JBoss Enterprise SOA Platform 4.3

Services Guide

Your guide to services available on the JBoss Enterprise SOA Platform 4.3 CP05

JBoss Enterprise SOA Platform 4.3 Services Guide
Your guide to services available on the JBoss Enterprise SOA Platform 4.3 CP05
Edition 4.3.5

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

Preface

1. Document Conventions

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

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

1.1. Typographic Conventions

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

`Mono-spaced Bold`

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

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

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

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

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

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

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

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

Proportional Bold

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

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

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

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

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

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

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

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

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

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

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

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

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

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

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

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

import javax.naming.InitialContext;
```

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

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

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

1.3. Notes and Warnings

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

Note

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

Important

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

Warning

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

2. We Need Feedback!

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

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

What is the Registry?

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. 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.

Note

The `Registry` may be replicated or federated to improve performance and reliability. It need not be a single point of failure.

1.2. How Does One Use It?

From a business analyst’s perspective, it is similar to an internet search engine, albeit one for business processes. From a developer's perspective, it is a registry used to publish services and query the registry to discover services matching various criteria.

1.3. Registry Versus Repository

A registry allows for the registration of services, discovery of metadata and classification of entities into predefined categories. Unlike a respository, it does not have the ability to store business process definitions or WSDL or any other documents that are required for trading agreements. A registry is essentially a catalogue of items, whereas a repository contains those items.

1.4. SOA Components

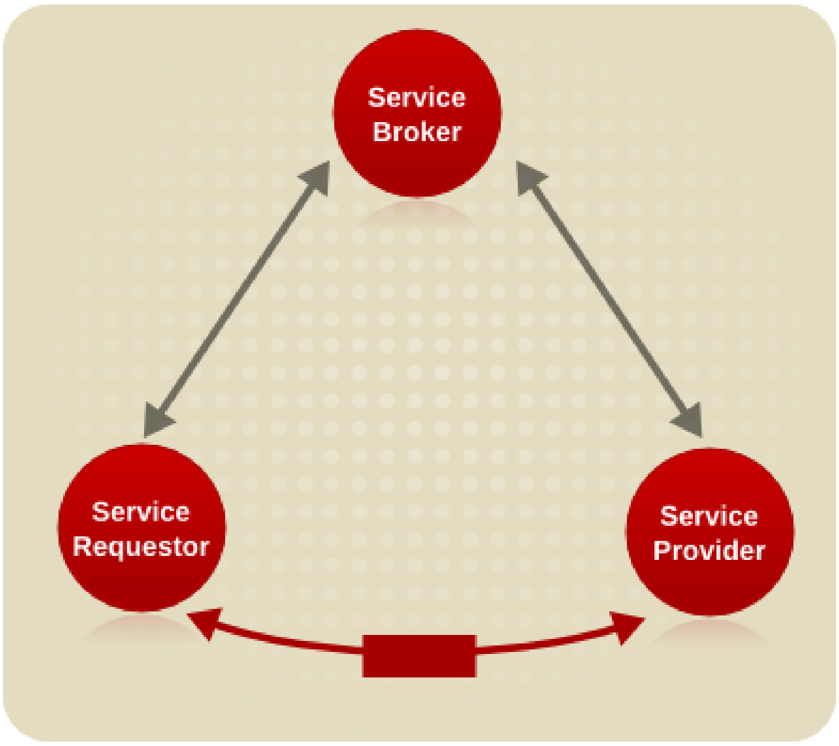
"A SOA is a specific type of distributed system in which the agents are 'services'."[1].

The key components of a Service Oriented Architecture are the messages that are exchanged, agents that act as service requesters and providers, and the shared transport mechanisms that allow the flow of messages. A description of a service that exists within an SOA is essentially just a description of the messages exchanged between itself and its users. Within an SOA there are three critical roles: requester, provider, and broker.

Service Provider
: A Provider allows access to services, creates a description of a service and publishes it to the service broker.

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

Service Requester
: A Requester is responsible for discovering a service by searching through the service descriptions given by the service broker. A requestor is also responsible for binding to services provided by the service provider.



1.5. The UDDI

The *Universal Description, Discovery and Integration* (UDDI) Registry is a directory service for Web Services. It facilitates service discovery through queries to the UDDI registry at design time or at run time. It also allows providers to publish descriptions of their services to the registry. The

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

registry typically contains a URL that locates the WSDL document for the web services and contact information for the service provider. Within UDDI information is classified into the following categories.

- *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.

1.6. 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.

Configuring the Registry

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

The JBoss SOA Platform Registry architecture allows for a great deal of flexibility when it comes to the configuration of either a Registry or Repository. By default the SOA Platform uses a JAXR implementation (**Scout**) and a UDDI (**jUDDI**), both of which are embedded.

In the ***${SOA_ROOT}*/server/*${CONFIG}*/deploy/jbossesb.sar/jbossesb-properties.xml** file there is section called registry which, as its name suggests, is used to configure the registry:

```
<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.interceptors" value=
  "org.jboss.internal.soa.esb.services.registry.InVMRegistryInterceptor"/>

  <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>
```

JBoss SOA Registry Properties

`org.jboss.soa.esb.registry.implementationClass`
: This is a class that implements the `jbossesb registry` interface. One implementation (the JAXRRegistry interface) has been provided by Red Hat.

`org.jboss.soa.esb.registry.factoryClass`
: This is the class name of the `JAXR ConnectionFactory` implementation.

`org.jboss.soa.esb.registry.queryManagerURI`
: This is the uniform resource indicator used by JAXR for querying.

`org.jboss.soa.esb.registry.lifeCycleManagerURI`
: This is the uniform resource indicator used by JAXR for editing.

`org.jboss.soa.esb.registry.user`
: This is the user-name used for editing.

`org.jboss.soa.esb.registry.password`
: This is the password for the specified user.

`org.jboss.soa.esb.scout.proxy.transportClass`
: This is the name of the class used by **Scout** to transport things to the UDDI.

2.1. The Registry Components

The `Registry` can be configured in many different ways. *Figure 2.1, “Blueprint of the Registry Component Architecture”* presents a blueprint of all of its components. From the top down one can see that the JBoss SOA Platform "funnels" all interaction with the `Registry` through the appropriately-named `Registry Interface`. It then calls a JAXR implementation of this interface, which by default is **Scout**. **Scout** then calls the **jUDDI** registry. However, although this is the default, there are many other configuration options.

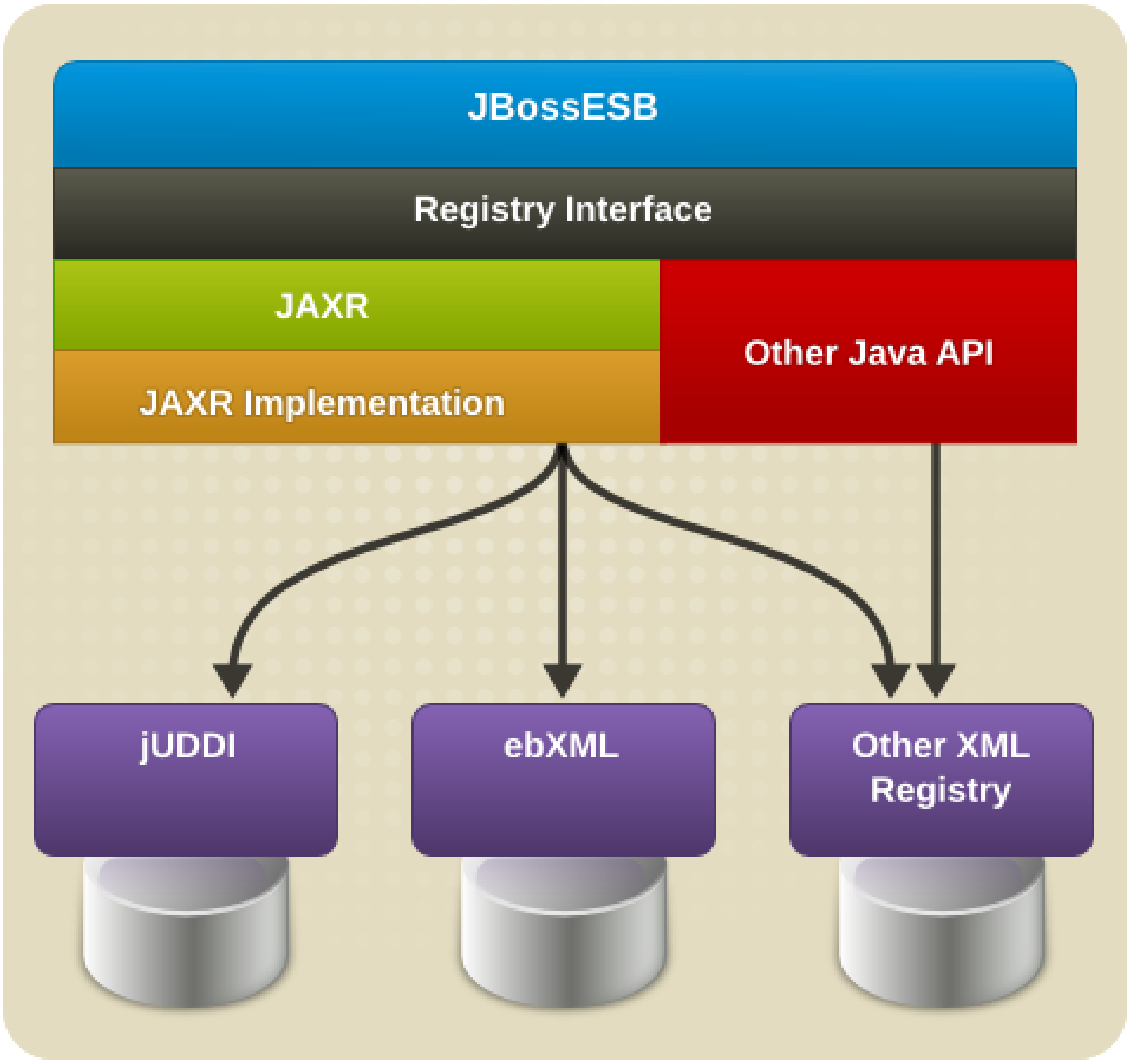


Figure 2.1. Blueprint of the Registry Component Architecture

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.

2.3. Using JAXR

org.jboss.soa.esb.registry.factoryClass

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`**.

Next, configure the JAXR implementation by providing the location of the `registry` that is to be used for querying and updating. Achieve this by editing the org.jboss.soa.esb.registry.queryManagerURI, org.jboss.soa.esb.registry.lifeCycleManagerURI and org.jboss.soa.esb.registry.securityManagerURI properties.

The user name and password for the UDDI Registry are set by editing the org.jboss.soa.esb.registry.user and org.jboss.soa.esb.registry.password properties respectively.

2.4. Using Scout and jUDDI

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

There is an additional, optional parameter that can be used with **Scout** and **jUDDI**. This is the transport class. There are two included implementations of this class, based upon Remote Method Invocation and Local (embedded) Java, respectively.

Note

Scout does have a transport class for SOAP, (which uses **Apache Axis** and is not currently supported in the JBoss Enterprise SOA Platform.) Additional information about this can be found on the **Apache Axis** website at *http://ws.apache.org/axis/*.

The **`transportClass`** settings for the different transports are listed below.

Important

When these are changed, the query and life-cycle uniform resource indicators also have to be updated..

Example 2.1. Using Remote Method Invocation

```
<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.scout.proxy.transportClass"
 value="org.apache.ws.scout.transport.RMITransport"/>
```

Example 2.2. Using Local

