCode", "2" use
namespaces "order=http://org.jboss.soa.esb/Order"
```

```
<!-- ESB XPath CBR Service -->
          <service category="Fun_CBRServices_ESB"
          name="XPath_FunCBRService_ESB" description="ESB
          Listener - for the native clients" invmScope="GLOBAL">
              <listeners>
                  <!-- Gateway -->
                  <jms-listener name="TheGateway"
                  busidref="xpathQuickstartGwChannel"
                  is-gateway="true" />
               </listeners>
              <actions mep="OneWay">
                  <action class="org.jboss.soa.esb.actions.ContentBasedRouter"
                  name="ContentBasedRouter">
                      <property name="cbrAlias" value="XPath"/>
                      <property name="destinations">
                          <namespace prefix="ord" uri="http://org.jboss.soa.esb/Order" />
                          <route-to service-category="BlueTeam"
                          service-name="GoBlue"
                          expression="/ord:Order[@statusCode='0']" />
                          <route-to service-category="RedTeam"
                          service-name="GoRed"
                          expression="/ord:Order[@statusCode='1']" />
                          <route-to service-category="GreenTeam"
                          service-name="GoGreen"
                          expression="/ord:Order[@statusCode='2']" />
                      </property>
                  </action>
              </actions>
          </service>
```

Example 3.4. Name-Space Example

3.1.4. Content-Based Routing using Regex

An easy way to perform content-based routing in JBoss Enterprise Service Bus is via the *Regex rules provider* on the **ContentBasedRouter** action. One will find this provider very easy to use and it supports both inline and external rule definitions.

3.1.4.1. Inline Rule Definitions

Defining inline Regex routing rules is very straightforward. One merely needs to set the cbrAlias property to **Regex** and then define the routing rules in the route-to configurations, found in the container destinations property.

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

```
   <property name="cbrAlias" value="Regex"/>
   <property name="destinations">
       <route-to service-category="BlueTeam"  service-name="GoBlue"  expression="#*111#*" />
       <route-to service-category="RedTeam"   service-name="GoRed"   expression="#*222#*" />
       <route-to service-category="GreenTeam" service-name="GoGreen" expression="#*333#*" />
   </property>
</action>
```

Example 3.5. Regex Example

3.1.4.2. External Rule Definitions

Defining external XPath routing rules is also quite simple. Again, one merely needs to set the cbrAlias property to **`Regex`** and then:

- define the routing expressions in a **`.properties`** file, where the property keys are the destination names and the property values are the Regex expressions for routing to the destination in question.
- define the routing rules in the **`route-to`** configurations in the container destinations property, with the destination-name attribute set to refer to the Regex rule key as defined in the external **`.properties`** file.

```
<action class="org.jboss.soa.esb.actions.ContentBasedRouter" name="ContentBasedRouter">
   <property name="cbrAlias" value="XPath"/>
   <property name="ruleSet" value="/rules/regex-rules.properties"/>
   <property name="ruleReload" value="true"/>
   <property name="destinations">
       <route-to destination-name="blue"  service-category="BlueTeam"  service-name="GoBlue" /
>
       <route-to destination-name="red"   service-category="RedTeam"   service-name="GoRed" />
       <route-to destination-name="green" service-category="GreenTeam" service-
name="GoGreen" />
   </property>
</action>
```

Example 3.6. External Rules

The XPath rules are in a **`.properties`** file. They are represented in this simple format:

```
blue=#*111#*
red=#*222#*
green=#*333#*
```

Example 3.7. XPath Rules File

3.2. Content-Based Routing Using JBoss Rules

3.2.1. Introduction

The *content-based router* used in the JBoss Enterprise Service Bus utilises **JBoss Rules** as its default rule provider "engine." The Enterprise Service Bus integrates with **JBoss Rules** through three different routing `action classes`. These are:

- a routing rule set, written in **JBoss Rules**' DRL (or, optionally, the DSL) language;

- the Enterprise Service Bus message content, which is the data that goes into the `rule engine` (it takes the form of either XML or objects within the message);
- the destination, (which is derived from the resultant information coming out of the rules engine.)

Important

There is no native support for **Freemarker** inside the **Enterprise Service Bus** and, hence, any use of this templating system must come from within the context of **Smooks**.

When a message is sent to the `content-based router`, a certain `rule set` will evaluate its content and return a set of service destinations. This chapter will teach how a rule set can be targeted, how the message content is evaluated and what can be achieved with the resulting destinations.

3.2.2. Three Different Routing Action Classes

The JBoss Enterprise Service Bus ships with three slightly different routing *action classes*. Each of these implements an *Enterprise Integration Pattern* (EIP). (The *JBossESB Wiki* contains more information about this subject.) These are the three supported action classes:

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

   This action class implements the content-based routing pattern. It routes a message to one or more destination services, based on the message content and the rule set against which it is evaluating that content. The content-based router throws an exception when no destinations are matched for a given rule set or message combination. This action will terminate any further pipeline processing, so it should be positioned last in one's pipeline.

2. **`org.jboss.soa.esb.actions.ContentBasedWireTap`**

   This implements the *WireTap* pattern. The `WireTap` is an Enterprise Integration Pattern through which a copy of the message is send to a control channel. The `WireTap` is identical in functionality to the standard content-based router, however it does not terminate the pipeline. It is this latter characteristic which makes it suitable to be used as a wire-tap.

3. **`org.jboss.soa.esb.actions.MessageFilter`**

   This implements the *message filter pattern*. The message filter pattern represents that case in which messages can simply be dropped if certain content requirements are not met. It is identical in functionality to the Content-Based Router but it does not throw an exception if the rule set does not match any destinations. If none are met, the message is simply filtered out.

3.2.3. Rule-Set Creation

Crate a rule-set by using the **JBoss Developer Studio** which includes a plug-in for **JBoss Rules**. *Figure 3.1, “Create a New Rule Set using the JBoss Developer Studio”* shows a screen-shot of this plug-in. For a detailed analysis of the subjects of rule creation and the **JBoss Rules** language itself, please see the **JBoss Rules** documentation.

To turn a regular rule-set into one that can be used for content-based routing, one must evaluate an ESB message and ensure that the rule match results in a list of strings containing the service destination names. Bear two things in mind whilst doing this:

- firstly, ensure the rule set imports the ESB message

```
import org.jboss.soa.esb.message.Message
```

- secondly, ensure that the rule set defines the following global variable which will create the list of destinations available to the Enterprise Service Bus:

```
global java.util.List destinations;
```

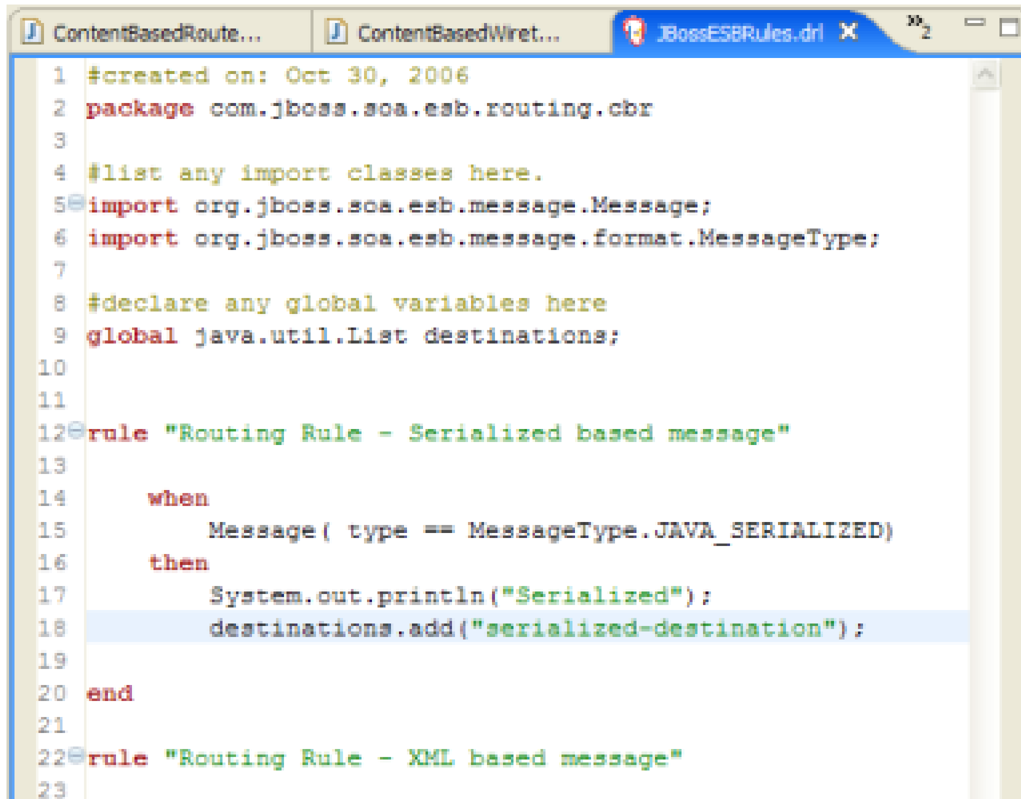


Figure 3.1. Create a New Rule Set using the JBoss Developer Studio

The message will now be added to the `rule engine`'s `working memory`. The figure shows an example in which the **`MessageType`** is used to determine to which destination the Message will be sent. This particular rule-set is shipped in the **`JBossESBRules.drl`** file. The rule also checks if the format type is XML or of the serialized.

3.2.4. XPath Domain Specific Language

It is convenient to undertake an *XPath*-based evaluation of XML-based messages. Red Hat supports this by shipping a *domain-specific language* implementation. Use this implementation to utilise XPath expressions in the rule file.

These expressions are defined in the **`XPathLanguage.dsl`** file. To use, simply reference it in the rule-set with:

```
expander XPathLanguage.dsl
```

Currently, the XPath Language makes sure the message is of the type **`JBOSS_XML`** and that it defines the following items:

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

The XPath Language is defined in a file called **`XPathLanguage.dsl`**. One can customise it if the need arises. Alternatively, one can define an entirely different domain-specific language.

Note

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

3.2.4.1. XPath and Name-Spaces

To use name-spaces with XPath, specify which name-space prefixes are to be used in the XPath expression. The name-space prefixes are specified in a comma-separated list of the following format: **`"prefix=uri,prefix=uri"`**. This same can done for all of the different kinds of XPath expressions that were mentioned above.

1. **`xpathMatch expr "<expression>" use namespaces "<namepaces>"`**
2. **`xpathEquals expr "<expression>", "<value>" use namespaces "<namspaces>"`**
3. **`xpathGreaterThan "<expression>", "<value>" use namespaces "<namspaces>"`**
4. **`xpathLowerThan expr "<expression>", "<value>" use namespaces "<namespaces>"`**

The name-space-aware statements differ in that they all need the extra **`expr`** keyword in front of the XPath expression. This avoids collisions with the non-XPath aware statements in the DSL file. The prefixes do not have to match those used in the XML to be evaluated: it only matters that the uniform resource identifier is the same.

3.2.4.2. Configuration

These individual pieces are all connected via configuration, which is undertaken in the **`jboss-esb.xml`** file. The `service configuration` below shows a service configuration fragment. In this fragment the service is listening to a Java Message Service queue.

Each ESB message is passed to the **`ContentBasedRouter`** action class, which is loaded with a certain rule-set. It moves the ESB message into working memory, "fires" the rules, obtains the list of destinations and routes copies of the ESB message to the services. It uses the **`JbossESBRules.drl`** rule-set, which matches two destinations, namely `xml-destination` and `serialized-destination`. These names are mapped to those of real services in the `route-to` section.

```
<service category="MessageRouting"
   name="YourServiceName" description="CBR Service">

   <listeners>
    <jms-listener name="CBR-Listener" busidref="QueueA" maxThreads="1">
 </jms-listener>
</listeners>

<actions>
 <action class="org.jboss.soa.esb.actions.ContentBasedRouter"
  name="YourActionName">
  <property name="ruleSet" value="JBossESBRules.drl"/>
  <property name="ruleReload" value="true"/>
  <property name="destinations">
   <route-to destination-name="xml-destination"
    service-category="category01"
    service-name="jbossesbtest1" />
   <route-to destination-name="serialized-destination"
    service-category="category02"
    service-name="jbossesbtest2" />
  </property>
  <property name="object-paths">
   <object-path esb="body.test1" />
   <object-path esb="body.test2" />
  </property>
 </action>

</actions>

</service>
```

Figure 3.2. Example of Content-Based Routing Service Configuration

This table shows `action` tag's attributes. These attributes specify which `action` is to be used and which name it is to be given:

Attribute	Description
Class	Action class, this being one of : **org.jboss.soa.esb.actions.ContentBasedRouter**, **org.jboss.soa.esb.actions.ContentBasedWireTap** or **org.jboss.soa.esb.actions.MessageFilter**
Name	Custom action name

Table 3.1. CBR Action Configuration Attributes

This table depicts the action properties. The properties specify which set of rules (`ruleSet`) is to be used within the action:

Property	Description
ruleSet	Name of the filename containing the **JBoss Rules ruleSet**, which is the set of rules used to evaluate content. Only one **ruleSet** can be given for each CBR instance.
ruleLanguage	This is an optional reference to a file containing the definition of a Domain Specific Language to be used for evaluating the rule set.
ruleAgentProperties	This property points to a "rule agent properties" file located on the class-path. The file can contain a property that points to pre-compiled rules packages on the file system, in a directory or identified by an uniform resource locator for

Property	Description
	integration with the **Business Rule Management System**. See the "RuleAgent" section below for more information.
ruleReload	This is an optional property which can be set to **`true`** in order to enable "hot" redeployment of rule sets. Note that this feature will cause some overhead on the rules processing. Note also that the rules will reload if the **`.esb`** archive in which they reside is redeployed.
stateful	This is an optional property which tells the RuleService to use a stateful session where facts will be remembered between invocations. See the "Stateful Rules" section for more information about this topic.
destinations	This is a set of route-to properties, each of which contains the logical name of the destination, along with the Service category and name as referenced in the registry. The logical name is the name which should be used in the rule set.
object-paths	This is an optional property to pass `message` objects into `working memory`.

Table 3.2. CBR Action Configuration Properties

3.2.4.3. Object Paths

Note that **JBoss Rules** treats objects as though they were "shallow" in order to achieve highly-optimised performance. To evaluate an object that resides in a location deeper than the "object tree," use the optional object-paths property to extract objects from the message, via an "ESB Message Object Path". MVEL is used to extract the object and the path used should therefore use the following syntax:

```
location.objectname.[beanname].[beanname]...
```

where,

location
: is one of {body, header, properties, attachment};

objectname
: is the name of the object. Attachments may be either named or numbered, so, in their case, this can be a number; too.

beannames
: is an optional you can specify to traverse a bean graph.

Here are some examples:

- **`properties.Order`** - obtains the property object named **`Order`**
- **`attachment.1`** - obtains the first attachment Object
- **`attachment.FirstAttachment`** - obtains the attachment named **`FirstAttachment`**
- **`attachment.1.Order`** - obtains `getOrder()` return object on the attached Object.
- **`body.Order1.lineitem`** - obtains the object named **`Order1`** from the body of the message. Next it will call `getLineitem()` on this object. More elements can be added to the query in order to traverse the bean graph.

It is important to remember that you have to add **java import** statements onto the objects you import into your rule-set.

Finally, the Object Mapper cannot flatten out entire collections, so if you have need to do that you firstly have to undertake a (**Smooks-**) transformation of the message. This is in order to unroll the collection.

3.2.4.4. Stateful Rules

Using stateful sessions means that facts will be remembered across invocations. When stateful is set to **true**, the working memory will not be cleared.

Tell stateful rule services when to continue with a current stateful session and when to dispose of it via message properties . To signal that the existing stateful session is to be continued, set these two message properties:

```
message.getProperties().setProperty(“dispose”, false);
message.getProperties().setProperty(“continue”, true);
```

When one invokes the rules for the last time, one must set “dispose” to "**true**" so that the working memory is disposed:

```
message.getProperties().setProperty(“dispose”, true);
message.getProperties().setProperty(“continue”, true);
```

For more details about the RuleService please see *Section 2.2, “ Updated Rule Services Using JBoss Rules ”*.

For an example showing how to use stateful rules, please refer to the **business_ruleservice_stateful** Quick Start.

3.2.4.5. RuleAgent

By using the **RuleAgent** property, one can utilise pre-compiled rules packages. These packages can be located on the local file system, in a local directory or pointing to an URL. For information about the configuration options that exist for the properties file, please refer to section *9.4.4.1. The Rule Agent*[1] of the **Drools** manual.

For more details about the RuleService, please see *Section 2.2, “ Updated Rule Services Using JBoss Rules ”*.

For an example of using a rule agent, please read the business_ruleservice_ruleAgent Quick Start.

3.2.4.6. RuleAgent and Business Rule Management System

By using the rule agent property, one can effectively integrate one's service with a Business Rule Management System (BRMS.) This can be accomplished by specifying a URL in the rule agent properties file. For information about the how to configure the URL and the other properties, please refer to section *9.4.4.1. The Rule Agent*[2] of the *JBoss Rules* documentation.

---

[1] http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html/ch09s04.html#d0e5889
[2] http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html/ch09s04.html#d0e5889

Note

For information about the how to install and configure the Business Rules Management System, please refer to the *JBoss Rules* manual.

3.2.4.7. Executing Business Rules

There is a close relationship between rule execution for modifying data in the message according to business processes and rule execution for routing. An example quick start called **`business_rule_service`** demonstrates this use case. This quick start uses the **`org.jboss.soa.esb.actions.BusinessRulesProcessor`** action class.

The functionality of the *Business Rule Processor* (BRP) is similar to that of a content-based router. However, it is not a router. It returns the modified ESB message for further `action pipeline` processing. One can mix business and routing rules in a single rule set if one so wishes. However, routing will only occur if one of those three routing **`action`** classes mentioned previously is used.

3.2.4.8. Changing Rule Service Implementations

To use a different rule service than that which is shipped with the JBoss Enterprise Service Bus, specify the preferred class in the action configuration:

```
<property name="ruleServiceImplClass" value="org.com.YourRuleService" />
```

The rule service is required to implement the `org.jboss.soa.esb.services.rules.RuleService` interface.

3.2.4.9. Deployment and Packaging

Note that one should package one's code by grouping it into units of functionality, using **`.esb`** packages. The idea of this is to collate the routing rules alongside the services that use those rule sets. *Figure 3.3, “Typical JBoss Rules .ESB Archive”* below shows the layout of the **`simple_cbr`** Quick Start in order to depict that which is typical of a package:

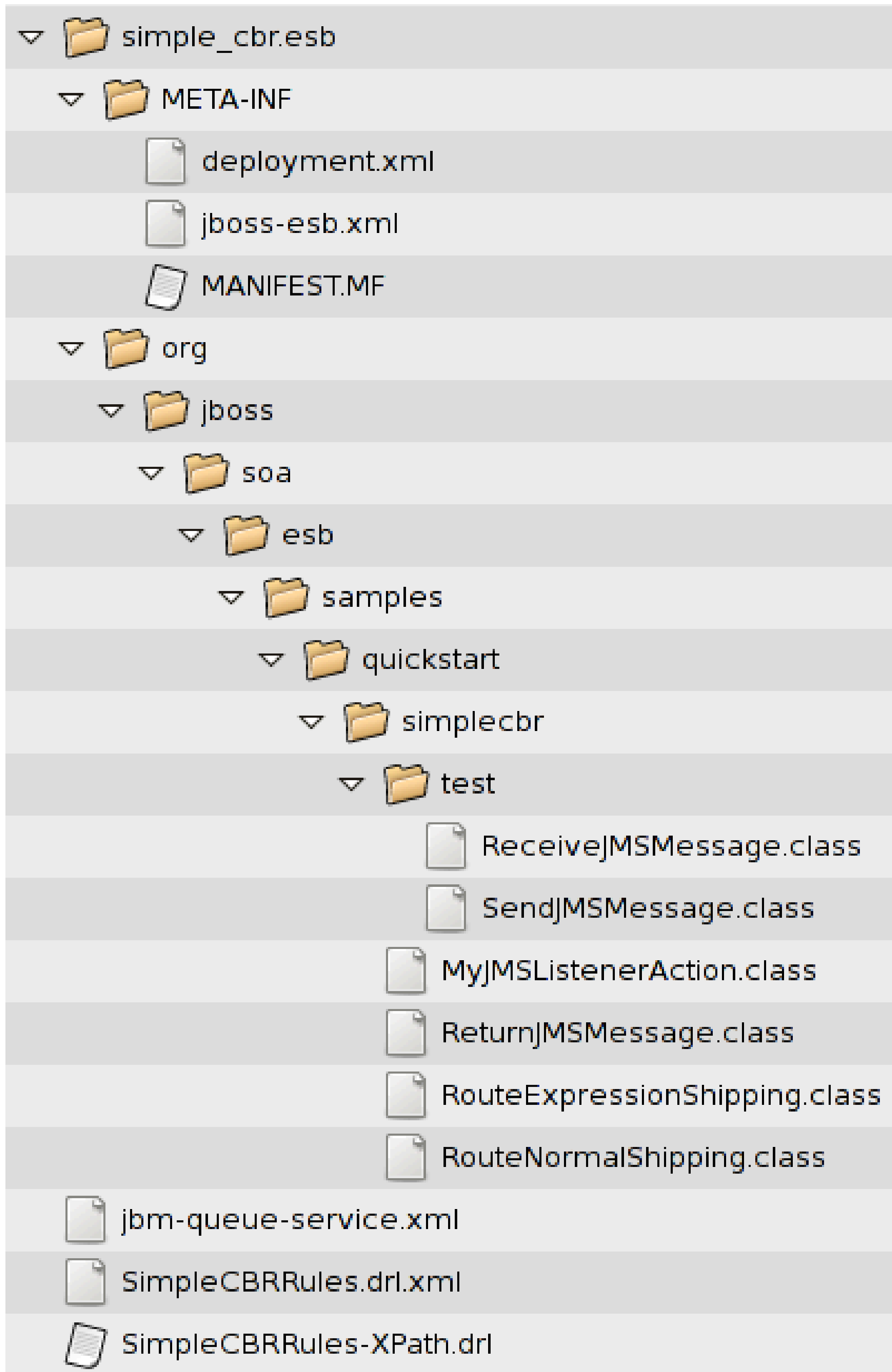


Figure 3.3. Typical JBoss Rules .ESB Archive

Finally, deploy and reference the **jbrules.esb** in the **deployment.xml** file, as per this example:

```
<jbossesb-deployment>
 <depends>jboss.esb:deployment=jbrules.esb</depends>
</jbossesb-deployment>
```

updated jBPM Integration

This section of the book examines the **JBoss Business Process Manager**. Read on to learn about the features of this powerful tool. (This document assumes that the readership is familiar with the basics of jBPM. If this is not the case, read the *jBPM Reference Guide* included with this software first.)

The **JBoss Business Process Manager** is a powerful workflow and *business process management* (BPM) engine. Use it to create business processes when there is a need to co-ordinate people, applications and services. The **jBPM** uses a modular architecture which combines easy development of work-flow applications with a process engine that is both flexible and scalable.

To represent the steps in a business procedure graphically, use the accompanying **jBPM Process Designer**. This can facilitate the formation of a strong working relationship between the business analyst and the technical developer.

The **JBoss Enterprise Service Bus** integrates with the **jBPM** for two reasons, these being:

1. Service Orchestration

   ESB services can be "orchestrated" using the **Business Process Manager**. To do so, create a *process definition* which calls upon them.

2. Human Task Management

   The **Business Process Manager** allows one to integrate machine-based services with the management of tasks undertaken by people.

4.1. Integration Configuration

The full jBPM run-time and console are included with the **`jbpm.esb`** deployment that ships with the **JBoss Enterprise Service Bus**. The runtime and the console share a common database. To create the database, start the Enterprise Service Bus **`DatabaseInitializer`** *M-Bean*. (The configuration settings for this M-Bean are found in the **`jbpm.esb/jbpm-service.xml`** file.)

```
<classpath codebase="deploy" archives="jbpm.esb"/>
<classpath codebase="deploy/jbossesb.sar/lib"
   archives="jbossesb-rosetta.jar"/>

<mbean code="org.jboss.internal.soa.esb.dependencies.DatabaseInitializer"
   name="jboss.esb:service=JBPMDatabaseInitializer">
   <attribute name="Datasource">java:/JbpmDS</attribute>
   <attribute name="ExistsSql">select * from JBPM_ID_USER</attribute>
   <attribute name="SqlFiles">
       jbpm-sql/jbpm.jpdl.hsqldb.sql,jbpm-sql/import.sql
   </attribute>
   <depends>jboss.jca:service=DataSourceBinding,name=JbpmDS</depends>
</mbean>

<mbean code="org.jboss.soa.esb.services.jbpm.configuration.JbpmService"
   name="jboss.esb:service=JbpmService">
</mbean>
```

Example 4.1. ESB DatabaseInitializer MBean Configuration

The first MBean configuration element contains the settings for the **`DatabaseInitializer`**.

Property	Description	Default
Data Source	The data source for the jBPM database	java:/JbpmDS
ExistsSql	Use this SQL command to confirm the existence of the database.	Select * from JBPM_ID_USER
SqlFiles	These files contain the SQL commands to create the jBPM database if it is not found.	jbpm-sql/jbpm.jpdl.hsqldb.sql, jbpm-sql/import.sql

Table 4.1. ESB DatabaseInitializer Mbean Default Values

The **DatabaseInitializer** MBean is configured (via the **jbpm-service.xml** file) to wait for the **JbpmDS** to be deployed, before it then deploys itself. The second MBean, **JbpmService**, ties the lifecycle of the **Business Process Manager**'s *job executor* to that of the **jbpm.esb**. It does so by launching a job executor instance on start-up. (It, of course, stops it on shutdown.)

The **JbpmDS** data source is defined in the **jbpm-ds.xml** file. By default, it uses a **Hypersonic** database. (Always change this to a production-quality database in a live environment.) Note that all **jbpm.esb** deployments should share the same database instance. This is so that the various Enterprise Service Bus nodes have access to the same *processes definitions*.

The **jBPM Console** is a web application. Access it from this address: *http://localhost:8080/jbpm-console*, after the server has been started.

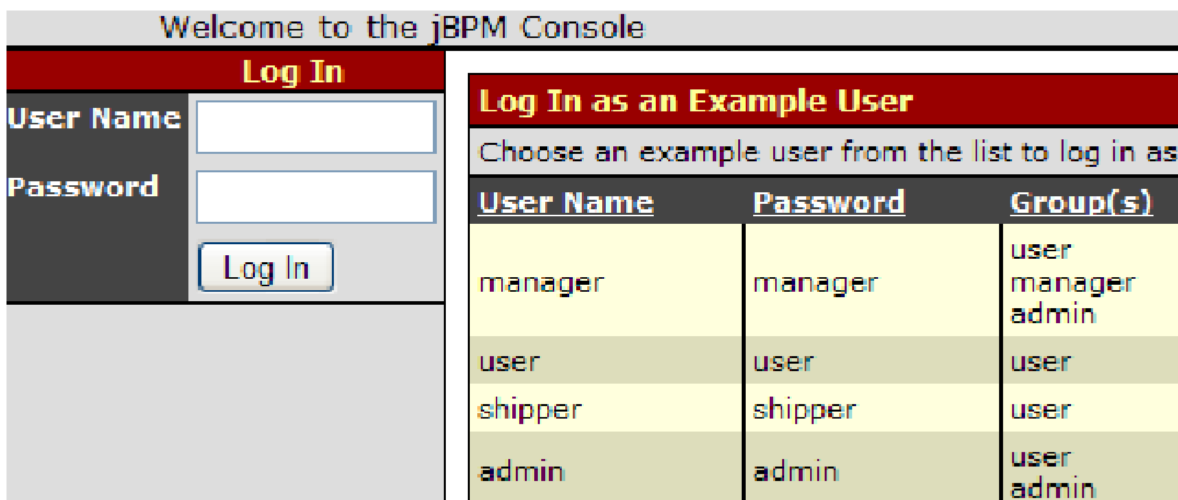


Figure 4.1. jBPM Console Log In

Please refer to the *jBPM Reference Guide* in order to learn how to change the security settings for this application. The process involves changing some settings in the **conf/login-config.xml** file. Use the console to deploy and monitor both jBPM processes and human task management procedures. (A customised tasklist will be shown for each user of the software, allowing them to work on their own tasks) The Quick Start entitled **bpm_orchestration4** demonstrates this feature.)

The **jbpm.esb/META-INF** directory contains the **deployment.xml** and **jboss-esb.xml** files.

The **deployment.xml** file specifies the two resources upon which the ESB archive will depend. (They are the **jbossesb.esb** and the **JbpmDS** data source files. The information in these files is used to determine the order of deployment.)

