JBoss Enterprise SOA Platform 4.3

Services Guide

Your guide to services available on the JBoss Enterprise SOA Platform

JBoss®
a division of Red Hat

JBoss Enterprise SOA Platform 4.3 Services Guide
Your guide to services available on the JBoss Enterprise SOA Platform
Edition 1.0

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

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

This book contains details of the services available with the 4.3 GA release of 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 key caps 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 key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

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

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

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, 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; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

---

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

> 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 **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.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

`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:

> When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules* (*MPMs*). Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types 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 are presented and highlighted 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

A note is a tip or shortcut or alternative approach 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 Important boxes won't cause data loss but may cause irritation and frustration.

Warning

A Warning 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_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?

1.1. Introduction

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

1.1.1. Why do I need it?

It is not difficult to discover, manage and interface with business partners on a small scale using manual or ad hoc techniques. However this approach does not scale with increases in the number of services, the frequency of interactions and the physical distribution of the environment. A registry solution based on agreed upon standards provides a common way to publish and discover services. It offers a central place where you query whether a partner has a service that is compatible with in-house technologies or to find a list of companies that support shipping services on the other side of the globe.

Service registries are central to most service oriented architectures and at runtime act as a contact point to correlate service requests to actual behaviors. A service registry has meta-data entries for all artifacts within the SOA that are used at both runtime and design time. Items inside a service registry may include service description artifacts such as WSDL, Service Policy descriptions, various XML schema used by services, artifacts representing different versions of services, governance and security artifacts (e.g., certificates, audit trails), etc. During the design phase, business process designers may use the registry to link together calls to several services to create a workflow or business process.

Note

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

1.1.2. How do I use it?

From a business analyst’s perspective, it is similar to an Internet search engine for business processes. From a developers perspective, they use the registry to publish services and query the registry to discover services matching various criteria.

1.1.3. Registry Vs 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.

1.1.4. SOA Components

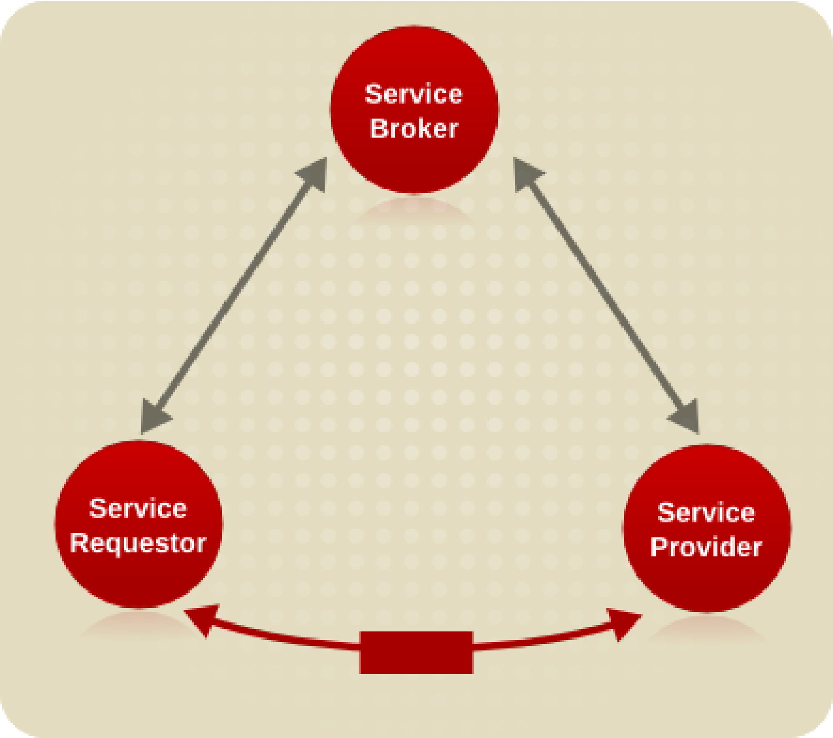
"A SOA is a specific type of distributed system in which the agents are 'services'."[1].

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.



1.1.5. UDDI

The Universal Description, Discovery and Integration (UDDI) registry is a directory service for Web Services. It enables 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 information about the company providing the service.
- *Yellow pages*: categorize businesses based on the industry their services cater to.
- *Green pages*: provide information that will enable a client to bind to the service that is being provided.

1.1.6. The Registry and the JBoss SOA Platform

The registry plays a central role within JBoss SOA. It is used to store endpoint references (EPRs) for the services deployed. It may be updated dynamically when services first start-up, or statically by an external administrator.

It is not possible for the registry to determine the status of the entities its data represents. For example if an EPR is registered with the registry then there can be no guarantee that the EPR is valid (it may be malformed) or it may represent a service that is no longer active. At present JBoss SOA does not perform life-cycle monitoring of the services that are deployed within it. If services fail or move elsewhere, their EPRs that may reside within the registry will remain until they are explicitly updated or removed by an administrator. Therefore, if you get warnings or errors related to EPRs obtained from the registry, you should consider informing those responsible for the services.