```
<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.scout.proxy.transportClass"
   value="org.apache.ws.scout.transport.LocalTransport"/>
```

To use **jUDDI**, two requirements must be fulfilled:

1. the database where its data will be stored must be accessible. In it, one must create the appropriate schema.

2. **jUDDI** must be configured in the ***${SOA_ROOT}*/server/*${CONFIG}*/deploy/jbossesb.sar/esb.juddi.xml** file.

The JBoss Enterprise SOA Platform includes a tool that performs 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*.

Registry Configuration Examples

3.1. Introduction

The JBoss SOA Platform uses the **Scout** implementation of the *JAXR application programming interface* by default. It also uses **jUDDI** to provide a `registry`. The following examples teach how to deploy these components.

3.2. Embedded jUDDI

Any server components with a relationship to the registry can share the latter between themselves. In other words, multiple instances of the **JBoss Enterprise SOA Platform** can use the same registry via a shared database.

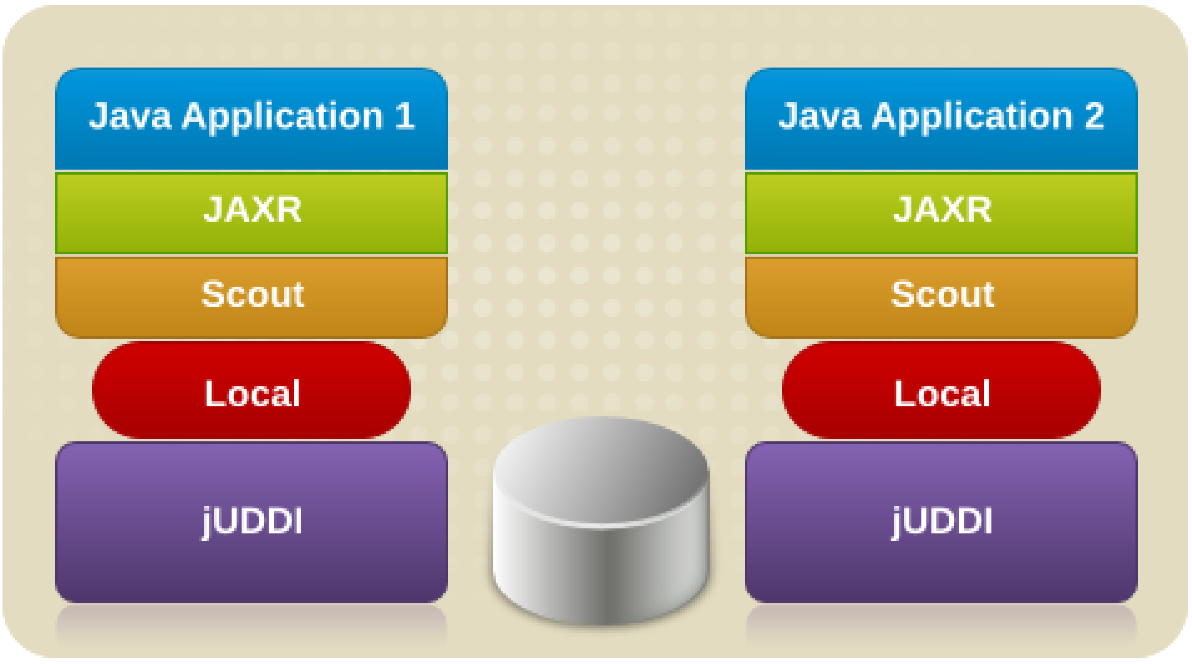


Figure 3.1. Embedded jUDDI

Example 3.1. Properties for 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.interceptors" value=
  "org.jboss.internal.soa.esb.services.registry.InVMRegistryInterceptor"/>

  <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>
```

3.3. Remote Method Invocation Using `jbossesb.sar`

The **`jbossesb.sar`** registers a RMI service for jUDDI.

Example 3.2. **`jbossesb.sar/jbossesb-properties.xml`** Settings for RMI

```
<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>
```

Add the following JNDI settings to the **`juddi.properties`** file:

```
# 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

The RMI clients need to have **`scout-client.jar`** in their class-paths.

3.4. Remote Method Invocation Using JNDI Registration of the RMI Service

It is possible to configure another JBoss SOA Platform component in the same Java Virtual Machine as **jUDDI**. Do this to register the Remote Method Invocation service.

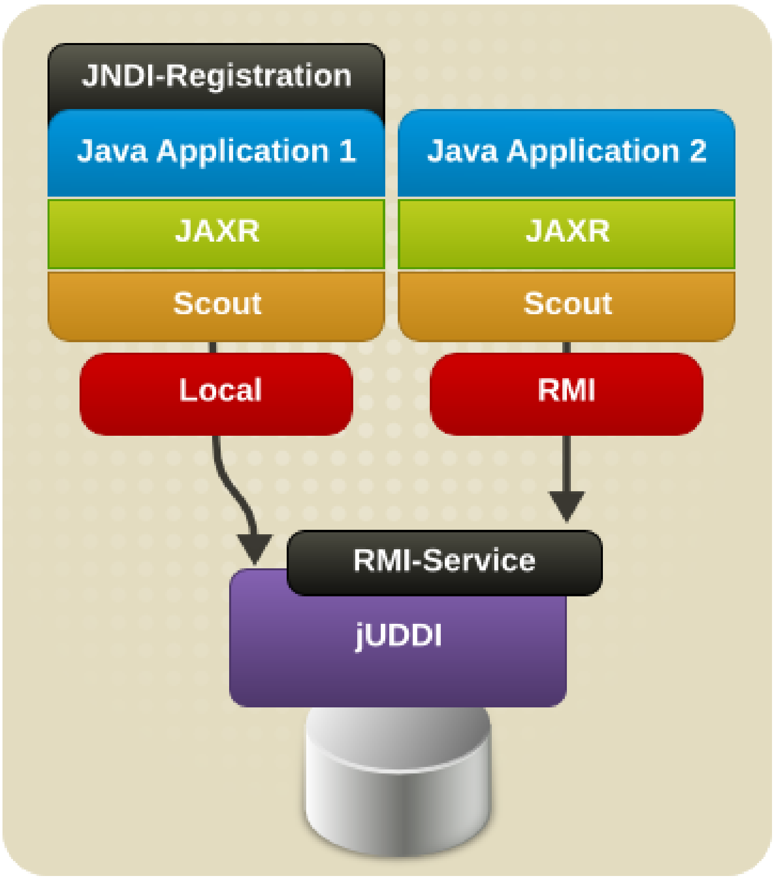


Figure 3.2. Remote Method Invocation Using a Custom JNDI Registration

In this example, **`Application1`** will need to be configured with the Local settings, whilst **`Application2`** will require the Remote Method Invocation settings.

Example 3.3. Local Settings Used for Application1

```
<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.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 3.4. RMI Settings Used for Application2

```
<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>
```

For `Application2` (using Remote Method Invocation), the host-name of the queryManagerURI and lifeCycleManagerURI properties need to be set to that of the host on which the **jUDDI** service is running.

Note

`Application1` needs to have access to a naming service.

Example 3.5. JNDI Registration Process for Application1

```
//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);

log.info("Creating Initial Context using: \n"
  + RegistryEngine.PROPNAME_JAVA_NAMING_FACTORY_INITIAL+"="+factoryInitial
  + "\n"
  + RegistryEngine.PROPNAME_JAVA_NAMING_PROVIDER_URL + "=" + providerURL
  + "\n"
  + RegistryEngine.PROPNAME_JAVA_NAMING_FACTORY_URL_PKGS + "="
  + factoryURLPkgs + "\n");

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);
```

3.5. SOAP

Scout can be configured to use SOAP to communicate with **jUDDI** via **Apache Axis**. Note that this is not currently supported in the JBoss SOA Platform although future versions may add SOAP support.

Note

Additional information about this can be found on the **Apache Axis** website at *http://ws.apache.org/axis/*.

Important

JBoss Application Server Version 4.2 ships with older versions of **Scout** and **jUDDI**. Red Hat recommends removing the **`juddi.sar`** file to prevent versioning issues if one intends to deploy this older version.

UDDI Browser

4.1. Introduction

The JBoss SOA Platform does not ship with an included UDDI browser.

The UDDI browser **ub** can be downloaded from *http://www.uddibrowser.org*. Before configuring **ub** make sure the **`juddi.war`** is deployed. This is required to enable webservice communication to jUDDI.

4.2. UB Setup

ub is a standalone Java application. Start **ub** and select **Edit** > **UDDI Registries**, and add an entry called jUDDI

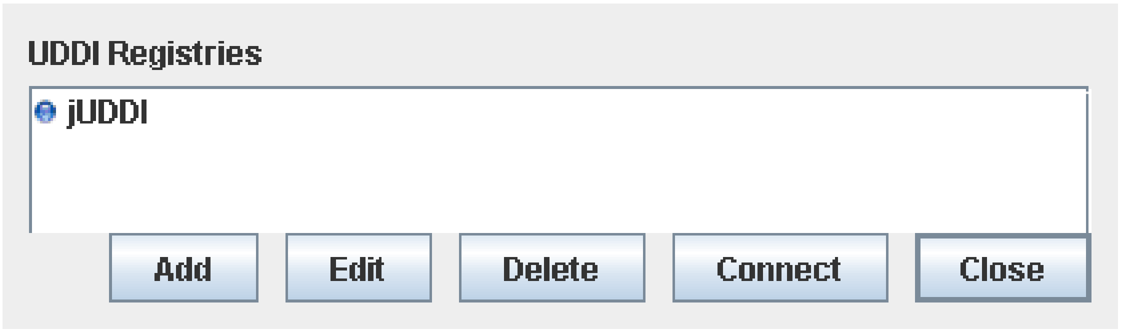
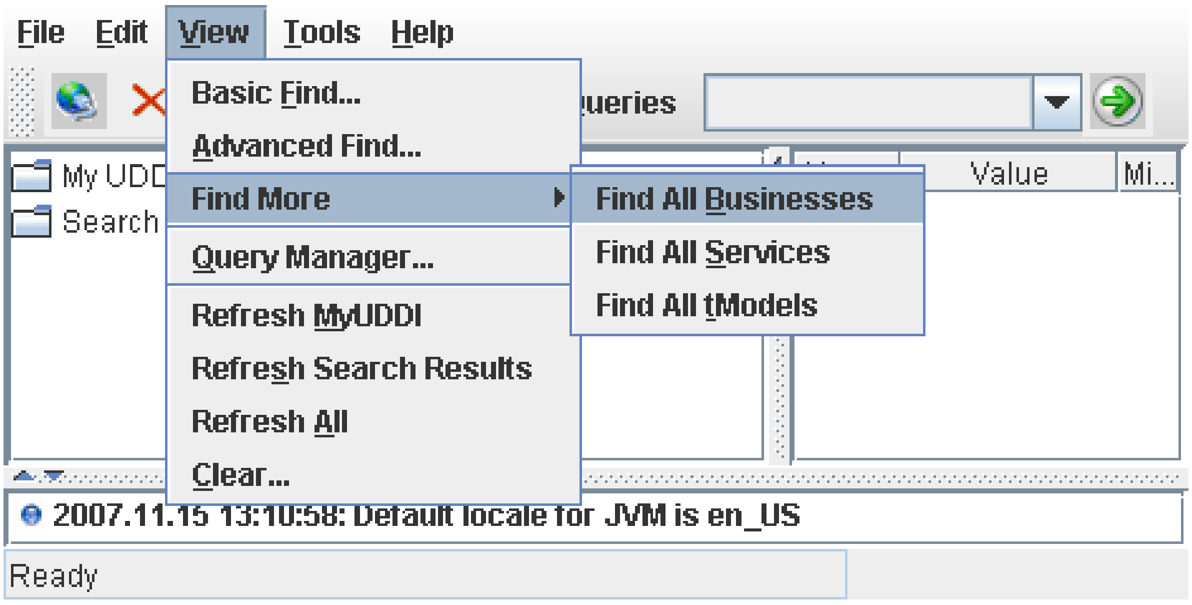


Figure 4.1. Add a connection

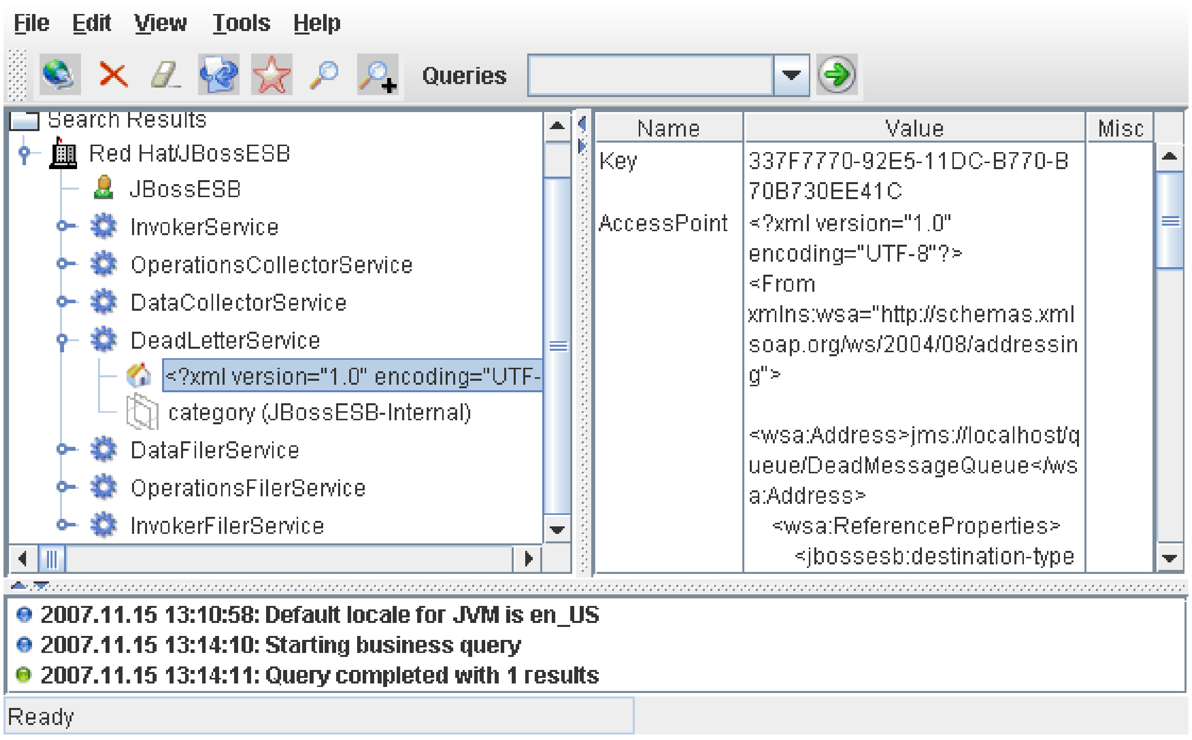
Click on **connect** and select **View** > **Find More** > **Find All Businesses**



2007.11.15 13:10:58: Default locale for JVM is en_US

Figure 4.2. View All Businesses

In the left pane, one will see the **`Red Hat/JBossESB organization`**. Navigate into the individual services and their `ServiceBindings`.