```
<jbossesb-deployment>
```

```
  <depends>jboss.esb:deployment=jbossesb.esb</depends>
  <depends>jboss.jca:service=DataSourceBinding,name=JbpmDS</depends>
</jbossesb-deployment>
```

Example 4.2. **deployment.xml** Dependancy Declarations

The **jboss-esb.xml** file deploys one internal service, called **JBpmCallbackService**:

```
<services>
  <service category="JBossESB-Internal" name="JBpmCallbackService"
    description="Service which makes Callbacks into jBPM">
    <listeners>
      <jms-listener name="JMS-DCQListener"
        busidref="jBPMCallbackBus" maxThreads="1" />
    </listeners>
    <actions mep="OneWay">
      <action name="action"
        class="org.jboss.soa.esb.services.jbpm.actions.JBpmCallback"/>
    </actions>
  </service>
</services>
```

Example 4.3. JBpmCallbackService

This internal service listens to the **jBPMCallbackBus**, which, by default, is set as either a **JBossMQ** (the **jbmq-queue-service.xml** file) or a **JBossMessaging** (the **jbm-queue-service.xml** file.) It is a messaging provider for the *Java Message Service Queue*. Ensure that only one of these files is deployed in the **jbpm.esb** archive. If one wants to use one's own messaging provider, simply modify the corresponding section in the **jboss-esb.xml** file to refer to it, in the way shown in this example:

```
<providers>
  <jms-provider name="CallbackQueue-JMS-Provider"
    connection-factory="ConnectionFactory">
    <jms-bus busid="jBPMCallbackBus">
      <jms-message-filter dest-type="QUEUE"
        dest-name="queue/CallbackQueue" />
    </jms-bus>
  </jms-provider>
</providers>
```

Example 4.4. Modifying the Provider Section in the **jboss-esb.xml** for One's Own Java Message Service

Note

Section 4.5, " jBPM-to-JBoss ESB " contains more information about the `JbpmCallbackService`.

4.2. Configuring the jBPM

The configuration of Business Process Manager itself is managed by three files, namely **jbpm.cfg.xml**, **hibernate.cfg.xml** and **jbpm.mail.templates.xml**.

The **jbpm.cfg.xml** file is programmed, to use the *JTA Transaction Manager* by default.

```
<service name="persistence">
  <factory>
    <bean class="org.jbpm.persistence.jta.JtaDbPersistenceServiceFactory">
      <field name="isTransactionEnabled"><false/></field>
      <field name="isCurrentSessionEnabled"><true/></field>
      <!--field name="sessionFactoryJndiName">
      <string value="java:/myHibSessFactJndiName" />
      </field-->
    </bean>
  </factory>
</service>
```

Figure 4.2. The Default Values in the **`jbpm.cfg.xml File`**

Other settings are left as the jBPM defaults.

The **`hibernate.cfg.xml`** file is also modified to use the `JTA Transaction Manager`.

```
<!-- JTA transaction properties (begin) ===
    ==== JTA transaction properties (end) -->
<property name="hibernate.transaction.factory_class">
  org.hibernate.transaction.JTATransactionFactory</property>

<property name="hibernate.transaction.manager_lookup_class">
  org.hibernate.transaction.JBossTransactionManagerLookup</property>
```

Figure 4.3. Default Values in the **`hibernate.cfg.xml`** File

Hibernate is not used to create the database schema. Rather, the `DatabaseInitializer` M-Bean referred to in *Section 4.1, “ Integration Configuration ”* is utilised.

The **`jbpm.mail.templates.xml`** file is empty by default.

Note

For more details on each of these configuration files, please see the *jBPM Guide*.

Important

The configuration files that formerly shipped with the **`jbpm-console.war`** have been removed. This was done to centralized all of the configuration files in the root of the **`jbpm.esb`** archive.

4.3. Creating and Deploying a Process Definition

Red Hat recommends using the **Eclipse**-based *jBPM Process Designer Plug-in* (KA-JBPM-GPD) to create *process definitions*. Either download and install it in **Eclipse** manually or use the **JBoss Developer Studio** to do so.

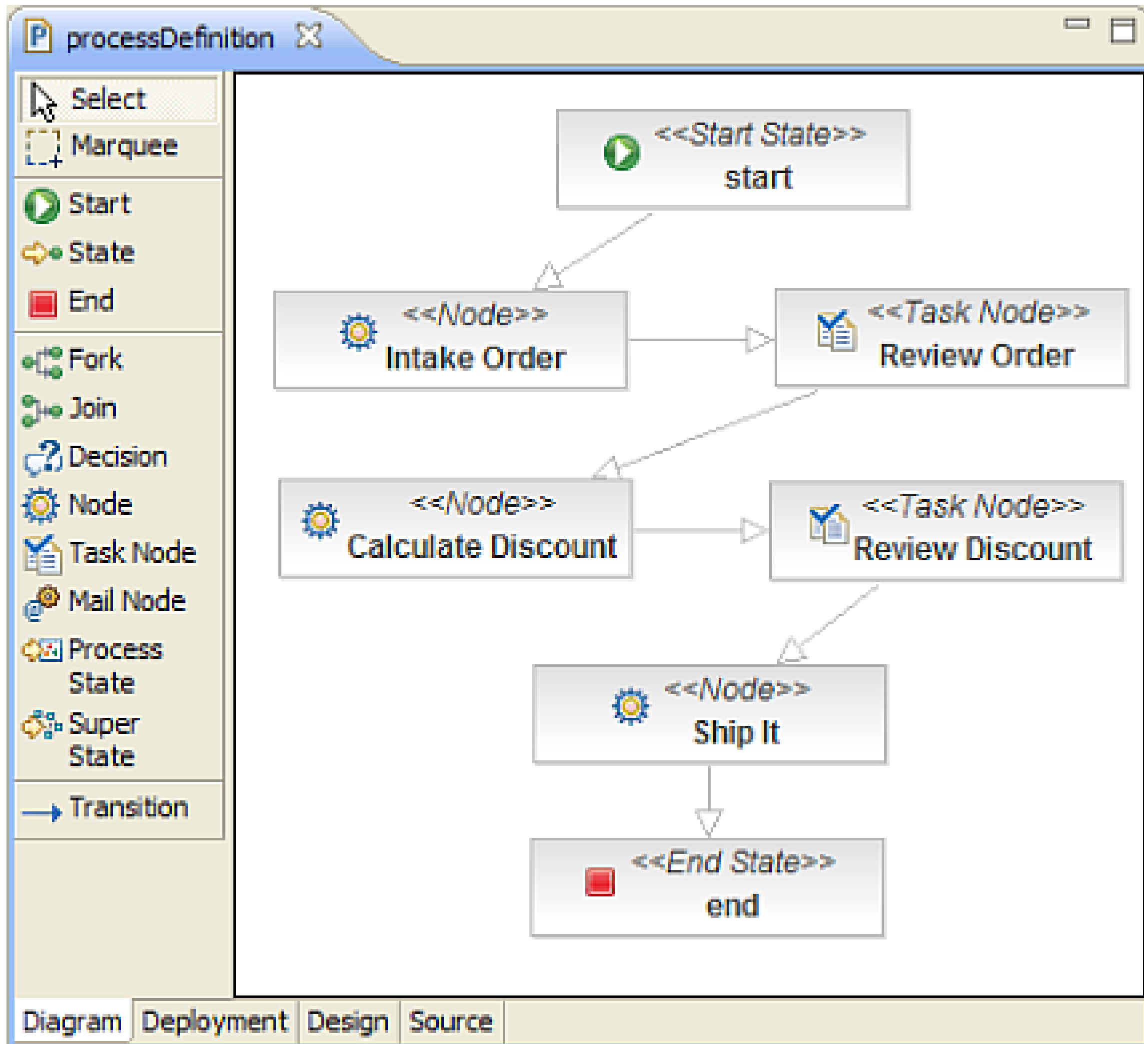


Figure 4.4. JBoss Developer Studio - jBPM Graphical Editor

Use the **jBPM Graphical Editor** to create a process definition visually. Nodes, (and transitions between nodes), can be added, modified and removed. Each process definition is saved in XML format. The saved file can then be stored in a directory and deployed to a jBPM instance (that is, a database.) Each time one deploys the process instance, the jBPM will "version" it and retain the older copies. This allows processes that are already underway to complete on the instance upon which they were started. New instances will use the latest version of the process definition.

In order to deploy a process definition, first check that the server is running. Next, activate a *Process Archive* (PAR) by going to the **Deployment** tab in the **Graphical Editor**:

Figure 4.5. JBoss Developer Studio - jBPM Deployment View

Sometimes it is sufficient to just deploy the **`processdefinition.xml`** file but, in most cases, one will be deploying other kinds of `artifacts` as well, such as *task forms*.

Warning

It is also possible to deploy Java classes into a **`process archive`**. This means that they will end up in the database, where they will be stored and versioned. Red Hat does not recommend doing this in the Enterprise Service Bus environment, the reason being that it can lead to class-loading issues. The recommended practice is to instead deploys the classes into the server's **`lib`** directory.

Use one of the following three mechanisms to deploy a process definition:

1. through **JBoss Developer Studio**, by clicking on the **Deploy Process Archive** button (having first configured the upload servlet used by the deployer.) This is visible in the **Deployment** view;

2. by saving the deployment to a local **`.par`** file from the **Deployment** view and then using the jBPM Console to activate the archive. (In order to do this, one needs to be able to log in to the console with the privileges of an administrator.)

3. by using the **`DeployProcessToServer`** jBPM **`ant`** task.

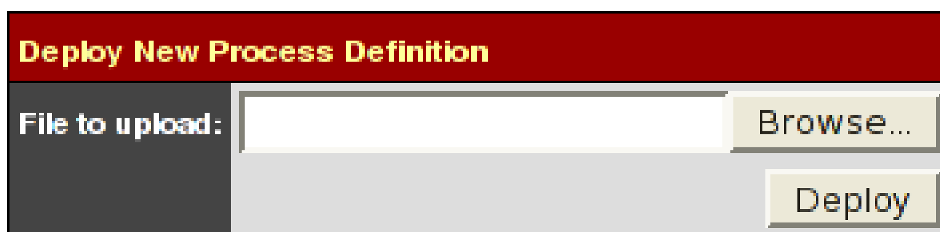


Figure 4.6. jBPM Console - Uploading a New Process Definition

4.4. From the Enterprise Service Bus to the jBPM

The **JBoss Enterprise Service Bus** can make calls into the Business Process Manager by using the **`BpmProcessor`** action. This action utilises the `jBPM Command` API. The following jBPM commands have been implemented at this stage:

Command	Description
`NewProcessInstanceCommand`	This command starts a new **`ProcessInstance`**, the associated process definition of which has already been deployed to the jBPM. The **`NewProcessInstanceCommand`** leaves the process instance in the **`start`** state. This is needed in the case of a task being associated with the `Start node`, an example being when there is one on an *actor*'s task-list.
`StartProcessInstanceCommand`	This is identical to the **`NewProcessInstanceCommand`** except that the new process instance is automatically moved from the **`start`** position to the first node.
`GetProcessInstanceVariablesCommand`	This command takes the root node variables for a process instance, by using the process instance identifier.
`CancelProcessInstanceCommand`	This command cancels a **`ProcessInstance`**. Use it in situations such as that which occurs when an event is received that should result in the cancellation of the entire **`ProcessInstance`**. (This action requires some jBPM context variables to be set on the message, most notably the **`ProcessInstance`** identifier.)

Table 4.2. jBPM commands

```
<action name="create_new_process_instance"
  class="org.jboss.soa.esb.services.jbpm.actions.BpmProcessor">
```

```
  <property name="command" value="StartProcessInstanceCommand" />
  <property name="process-definition-name" value="processDefinition2"/>
  <property name="actor" value="FrankSinatra"/>


  <property name="esbToBpmVars">
  <!-- esb-name maps to getBody().get("eVar1") -->
    <mapping esb="eVar1" bpm="counter" default="45" />
    <mapping esb="BODY_CONTENT" bpm="theBody" />
  </property>