Configuring the Registry

2.1. Introduction

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 we use a JAXR implementation (Scout) and a UDDI (jUDDI), in an embedded way.

The following properties can be used to configure the JBoss SOA Registry. In the **`jbossesb-properties.xml`** there is section called 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.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`
A class that implements the jbossesb Registry interface. We have provided one implementation (JAXRRegistry interface).

`org.jboss.soa.esb.registry.factoryClass`
The class name of the JAXR ConnectionFactory implementation.

`org.jboss.soa.esb.registry.queryManagerURI`
The URI used by JAXR to query.

`org.jboss.soa.esb.registry.lifeCycleManagerURI`
The URI used by JAXR to edit.

`org.jboss.soa.esb.registry.user`
The username used for edits.

`org.jboss.soa.esb.registry.password`
The password for the specified user.

`org.jboss.soa.esb.scout.proxy.transportClass`
The name of the class used by Scout to do the transport from Scout to the UDDI.

2.2. The Components Involved

The registry can be configured in many ways. *Figure 2.1, “Blue print of the Registry component architecture”* shows a blue print of all the registry components. From the top down we can see that JBoss SOA funnels all interaction with the registry through the Registry Interface. By default it then calls into a JAXR implementation of this interface. The JAXR API needs an implementation, which by default is Scout. The Scout JAXR implementation calls into a jUDDI registry. However there are many other configuration options.

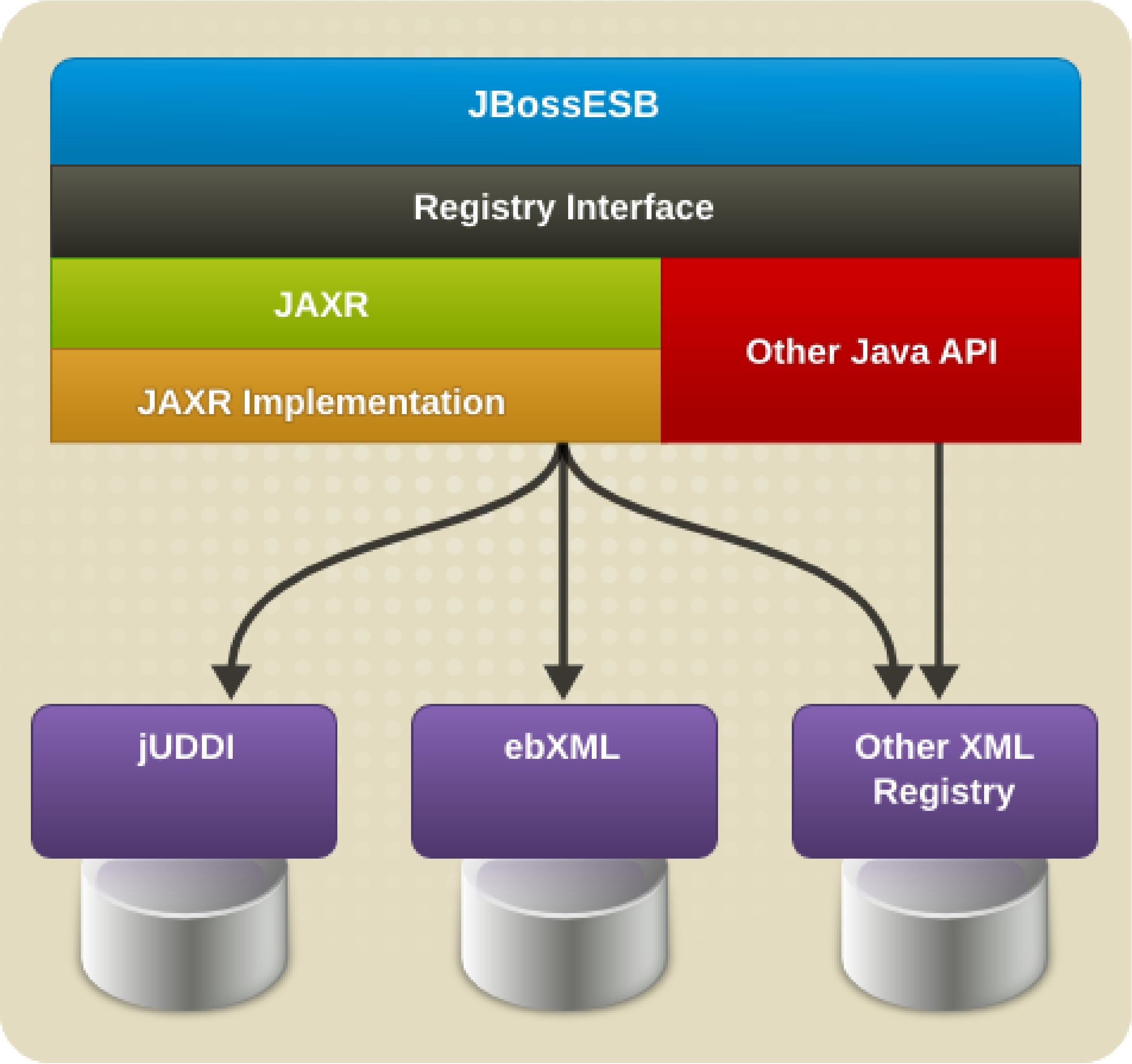


Figure 2.1. Blue print of the Registry component architecture

2.3. The Registry Implementation Class

org.jboss.soa.esb.registry.implementationClass

By default we use the JAXR API. The JAXR API is a convenient API since it allows us to connect any kind of XML based registry or repository. However, if for example you want to use Systinet's Java API you can do that by writing your own SystinetRegistryImplentation class and referencing it in this property.