Name	Value	Misc
Key	337F7770-92E5-11DC-B770-B70B730EE41C	
AccessPoint	<?xml version="1.0" encoding="UTF-8"?> <From xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"> <wsa:Address>jms://localhost/queue/DeadMessageQueue</wsa:Address> <wsa:ReferenceProperties> <jbossesb:destination-type	

```
2007.11.15 13:10:58: Default locale for JVM is en_US
2007.11.15 13:14:10: Starting business query
2007.11.15 13:14:11: Query completed with 1 results
```

Figure 4.3. View Services and ServiceBindings

The `AccessPoint` for each `ServiceBinding` contains an end-point reference.

Some of **ub**'s features may not work but it should provide enough functionality to maintain the **jUDDI**. The JBoss Enterprise Service Bus community project is currently looking for a good web-based console through which to control the **jUDDI**.

Registry Troubleshooting

5.1. Scout and jUDDI Pitfalls

- If Remote Method Invocation is being used, add the **`juddi-client-2.0rc5.jar`** **jUDDI** client file, found at ***`${SOA_ROOT}`*****`/server/production/deploy/jbossesb.sar/lib/`**.
- Make sure that the **`jbossesb-properties.xml`** file is in the classpath and that it is readable or the registry will try to instantiate classes named **`null`**.
- Make sure there is a **`juddi.properties`** file on the class-path so that the **jUDDI** can configure itself. (The JBoss SOA Platform uses **`esb.juddi.xml`** but generates the **`esb.juddi.properties`** file for **jUDDI** to read.)
- The JBoss Enterprise SOA Platform includes a database configuration tool which performs these configuration steps for **jUDDI**. This application can be found in the ***`${SOA_ROOT}`*****`/tools/schema/`** directory. Directions for using it are found in the "Switching Databases" section of the *Administration Guide*.
- If a service fails or does not shut down cleanly, old entries may persist within a registry. Remove these manually.

5.2. More Information

Further community resources on this topic can be found at:

- The JBoss jUDDI wiki *http://www.jboss.org/community/docs/DOC-11217*
- JBossESB user forum: *http://www.jboss.com/index.html?module=bb&op=viewforum&f=246*.

What is a Rule Service?

6.1. Introduction

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.

The **JBoss Enterprise SOA Platform**'s *Rule Service* allows one to deploy rules that have been created in **JBoss Rules** as services. This has two major benefits: firstly, the amount of client code required to integrate the rules into one's application environment is dramatically reduced; secondly, rules can be accessed either as part of an *action chain* or within an *orchestrated* business process.

Note

The **JBoss Business Rules Management System** is supported but one can also use other rules engines if required to do so.

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 classpath. 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`.

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. For instance, rules might be sourced from the BRMS **Business Rules Editor**, imported DRL files, *Domain Specific Language* files and *Decision Tables*.

Stateless Rule Services

Most rule services will be "stateless." In the stateless model, a message is sent to the Rule Service. All the facts that are to be inserted into the rules engine are included in the message body. The rules execute and update either the message or the facts.

Stateful Rule Services

"Stateful" execution takes place over time, with several messages being sent to the `rule service`. The rules are executed each time and they update either the message or the facts until a final message is received by the service that tells it to dispose of the stateful session. This configuration model is currently limited, in the sense that there can only be a single stateful `rule service` in the message flow.

Rule Services Using JBoss Rules

7.1. Introduction

JBoss Rules is the engine that provides the **SOA Platform** with *rule service* support. **JBoss Rules** is integrated through the following:

- the **`BusinessRulesProcessor`** action class
- rules written in DRL, DSL, a decision table, or the business rules editor.
- ESB messages
- The objects in the ESB message's content, this being the data going into the rules engine.

When a message is sent to the BusinessRulesProcessor a rule set executes over the objects in the message and updates either of those objects or the message.

7.2. Rule Set Creation

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

Note

For more information about rule creation and the JBoss Rules language itself, please refer to the *JBoss Rules Reference Guide*.

There are only three requirements when writing rules for deployment on the JBoss SOA Platform as a service:

1. they must all define the ESB message as a global.

   Most rule services will want to update the message as a way of communicating results to other services in the flow, so the `BusinessRulesProcessor` and the `DroolsRuleService` will always set the message as a global.

   Example 7.1. Defining an ESB message as a Global

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

2. If other globals are required in addition to the ESB message, they must be set in a higher *salience* rule.

   Neither the `BusinessRulesProcessor` nor the `DroolsRuleService` provide a means to set globals in **`jboss-esb.xml`**. This might be added in the future.

   Example 7.2. Declaring a Global in a Rule with Higher Salience

   ```
   rule "Set a global"
    salience 100
   ```

```
 when
 then
  drools.setGlobal("foo", new Foo());
end
```

3. The `ESBRuleService` does not provide a means to start a `RuleFlow` from the service itself. This feature might be added in the future.

7.3. Rule Service Consumers

A rule service consumer has little about which to be concerned. There is no need for the consumer to create `rule-bases` or `working memories` or to deal with rule executions. Instead, it just adds facts and, sometimes, properties to the message.

In some cases the client is *JBoss SOA aware* and will add the objects directly to the message.

Example 7.3. Adding Objects Directly to a Message

```
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);
```

In other cases the data may be in an XML message. If so, a transformation service will be added to the message flow to transform the XML to *Plain Old Java Objects* (POJOs) before the rule service is invoked.

Stateful Rule Execution

Stateful rule execution requires a few properties to be added the messages.

For the first message:

```
message.getProperties().setProperty("dispose", false);
message.getProperties().setProperty("continue", false); // this is the default
```

For all the subsequest messages but the final message:

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

For the final message:

```
message.getProperties().setProperty("dispose", true); // this is the default
message.getProperties().setProperty("continue", true);
```

Important

These can be added directly by an JBoss SOA aware client but a client that is not JBoss SOA aware will have to communicate the position of the message (first, ongoing, last) in the data. You will also need to add an action class to the pipeline to add the properties to the ESB message.

`quickstarts/business_ruleservice_stateful` is an example of this type of service.

7.4. Configuration

A rule service is configured in the jboss-esb action element for the service.

The action class and name is 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 that tells the rule agent how to find the rule package. This could specify a url or a local file.

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

Several example configurations follow:

Example 7.4. Rules are in a drl, execution is stateless

```
<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 7.5. Rules are in a drl, execution is stateful

```
<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>
```

In this scenario the client may send multiple messages over time to the rule service. For example, the first message may contain a customer object, and the next several messages contain orders for that customer. Each time a message is received, the rules will be fired. On the final message, the client can add a property to the message to tell the rule service to dispose of the working memory.

Example 7.6. Rules in a Domain Specific Language, stateless execution

```
<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 7.7. Rules in a DecisionTable, 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 7.8. Rules in the BRMS, 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>
```

The Action Configuration Attributes to the action tag specify which action is to be used and which name this action is to be given.

The Action Configuration Attributes specify the set of rules (ruleSet) to be used in 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.
`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`.

7.5. Object Paths

Note that **JBoss Rules** treats objects as if they were "shallow" in order to achieve highly-optimized performance. Use the optional object-paths property to evaluate an object residing in a location that is deeper than the `object tree`. (Setting this property results in those objects with an **`ESB Message Object Path`** being extracted.)

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`
`attachment.1`	gets 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.)

7.6. 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.

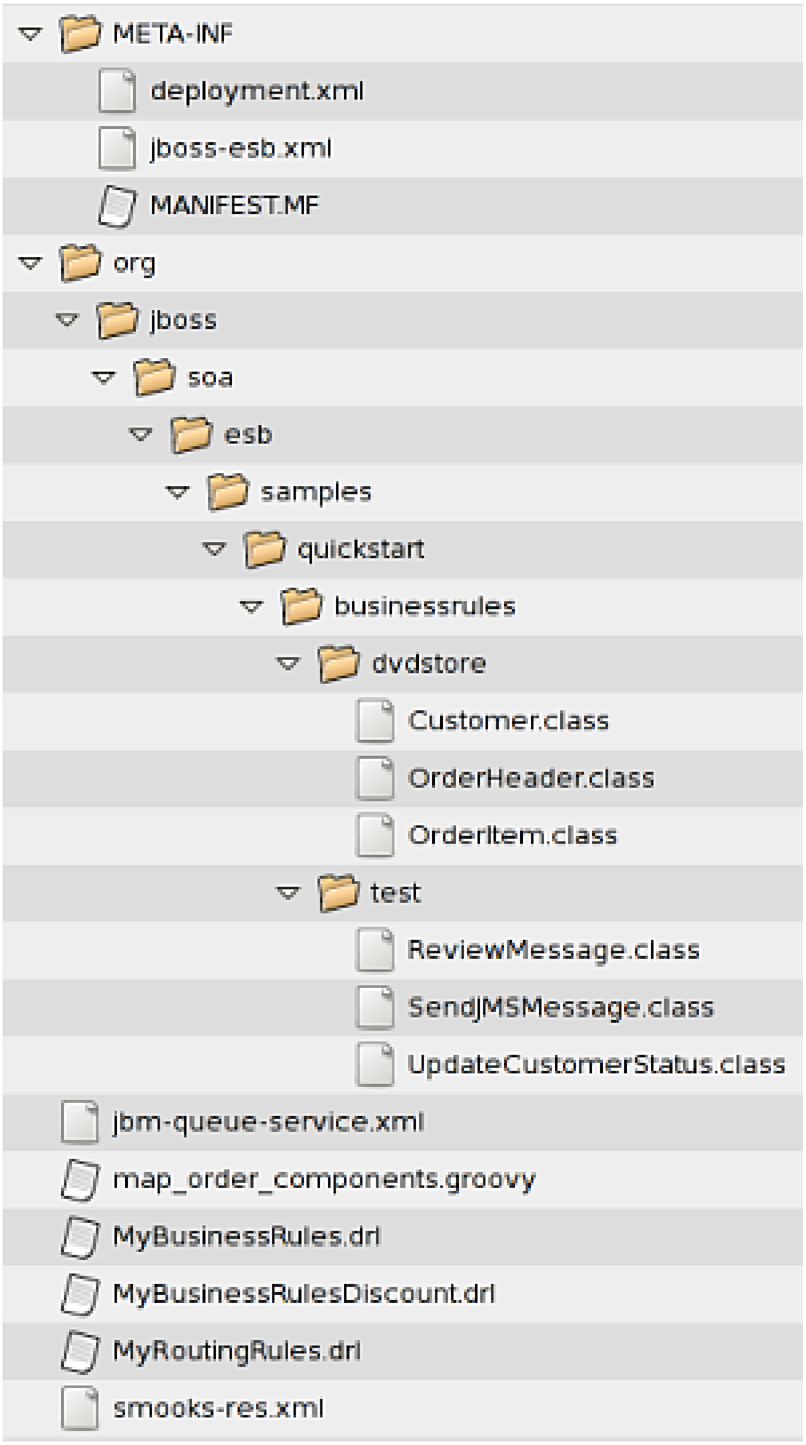


Figure 7.1. Typical .esb archive which uses JBoss Rules.

Finally make sure to deploy and reference the **`jbrules.esb`** in your **`deployment.xml`**.

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

What is Content-Based Routing?

8.1. Introduction

8.1.1. 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?

8.1.2. 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.

8.2. 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 illustrates a very basic example of the content-based routing architecture. The client sends a message to **`Service A`**. **`Service A`** forwards the message to **`Router`** which, based on the content of the message, forwards the message to either **`Service B`** or **`Service C`**.

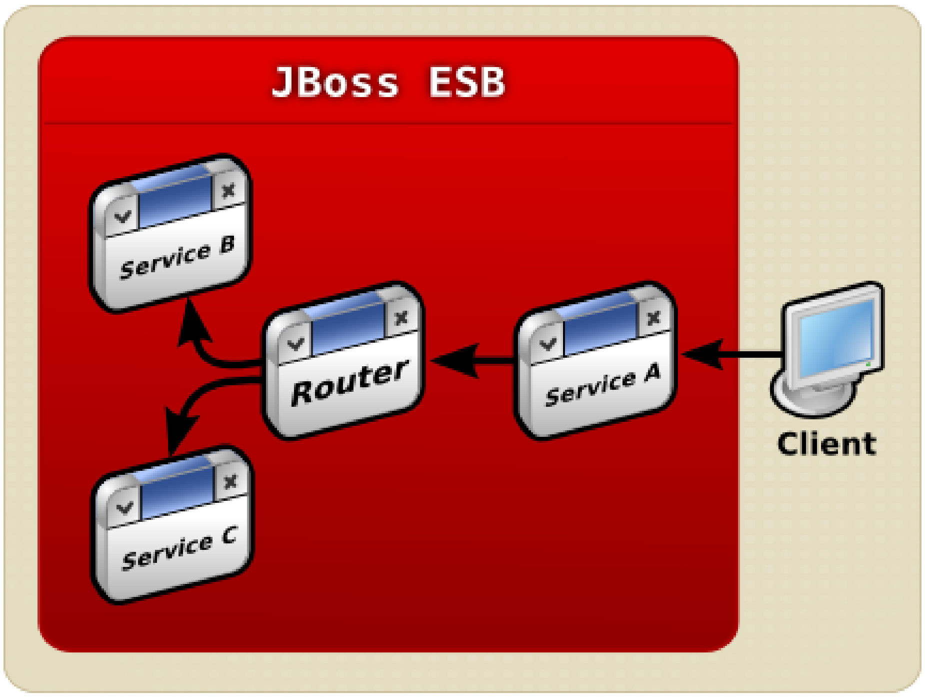


Figure 8.1. A basic content-based routing scenerio

Content Based Routing Using JBoss Rules

9.1. 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 rules 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.)

Note

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

9.2. 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:

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.

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.

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.

9.3. Rule Set Creation

A rule set can be created using either the **JBoss Developer Studio** which includes a plug-in for **JBoss Rules**. *Figure 9.1, "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.

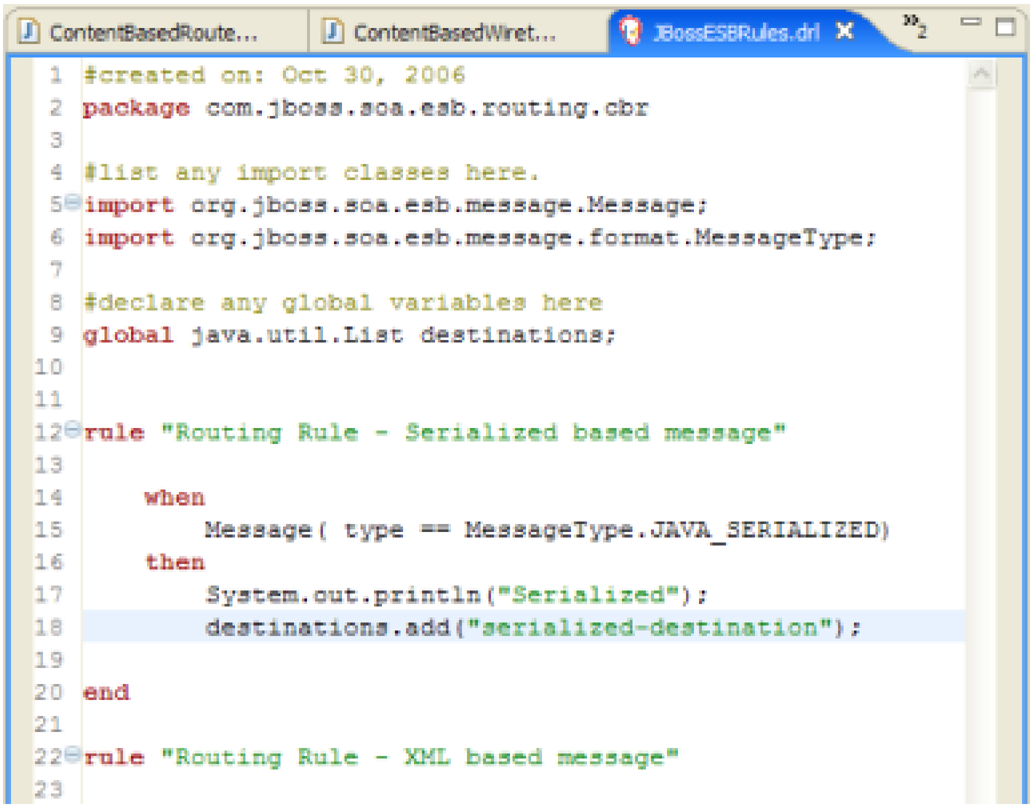
To turn a regular rule-set into one that can be used for content-based routing, one must evaluate an ESB message and ensure that the rule match results in a list of strings containing the service destination names. Bear two things in mind whilst doing this:

- 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;
  ```



```
#created on: Oct 30, 2006
package com.jboss.soa.esb.routing.cbr

#list any import classes here.
import org.jboss.soa.esb.message.Message;
import org.jboss.soa.esb.message.format.MessageType;

#declare any global variables here
global java.util.List destinations;


rule "Routing Rule - Serialized based message"

    when
        Message( type == MessageType.JAVA_SERIALIZED)
    then
        System.out.println("Serialized");
        destinations.add("serialized-destination");

end

rule "Routing Rule - XML based message"