</action>
```

Example 4.5. **BpmProcessor** Action Configuration in **jboss-esb.xml**

Two action attributes are required:

1. name

   Use any value for this name attribute, as long as it is unique in the `action pipeline`.

2. class

   Always set this attribute to **org.jboss.soa.esb.services.jbpm.actions.BpmProcessor**.

One can also set these configuration properties:

Property	Description	Required?
command	This must be one of: **NewProcessInstanceCommand**, **StartProcessInstanceCommand**, **GetProcessInstanceVariablesCommand** or **CancelProcessInstanceCommand**.	Yes
process-definition-name	This property is required for the **NewProcessInstanceCommands** and **StartProcessInstanceCommands** if the process-definition-id property is not used. The value of this property should reference the already-deployed process definition for which one wishes to create a new instance. (This property does not apply to the **CancelProcessInstanceCommand**.)	Sometimes
process-definition-id	This is a required property for the **NewProcessInstanceCommands** and **StartProcessInstanceCommands** if the process-definition-name property is not used. The value of this property should refer to the already-deployed process definition for which a new instance is to be created. (This property does not apply to the **CancelProcessInstanceCommand**.)	Sometimes
actor	Use this property to specify the **jBPM actor** identifier. (It only applies to the **NewProcessInstanceCommand** and the **StartProcessInstanceCommand**.)	No
key	Use this property to specify the value of the jBPM key. The key is a string based business key	No

Property	Description	Required?
	property on the process instance. The combination of business key and process definition must be unique if a business key is supplied. The key value can hold an MVEL expression to extract the desired value from the EsbMessage. For example, if one were to have a named parameter called `businessKey` in the body of a message, body.businessKey would be used. (This property only applies to **`NewProcessInstanceCommand`** and **`StartProcessInstanceCommands`**.)	
transition-name	This only applies to **`StartProcessInstanceCommand`**. Use it only if there is more than one transition out of the current node. If this property is not specified, then the default transition out of the node is taken. The default transition is the first transition in the list of transitions defined for that node in the jBPM **`processdefinition.xml`**.	No
esbToBpmVars	This is a optional property for the **`New-`** and **`StartProcessInstanceCommand`**s. It defines a list of variables which need to be extracted from the ESB Message and set into the jBPM context for that particular process instance. The list consists of mapping elements, each of which can have the following attributes: • esb This is a required attribute. Place an MVEL expression in it and use it to extract a value from anywhere in the ESB message. • bpm This is a optional attribute containing the name to use on the jBPM side. (If it is omitted, the Enterprise Service Bus name is used instead.) • default This is a optional attribute which can hold a default value if the ESB's MVEL expression does not find a value set in the ESB message. • bpmToEsbVars This is structurally identical to the esbToBpmVars property above. Use it in conjuction with the **`GetProcessInstanceVariablesCommand`** to map jBPM `process instance variables` (`root token` variables) to the ESB message.	No

Property	Description	Required?
	• reply-to-originator This is an optional property for the **`New`**- and **`StartProcessInstanceCommands`**. Specify a value of **`true`**, to make the process instance store the `ReplyTo/FaultTo` values of the invoking message's end-point references ' within the process instance. These values can then be used within subsequent `EsbNotifier/EsbActionHandler` invocations to deliver a message to the `ReplyTo/FaultTo` addresses.	
jbpmProcessInstId	This is a required ESB message body parameter that applies to the **`GetProcessInstanceVariablesCommand`** and the **`CancelProcessInstanceCommand`** commands. This value must be set as a named parameter on the ESB message's body.	Yes

Table 4.3. Configuration Properties

4.4.1. ESB to jBPM Exception Handling

A `JbpmException` can be thrown from the jBPM Command API when ESB calls are made. This exception is not handled by the integration. Instead, it is passed through to the `action pipeline`'s code. The action pipeline will log the error, send the message to the **`DeadLetterService`**, and send an error message to the `faultTo` end-point reference, if this has been set.

4.5. jBPM-to-JBoss ESB

jBPM-to-JBossESB communication provides one with the capability to use jBPM for *service orchestration*. (Service Orchestration itself is discussed in more detail in the next chapter. Firstly, though, one must learn about the details of this integration.)

The integration implements two jBPM action handler classes, namely**`EsbActionHandler`** and **`EsbNotifier`**. The **`EsbActionHandler`** is a request-reply type action, which sends a message to a service and then awaits a response. The **`EsbNotifier`**, by contrast, does not wait for such a response. The interaction with the Enterprise Service Bus is asynchronous in nature and, therefore, does not block the process instance whilst the service executes.

The **`EsbNotifier`** will be examined first, as it implements a subset of the configuration of the **`EsbActionHandler`**.

4.5.1. ESBNotifier

The **`EsbNotifier`** action should be attached to an outgoing transition. This is so that the jBPM processing can continue whilst the request to the ESB service is processed in the background. One needs to attach the **`EsbNotifier`** to the outgoing transition in the jBPM **`processdefinition.xml`** file.

```
<node name="ShipIt">
  <transition name="ProcessingComplete" to="end">
    <action name="ShipItAction"
```

```
      class="org.jboss.soa.esb.services.jbpm.actionhandlers.EsbNotifier">
      <esbCategoryName>BPM_Orchestration4</esbCategoryName>
      <esbServiceName>ShippingService</esbServiceName>
      <bpmToEsbVars>
        <mapping bpm="entireCustomerAsObject" esb="customer" />
        <mapping bpm="entireOrderAsObject" esb="orderHeader" />
        <mapping bpm="entireOrderAsXML" esb="entireOrderAsXML" />
      </bpmToEsbVars>
    </action>
  </transition>
</node>
```

Example 4.6. *Ship It* Node with **`EsbNotifier`** Attached

The following attributes can be specified:

- name

  This is a required attribute. It is the user-specified name of the action.

- class

  This is a required attribute. Set it to **`org.jboss.soa.esb.services.jbpm.actionhandlers.EsbNotifier`**.

The following sub-elements can be specified.

- **`esbCategoryName`**

  The category name of the ESB service. This is required if the **`reply-to-originator`** functionality is not in use.

- **`esbServiceName`**

  This is the name of the ESB service. It is required if the **`reply-to-originator`** functionality is not in use.

- **`replyToOriginator`**

  This specifies the *reply* or *fault* originator address. Upon its creation, this address was stored in the process instance.

- **`globalProcessScope`**

  This element is an optional Boolean-valued parameter. Use it to set the default scope within which the **`bpmToEsbVars`** variables are to be found. If the **`globalProcessScope`** is set to **`true`**, it searches for the variables within the *token hierarchy* (that is, the **`process-instance`** scope.) If, by contrast, it is set to **`false`**, it retrieves the variables in the scope of the token. If the token itself does not possess a variable for a given name, then the token hierarchy is used to search for that variable. If the element is omitted altogether, the **`globalProcessScope`** defaults to **`false`**.

- **`bpmToEsbVars`**

  This element is optional. It takes a list of sub-elements and uses them to map a jBPM context variable to an ESB Message location. Each of these mapping sub-elements can have the following attributes:

  - **`bpm`**

This is a required attribute. It is the name of the variable within the jBPM context. This name can be an MVEL-type expression and, therefore, can be used to extract a specific field from a larger object. The MVEL root is set to the value of the jBPM **ContextInstance**:

```
<mapping bpm="token.name" esb="TokenName" />
<mapping bpm="node.name" esb="NodeName" />
<mapping bpm="node.id" esb="esbNodeId" />
<mapping bpm="node.leavingTransitions[0].name" esb="transName" />
<mapping bpm="processInstance.id" esb="piId" />
<mapping bpm="processInstance.version" esb="piVersion" />
```

Example 4.7. Mapping jBPM Context Variable to a Location in the ESB Message

The jBPM context-variable names can also be referenced directly.

- **esb**

  This attribute is optional. It is the name of the variable in the Enterprise Service Bus Message. It can be an MVEL-type expression. (In other words, the attribute value TokenName in the example above is equal to **body.TokenName**. A special value called **BODY_CONTENT** "addresses" the body directly.) By default, the variable is set as a named parameter on the body of the ESB Message. In order to omit the **esb** attribute, replace it with the value of the **bpm** attribute.

- default

  This attribute is optional. If the variable is not found within the jBPM context, the value of this field is taken instead.

- **process-scope**

  This attribute is optional. It is a parameter that can contain a Boolean value used to override the setting of the **globalProcessScope** for this mapping.

Important

Always activate debug-level logging when working on the variable mapping configuration.

4.5.2. ESB Action Handler

The **EsbActionHandler** is designed to work as a reply-response type call into the Enterprise Service Bus. Attach it to the node. This is so that the action is called when the node is entered. The **EsbActionHandler** executes, leaving the node waiting for a transition signal, (which can come from any other thread of execution but will normally be sent by the **JBossESB callback** service.)

```
<action name="create_new_process_instance"
  class="org.jboss.soa.esb.services.jbpm.actions.BpmProcessor">

  <property name="command" value="StartProcessInstanceCommand" />
  <property name="process-definition-name" value="processDefinition2"/>
  <property name="actor" value="FrankSinatra"/>


  <property name="esbToBpmVars">
```

```
  <!-- esb-name maps to getBody().get("eVar1") -->
    <mapping esb="eVar1" bpm="counter" default="45" />
    <mapping esb="BODY_CONTENT" bpm="theBody" />
  </property>

</action>
```

Example 4.8. Configuration for the **EsbActionHandler**

The **EsbActionHandler** action's configuration extends to the settings for the **EsbNotifier**. The extensions consist of the following sub-elements:

Property	Description	Required?
esbToBpmVars	This identical to the **esbToBpmVars** property (mentioned in *Section 4.4, " From the Enterprise Service Bus to the jBPM "*) for the **BpmProcessor** configuration. The sub-element defines a list of variables that need to be extracted from the ESB message and set in the Business Process Manager context for that particular process instance. If left unspecified, the **globalProcessScope** value defaults to **true** when the variables are set. The list consists of mapping elements, each of which can have the following attributes: • **esb** This is a required attribute which can contain an MVEL expression. Use it to extract a value and put it into the ESB Message from anywhere. • **bpm** This is an optional attribute containing the name which is to be used by the jBPM. If it is not supplied, then the name in **esb** is used instead. • default Use this is an optional attribute to hold a default value if the **esb** MVEL expression cannot find one that is set in the Enterprise Service Bus message. • **process-scope** This is an optional parameter consisting of a Boolean value. Use it to override the setting of this mapping's e **globalProcessScope**.	No
exceptionTransition	This the name of the transition to utilize if an exception occurs whilst the service is being processed. This element requires the current node to have more than one outgoing transition and	No

Property	Description	Required?
	for one of those transitions to handle *exception processing*.	

Table 4.4. Sub-Elements

A time-out value can be specified for this action (it is optional.) To do so, use a jBPM-native *timer* on the node. *Example 4.9, “Specifying a Time-Out Value for an Action”* demonstrates how to add a time-out value so that, if no signal is received within ten seconds of entering this node, a transition called **`time-out`** is triggered:

```
<timer name='timeout' duedate='10 seconds' transition='time-out'/>
```

Example 4.9. Specifying a Time-Out Value for an Action

4.5.3. jBPM-to-ESB Exception Handling

There are two scenarios in which exceptions can arise:

1. The **`ServiceInvoker`** will throw a `MessageDeliveryException` when delivery of the message to the **Enterprise Service Bus** fails. This happens when the user has mis-spells the name of the service that he or she is trying to reach. This type of exception can be thrown from both the **`EsbNotifier`** and the **`EsbActionHandler`**. It is possible to add an **`ExceptionHandler`** (**`TB-JBPM-USER`**) that can deal with this situation to the jBPM node. (See *http://docs.jboss.com/jbpm/v3/userguide/processmodelling.html* for more information.)

2. The second type of exception occurs when the service receives a request successfully only for something to go wrong during subsequent processing. Only if the call was made from the **`EsbActionHandler`** does it makes sense to report the exception back to **Business Process Manager**. This is due to the fact that, if the call was made from the **`EsbNotifier`**, then jBPM processing has already moved on, and it would, therefore, be of little value to notify the process instance of the problem.

The following scenarios illustrate the kind of error handling that it is now possible to achieve using standard jBPM features: *Figure 4.7, “Three Exception Handling Scenarios: Time-Out, Exception-Transition and Exception-Decision.”*.

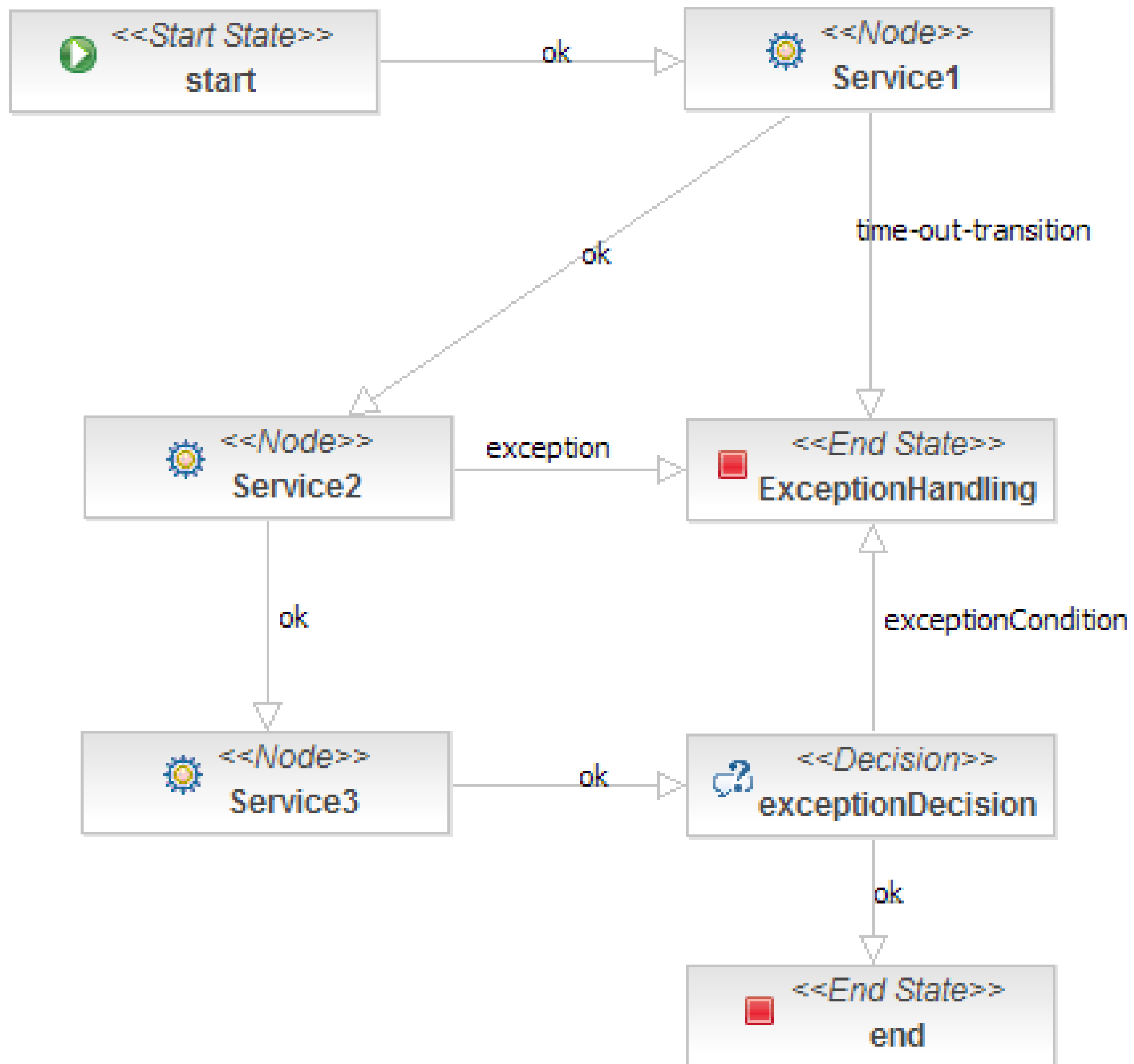


Figure 4.7. Three Exception Handling Scenarios: Time-Out, Exception-Transition and Exception-Decision.

4.5.4. Scenerio One: Time-out

If one is using the **`EsbActionHandler`** action and the node is awaiting a callback, then it may be advantageous to limit the waiting period. To do so, add a timer to the node. That is how **`Service1`** is configured in the diagram. The timer can be set for a certain period, which, in this case, is ten seconds:

```
<node name="Service1">

  <action class=
    "org.jboss.soa.esb.services.jbpm.actionhandlers.EsbActionHandler">
    <esbCategoryName>MockCategory</esbCategoryName>
      <esbServiceName>MockService</esbServiceName>
  </action>

  <timer name='timeout' duedate='10 seconds'
    transition='time-out-transition'/>
  <transition name="ok" to="Service2"></transition>
  <transition name="time-out-transition" to="ExceptionHandling"/>
```

```
</node>
```

Service1 has two outgoing transitions. The first of these is called **ok** whilst the second one is named **time-out-transition**. Under normal processing conditions, the call-back would signal the default transition, which is the **ok**, since it is defined as the first. However, if the processing of the service takes more then ten seconds, the timer will execute. The transition attribute of the timer is set to **time-out-transition**, meaning that this transition will be taken on time-out. Look at the diagram and observe that the processing ends up in the **ExceptionHandling** node. From here, one can perform compensatory work.

4.5.5. Scenerio Two: Exception Transition

One can define an **exceptionTransition** to handle any exceptions that may occur whilst the service is being processed. Doing so results in the **faultTo** end point reference being set on the message, meaning that the Enterprise Service Bus will make a call-back to this node. It is this call-back that signals the **exceptionTransition**. **Service2** has two outgoing transitions: Transition **ok** will be taken under normal processing, whilst the **exception** transition will be taken when the service has, as its name inidcates, thrown an exception during processing.

```
<node name="Service2">
  <action class=
  "org.jboss.soa.esb.services.jbpm.actionhandlers.EsbActionHandler">
     <esbCategoryName>MockCategory</esbCategoryName>
     <esbServiceName>MockService</esbServiceName>
     <exceptionTransition>exception</exceptionTransition>
   </action>
   <transition name="ok" to="Service3"></transition>
   <transition name="exception" to="ExceptionHandling"/>
</node>
```

Example 4.10. Definition of Service Two

In the preceding definition of **Service2**, the action's exceptionTransition is set to “exception.” Note that, in this scenario the process also ends up in the **ExceptionHandling** node.

4.5.6. Scenerio Three: Exception Decision

In order to understand this final scenario, study the configuration of **Service3** and the **exceptionDecision** node that follows it. As can be seen **Service3** processes and completes normally and the default transition out of its node occurs as one would expect. However, at some point during the service execution, an **errorCode** was set, and the **exceptionDecision** node checks if a variable of the same name has been called here:

```
<node name="Service3">
  <action class=
  "org.jboss.soa.esb.services.jbpm.actionhandlers.EsbActionHandler">
     <esbCategoryName>MockCategory</esbCategoryName>
     <esbServiceName>MockService</esbServiceName>
     <esbToBpmVars>
        <mapping esb="SomeExceptionCode" bpm="errorCode"/>
     </esbToBpmVars>
  </action>
  <transition name="ok" to="exceptionDecision"></transition>
</node>

<decision name="exceptionDecision">
```

```
   <transition name="ok" to="end"></transition>
   <transition name="exceptionCondition" to="ExceptionHandling">
      <condition>#{ errorCode!=void }</condition>
   </transition>
</decision>
```

Example 4.11. Definition of Service Three

In the above example, the **esbToBpmVars** mapping element extracts the **errorCode** called **SomeExceptionCode** from the Enterprise Service Bus message body and sets in the jBPM context. (This is provided that the **SomeExceptionCode** is set.) In the next node, named **exceptionDecision**, the **ok** transition is taken if processing is normal, but if a variable called **errorCode** is found in the jBPM context, the **exceptionCondition** transition is taken instead. This is achieved by using the jBPM's *decision node* feature, by means of which transitions can nest within a condition.

To learn more about conditional transitions, please refer to the *jBPM Reference Guide*.

Service Orchestration

Read this chapter to gain an understanding of how to use the integration functionality discussed earlier to perform Service Orchestration with the **Business Process Manager**.

The term, *service orchestration*, refers to the arrangement of business processes. Traditionally, the *Business Process Execution Language* (BPEL) has been used to execute SOAP-based web services. Red Hat recommends using jBPM to orchestrate processes, regardless of their end-point type, within the **JBoss Enterprise SOA Platform**.

5.1. Orchestrating Web Services

Read the *Message Action Guide* to gain an understanding of how the **JBoss Enterprise Service Bus** provides *Web Service-BPEL* (WS-BPEL) support. This *Guide* also provides information about how to configure the main components of this.

Note

JBoss and the JBoss Enterprise Service Bus team also have a special support agreement with *ActiveEndpoints* [1] who built the award-winning **ActiveBPEL** WS-BPEL Engine.

The JBoss Enterprise Service Bus software includes *ActiveBPEL* [2] which can be used to collaborate effectively to provide a WS-BPEL-based orchestration layer on top of a set of services that do not expose Webservice Interfaces. The JBoss Enterprise Service Bus provides the web service integration and **ActiveBPEL** provides the *Process Orchestration*. A number of **Flash**- based "walk-throughs" of this Quick Start are also available online here: *http://labs.jboss.com/jbossesb/resources/tutorials/bpel-demos/bpel-demos.html*.

Note

The **ActiveEndpoints WS-BPEL Engine** has been incompatible with the **JBoss Application Server** since Release 4.0.5. However, it can be deployed and run successfully on **Tomcat**, as demonstrated in the examples below.

5.2. Orchestration Diagram

A flow chart-like design tool must be used to plan and deploy Service Orchestration processes. The **jBPM Integrated Development Environment** (IDE) can be utilised for this purpose. The following shows an example of just such a flow-chart, representing a simplified ordering process. (This example is taken from the **`bpm_orchestration4`** Quick Start which ships with the **JBoss Enterprise Service Bus**.)

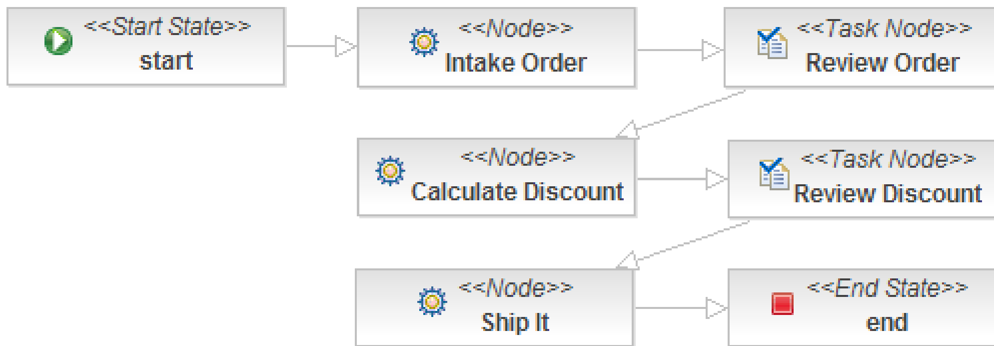


Figure 5.1. Orchestration diagram for the **`bpm_orchestration4`** QuickStart

The classnames of the *Figure 5.1, “Orchestration diagram for the* ***`bpm_orchestration4`*** *QuickStart ”* nodes are JBoss ESB Services. They are called **`Intake Order`**, **`Calculate Discount`** and **`Ship It`**. The regular type of `Node` was used for them, which is why they are labeled with *<<Node>>*. Each of these nodes has the **`EsbActionHandler`** attached to itself. This means that the **Business Process Manager** node will send a request to the service and then it will remain in a "wait" state, until the Enterprise Service Bus calls back with the response from the service. This response can then be used within a **Business Process Manager** context.

For example, when the **`Intake Order`** Service responds, that response is used to populate the **`Review Order`** form. The **`Review Order`** node is a *task node*, which means that it was designed for human interaction. (In this case, someone is required to review the order before the Order Process can occur.)

To create the diagram in *Figure 5.1, “Orchestration diagram for the* ***`bpm_orchestration4`*** *QuickStart ”*, select **File > New > Other** and, from the `Selection` wizard, choose **JBoss jBPM Process Definition**. The wizard will direct one to save the process definition. Red Hat recommends that a single directory be used for each process definition, as this makes the most sense from an organisational point of view. This is because one will usually end up with multiple files associated with each process design.

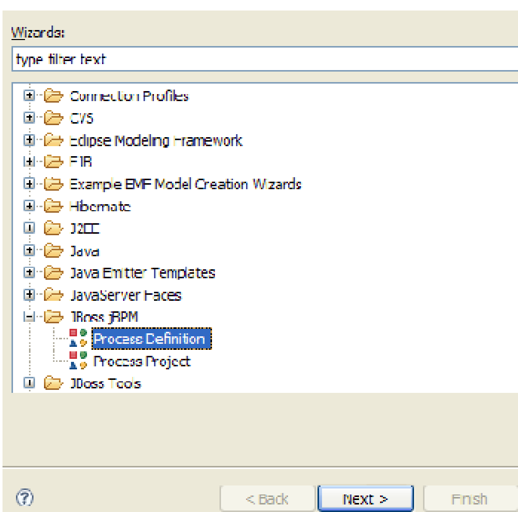


Figure 5.2. Select the New JBoss jBPM Process Definition

After creating a new process definition, start to "drag-and-drop" items from the **jBPM Integrated Development Environment**'s menu palette into the **Process Design** view. One can switch between

the design and source modes if need be, in order to check the XML elements being added, or in order to add those XML fragments that are needed for the integration. (Recently, a new type of node called *ESB Service* was added.)

Before building the order process diagram depicted in *Figure 5.1, “Orchestration diagram for the **bpm_orchestration4** QuickStart ”*, create and test the three services. These are ordinary ESB services and they are defined in the **jboss-esb.xml** file. Study the **jboss-esb.xml** file in the **bpm_orchestration4** Quick Start to learn more about them but the only essential things to know in relation to Service Orchestration are the names and categories of the services themselves. These are as shown in the following sample **jboss-esb.xml** file fragment:

```
            <services>
  <service category="BPM_orchestration4_Starter_Service"
  name="Starter_Service"
  description="BPM Orchestration Sample 4: Use this service to start a
process instance">
   <!-- .... -->
  </service>
  <service category="BPM_Orchestration4" name="IntakeService"
  description="IntakeService: transforms, massages, calculates priority">
   <!-- .... -->
  </service>
  <service category="BPM_Orchestration4" name="DiscountService"
    description="DiscountService">
  </service>
  <service category="BPM_Orchestration4" name="ShippingService"
    description="ShippingService">
   <!-- .... -->
  </service>
</services>
```

Reference these services by using the **EsbActionHandler** or **EsbNotifier** action handlers. (The **EsbActionHandler** should be used when the **Business Process Manager** expects a response, whilst the **EsbNotifier** is to be utilised when none is needed.)

Now that the ESB services are known, drag the `Start` state node into the design view. A new process instance will begin at this node. Next, drag in a `node` (or ESB Service, if one is available.) Name this node `Intake Order`. It is possible to connect the `Start` and the `Intake Order` nodes by selecting **Transition** from the menu and then clicking on them both. (An arrow connecting them should appear. It will be pointing towards the first `Intake Order`.)

Now add the Service and Category names to the Intake Node. Select the **Source** view. The source code of the `Intake Order` node should look like this:

```
<node name="Intake Order">
    <transition name="" to="Review Order"></transition>
</node>
```

Next, add the **EsbHandlerAction** class reference, then the sub-element configurations for the Service Category and Name, the `BPM_Orchestration4` and the `IntakeService`, as per this example code:

```
<node name="Intake Order">
  <action name="esbAction" class=
```

```
    "org.jboss.soa.esb.services.jbpm.actionhandlers.EsbActionHandler">
    <esbCategoryName>BPM_Orchestration4</esbCategoryName>
    <esbServiceName>IntakeService</esbServiceName>
    <!-- async call of IntakeService -->
  </action>
  <transition name="" to="Review Order"></transition>
</node>
```

Having done that, send some **Business Process Manager** context variables along with the service call. In this example, there is a variable named entireOrderAsXML, which is to be set in the default position on the body of the Enterprise Service Bus message. To do this, simply add the following code:

```
<bpmToEsbVars>
  <mapping bpm="entireOrderAsXML" esb="BODY_CONTENT" />
</bpmToEsbVars>
```

This will cause the XML-based contents of the entireOrderAsXML variable to end up in the body of the Enterprise Service Bus message. This, in turn, means that the **`IntakeService`** can now access the message and can process it, by letting it flow through each action in the `Pipeline`. When the last action is reached, the replyTo property is checked and the Enterprise Service Bus message is sent to the **`JBpmCallBack`** service. The latter makes a call back into the **Business Process Manager**, signaling the transition from the `Intake Order` node to the next one (`Review Order`.) This time, one will want to send some variables from the Enterprise Service Bus message to the **Business Process Manager**. Note that entire objects can be sent, as long both contexts can load the object's class. In order to retain the ability to "map back" to the **Business Process Manager**, add an esbToEsbVars element.

Putting all of these components together will result in the following code:

```
<node name="Intake Order">
<action name="esbAction" class=
 "org.jboss.soa.esb.services.jbpm.actionhandlers.EsbActionHandler">
<esbCategoryName>BPM_Orchestration4</esbCategoryName>
<esbServiceName>IntakeService</esbServiceName>
<bpmToEsbVars>
<mapping bpm="entireOrderAsXML" esb="BODY_CONTENT" />
</bpmToEsbVars>
<esbToBpmVars>
<mapping esb="body.entireOrderAsXML" bpm="entireOrderAsXML"/>
<mapping esb="body.orderHeader" bpm="entireOrderAsObject" />
<mapping esb="body.customer" bpm="entireCustomerAsObject" />
<mapping esb="body.order_orderId" bpm="order_orderid" />
<mapping esb="body.order_totalAmount" bpm="order_totalamount" />
<mapping esb="body.order_orderPriority" bpm="order_priority" />
<mapping esb="body.customer_firstName" bpm="customer_firstName" />
<mapping esb="body.customer_lastName" bpm="customer_lastName" />
<mapping esb="body.customer_status" bpm="customer_status" />
</esbToBpmVars>
</action>
<transition name="" to="Review Order"></transition>
</node>
```

When this service returns, the following variables will be stored in the **Business Process Manager** context:

- `entireOrderAsXML`,
- `entireOrderAsObject` and

- `entireCustomerAsObject`.

In addition, for demonstration purposes, there are also some flattened variables:

- `order_orderid`,
- `order_totalAmount`,
- `order_priority`,
- `customer_firstName`,
- `customer_lastName` and
- `customer_status`.

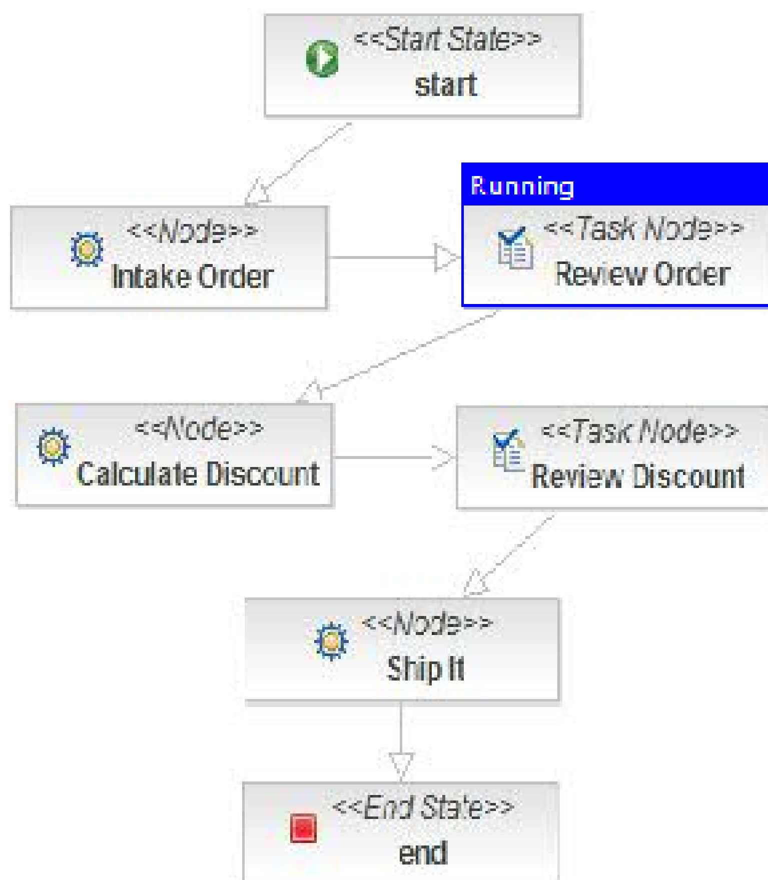


Figure 5.3. The Order Process Has Reached the “Review Order” Node

A human will be required to review the order process. Therefore, add a `Task Node` and the task `Order Review`. These need to be performed by someone with the actor_id **`user`**. The XML fragment should look like this:

```
<task-node name="Review Order">
<task name="Order Review">
<assignment actor-id="user"></assignment>
 <controller>
<variable name="customer_firstName"
access="read,write,required"></variable>
```

```
<variable name="customer_lastName" access="read,write,required">
<variable name="customer_status" access="read"></variable>
<variable name="order_totalamount" access="read"></variable>
<variable name="order_priority" access="read"></variable>
<variable name="order_orderid" access="read"></variable>
<variable name="order_discount" access="read"></variable>
<variable name="entireOrderAsXML" access="read"></variable>
</controller>
</task>
<transition name="" to="Calculate Discount"></transition>
</task-node>
```

Create an XHTML data-form. Do this so that these variables can display in a form in the **jbpm-console** (see the **`Review_Order.xhtml`** file in the *bpm_orchestration4* Quick Start [JBESB-QS] for more information about this.) "Tie" this data-form to the TaskNode via the **`forms.xml`** file:

```
<forms>
<form task="Order Review" form="Review_Order.xhtml"/>
<form task="Discount Review" form="Review_Order.xhtml"/>
</forms>
```

Note that, in this case, the same form is applied to two task nodes. The variables are referenced in the **Review Order** form as shown in the following sample code. (This, in turn, references the variables that are set in the **Business Process Manager** context.)

```
<jbpm:datacell>
<f:facet name="header">
<h:outputText value="customer_firstName"/>
</f:facet>
<h:inputText value="#{var['customer_firstName']}" />
</jbpm:datacell>
```

When the process reaches the `Review Node`, (depicted in *Figure 5.3, “The Order Process Has Reached the “Review Order” Node”*), the user can log into the **jbpm-console** and click on "Tasks" to see a list of items, (as shown in *Figure 5.4, “The Task List for User 'User'”*.) He or she can examine the task by clicking on it. A form will appear, (as seen in *Figure 5.5, “The 'Order Review' form”*.) He or she can then update some of the values and conclude by clicking **Save and Close**, at which point the process will move on to the next node.

Manage: Processes | Tasks

Tasks — First Prev - Page 1 of 1 - Next Last

ID	Name	Pooled Actors	Assigned To	Status	Start Date	End Date	Actions
				N R S C			Apply Filter Clear Filter
1	Order Review		user	Not Started			Examine Suspend Start

Figure 5.4. The Task List for User 'User'

Order Review	
customer_firstName	Rex
customer_lastName	Myers
customer_status	60
order_totalamount	64.92
order_priority	3
order_orderid	2
order_discount	
entireOrder	<Order netAmount="59.97
Actions	Save / Cancel / Save and Close

Figure 5.5. The 'Order Review' form

The next one is the `Calculate Discount` node. This is, once again, an ESB service node, the configuration file for which looks like this:

```
<node name="Calculate Discount">
<action name="esbAction" class="
org.jboss.soa.esb.services.jbpm.actionhandlers.EsbActionHandler">
<esbCategoryName>BPM_Orchestration4</esbCategoryName>
<esbServiceName>DiscountService</esbServiceName>
<bpmToEsbVars>
<mapping bpm="entireCustomerAsObject" esb="customer" />
<mapping bpm="entireOrderAsObject" esb="orderHeader" />
<mapping bpm="entireOrderAsXML" esb="BODY_CONTENT" />
</bpmToEsbVars>
<esbToBpmVars>
<mapping esb="order"
bpm="entireOrderAsObject" />
<mapping esb="body.order_orderDiscount" bpm="order_discount" />
</esbToBpmVars>
</action>
<transition name="" to="Review Discount"></transition>
</node>
```

The service receives both the **`customer`** and the **`orderHeader`** objects, as well as the **`entireOrderAsXML`** data. It then computes a discount. The response maps the **`body.order_orderDiscount`** value onto a **Business Process Manager** context variable called order_-discount. The process is then signaled, which tells it to move to the `Review Discount` node:

Discount Review	
customer_firstName	Rex
customer_lastName	Myers
customer_status	60
order_totalamount	64.92
order_priority	3
order_orderid	2
order_discount	8.5
entireOrder	<Order netAmount="59.97
Actions	Save Cancel Save and Close

Figure 5.6. The "Discount Review" Form

Here, the user is asked to review the discount, which is set to a value of 8.5 (see *Figure 5.6, “The "Discount Review" Form”*.) When he or she clicks **Save and Close**, the process moves to the `Ship It` node, which is, once again, an ESB service. To circumvent the Order process before the `Ship It` service completes, use the **`EsbNotifier`** action handler by attaching it to the outgoing transition:

```
<node name="ShipIt">
<transition name="ProcessingComplete" to="end">
<action name="ShipItAction" class=
"org.jboss.soa.esb.services.jbpm.actionhandlers.EsbNotifier">
<esbCategoryName>BPM_Orchestration4</esbCategoryName>
<esbServiceName>ShippingService</esbServiceName>
 <bpmToEsbVars>
<mapping bpm="entireCustomerAsObject" esb="customer" />
 <mapping bpm="entireOrderAsObject" esb="orderHeader" />
 <mapping bpm="entireOrderAsXML" esb="entireOrderAsXML" />
 </bpmToEsbVars>
 </action>
</transition>
</node>
```

After notifying the **`ShippingService`**, the order process moves to the "end" state and terminates. (The **`ShippingService`** itself may still be finishing.) **JBoss Rules** is used in the **`bpm_orchestration4`** file to determine whether this order should be shipped via the "normal" or "express" method.

5.3. Process Deployment and "Instantiation"

In the previous section, an assumption was made that an instance of the process definition was running. This was in order to explain the process flow. However, now that the

processdefinition.xml file has been created, it can be deployed to the **Business Process Manager**, by using any one of the following: the integrated development environment, **ant** or the **jbpm-console**. (The integrated development environment will be used in the following example.)

The following files will be deployed:

- **Review_Order.xhtml**,
- **forms.xml**,
- **gpd.xml**,
- **processdefinition.xml** and
- **processimage.jpg**.

The integrated development environment creates a **PAR** archive and deploys it to the **Business Process Manager**'s database.

Warning

Red Hat recommends against deploying Java code in **PAR** archives as it may cause class-loading issues. Instead, use either **JAR** or **ESB** archives to deploy classes.

Create a new process instance once the process definition is deployed. (Note that **StartProcessInstanceCommand** can be used. The allows one to create a process instance with some pre-set initial values.) Study this code sample:

```
<service category="BPM_orchestration4_Starter_Service"
name="Starter_Service"
description="BPM Orchestration Sample 4: Use this service to start a
 process instance">
<listeners>
</listeners>
<actions>
<action name="setup_key" class=
"org.jboss.soa.esb.actions.scripting.GroovyActionProcessor">
<property name="script"
value="/scripts/setup_key.groovy" />
</action>
<action name="start_a_new_order_process" class=
"org.jboss.soa.esb.services.jbpm.actions.BpmProcessor">
<property name="command"
value="StartProcessInstanceCommand" />
<property name="process-definition-name"
value="bpm4_ESBOrderProcess" />
<property name="key" value="body.businessKey" />
<property name="esbToBpmVars">
 <mapping esb="BODY_CONTENT" bpm="entireOrderAsXML" />
</property>
</action>
</actions>
</service>
```

The new process instance is now invoked and using a script. The jBPM key is set to the value of OrderId from an incoming order XML file. This same XML is subsequently put into a **Business Process Manager** context, through use of the the **esbToBpmVars** mapping. In

the **bpm_orchestration4** Quick Start, the XML came from the **Seam DVD Store** and the **SampleOrder.xml** looks like this:

```
<Order orderId="2" orderDate="Wed Nov 15 13:45:28 EST 2006" statusCode="0"
 netAmount="59.97" totalAmount="64.92" tax="4.95">
<Customer userName="user1" firstName="Rex" lastName="Myers" state="SD"/>
<OrderLines>
<OrderLine position="1" quantity="1">
<Product productId="364" title="Gandhi"
price="29.98"/>
</OrderLine>
<OrderLine position="2" quantity="1">
<Product productId="299" title="Lost Horizon" price="29.99"/>
</OrderLine>
</OrderLines>
</Order>
```

Note that both the Enterprise Service Bus and the Business Process Manager deployments are "hot." The jBPM has a special feature that results in process deployments being "versioned": newly created process instances will use the latest version, whilst existing ones will finish using the process deployment on which they were started.

5.4. Conclusion

From studying the examples in this chapter, you have learned how the **Business Process Manager** can be used to orchestrate services and, in addition, perform "Human Task Management." Note that you are free to use any jBPM feature. For instance, look at the Quick Start entitled **bpm_orchestration2**, in order to learn how to use the Business Process Manager's `fork` and `join` functionality.

Message Transformation

The **JBoss Enterprise Service Bus** supports *message data transformation* functionality through several mechanisms.

6.1. Smooks

Smooks is, amongst other things, a *Fragment-Based Data Transformation and Analysis Tool*. It can "understand" a wide range of source and target data formats, including XML, EID, CSV and Java. It features a wide range of data processing and manipulation functionality. Many transformation technologies are supported, all within this single framework.

The **`SmooksAction`** component supports message transformation on the **JBoss Enterprise Service Bus**. Use this ESB Action component to "plug" the Smooks Data Transformation/Processing Framework into an ESB Action Processing Pipeline.

Samples and Tutorials

There are a number of quick start examples demonstrating transformations included with the **JBoss Enterprise SOA Platform**. These can be found in the **`samples/quickstarts`** directory. (The name of each transformation Quick Start sub-directory is prefixed with the word **`transform_`**.)

The *JBoss SOA Platform Programmers' Guide* contains further detailed information about this topic. It also provides links to additional resources that can be found on the **Smooks** website.

Note

Some of the quick starts use the old **`SmooksTransformer`** `action class` instead of its successor, **`SmooksAction`**. Please bear in mind that **`SmooksTransformer`** will be deprecated in a future release.

6.2. XSL Transformations

The **`XstlAction`** class supports XSLT transformations. Read the section entitled “XSLTAction” in the *Programmers' Guide* to learn more about this.

The Message Store

The Enterprise Service Bus' *MessageStore* mechanism has been designed for the purpose of audit-tracking. As with other ESB services, it is *pluggable*, which means that the developer can plug in his or her own persistence mechanism should there be the need to do so. (A database persistence mechanism is supplied.) For instance, to create a file persistence mechanism, simply code a service to create it, then over-ride the default behavior with a configuration change.

Note

Note that this **`MessageStore`** is a base implementation only. Red Hat will be working with the community and partners to improve the functionality of this software to the point where, at a future point in time, it will support advanced auditing and management requirements. At present, this program is solely intended as a starting point.

Important

The **`MessageStore`** is also used for holding messages that need to be re-delivered in the event of a failure. Additional information on this topic si found in the *Programmers' Guide*.

7.1. Message Store Interface

The **`MessageStore`** is responsible for reading and writing messages upon request. Each message must be uniquely identified within the context of the store. (Each **`MessageStore`** implementation uses a uniform resource identifier to accomplish this. The URI acts as the “key” for messages in the database.)

```
public interface MessageStore
{
  public MessageURIGenerator getMessageURIGenerator();
  public URI addMessage (Message message, String classification)
        throws MessageStoreException;
  public Message getMessage (URI uid) throws MessageStoreException;
  public void setUndelivered(URI uid) throws MessageStoreException;
  public void setDelivered(URI uid) throws MessageStoreException;
  public Map<URI, Message> getUndeliveredMessages(String classification)
         throws MessageStoreException;
  public Map<URI, Message> getAllMessages(String classification)
         throws MessageStoreException;
public Message getMessage (URI uid, String classification)
         throws MessageStoreException;
public int removeMessage (URI uid, String classification)
         throws MessageStoreException;
}
```

Figure 7.1. The **`org.jboss.soa.esb.services.persistence.MessageStore`** Interface

Important

Each **MessageStore** implementation uses a different format for uniform resource identifiers.

Messages can be stored using a classification derived from **addMessage**. If the classification is not defined, then it is up to the individual implementation of the **MessageStore** to determine for itself how it will store the message. Furthermore, the classification is only a guide: one's implementation can ignore this field if necessary.

It is dependent on the implementation as to whether or not the **MessageStore** imposes any kind of concurrency control on individual messages. Therefore, use the **removeMessage** operation with care.

Do not use the **setUndelivered/setDelivered** commands or other associated operations unless they are applicable. This is because the current **MessageStore** interface is designed to support both audit trail and re-delivery functionality.

The **org.jboss.internal.soa.esb.persistence.format.db.DBMessageStoreImpl** class provides the default implementation of the **MessageStore**. . The methods in this implementation make the required database connections via a pooled database manager, called **DBConnectionManager**.

Use the **MessageActionGuide** and the **MessagePersister** actions to override the **MessageStore** implementation.

Note

The **MessageStore** interface does not currently support transactions. Any use of the **MessageStore** within the scope of a global transaction will, therefore, be unco-ordinated. The implication of this is that each **MessageStore** update or read will be undertaken separately and independently. However, future versions of the software shall provide control over whether or not specific interactions are to be conducted within the scope of an "enclosing" transactional context.

7.2. Configuring the Message Store

To configure the **MessageStore**, firstly over-ride the default service implementation. Do this by editing the settings found in the **jbossesb-properties.xml** file:

```
<properties name="dbstore">
  <!--  connection manager type -->
  <property name="org.jboss.soa.esb.persistence.db.conn.manager" value=
"org.jboss.internal.soa.esb.persistence.manager.StandaloneConnectionManager"/>
    <!-- this property is only used for the j2ee connection manager -->
    <property name="org.jboss.soa.esb.persistence.db.datasource.name"
      value="java:/JBossesbDS"/>
    <!-- standalone connection pooling settings -->
    <!--  mysql
    <property name="org.jboss.soa.esb.persistence.db.connection.url"
      value="jdbc:mysql://localhost/jbossesb"/>
    <property name="org.jboss.soa.esb.persistence.db.jdbc.driver"
      value="com.mysql.jdbc.Driver"/>
    <property name="org.jboss.soa.esb.persistence.db.user"
      value="kstam"/> -->
```

```
    <!--  postgres
    <property name="org.jboss.soa.esb.persistence.db.connection.url"
      value="jdbc:postgresql://localhost/jbossesb"/>
    <property name="org.jboss.soa.esb.persistence.db.jdbc.driver"
      value="org.postgresql.Driver"/>
    <property name="org.jboss.soa.esb.persistence.db.user"
      value="postgres"/>
    <property name="org.jboss.soa.esb.persistence.db.pwd"
      value="postgres"/> -->
    <!-- hsqldb -->
    <property name="org.jboss.soa.esb.persistence.db.connection.url"
      value="jdbc:hsqldb:hsql://localhost:9001/jbossesb"/>
    <property name="org.jboss.soa.esb.persistence.db.jdbc.driver"
      value="org.hsqldb.jdbcDriver"/>
    <property name="org.jboss.soa.esb.persistence.db.user" value="sa"/>
    <property name="org.jboss.soa.esb.persistence.db.pwd" value=""/>
    <property name="org.jboss.soa.esb.persistence.db.pool.initial.size"
      value="2"/>
    <property name="org.jboss.soa.esb.persistence.db.pool.min.size"
      value="2"/>
    <property name="org.jboss.soa.esb.persistence.db.pool.max.size"
      value="5"/>
    <!--table managed by pool to test for valid connections
        created by pool automatically -->
    <property name="org.jboss.soa.esb.persistence.db.pool.test.table"
      value="pooltest"/>
    <property name="org.jboss.soa.esb.persistence.db.pool.timeout.millis"
      value="5000"/>
</properties>
```

In this file, there is a section entitled “dbstore.” It is here that all of the settings required by the **Message Store**'s database implementation are located. Modify the standard settings, like URL, db user, password, pool size and so forth here.

The scripts for the "required database" schema are found in the **lib/jbossesb.esb/message-store-sql/<db_type>/create_database.sql** file in the **JBoss Enterprise Service Bus** installation.

The SQL code in the following sample shows the structure of this file:

```
CREATE TABLE message
(
   uuid varchar(128) NOT NULL,
   type varchar(128) NOT NULL,
   message text(4000) NOT NULL,
   delivered varchar(10) NOT NULL,
   classification varchar(10),
   PRIMARY KEY (`uuid`)
);
```

Example 7.1. Sample SQL for Message Store Table Creation

The `UUID` column is used to store a unique key for the message. It takes the form of a standard uniform resource identifier. Message keys look like this:

```
urn:jboss:esb:message:UID: + UUID.randomUUID()_
```

This logic exploits the UUID's random number generator. The `type` will be equal to that of the stored message. The **JBoss Enterprise Service Bus** currently ships with two different `type`, these being `JBOSS_XML` and `JAVA_SERIALIZED`, respectively.

The `message` column contains the contents of the actual message itself.

The database message store implementation supplied with the Enterprise Service Bus works by invoking a connection to one's already-configured database. Both a stand-alone connection manager, and another for using a JNDI data-source, are also supplied with the ESB.

To configure the database connection manager, add its implementation details to the **`jbossesb-properties.xml`** file. The properties to change are as follows:

```
<!--  connection manager type -->
<property name="org.jboss.soa.esb.persistence.db.conn.manager"
  value="org.jboss.internal.soa.esb.persistence.format.db.Standalone
ConnectionManager"/>
<!--  property name="org.jboss.soa.esb.persistence.db.conn.manager"
value="org.jboss.soa.esb.persistence.manager.J2eeConnectionManager"/ -->
<!-- this property is only used for the j2ee connection manager -->
<property name="org.jboss.soa.esb.persistence.db.datasource.name"
  value="java:/JBossesbDS"/>
```

The two pre-supplied connection managers for the database pool are: **`org.jboss.soa.esb.persistence.manager.J2eeConnectionManager`** and **`org.jboss.soa.esb.persistence.manager.StandaloneConnectionManager`**.

The *Stand-Alone Manager* uses *C3PO* to manage the connection pooling logic whilst the *J2eeConnectionManager*, by contrast, employs a data-source. Use this when deploying Enterprise Service Bus end points inside a container such as the **JBoss Application Server** or **Tomcat**.

Another option is to "plug in" a custom connection pool manager. Firstly, implement this interface: **`org.jboss.internal.soa.esb.persistence.manager.ConnectionManager`**. Next, update the **`Properties`** file with the name of the new class. Having done so, the *Connection Manager Factory* will now be able to utilize the new implementation.

updated Security

JBoss ESB services can be made secure, in the sense that one can configure the platform so that they will only be executed if authentication succeeds and the caller also has the suitable level of authority.

There are two ways in which to invoke a service:

1. through a gateway
2. directly via the **Enterprise Service Bus** via the *ServiceInvoker*.

When one uses the first option, the gateway is responsible for obtaining the security information needed to authenticate the caller. It does this by extracting the information from the transport that it handles. Once it has obtained this information, it creates an authentication request that is encrypted and then passed to the **Enterprise Service Bus**.

If one uses the **`ServiceInvoker`** instead, the gateway is not utilised. Instead, it becomes the client's responsibility to create the authentication request prior to invoking the service.

Both of these options will be discussed in more detail the following sections.

The default security implementation is based on the *Java Authentication and Authorization Service* (JAAS). (It can be reconfigured if one wishes to use an alternative system.) The following sections describe how to set the JAAS security components.

8.1. Security Service Configuration

To configure the security service, edit the **`jbossesb-properties.xml`** file.