2.4. Using JAXR

org.jboss.soa.esb.registry.factoryClass

If you decided to use JAXR then you will have to pick a JAXR implementation to use. This property is used to configure that class. JBoss SOA defaults to using Scout, and this property is set to the Scout factory class 'org.apache.ws.scout.registry.ConnectionFactoryImpl'.

The next step is to tell the JAXR implementation the location of the registry or repository for querying and updating. This is done by setting the org.jboss.soa.esb.registry.queryManagerURI, and org.jboss.soa.esb.registry.lifeCycleManagerURI. The username and password for the UDDI are set in org.jboss.soa.esb.registry.user and org.jboss.soa.esb.registry.password respectively.

2.5. Using Scout and jUDDI

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

When using Scout and jUDDI there is an additional optional parameter. This is the transport class that should be used for communication between Scout and jUDDI. Thus far there are four implementations of this class which are based on SOAP, SAAJ, RMI and Local (embedded Java).

Important

Note that when you change the transport, you will also have to change the query and lifecycle URIs.

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

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

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

Example 2.1. Using SOAP

```
<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 RMI

```
<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"/>
```

Example 2.3. Using Local

You have two requirements when using jUDDI:

1. You need access to the jUDDI database. You will need to create a schema in your database, and add the jbossesb publisher. The jUDDI-registry directory contains database creation scripts for several common database systems.

2. The configuration of jUDDI is done in **`esb.juddi.xml`**. If you do not use a datasource you need to take special care to set the following properties:

   ```
   <entry key="juddi.isUseDataSource">false</entry>
   <entry key="juddi.jdbcDriver">com.mysql.jdbc.Driver</entry>
   <entry key="juddi.jdbcUrl">jdbc:mysql://localhost/juddi</entry>
   <entry key="juddi.jdbcUsername">juddi</entry>
   <entry key="juddi.jdbcPassword">juddi</entry>
   ```

   If you do use a datasource you need something like:

   ```
   <entry key="juddi.isUseDataSource">true</entry>
   <entry key="juddi.dataSource">java:comp/env/jdbc/juddiDB</entry>
   ```

The database can be automatically created if the specified user has enough rights to create tables. You must also ensure that the *`isCreateDatabase`* flag is set to true, and that the *`sqlFiles`* parameter settings indicates the database that you are using. The jUDDI creation scripts are located in the **`juddi.jar`**.

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

jUDDI supports the following databases:

- Daffodildb
- DB2

- Derby
- Firebird
- HSQLDB
- informix
- jdatastore
- mysql
- oracle
- postgresql
- Sybase (can be used for Microsoft SQLServer)
- totalxml

Registry Configuration Examples

3.1. Introduction

By default the JBoss SOA Platform is configured to use the JAXR API using the Scout JAXR implementation and jUDDI as the registry. Here are some examples of how you can deploy them.

3.2. Embedded

All JBoss SOA components [1] can embed the registry. They all can connect to the same database or use different ones.

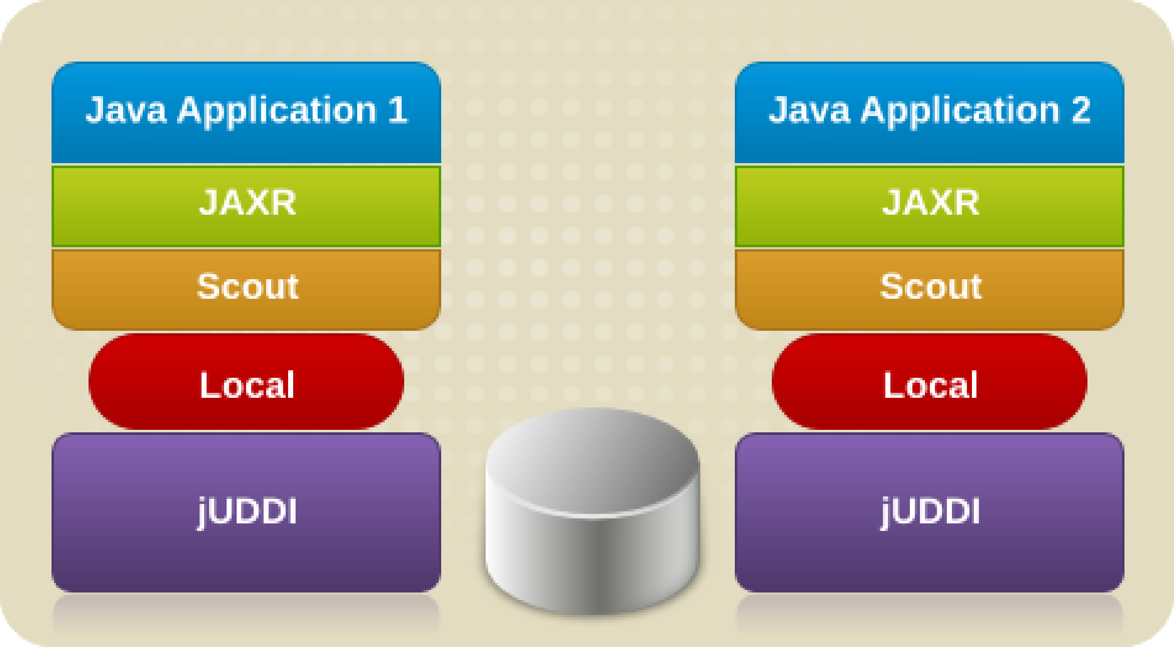


Figure 3.1. Embedded jUDDI

---

[1] In this case the "component" being refered to is actually the JVM.