```

Figure 9.1. Create a New Rule Set using the JBoss Developer Studio

The message will now be added to the working memory of the rules engine. 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 of the XML or of the serializable kind.

9.4. XPath Domain Specific Language

For XML-based messages, it is convenient to undertake *XPath*-based evaluation. In order to support this, Red Hat ship a *Domain Specific Language* implementation which allows you to use XPath expressions in the rule file. These expressions are defined in the **`XPathLanguage.dsl`** file. To use, 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`** and can be customized if need be. Alternatively, one can define an entirely different DSL altogether. The quick start called **`fun_cbr`** demonstrates this use of XPath.

9.4.1. XPath and Name-Spaces

To use name-spaces with XPath, one needs to specify which name-space prefixes are to be used in the XPath expression. The names-pace prefixes are specified in a comma-separated list in 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.

The `XPathLanguage.dsl` is defined in the **`XPathLanguage.dsl`** file. This can be customized if needed. Users can even define their own DSL. (The quick start called **`fun_cbr`** demonstrates this use of XPath.)

9.5. 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.

Example 9.1. Example Content Based Routing Service Configuration

```
<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>
```

The attributes of the `action` tag are shown in the following table. These attributes specify which `action` is to be used and what name it is to be given.

Table 9.1. Content-Based Routing Action Configuration Attributes

Attribute	Description
`Class`	Action class, 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

The action properties are shown in the following table. The properties specify the set of rules (`ruleSet`) to be used in this action.

Table 9.2. CBR Action Configuration Properties

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.

Property	Description
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 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`.

9.6. "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 invoking the rules for the last time, set dispose to **`true`** so that the working memory is cleared:

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

Note

To learn more about the `RuleService`, please refer to *Chapter 7, Rule Services Using JBoss Rules* .

Note

For an example showing how to use stateful rules, please refer to the `business_ruleservice_stateful` quick start.

9.7. The RuleAgent and the Business Rules Management System

By using the RuleAgent, one can effectively integrate one's service with a Business Rules Management System (BRMS.) This is 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 the *JBoss Rules* documentation.

Note

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

9.8. 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.

9.9. 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.

9.10. Deployment and Packaging

Package 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 9.2, “Typical .ESB Archive Employing JBoss Rules.”* below shows the layout of the **`simple_cbr`** quick start in order to depict a typical package:

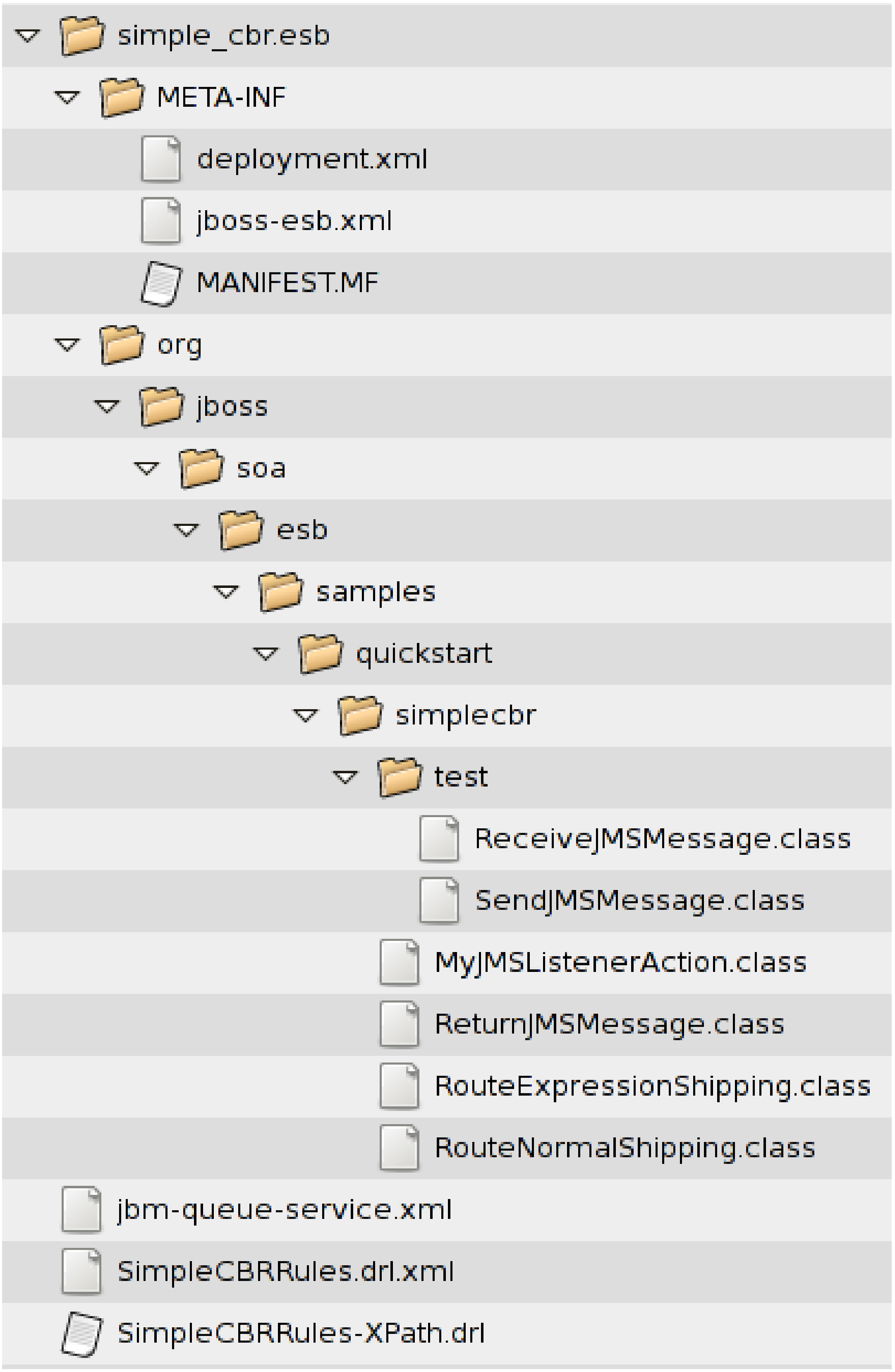


Figure 9.2. Typical .ESB Archive Employing JBoss Rules.

Finally, make sure to both 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>
```

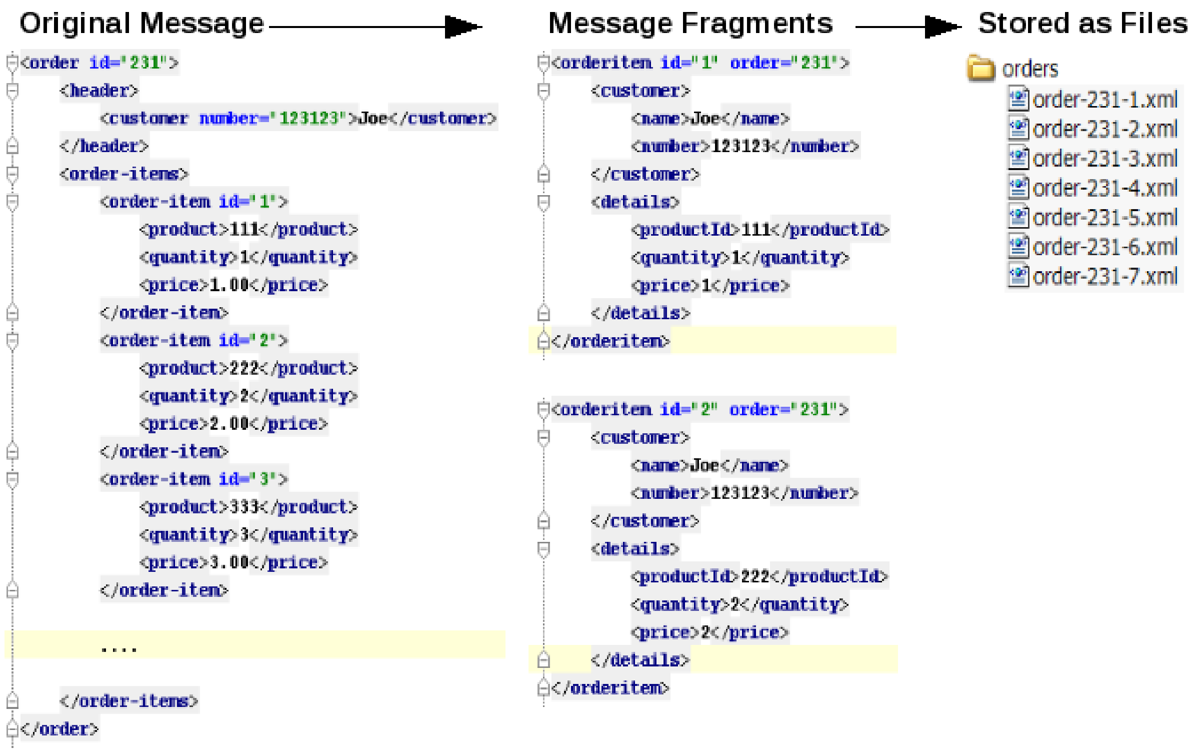
Content-Based Routing Using Smooks

The `SmooksAction` can be used for splitting messages into fragments and performing content-based routing on those fragments.

Important

Red Hat is offering this functionality as a Technical Preview only. What this example does not show is how to perform the content-based routing using <condition> elements on the resources. It also doesn't demonstrate how to route fragments to message-aware end-points. A future version will include a quick start dedicated to demonstrating these features of the Enterprise Service Bus.

An order message with many order items per message is one example example of a situation in which message splitting could be used. If different services are used to process different types of items then one would need to split the message into one fragment per order item. Each of these new messages could then be routed to their respective destinations based on their content.