```
<properties name="security">
<property name="org.jboss.soa.esb.services.security.implementationClass"
value="org.jboss.internal.soa.esb.services.security.JaasSecurityService"/>

<property name="org.jboss.soa.esb.services.security.callbackHandler"
value=
"org.jboss.internal.soa.esb.services.security.UserPassCallbackHandler"/>

<property name="org.jboss.soa.esb.services.security.sealAlgorithm"
value="TripleDES"/>

<property name="org.jboss.soa.esb.services.security.sealKeySize"
value="168"/>

<property name="org.jboss.soa.esb.services.security.contextTimeout"
value="30000"/>

<property name=
"org.jboss.soa.esb.services.security.contextPropagatorImplemtationClass"
value=
"org.jboss.internal.soa.esb.services.security.JBossASContextPropagator"/>

<property name="org.jboss.soa.esb.services.security.publicKeystore"
value="/publicKeyStore"/>


<property name="org.jboss.soa.esb.services.security.publicKeystorePassword"
```

```
value="testKeystorePassword"/>

<property name="org.jboss.soa.esb.services.security.publicKeyAlias"
value="testAlias"/>

<property name="org.jboss.soa.esb.services.security.publicKeyPassword"
value="testPassword"/>

<property name="org.jboss.soa.esb.services.security.publicKeyTransformation"
value="RSA/ECB/PKCS1Padding"/>

</properties>
```

Property	Description	Require
org.jboss.soa.esb.services.security.implementationClass	This is the "concrete" *SecurityService* implementation that should be used. The default setting is **JaasSecurityService**.	Yes
org.jboss.soa.esb.services.security.callbackHandler	This is a default **CallbackHandler** implementation, utilised when a JAAS-based **SecurityService** is employed. See “Customizing Security” for more information about the **CallbackHandler** property.	No
org.jboss.soa.esb.services.security.sealAlgorithm	This is the algorithm to use when "sealing" the **SecurityContext**.	No
org.jboss.soa.esb.services.security.sealKeySize	This is the size of the secret/ symmetric key used to encrypt/decrypt the **SecurityContext**.	No
org.jboss.soa.esb.services.security.contextTimeout	This is the amount of time (in milliseconds) for which a security context is valid. A global setting, this may be over-ridden on a per-service basis. To do so, specify the property of the same name that exists on the security element in the **jboss-esb.xml** file.	No

Property	Description	Require
`org.jboss.soa.esb.services.security.contextPropagatorImplementationClass`	Use this to configure a global **`SecurityContextPropagator`**. (For more details on the **`SecurityContextPropagator`**, please refer to the section on "Security Context Propagation."	No
`org.jboss.soa.esb.services.security.publicKeystore`	This is the path to the *Keystore* which holds the keys used to encrypt and decrypt that data which is external to the Enterprise Service Bus. The Keystore is used to encrypt the **`AuthenticationRequest`**.	No
`org.jboss.soa.esb.services.security.publicKeystorePassword`	This is the password for the public keystore.	No
`org.jboss.soa.esb.services.security.publicKeyAlias`	This is the alias to use.	No
`org.jboss.soa.esb.services.security.publicKeyPassword`	This is the password for the alias if one was specified upon creation.	No
`org.jboss.soa.esb.services.security.publicKeyPassword`	This is a cipher transformation. It is in this format: **`algorithm/mode/padding`**. If this is not specified, the "keys" algorithm will be used by default.	No

Table 8.1. **`jbossesb-properties.xml`** Security Settings

Configure the JAAS log-in modules via the ***`$SOA_ROOT`*****`/server/`*****`$PROFILE`*****`/conf/login-config.xml`** file. Use either a pre-configured one or create a custom solution.

Warning

The **JBoss Enterprise Service Bus** ships with an example key-store. Do not be use this in a production environment. It is only provided as a sample to help users achieve a working security configuration "out-of-the-box."

Note

One can update the sample key-store with a custom-generated pairs of keys.

8.1.1. Configuring Security on Services

Security is configured on a per-service basis. An ESB service can be declared secure and requiring authentication.

To configure a service, find it in the **jbossesb.xml** file and add a security element there. This code sample demonstrates this:

```
<service category="Security" name="SimpleListenerSecured">
   <security moduleName="messaging" runAs="adminRole" rolesAllowed=”adminRole,  normalUsers”

  callbackHandler="org.jboss.internal.soa.esb.services.security.UserPassCallbackHandler">
     <property name="property1" value="value1"/>
     <property name="property2" value="value2"/>
   </security>
   ...
</service>
```

Property	Description	Required?
moduleName	This is a named module that exists in the **conf/login-config.xml** file.	No
runAs	This is the runAs role.	No
rolesAllowed	This is an comma-separated list of those roles that have been granted the ability to execute the service. This is used as a check that is performed after a caller has been authenticated, in order to verify that they are indeed belonging to one of the roles specified. The roles will have been assigned after a successful authentication by the underlying security mechanism.	No
callbackHandler	This is the **CallbackHandler** that will override that which was defined in the **jbossesb-properties.xml** file.	No
property	These are optional properties that, once defined, will be made available to the **CallbackHandler** implementation.	No

Table 8.2. Security Properties

Property	Description	Required?
org.jboss.soa.esb.services.security.contextTimeout	This property lets the service over-ride the global security context time-out (milliseconds) that is specified in the **jbossesb-properties.xml** file.	No
org.jboss.soa.esb.services.security.contextPropagatorImplementationClass	This property lets the service to override the "global security context propagator" class implementation, that is specified in the **jbossesb-properties.xml** file.	No

Table 8.3. Security Property Over-rides:

This example demonstrates how to override global configuration settings:

```
<security moduleName="messaging"
  runAs="adminRole" rolesAllowed="adminRole">

<property
    name="org.jboss.soa.esb.services.security.contextTimeout"
    value="50000"/>

<property name=
"org.jboss.soa.esb.services.security.contextPropagatorImplementationClass"
    value="org.xyz.CustomSecurityContextPropagator" />

</security>
```

8.2. Authentication

Security information needs to be provided in order to authenticate a caller. If the call to the service is coming through a gateway, then that gateway will extract the required information from the transport with which it works. For a web service call, this would entail extracting either the **UsernameToken** or the **BinarySecurityToken** from the security element in the SOAP header.

An authentication process will run if one service requiring authentication needs to call upon another. Therefore, having a chain of services that are all configured for authentication will cause multiple authentications to be performed. In order to minimize the overhead, the Enterprise Service Bus will store an encrypted **SecurityContext**. This **SecurityContext** will be passed on to the ESB message object between services. If the ESB detects that a Message has a **SecurityContext**, it will check that it is still valid and, if so, re-authentication is not performed. Note that the **SecurityContext** is only valid for a single Enterprise Service Bus node. If the message is routed to a different ESB node, a re-authentication will be required.

8.2.1. Authentication Request

An **AuthenticationRequest** is designed to carry the security information needed for authentication between either a gateway and a service or between two services.

An instance of this class should be set on the message object prior to that service which has been configured for authentication being called:

```
byte[] encrypted = PublicCryptoUtil.INSTANCE.encrypt((Serializable)
 authRequest);
message.getContext.setContext(SecurityService.AUTH_REQUEST, encrypted);
```

Note that the authentication context is encrypted and then set within the message context. It will later be de-crypted by the Enterprise Service Bus in order to perform the authentication. See *Section 8.1, “ Security Service Configuration ”* for information on how to configure the *public keystore* for this purpose.

The **security_basic** Quick Start shows an example of an external client in use. The Quick Start explains shows how to prepare the message before using the **ServiceInvoker**. (See the **SendEsbMessage** class for more information.) It also demonstrates how one can configure the **jbossesb-properties.xml** file for client usage.

8.3. The JBoss Enterprise Service Bus Security Context

In the JBoss Enterprise Service Bus, a **`SecurityContext`** is an object that is local to a specific ESB node or to the Java Virtual Machine of that node. The **`SecurityContext`** is created after a successful authentication has be performed. It will be used locally in the Enterprise Service Bus in which it was created. This is in order to avoid having to re-authenticate with every call.

A time-out (in milliseconds) is specified for the context in which it is valid. This time value can be either specified globally (by editing the **`jbossesb-properties.xml`** file) or it can be over-ridden on a per-service basis. This latter is achieved by specifying it in the **`jboss-esb.xml`** file. Additional information about this topic can be found in *Section 8.1.1, “Configuring Security on Services”* and *Section 8.1, “ Security Service Configuration ”*.

8.4. Security Context Propagation

In this case, the term "*propagation*" refers to the process of propagating security context information in a way specific to an external system. For example, one might want to use the same credentials to call both the Enterprise Service Bus and an *Enterprise Java Beans* (EJB) method. One can accomplish this by specifying a **`SecurityContextPropagator`**, which, as its name suggests, will perform the security-context propagation specific to the destination environment.

A **`SecurityContextPropagator`** can be configured either globally (by specifying the **`org.jboss.soa.esb.services.security.contextPropagatorImplementationClass`** class in the **`jbossesb-properties.xml`** file) or, on a per-service basis (by specifying that same property in the **`jboss-esb.xml`** file.) *Section 8.1.1, “Configuring Security on Services”* and *Section 8.1, “ Security Service Configuration ”* contain more examples of this.

Implementations of **`SecurityContextPropagator`**

Package: **`org.jboss.internal.soa.esb.services.security`** **`Class: JBossASContextPropagator`**

> This will pass on the security credentials to a JBoss Application Server. If one has the need to create one's own implementation, a class must be written that implements **`org.jboss.internal.soa.esb.services.security.SecurityContextPropagator`**. After that, the new implementation must be specified in either the **`jbossesb-properties.xml`** or the **`jboss-esb.xml`** file, as was noted above.

8.5. Customising Security

The default security implementation in the JBoss Enterprise Service Bus is based on JAAS. It is named **`JaasSecurityService`**. Custom log-in modules can be added to the **`conf/login-config.xml`** file of a JBoss Application Server.

Since different log-in modules will require different information, the callback handler to be used can be specified in the security configuration for that service. This can be accomplished by specifying the *callbackHandler* attribute belonging to the security element defined on the service.

The **`callbackHandler`** should specify a "fully qualified" classname for that class which implements the **`EsbCallbackHandler`** interface:

```
public interface EsbCallbackHandler extends CallbackHandler
{
  void setAuthenticationRequest(final AuthenticationRequest authRequest);
  void setSecurityConfig(final SecurityConfig config);
```

```
}
```

The **AuthenticationRequest** class will contain both the principal and the credentials needed to authenticate a caller.

The **SecurityConfig** class will grant access to the security configuration specified in the **jboss-esb.xml** file.

Both of these are made available to the **CallbackHandler**. It can use them to populate the callback instances that are required by the log-in module.

8.6. Provided Log-in Modules

This section lists the log-in modules provided with JBoss Enterprise Service Bus. Please note that all of the log-in modules available with JBoss Application Server are also available here. Custom log-in modules should also be easy to add.

8.6.1. Certificate Log-in Module

This log-in module performs authentication by verifying the certificate that is passed with the call to the Enterprise Service Bus against a certificate held in a local keystore.

Upon successful authentication, the certificate's *Common Name* (CN) creates a *principal*. If role-mapping is in use, then it is the Common Name that will be used for this. Please refer to *Section 8.6.2, “ Role Mapping ”* for details about the role-mapping functionality.

```
<security moduleName="CertLogin" rolesAllowed=”worker”
  callbackHandler="org.jboss.soa.esb.services.security.auth.loginUserPass
CallbackHandler">
  <property name="alias" value="certtest"/>
</security>
```

CertificateLogin Module Properties

moduleName
: This identifies the specific JAAS log-in module to use. This module will be specified in the JBoss Application Server's **login-config.xml** file.

rolesAllowed
: This comma-separated list contains those roles that have permission to execute the service.

alias
: This is the alias for which to search in the local keystore. It is used to verify the caller's certificate.

Here is a portion of the **login-config.xml** file:

```
<application-policy name="CertLogin">
<authentication>
  <login-module
code="org.jboss.soa.esb.services.security.auth.login.CertificateLoginModule"
flag = "required" >
  <module-option name="keyStoreURL">
    file://pathToKeyStore
  </module-option>
  <module-option name="keyStorePassword">storepassword</module-option>
  <module-option name="rolesPropertiesFile">
```

```
    file://pathToRolesFile
  </module-option>
  </login-module>
</authentication>
</application-policy>
```

Properties

keyStoreURL
: This is the path to that keystore which is used to verify the certificates. This keystore can take the form of a file on either the local file system or on the classpath.

keyStorePassword
: This is the password for the above keystore.

rolesPropertiesFile
: This is optional. It is the path to a file containing role mappings. Refer to *Section 8.6.2, “ Role Mapping ”* for additional details.

8.6.2. Role Mapping

This file is optional. It can be specified in **login-config.xml** by using the rolesPropertiesFile property. This property can point to a file located either on the local file system or on the classpath. The file contains a mapping of users to roles, as shown in the following example:

```
# user=role1,role2,...
guest=guest
esbuser=esbrole
# The current implementation will use the Common Name(CN) specified
# for the certificate as the user name.
# The unicode escape is needed only if your CN contains a space
Austin\u0020Powers=esbrole,worker
```

For an example, please look at the **security_cert** Quick Start.

8.7. Password Encryption

JBoss Enterprise Service Bus configuration files sometimes require passwords. In the past, these had been stored in clear text in the configuration files themselves. This was obviously a security risk. There is now the option to specify a path to a file that contains an encrypted password that can be read whenever it is required.

8.7.1. Creating an Encrypted Password File

Follow these steps to create an encrypted password file:

1. Go to the JBoss Server instance's **default/conf** directory.

2. **java -cp ../lib/jbosssx.jar org.jboss.security.plugins.FilePassword welcometojboss 13 testpass esb.password**

Option	Description
Salt	The "salt" used to encrypt. (In the example above, this is the **welcometojboss** string .)

Option	Description
Iteration	The number of iterations. (In the example above, this is the number **13**.)
Clear Text Password	The password one wishes to encrypt. (In the example above, this is the string **testpass**.)
Password File Name	The name of the file in which the encrypted password will be saved. (In the example above, this is the **esb.password** string.)

Table 8.4. Encrypted Password Options

8.7.1.1. Configuring Encrypted Password Files

To configure encrypted passwords files, simply replace the existing clear-text password with the path to the other file containing the encrypted password.

8.7.2. Security Service

The `SecurityService` interface is the Enterprise Service Bus central security component. Here it is:

```
public interface SecurityService
{
    void configure() throws ConfigurationException;

    void authenticate(
        final SecurityConfig securityConfig,
        final SecurityContext securityContext,
        final AuthenticationRequest authRequest)
        throws SecurityServiceException;

    boolean checkRolesAllowed(
        final List<String> rolesAllowed,
        final SecurityContext securityContext);

    boolean isCallerInRole(
        final Subject subject,
        final Principal role);

    void logout(final SecurityConfig securityConfig);

    void refreshSecurityConfig();
}
```

The default implementation is based on JAAS but it can be customised if one implements the above interface and configures the **jbossesb-properties.xml** file to use a custom `SecurityService`. For more information relating to the **SecurityService** interface, please refer to the Java documentation.

Appendix A. Revision History

Revision 1.3 **Wed Jul 14 2010** **David Le Sage** *`dlesage@redhat.com`*

Updated for SOA 5.1

Revision 1.2 **Wed May 26 2010** **David Le Sage** *`dlesage@redhat.com`*

Updated for SOA 5.0.2

Revision 1.1 **Tue Apr 20 2010** **David Le Sage** *`dlesage@redhat.com`*

Updated for SOA 5.0.1

Revision 1.0 **Fri Jan 22 2010** **David Le Sage** *`dlesage@redhat.com`*

Created