```
<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 3.1. Properties for Embedded jUDDI

3.3. RMI using the juddi.war or jbossesb.sar

The JBoss SOA includes **`juddi.war`** in the **`jUDDI-registry`** directory. When deployed this brings up the regular webservices but also an RMI service. You also need to deploy a datasource which points to your jUDDI database. An example file is supplied for MySQL.

The **`jbossesb.sar`** only registers a RMI service. So you would only need to deploy the **`juddi.war`** if you need webservice access.

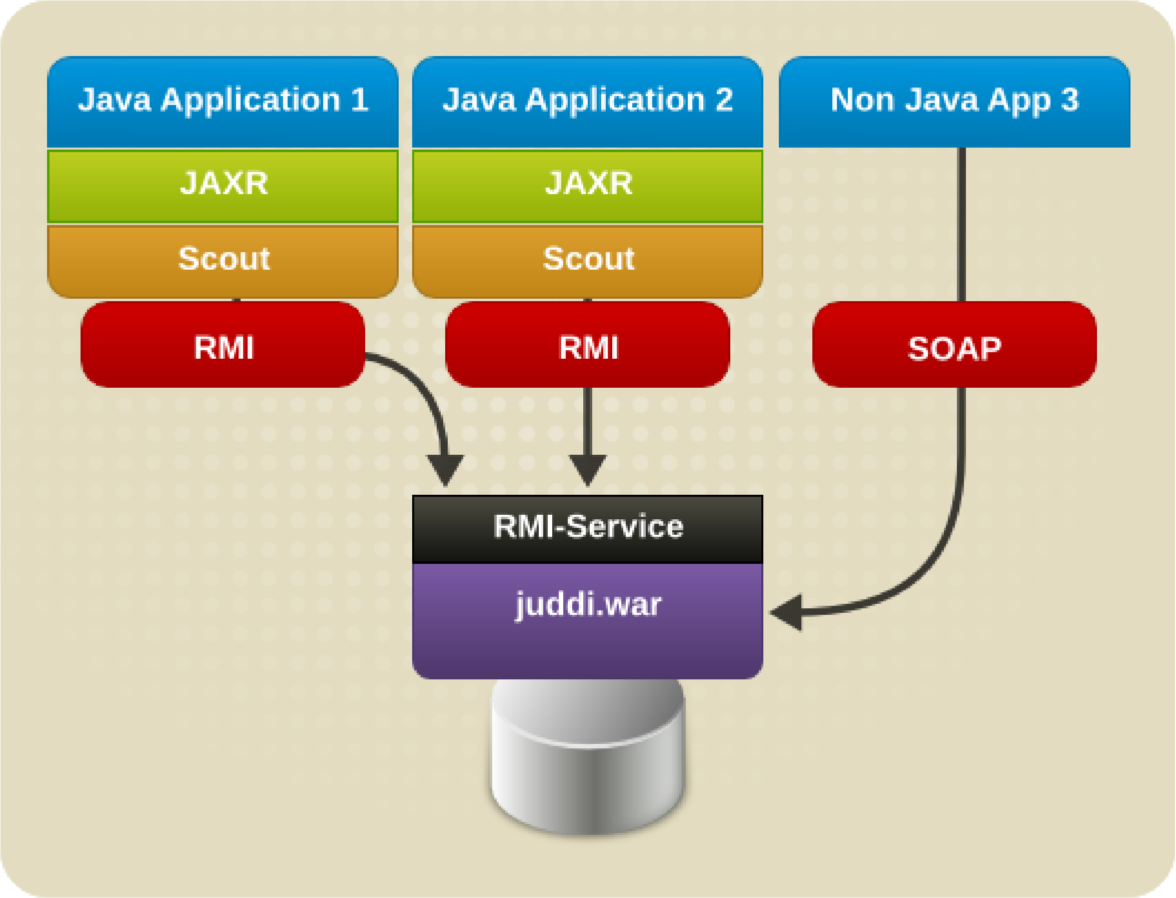


Figure 3.2. RMI using the juddi.war

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

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

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

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

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

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

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

Example 3.2. Properties

The **`juddi.war`** provides a RMI Service if enabled by the following setting in the **`web.xml`**

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

Make sure to include, for example, the following JNDI settings in your **`juddi.properties`**:

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

Important

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

3.4. RMI using your own JNDI Registration of the RMI Service

If you don't want to deploy the **`juddi.war`** you can setup another JBoss SOA component in the the same JVM as jUDDI to register the RMI service.