```
<order id='231'>
    <header>
        <customer number='123123'>Joe</customer>
    </header>
    <order-items>
        <order-item id='1'>
            <product>111</product>
            <quantity>1</quantity>
            <price>1.00</price>
        </order-item>
        <order-item id='2'>
            <product>222</product>
            <quantity>2</quantity>
            <price>2.00</price>
        </order-item>
        <order-item id='3'>
            <product>333</product>
            <quantity>3</quantity>
            <price>3.00</price>
        </order-item>

        ....

    </order-items>
</order>
```

```
<orderitem id="1" order="231">
    <customer>
        <name>Joe</name>
        <number>123123</number>
    </customer>
    <details>
        <productId>111</productId>
        <quantity>1</quantity>
        <price>1</price>
    </details>
</orderitem>

<orderitem id="2" order="231">
    <customer>
        <name>Joe</name>
        <number>123123</number>
    </customer>
    <details>
        <productId>222</productId>
        <quantity>2</quantity>
        <price>2</price>
    </details>
</orderitem>
```

Figure 10.1. Message Splitting

Figure 10.1, “Message Splitting” shows how to perform the by-order-item splitting operation and how to route the split messages to files. The split messages contain a full XML document with data merged from the order header and the order item in question. It is not just a dumb split, but an actual de-normalization of the message data. You could route all the message fragments to files, a Java Message Service service or a database in any number of different formats (EDI, populated Java Objects and so forth.)

The **Smooks** configurations for the examples above look like this:

Example 10.1. Resource Configuration One

```
<jb:bindings beanId="order" class="java.util.HashMap"
    createOnElement="order">
    <jb:value property="orderId" decoder="Integer" data="order/@id"/>
    <jb:value property="customerNumber"
        decoder="Long" data="header/customer/@number"/>
    <jb:value property="customerName" data="header/customer"/>
    <jb:wiring property="orderItem" beanIdRef="orderItem"/>
</jb:bindings>
```

Example 10.2. Resource Configuration Two

```
<jb:bindings beanId="orderItem" class="java.util.HashMap"
    createOnElement="order-item">
    <jb:value property="itemId" decoder="Integer" data="order-item/@id"/>
    <jb:value property="productId" decoder="Long"
        data="order-item/product"/>
    <jb:value property="quantity" decoder="Integer"
        data="order-item/quantity"/>
    <jb:value property="price" decoder="Double" data="order-item/price"/>
</jb:bindings>
```

Example 10.3. Resource Configuration Three

```
<file:outputStream openOnElement="order-item"
    resourceName="orderItemSplitStream">

    <file:fileNamePattern>
        order-${order.orderId}-${order.orderItem.itemId}.xml
    </file:fileNamePattern>

    <file:destinationDirectoryPattern>
        target/orders
    </file:destinationDirectoryPattern>

    <file:listFileNamePattern>
        order-${order.orderId}.lst
    </file:listFileNamePattern>

    <file:highWaterMark mark="3"/>

</file:outputStream>
```

Example 10.4. Resource Configuration Four

```
<ftl:freemarker applyOnElement="order-item">
    <ftl:template>target/classes/orderitem-split.ftl</ftl:template>
    <ftl:use>
        <ftl:outputTo outputStreamResource="orderItemSplitStream"/>
    </ftl:use>
</ftl:freemarker>
```

Resource Configurations One and Two configure the bind data from the source message into Java Objects in the Smooks bean context. In this example, we're just binding the data into HashMaps. The

Map being populated in configuration Two is recreated and repopulated for every order item as the message is being filtered.

The populated Java Objects are used to populate the **FreeMarker** template defined in Resource Four. This template gets applied on every order item and the output is sent to a **`FileOutputStream`** (defined in Resource Three) which manages the file output for the split messages.

Note

This example does not show how to perform content-based routing by using <condition> elements on the resources. It also does not demonstrate how to route fragments to message-aware end-points.

Message Transformation

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

11.1. 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.

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 the **`SmooksTransformer`** will be deprecated in a future release.

11.2. XSL Transformations

The Enterprise Service Bus supports message transformation through the standard XSLT usage model, as well as through **Smooks**. (Native XSLT may be added in a future release.) Create support for XSLT by adding a custom `org.jboss.soa.esb.actions.ActionProcessor` implementation.

11.3. ActionProcessor Data Transformation

If **Smooks** cannot handle a specific transformation usecase, implement a custom transformation solution by using the `org.jboss.soa.esb.actions.ActionProcessor` interface.

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 you are not, 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.

Use the accompanying **jBPM Process Designer** to represent the steps in a business procedure graphically. This can form a strong link 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.

12.1. 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>
```

Figure 12.1. ESB DatabaseInitializer mbean configuration

The first M-Bean configuration element contains the settings for the **`DatabaseInitializer`**.

Table 12.1. ESB DatabaseInitializer mbean default values

Property	description	Default
`Datasource`	This is the data-source for the jBPM database	java:/JbpmDS

Property	description	Default
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

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 accessible at *http://localhost:8080/jbpm-console* when the server is started.

Please refer to the *jBPM Reference Guide* in order to learn how to change this application's security settings. The process involves modifying the configuration found in the **conf/login-config.xml** file. Use the **Console** to deploy and monitor both jBPM processes and human task management procedures. (A customised task-list 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 **deployment.xml** file specifies the resources upon which this ESB archive depends, namely the **jbossesb.esb** and the JbpmDS data-source. This information is used to determine the deployment order.

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

Figure 12.2. deployment.xml dependancy declarations

The **jboss-esb.xml** file deploys an internal service called the `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>
```

Figure 12.3. JBpmCallbackService

This service listens to the jBPMCallbackBus, which is a Java Message Service queue that uses the JBossMessaging (**jbm-queue-service.xml**) messaging provider. To use a different one, simple modify the provider section of the **jboss-esb.xml** file.

```
<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>
```

Figure 12.4. Modifying the Provider Section in the **`jboss-esb.xml`** for Another JMS

Note

Section 12.5, “ jBPM-to-ESB ” contains more information about the `JbpmCallbackService`.

12.2. 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 12.5. 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 12.6. 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 12.1, “ 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.

12.3. 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.

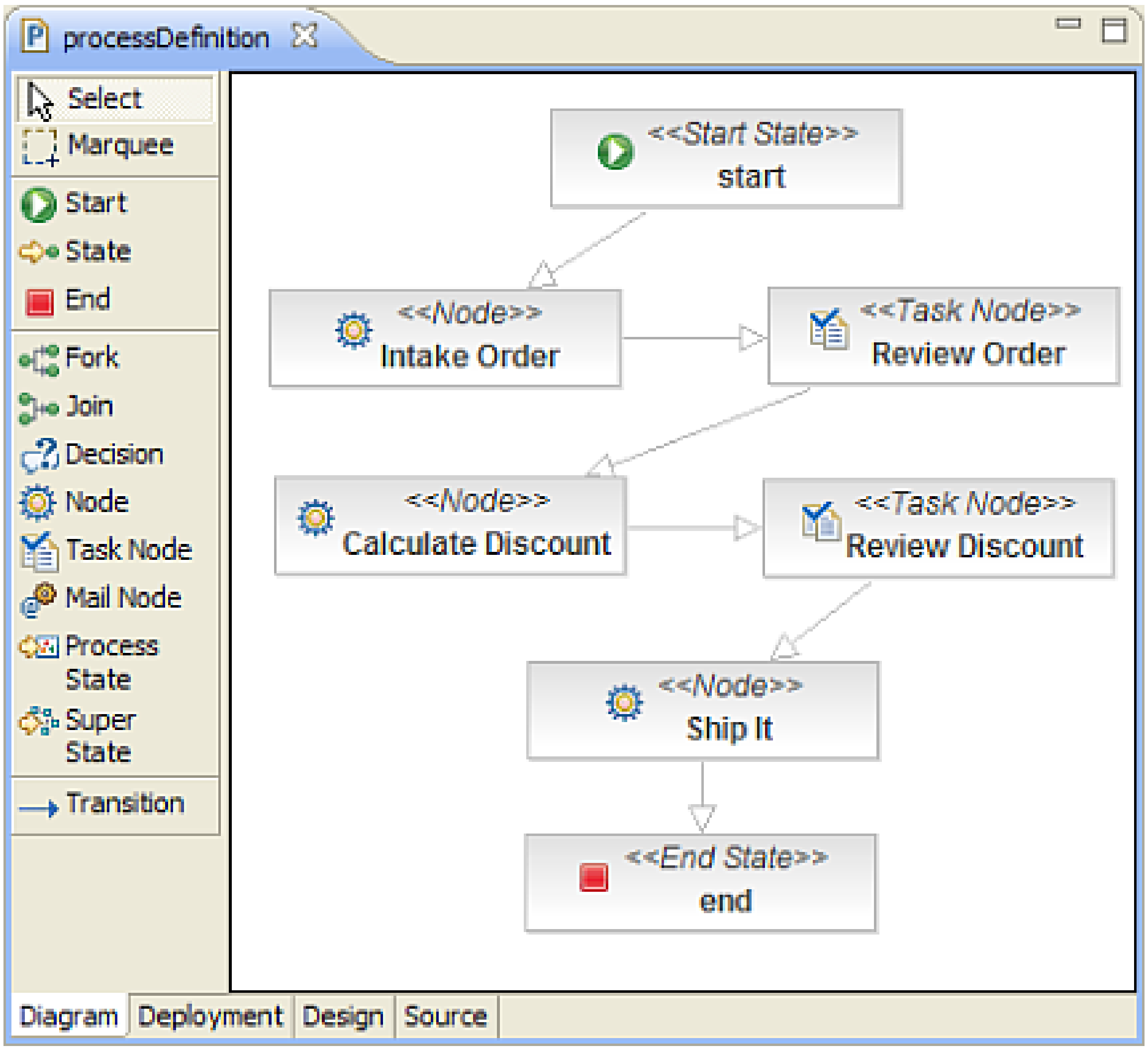


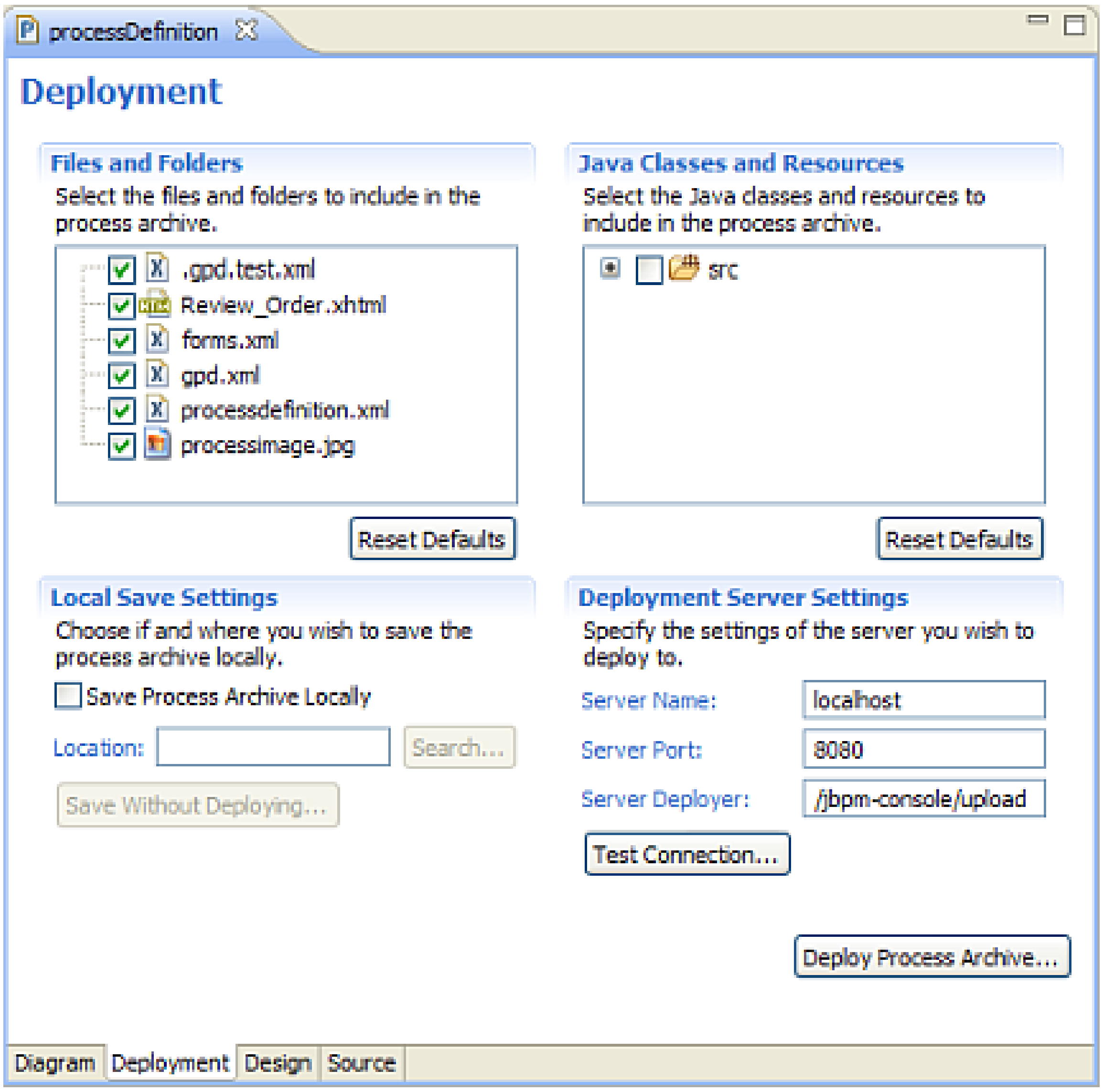
Figure 12.7. JBoss Developer Studio - jBPM Graphical Editor

Important

The server must be configured to accept jBPM process deployments. Details of this are found in the *JBoss Enterprise SOA Platform Administration Guide*.

The **Graphical Designer** allows one to create a process definition visually. Nodes, and transitions between nodes, can be added, modified or removed. The process definition is saved to the file system as an XML document which can be deployed to a jBPM instance (database). Each time a process instance is deployed, the jBPM will version it and will keep the older copies. This allows processes that are in use to complete the process instance on which they were started. New process instances will use the latest version of the process definition.

To deploy a process definition, start the server and go to the **Deployment** tab in the **Graphical Designer** to deploy a **`process archive`** (PAR) file.



Select the files and folders to include in the process archive.

Select the Java classes and resources to include in the process archive.

Choose if and where you wish to save the process archive locally.

Specify the settings of the server you wish to deploy to.

Server Name: localhost

Server Port: 8080

Server Deployer: /jbpm-console/upload

Figure 12.8. 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.

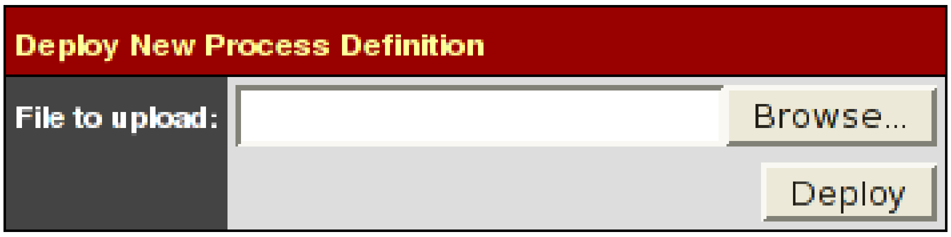


Figure 12.9. jBPM Console - Uploading a New Process Definition

12.4. 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:

`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.)