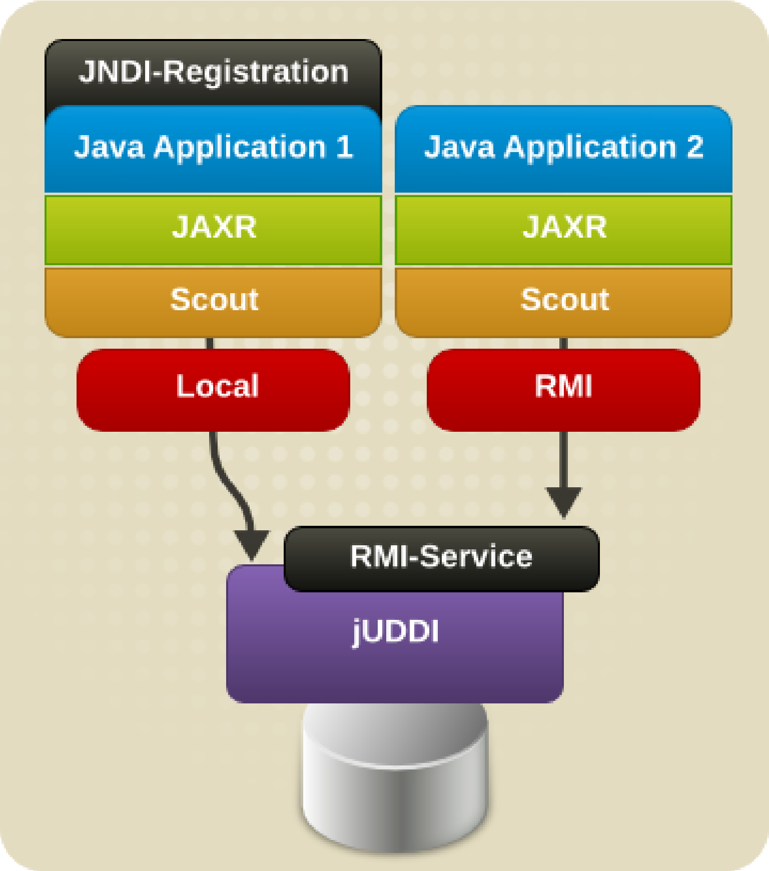


Figure 3.3. RMI using your own JNDI registration

In this example Application1 will need to be configured with the Local settings, and Application2 will need the RMI settings.

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

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

 <property name="org.jboss.soa.esb.registry.queryManagerURI"
  value="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 3.3. 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="jnp://localhost:1099/InquiryService?
org.apache.juddi.registry.rmi.Inquiry#inquire"/>

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

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

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

Example 3.4. RMI settings, used for Application2

For Application2, using RMI, the hostname of the queryManagerURI and lifeCycleManagerURI properties need to be set to that of the host on which the jUDDI service is running.

Obviously application1 needs to have access to a naming service.

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

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

Example 3.5. JNDI registration process for Application1

3.5. SOAP

Communication between Scout and jUDDI can also be provided via SOAP based webservices. As with RMI you need to deploy the **`juddi.war`** and configure the datasource.

If you are not also using RMI you should disable the RMI service by commenting out the RegisterServicesWithJNDI servlet in the **`web.xml`**.