```
<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>
```

Figure 12.10. BpmProcessor Action Configuration in the **`jboss-esb.xml`** File

Two action attributes are required:

1. name

   Any value can be used for the 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`**.

The following configuration properties can also be set:

1. command

   This is a required property. It needs to be one of **`NewProcessInstanceCommand`**, **`StartProcessInstanceCommand`**, **`GetProcessInstanceVariablesCommand`** or **`CancelProcessInstanceCommand`**.

2. processdefinition

   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 that you want to create a new instance of. This property does not apply to the **`Signal`**- and **`CancelProcessInstanceCommands`**.

3. process-definition-id

   This is a required property for the **`NewProcessInstanceCommands`** and **`StartProcessInstanceCommands`** if the processdefinition property is not used. The value of this property should refer to the already-deployed process definition of which one wants to create a new instance. This property does not apply to the **`Signal`**- and **`CancelProcessInstanceCommands`**.

4. actor

   Use this optional property to specify the jBPM `actor` identifier. (It applies only to **`NewProcessInstanceCommand`** and **`StartProcessInstanceCommand`**.)

5. key

This is a optional property to specify the value of the jBPM key. The key is a string based business key property on the process instance. The combination of business key and process definition must be unique if a business key is supplied. The key value can hold an MVEL expression to extract the desired value from the EsbMessage. For example, if 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`**.)

6. transition-name

   This is a optional property. It only applies to **`StartProcessInstanceCommand`** and **`Signal`**. It is used if there is more then 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`**.

7. esbToBpmVars

   This is a optional property for the **`New`**- and **`StartProcessInstanceCommands`**. This property defines a list of variables that need to be extracted from the ESB message and added to the jBPM context for the particular process instance. The list consists of mapping elements. Each mapping element 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.

   - 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.

8. 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.

12.4.1. 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.

12.5. jBPM-to-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`**.

12.5.1. 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>
```

Figure 12.11. *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 you are not using the reply-to-originator functionality.

- esbServiceName

  The name of the ESB service. This is required if you are not using the reply-to-originator functionality.

- replyToOriginator

  Specify the *reply* or *fault* originator address previously stored in the process instance on creation.

- globalProcessScope

  This element is optional. This boolean valued parameter sets the default scope in which the bpmToEsbVars are looked up. If the globalProcessScope is set to true the variables are looked for up the token hierarchy (process-instance scope). If set to false it retrieves the variables in the scope of the token. If the token does not have a variable for the given name, the variable is searched for up the token hierarchy. If omitted the globalProcessScope is set to false for retrieving variables.

- bpmToEsbVars

  This element is optional. This element takes a list of mapping sub-elements to map a jBPM context variable to a location in the EsbMessage. Each mapping element can have the following attributes.

  - bpm

    This is a required attribute. The name of the variable in jBPM context. The name can be MVEL type expression so you can extract a specific field from a larger object. The MVEL root is set to the jBPM *ContextInstance*.

    Example 12.1. Mapping jBPM context variable to a location in the EsbMessage

    ```
    <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" />
    ```

    You can also reference jBPM context variable names directly.

  - esb

    This attribute is optional. The name of the variable on the EsbMessage. The name can be a MVEL type expression. By default the variable is set as a named parameter on the body of the EsbMessage. If you decide to omit the esb attribute, the value of the bpm attribute is used.

  - 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 can contain a Boolean value used to override the setting of the **globalProcessScope** for this mapping.

Note

Red Hat recommends activating the debug-level logging when working on the variable mapping configuration.

12.5.2. ESBActionHandler

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.)

Example 12.2. Configuration for the EsbActionHandler

```
<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>
```

The configuration for the **EsbActionHandler** action extends that for the **EsbNotifier**. The extensions are the as follows:

- **esbToBpmVars**

  This sub-element is optional. It is identical to the **esbToBpmVars** property (mentioned in *Section 12.4, “ 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**.

- **exceptionTransition**

  This is an optional element. It is 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 for one of those transitions to handle *exception processing*.

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 12.3, "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.

Example 12.3. Specifying a Time-Out Value for an Action

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

12.5.3. jBPM-to-ESB Exception Handling

There are two scenarios in which exceptions can arise:

1. A **MessageDeliveryException** will be thrown by the **ServiceInvoker** 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`) to the jBPM node that will deal with this situation.

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.

Figure 12.12, "Three exception handling scenarios: time-out, exception-transition and exception-decision." illustrates several error handling scenerios that will be discussed in detail below.

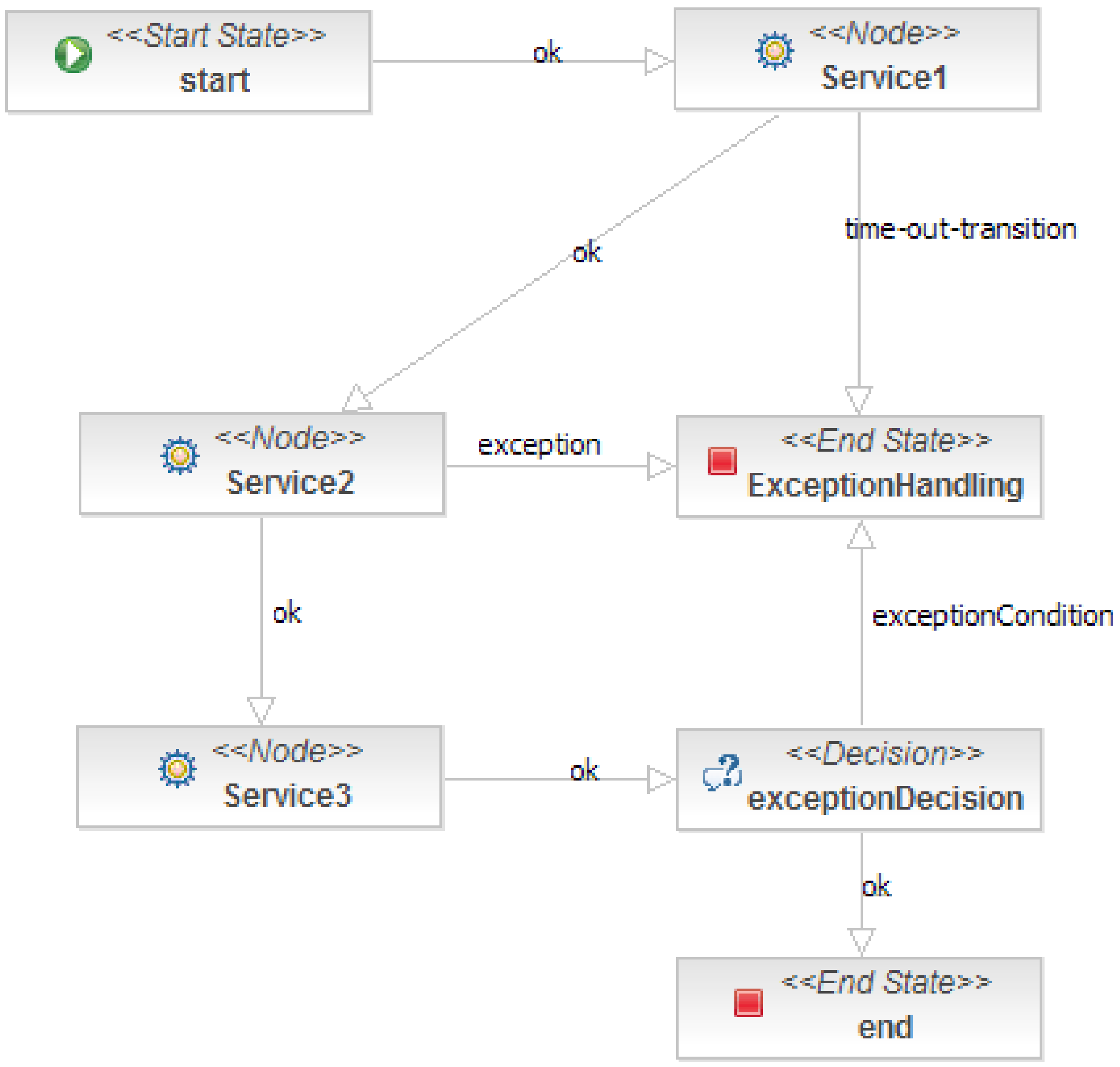


Figure 12.12. Three exception handling scenarios: time-out, exception-transition and exception-decision.

12.5.4. 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.

12.5.5. 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.

Example 12.4. Definition of Service Two

```
<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>
```

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.

12.5.6. 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:

Example 12.5. Definition of Service Three

```
<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>
```

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.

For more information about conditional transitions please refer to the *jBPM Reference Guide*.

Service Orchestration

13.1. Introduction

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

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**.

13.2. Orchestrating Web Services

JBossESB provides WS-BPEL support via its Web Service components. For details on these components and how to configure and use them, refer to the Web Services Chapter of the SOA Platform Programmers Guide [1].

JBoss and JBossESB also have a special support agreement with *ActiveEndpoints* [2] for their award wining ActiveBPEL WS-BPEL Engine. JBoss ESB includes with the **`webservice_bpel`** QuickStart which demonstrates how the JBoss ESB and ActiveBPEL can collaborate effectively to provide a WS-BPEL based orchestration layer on top of a set of Services that don't expose Webservice Interfaces. JBossESB provides the Webservice Integration and ActiveBPEL provides the Process Orchestration. A number of flash based walk-thrus of this Quickstart are also *available online* [3].

Note

ActiveEndpoints WS-BPEL engine does not run on versions of JBossAS since 4.0.5. However, it can be deployed and run successfully on Tomcat as our examples illustrate.

13.3. Orchestration Diagram

A key component of Service Orchestration is to use a flow-chart like design tool to design and deploy processes. The jBPM IDE can be used for just this. *Figure 13.1, “Orchestration diagram for the **`bpm_orchestration4`** QuickStart ”* shows an example of such a flow-chart, which represents a simplified order process. This example is taken from the **`bpm_orchestration4`** QuickStart which ships with JBossESB.