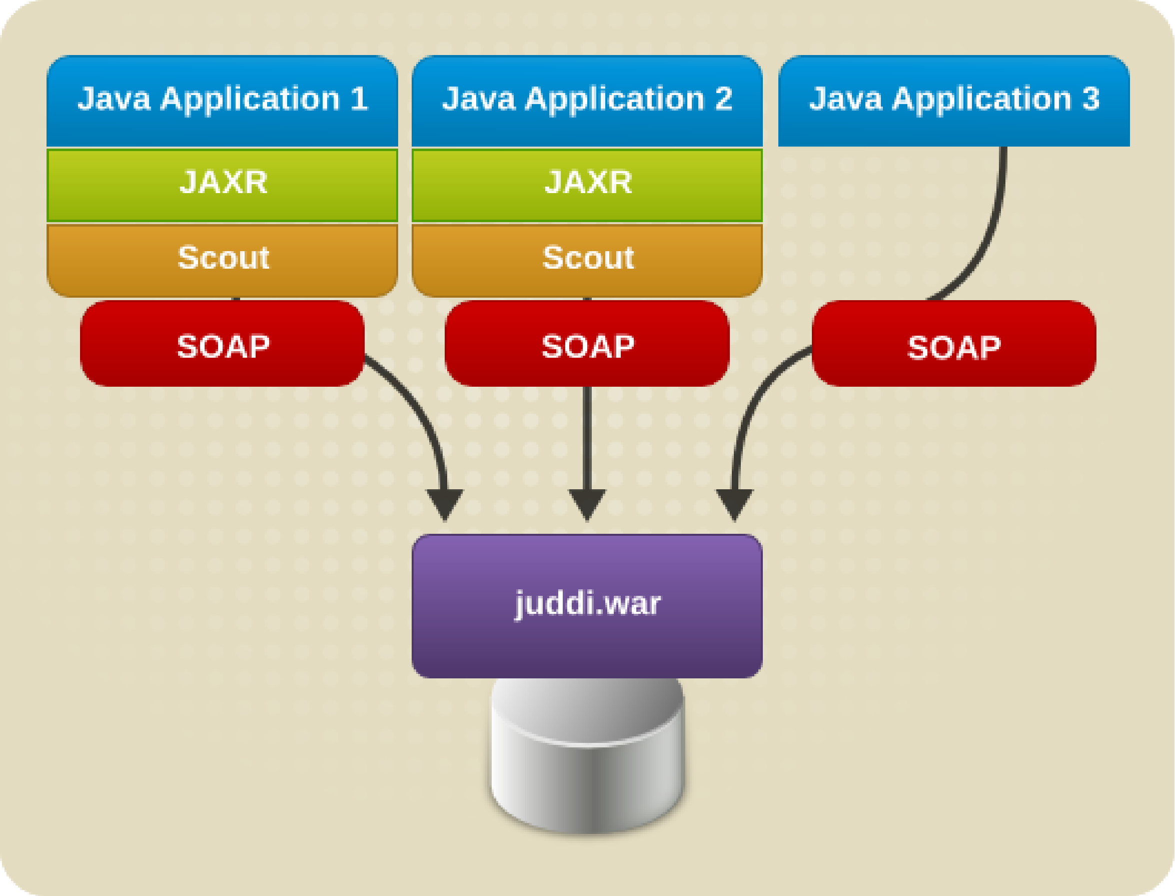


Figure 3.4. Accessing a jUDDI registry using SOAP

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

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

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

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

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

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

Example 3.6. Properties for accessing a jUDDI registry using SOAP

Note

JBossAS 4.2 ships with older versions of Scout and jUDDI. It is recommended to remove the **`juddi.sar`** to prevent versioning issues if you must deploy using 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

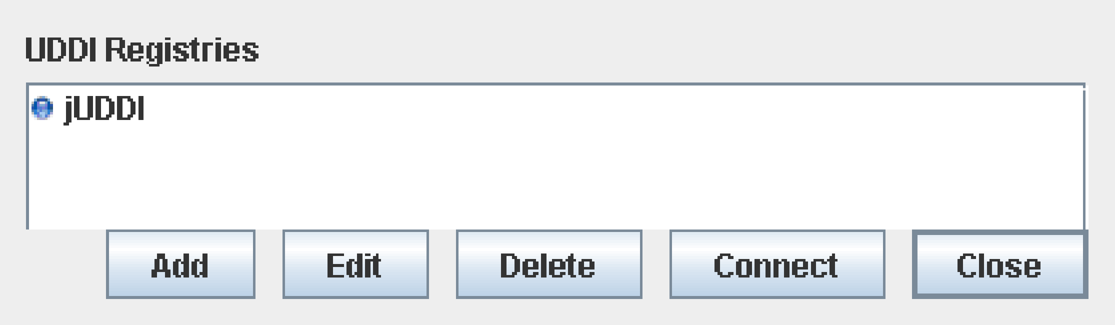
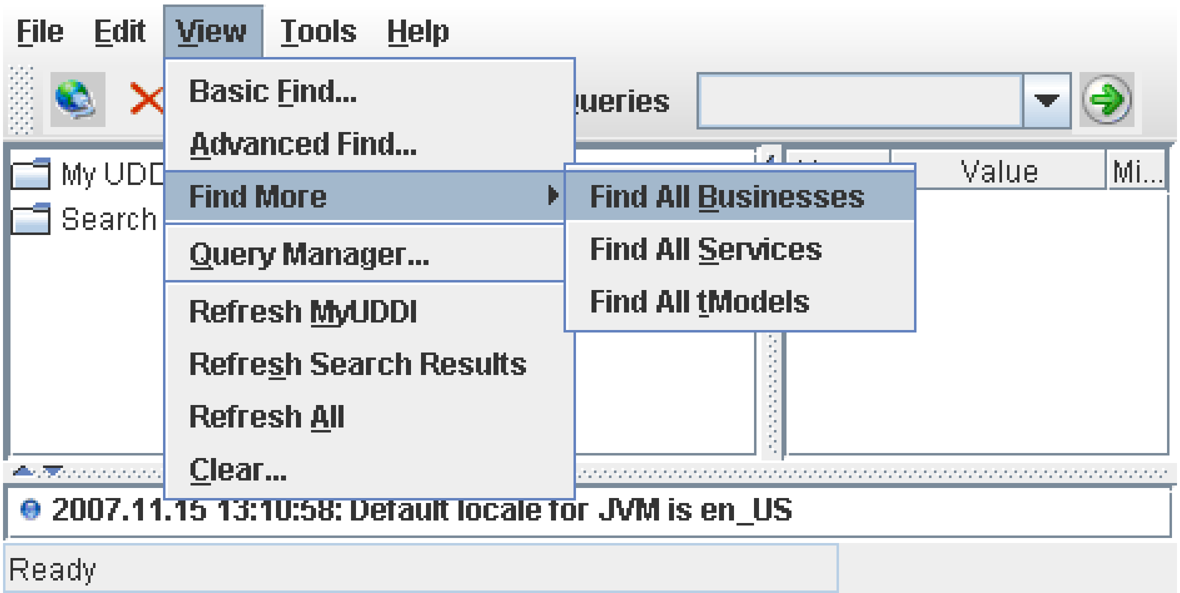


Figure 4.1. Add a connection

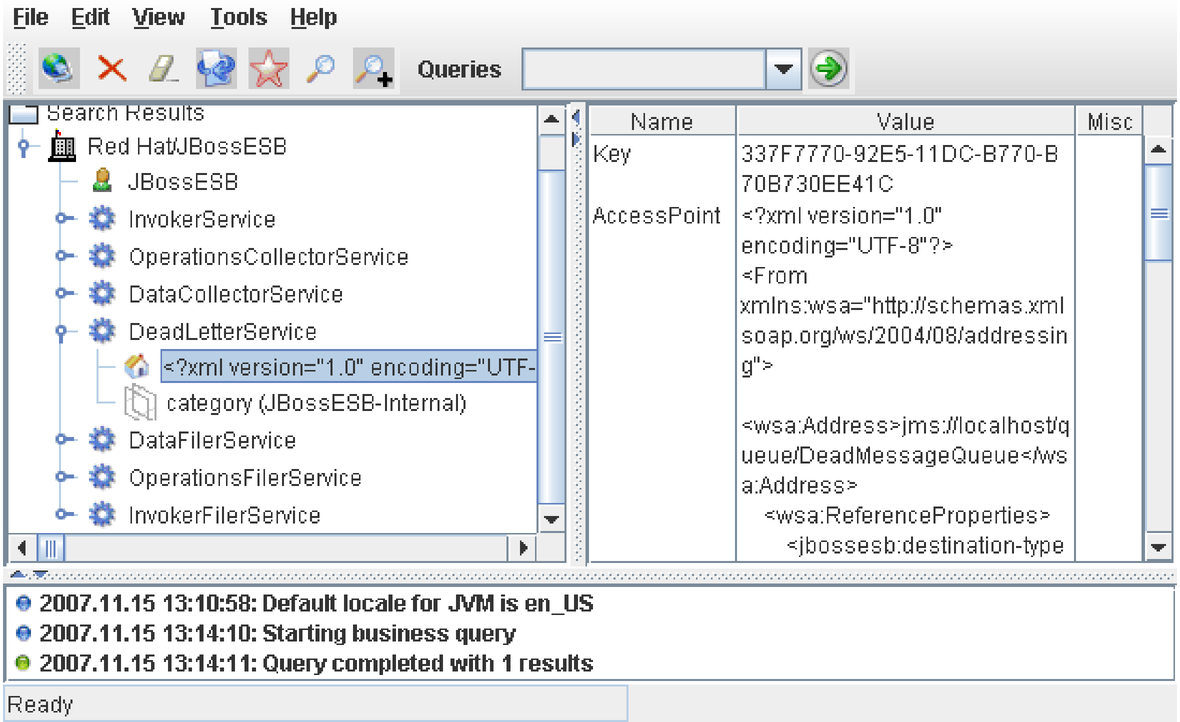
Click on 'connect' and select **View** > **Find More** > **Find All Businesses**



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

Figure 4.2. View all Businesses

In the left pane you should see the Red Hat/JBossESB organization. You can 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

Each ServiceBinding contains an EPR in its AccessPoint.

Some features of **ub** may not work, but it should give enough functionality to maintain jUDDI. The JBoss ESB community project is currently looking for a good web based console for maintaining jUDDI.

Registry Troubleshooting

5.1. Scout and jUDDI pitfalls

- Make sure to put our version of the **`jaxr-api-1.0.jar`**, **`scout-0.7rc2-embedded.jar`** and the **`juddi-embedded.jar`** first. The versions of these libraries that are included with the JBoss Application Server are incompatiable. This should get resolved in future release of JBoss Application Server.
- If you use RMI you need the **`juddi-client.jar`**.
- Make sure the **`jbossesb-properties.xml`** file is in the classpath and readable or else the registry will try to instantiate classes with the name of 'null'.
- Make sure you have a **`juddi.properties`** file on your classpath for jUDDI to configure itself. JBoss SOA uses **`esb.juddi.xml`**, but generates the **`juddi.properties`** file for jUDDI to read.
- Make sure to read the README in the **`jUDDI-registry`** directory, for instructions regarding prepopulating your own jUDDI database.
- In the event that a service fails or does not shut down cleanly, it is possible that old entries may persist within a registry. You will have to remove these manually.