[1] The JBoss Enterprise SOA Programmers Guide is provided as the file **`Programmers_Guide.pdf`** or can be viewed online at *http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/*
[2] http://www.active-endpoints.com/
[3] http://labs.jboss.com/jbossesb/resources/tutorials/bpel-demos/bpel-demos.html

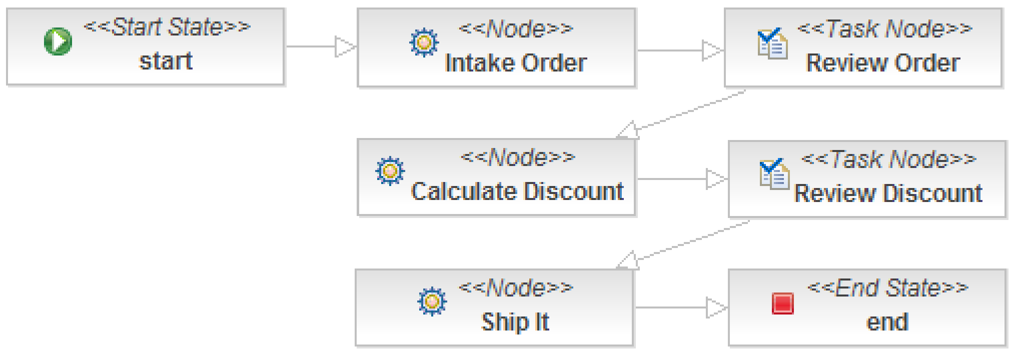


Figure 13.1. Orchestration diagram for the **`bpm_orchestration4`** QuickStart

In *Figure 13.1, “Orchestration diagram for the **`bpm_orchestration4`** QuickStart ”* three of the nodes are JBossESB Services, the *Intake Order*, *Calculate Discount* and the *Ship It* nodes. For these nodes the regular *Node* type was used, which is why these are labeled with <<*Node*>>. Each of these nodes have the **`EsbActionHandler`** attached to the node itself. This means that the jBPM node will send a request to the Service and then it will remain in a wait state, waiting for the ESB to call back into the node with the response of the Service. The response of the service can then be used within jBPM context.

For example when the Service of the *Intake Order* responds, the response is then used to populate the *Review Order* form. The *Review Order* node is a *Task Node*. Task Nodes are designed for human interaction. In this case someone is required to review the order before the Order Process can process.

To create the diagram in *Figure 13.1, “Orchestration diagram for the **`bpm_orchestration4`** QuickStart ”*, select **File > New > Other**, and from the Selection wizard select **JBoss jBPM Process Definition**. The wizard will direct you to save the process definition. From an organizational point of view it is recommended use one directory per process definition, as you will typically end up with multiple files per process design.

After creating a new process definition. You can drag and drop any item from jBPM IDE menu palette into the process design view. You can switch between the design and source modes if needed to check the XML elements that are being added, or to add XML fragments that are needed for the integration. Recently a new type of node was added called *ESB Service*.

Before building the *Order Process* diagram of *Figure 13.1, “Orchestration diagram for the **`bpm_orchestration4`** QuickStart ”* you need to create and test the three Services. These services are ordinary ESB services and are defined in the **`jboss-esb.xml`**. Check the **`jboss-esb.xml`** of the **`bpm_orchestration4`** QuickStart if you want details on them, but they only thing of importance to the Service Orchestration are the Services names and categories as shown in the following **`jboss-esb.xml`** 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>
```

These Service can be referenced using the **`EsbActionHandler`** or **`EsbNotifier`** Action Handlers.The **`EsbActionHandler`** is used when jBPM expects a response, while the **`EsbNotifier`** can be used if no response back to jBPM is needed.

Now that the ESB services are known we drag the *Start* state node into the design view. A new process instance will start a process at this node. Next we drag in a *Node* (or *ESB Service* if available). Name this Node *Intake Order*. We can connect the Start and the Intake Order Node by selecting *Transition* from the menu and by subsequently clicking on the Start and Intake Order Node. You should now see an arrow connecting these two nodes, pointing to the Intake Order Node.

Next we need to add the Service and Category names to the Intake Node. Select the **Source** view. The *Intake Order* Node should look like:

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

Then we add the EsbHandlerAction class reference and the subelement configuration for the Service Category and Name, BPM_Orchestration4 and *IntakeService* respectively.

```
<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>
```

Next we want to send the some jBPM context variables along with the Service call. In this example we have a variable named *entireOrderAsXML* which we want to set in the default position on the EsbMessage body. For this to happen we add:

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

which will cause the XML content of the variable “entireOrderAsXML” to end up in the body of the EsbMessage, so the IntakeService will have access to it, and the Service can work on it, by letting it flow through each action in the Action Pipeline. When the last action is reached it the replyTo is checked and the EsbMessage is send to the JBpmCallBack Service, which will make a call back into jBPM signaling the “Intake Order” node to transition to the next node (“Review Order”). This time we will want to send some variables from the EsbMessage to jBPM. Note that you can send entire objects as long both contexts can load the object's class. For the mapping back to jBPM we add an “esbToEsbVars” element. Putting it all together we end up with:

```
<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>
```

So after this Service returns we have the following variables in the jBPM context for this process: entireOrderAsXML, entireOrderAsObject, entireCustomerAsObject, and for demo purposes we also added some flattened variables: order_orderid, order_totalAmount, order_priority, customer_firstName, customer_lastName and customer_status.

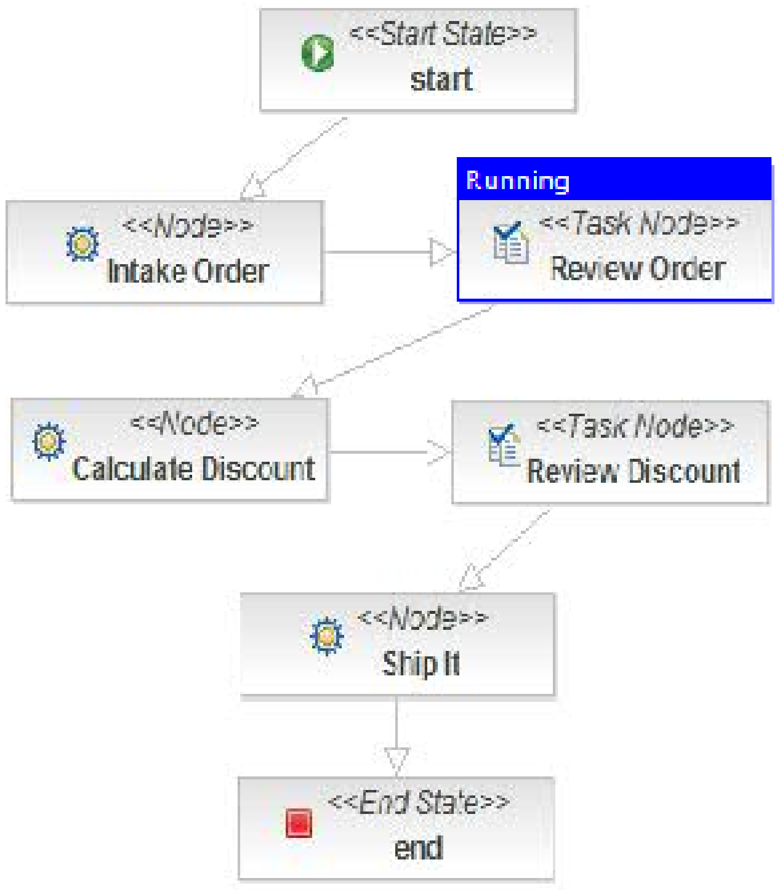


Figure 13.2. The Order process reached the “Review Order” node

In our Order process we require a human to review the order. We therefore add a “Task Node” and add the task “Order Review”, which needs to be performed by someone with actor_id “user”. The XML-fragment looks like:

```
<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>
```

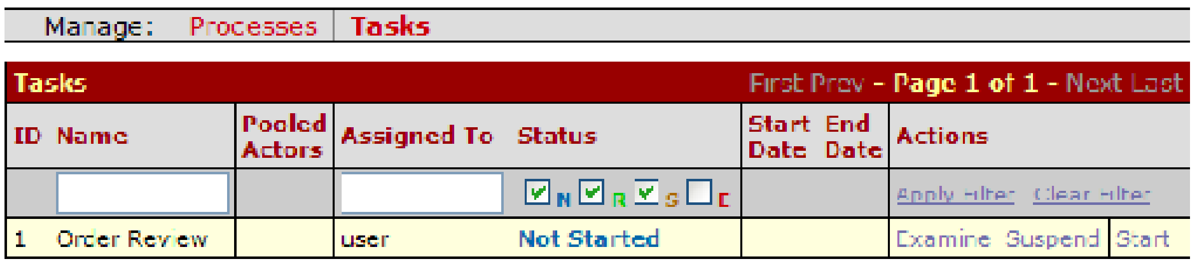
In order to display these variables in a form in the jbpm-console we need to create an xhtml dataform (see the Review_Order.xhtml file in the bpm_orchestration4 quick start [JBESB-QS] and tie this for this TaskNode using the forms.xml file:

```
<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 used in two task nodes. The variables are referenced in the Review Order form like this, which references the variables set in the jBPM context:

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

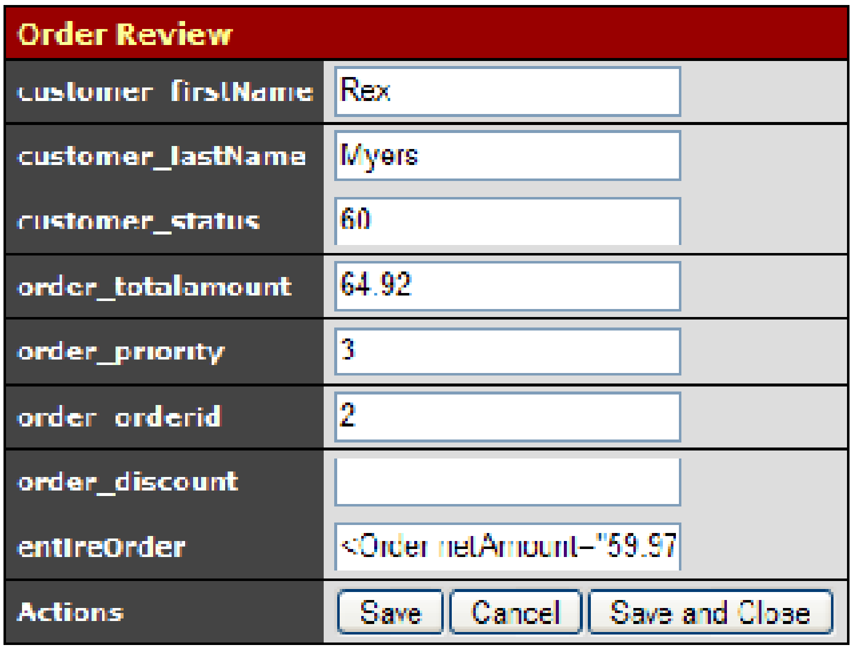
When the process reaches the “Review Node”, as shown in *Figure 13.2, “The Order process reached the “Review Order” node”*. When the 'user' user logs into the jbpm-console the user can click on 'Tasks” to see a list of tasks, as shown in *Figure 13.3, “The task list for user 'user'”*. The user can 'examine' the task by clicking on it and the user will be presented with a form as shown in *Figure 13.4, “The "Order Review" form”*. The user can update some of the values and click “Save and Close” to let the process move 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 ☐ E			Apply filter Clear filter	
1	Order Review		user	Not Started			Examine Suspend	Start

Figure 13.3. The task list for user 'user'



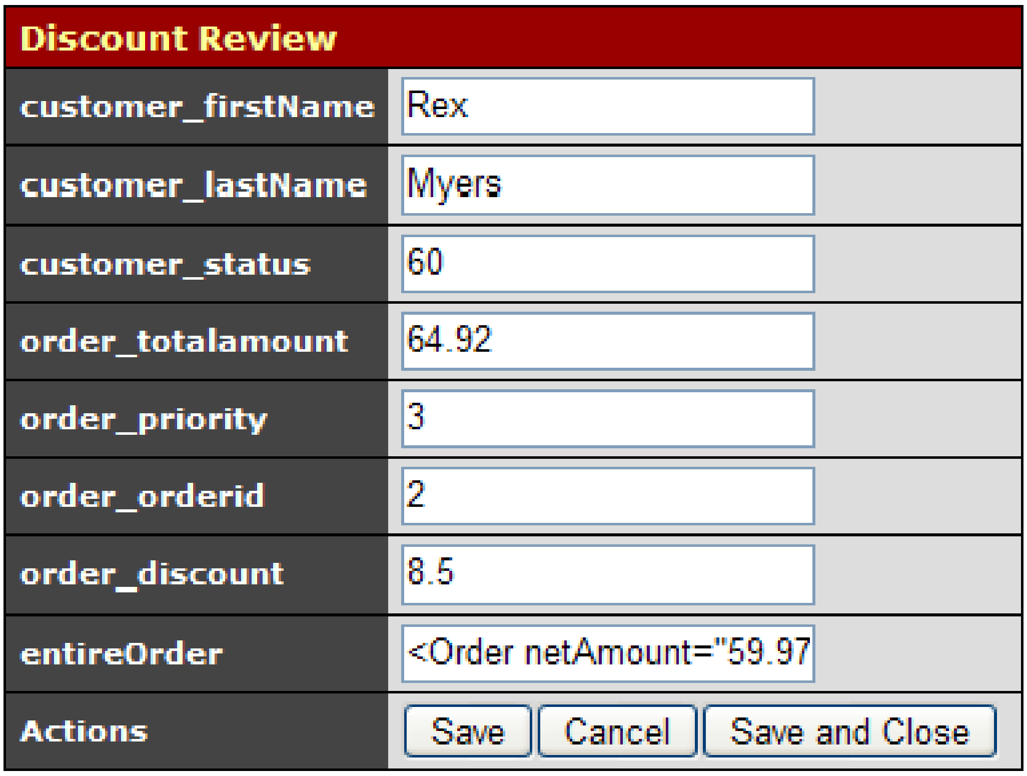
Order Review	
customer_firstName	Rex
customer_lastName	Myers
customer_status	60
order_totalamount	64.92
order_priority	3
order_orderid	2
order_discount	
entireOrder	<Order netAmount="59.97
Actions	Save Cancel Save and Close

Figure 13.4. The "Order Review" form

The next node is the “Calculate Discount” node. This is an ESB Service node again and the configuration looks like:

```
<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 the customer and orderHeader objects as well as the entireOrderAsXML, and computes a discount. The response maps the body.order_orderDiscount value onto a jBPM context variable called “order_-discount”, and the process is signaled to move to the “Review Discount” task 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 13.5. The "Discount Review" form

The user is asked to review the discount, which is set to 8.5. On “Save and Close” the process moves to the “Ship It” node, which again is an ESB Service. If you don't want the Order process to wait for the Ship It Service to be finished you can use the EsbNotifier action handler and attach 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 up. In **`bpm_orchestration4`** it uses drools to determine whether this order should be shipped 'normal' or 'express'.

13.4. Process Deployment and Instantiation

In the previous paragraph we create the process definition and we quietly assumed we had an instance of it to explain the process flow. But now that we have created the **`processdefinition.xml`**, we can deploy it to jBPM using the IDE, ant or the jbpm-console as explained in *Chapter 12, jBPM Integration* . In this example we use the IDE and deployed the files: Review_Order.xhtml, forms.xml, gpd.xml, processdefinition.xml and the processimage.jpg. On deployment the IDE creates a par achive and deploys this to the jBPM database. We do not recommend deploying Java code in par archives as it may cause class loading issues. Instead we recommend deploying classes in jar or esb archives.

When the process definition is deployed a new process instance can be created. It is interesting to note that we can use the 'StartProcessInstanceCommand” which allows us to create a process instance with some initial values already set. Take a look at:

```
<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>
```

where new process instance is invoked and using some groovy script, and the jBPM key is set to the value of 'OrderId' from an incoming order XML, and the same XML is subsequently put in jBPM context using the esbToBpmVars mapping. In the **`bpm_orchestration4`** QuickStart the XML came from the Seam DVD Store and the **`SampleOrder.xml`** looks like:

```
<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 ESB as well as jBPM deployments are hot. An extra feature of jBPM is that process deployments are versioned. Newly created process instances will use the latest version while existing process instances will finish using the process deployment on which they where started.

13.5. Conclusion

We have demonstrated how jBPM can be used to orchestrate Services as well as do Human Task Management. Note that you are free to use any jBPM feature. For instance look at the QuickStart **`bpm_orchestration2`** how to use the jBPM fork and join concepts.

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*.

14.1. 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 14.1. 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.

14.2. 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>
```

The section in the property file called “dbstore” has all the settings required by the database implementation of the message store. The standard settings, like URL, db user, password, pool sizes can all be modified here.

The scripts for the required database schema are very simple. They can be found under **`lib/jbossesb.esb/message-store-sql/<db_type>/create_database.sql`** of your JBoss ESB installation.

The structure of the table can be seen from the sample SQL.

Example 14.1. Sample SQL for message store table creation

```
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`)
);
```

the uuid column is used to store a unique key for this message, in the format of a standard URI. A key for a message would look like:

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

This logic uses the UUID random number generator and the type will be the type of the stored message. JBossESB ships with JBOSS_XML and JAVA_SERIALIZED currently.

The “message” column will contain the actual message content.

The supplied database message store implementation works by invoking a connection manager to your configured database. Supplied with Jboss ESB is a standalone connection manager, and another for using a JNDI datasource.

To configure the database connection manager, you need to provide the connection manager implementation in the **`jbossesb-properties.xml`** file. The properties that you would need to change are:

```
<!--  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 supplied connection managers for managing 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.

Once you have implemented this interface, you update the properties file with your new class, and the connection manager factory will now use your implementation.

Security

Services in JBossESB can be configured to be secure which means that the service will only be executed if authentication succeeds and if the caller is authorized to execute the service.

A service can be invoked in one of two ways: 1.) Through a gateway or 2.) Directly via the Enterprise Service Bus by using 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, it creates an authentication request that is encrypted and then passed to the Enterprise Service Bus.

If one uses the **`ServiceInvoker`** instead, the gateway will not be utilised. Rather, it becomes the responsibility of the client to create the authentication request prior to invoking the service. Both of these options will be the objects of study in the following sections.

The default security implementation is based on the *Java Authentication and Authorization Service* (JAAS) but it is highly configurable, so it can be altered if one wishes to use an alternative system. The following sections describe the JAAS security components and how they can be configured.

15.1. Security Service Configuration

The Security Service, along with all the other settings, can be configured by editing 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>
```

`jbossesb-properties.xml` security settings

`org.jboss.soa.esb.services.security.implementationClass`
: This is the "concrete" *SecurityService* implementation that should be used. Required. The default setting is **`JaasSecurityService`**.

`org.jboss.soa.esb.services.security.callbackHandler`
: This is optional. It is a default **`CallbackHandler`** implementation, utilised when a JAAS-based **`SecurityService`** is employed. See "Customizing Security" for more information about the **`CallbackHandler`** property.

`org.jboss.soa.esb.services.security.sealAlgorithm`
: This is the algorithm to use when "sealing" the **`SecurityContext`**.

`org.jboss.soa.esb.services.security.sealKeySize`
: This is the size of the secret/symmetric key used to encrypt/decrypt the **`SecurityContext`**.

`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 by specifying the property of the same name that exists on the security element in the **`jboss-esb.xml`** file.

`org.jboss.soa.esb.services.security.contextPropagatorImplementationClass`
: This is an optional property that is used to configure a global **`SecurityContextPropagator`**. (For more details on the **`SecurityContextPropagator`**, please refer to the section on "Security Context Propagation."

`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`**.

`org.jboss.soa.esb.services.security.publicKeystorePassword`
: This is the password for the public keystore.

`org.jboss.soa.esb.services.security.publicKeyAlias`
: This is the alias to use.

`org.jboss.soa.esb.services.security.publicKeyPassword`
: This is the password for the alias if one was specified upon creation.

`org.jboss.soa.esb.services.security.publicKeyPassword`
: This is an optional cipher transformation. It is in the format, "**`algorithm/mode/padding`**." If this is not specified, the "keys" algorithm will be used by default.

The JAAS log-in modules are configured using the **`login-config.xml`** file located in the ***`$SOA_ROOT`*`/server/`*`$PROFILE`*`/conf/`** directory of one's JBoss Application Server. One can either use those that came pre-configured or add one's own.

The JBoss Enterprise Service Bus ships with an example keystore. This should not be used in a production environment. It is only provided as a sample to help users achieve a working security configuration "out-of-the-box." The sample keystore can be updated with custom-generated pairs of keys.

15.1.1. Configuring Security on Services

Security is configured for each service. A service in JBossESB can be declared as being secured and that it requires authentication.

Services are configured by adding a “security” element to the service in **`jbossesb.xml`**:

```
<security moduleName="messaging" runAs="adminRole"
  rolesAllowed="adminRole, normalUsers" callbackHandler=
  "org.jboss.internal.soa.esb.services.security.User PassCallbackHandler">
<property name="property1" value="value1"/>
<property name="property2" value="value2"/>
</security>
```

Security properties description

moduleName
: This is a named module that exists in the **`conf/login-config.xml`** file.

runAs
: This is an optional runAs role.

rolesAllowed
: This is an optional, 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.

callbackHandler
: An optional **`CallbackHandler`** that will override the one defined in **`jbossesb-properties.xml`**.

property
: Optional properties can be defined which will be made available to the **`CallbackHandler`** implementation.

Security properties overrides:

`org.jboss.soa.esb.services.security.contextTimeout`
: Optional property that lets the service override the global security context timeout (ms) specified in **`jbossesb-properties.xml`**.

`org.jboss.soa.esb.services.security.contextPropagatorImplementationClass`
: Optional property that lets the service override the global security context propagator class implementation specified in **`jbossesb-properties.xml`**.

Example 15.1. Overriding 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>
```

15.2. 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 be performed 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.

15.2.1. AuthenticationRequest

An AuthenticationRequest is intended to carry security information needed for authentication between a gateway and a service, or between two services.

An instance of this class should be set on the message object before calling the service configured for authentication:

```
byte[] encrypted =
    PublicCryptoUtil.INSTANCE.encrypt((Serializable) authRequest);

message.getContext.setContext(SecurityService.AUTH_REQUEST, encrypted);
```

The authentication context is encrypted and then set in the message context. This will be decrypted by the ESB to perform authentication. See *Section 15.1, “ Security Service Configuration ”* for information on how to configure the public keystore for this purpose.

The **`security_basic`** QuickStart shows an example of using a external client and how to prepare the Message before using the **`ServiceInvoker`**, see the **`SendEsbMessage`** class for more information. This quickstart also shows how you can configure **`jbossesb-properties.xml`** for client usage.

15.3. JBossESB SecurityContext

A SecurityContext in JBossESB is an object that is local to a specific ESB node, or really to the JVM of the node. The SecurityContext is created after a successful authentication has be performed and it will be used locally in the ESB where it was created to save having to re-authenticate with every call.

A timeout is specified for the context which is the time in milliseconds that the context is valid for. This value can be specified globally in **`jbossesb-properties.xml`** and can be overridden by specifying the value in the **`jboss-esb.xml`** of the specific service. Additional details can be found in *Section 15.1.1, “Configuring Security on Services”* and *Section 15.1, “ Security Service Configuration ”*.

15.4. 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 15.1.1, "Configuring Security on Services"* and *Section 15.1, " 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.

15.5. Customizing security

The default security implementation in JBossESB is based on JAAS and named JaasSecurityService. Custom login modules can be added in **`conf/login-config.xml`** of an JBoss Application Server.

Since different login 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 class name of a class that implements the **`EsbCallbackHandler`** interface:

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

The **`AuthenticationRequest`** will contain the principal and credentials needed authenticate a caller.

The **`SecurityConfig`** will give access to the security configuration in **`jboss-esb.xml`**.

Both of these are made available to the **`CallbackHandler`** which it can use to populate the **`Callback`** instances required by the login module.

15.6. Provided Login Modules

This section lists the login modules provided with JBossESB. Please note that all login modules available with JBoss AS are available as well and custom login modules should be easy to add.

15.6.1. CertificateLoginModule

This login module performs authentication by verifiying that a certificate passed with the call to the ESB, can be verified against a certificate in a local keystore.

Upon successful authentication the certificates Common Name(CN) will be used to create a principal. If role mapping is in use then it is the CN that will be used in the role mapping. Refer to *Section 15.6.2, “Role Mapping”* for details on the role mapping functionality.

Example 15.2. CertificateLoginModule configuration

```
<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
: Identifies the JAAS Login module to use. This module will be specified in JBossAS login-config.xml.

rolesAllowed
: Comma separated lite of roles that are allowed to execute this service.

alias
: The alias to look up in the local keystore which will be used to verify the callers certificate.

Here is an example of a fragment from 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 15.6.2, “Role Mapping”* for additional details.

15.6.2. Role Mapping

This file is can be optionally specified in **`login-config.xml`** by using rolesPropertiesFile. This can point to a file on the local file system or to a file on the classpath. This file contains a mapping of users to roles:

```
# 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`** QuickStart.

15.7. `SecurityService`

The **`SecurityService`** interface is the central component in Enterprise Service Bus security. It is shown below:

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

Updated for 4.3.CP05 Release

Revision 1.4	**Tue Apr 23 2010**	**David Le Sage** *dlesage@redhat.com*

Updated for 4.3.CP04

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

Updated for SOA 4.3.CP03

Revision 1.1	**Fri 18 Sep 2009**	**Darrin Mison** *dmison@redhat.com*

Updated for 4.3.CP02
SOA-1107 - Clarified RMI JUDDI content. Section 3.3
SOA-1127 - Clarified EPR registration comments. Section 1.6
SOA-1337 - Updated troubleshooting details. Section 5.1
SOA-1352 - Updated SOAP content to reflect current support status of the feature. Section 3.5
SOA-1426 - Added XPath and Namespaces content. Section 9.4.1

Revision 1.0	**Tue 9 Sep 2008**	**Darrin Mison** *dmison@redhat.com*

Updated for 4.3.CP01

Revision 1.0	**Tue 9 Sep 2008**	**Darrin Mison** *dmison@redhat.com*

Created