5.2. More Information

Further community resources for Registry Troubleshooting 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

The JBoss SOA Rule Service allows you to deploy rules created in JBoss Drools as services. This has two major benefits. First, the amount of required client code to integrate rules into your application environment is dramatically reduced. Secondly, rules can be accessed as part of a action chain or orchestarted business process. An understanding of JBoss Drools will aid the reader in understanding these types of services.

> **Note**
>
> JBoss Drools is supported out of the box but it is possible to use other rule engines in its place.

Rule Services are supported by the BusinessRuleProcessor action class and the DroolsRuleService which implements the RuleService interface.

The BusinessRuleProcessor supports rules loaded from the classpath that are defined in **`.drl`** and **`.dsl`** files, dslr files (domain specific language support), and decision tables using **`.xls`** files. However there is no way to specify multiple rule files in the **`jboss-esb.xml`** file. These file-based rules are primarily for testing, prototypes, and very simple rule services.

Complex rule services need to use the Drools RuleAgent.

The RuleService uses the RuleAgent to access rule packages from the Drools BRMS or local file system. These rule packages can contain thousands of rules, created through the Drools BRMS business rule editor, imported DRL files, rules written in a Domain Specific Language, and rules from Decision Tables.

Use of the Drools RuleAgent is the recommended approach for production systems.

The BusinessRuleProcessor action supports both Drools stateless and stateful execution models.

Stateless Rule Services

Most rule services will be stateless. In the stateless model, a message is sent to the rule service that includes all the facts in the message body to be inserted into the rule engine. The rules execute and update either of 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, updating either of the message or the facts until a final message is received that tells the rule service to dispose of the stateful session. This configuration is currently limited in that there can only be a single stateful rule service in the message flow in this model.

Rule Services Using Drools

7.1. Introduction

The Rule Service support in the JBoss SOA Platform uses JBoss Rules as its rule engine. This integration is acheived using:

- The BusinessRulesProcessor action class
- Rules written in Drools drl, dsl, decision table, or business rule editor.
- The ESBMessage
- The objects in the ESBMessage content, which is the data going into the rules engine.

When a message gets send 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

A rule set can be created using **Red Hat Developer Studio**.Since the message is added as a global, you need to add **`jbossesb-rosetta.jar`** to your Drools project.

For a detailed discussion on rule creation and the Drools language itself please refer to the included JBoss Rules Reference Guide.

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

1. All rules deployed as a rule service must define the ESBMessage 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 or DroolsRuleService will always set the message as a global.

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

   Example 7.1. defining ESBMessage as a global

2. If additional globals other than the ESBMessage are required, they must be set in higher *salience*> rule.

   The BusinessRulesProcessor and DroolsRuleService does not provide a means to set globals in **`jboss-esb.xml`**. This could be supported in the future.

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

Example 7.2. declaring a global in a higher salience rule

3. The ESBRuleService does not provide a means to start a RuleFlow from the rule service itself. This could be supported in the future.

7.3. Rule Service Consumers

The consumer of a rule service has little to worry about. There is no need for the consumer to creating rulebases or working memories, inserting facts or firing the rules. Instead the consumer just has to worry about adding facts, and possibly some properties, to the message.

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

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

Example 7.3. Adding objects to a message directly

In other cases the data may be in an XML message, and a transformation service will be added to the message flow to transform the XML to 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" />
  ```

- A 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:

```
<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 7.4. Rules are in a drl, execution is stateless

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

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 7.5. Rules are in a drl, execution is stateful

```
<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 7.6. Rules in a Domain Specific Language, 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 7.7. Rules in a DecisionTable, stateless execution

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

Example 7.8. Rules in the BRMS, stateless execution

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	Optional reference to a file containing the ruleSet. The set of rules that is used to evaluate the content. Only 1 ruleSet can be given for each rule service instance.
ruleLanguage	Optional reference to a file containing the definition of a Domain Specific Language to be used for evaluating the rule set. If this is used, the file in ruleSet should be a dslr file.
ruleReload	Optional property which can be set to true to enable *hot redeployment* of rule sets. 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.

Property	Description
decisionTable	Optional reference to a file containing the definition of a spreadsheet containing rules.
ruleAgentProperties	Optional reference to a properties file containing the location (URL or file path) to the compiled rule packages. Note there is no need to specify ruleReload with a ruleAgent as this is controlled through the properties file.
stateful	Optional property which can be set to true to specify that the rule service will receive multiple messages over time, adding the new facts to the rule engine working memory and re-executing the rules each time.
object-paths	Optional property to pass Message objects into JBoss Rules working memory.

7.5. Object Paths

Note that JBossRules treats objects as shallow objects to achieve highly optimized performance. To evaluate an object deeper than the object tree the optional object-paths property can be used. This results in the extraction of objects from the message, using an “ESB Message Object Path”.

The expresssion language MVEL is used to extract the object and the path used should follow the syntax:

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

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

objectname
: name of the object. Attachments can be named or numbered, so for attachments this can be a number.

beannames
: optionally you traverse a bean graph by specifying bean names

Example MVEL expressions

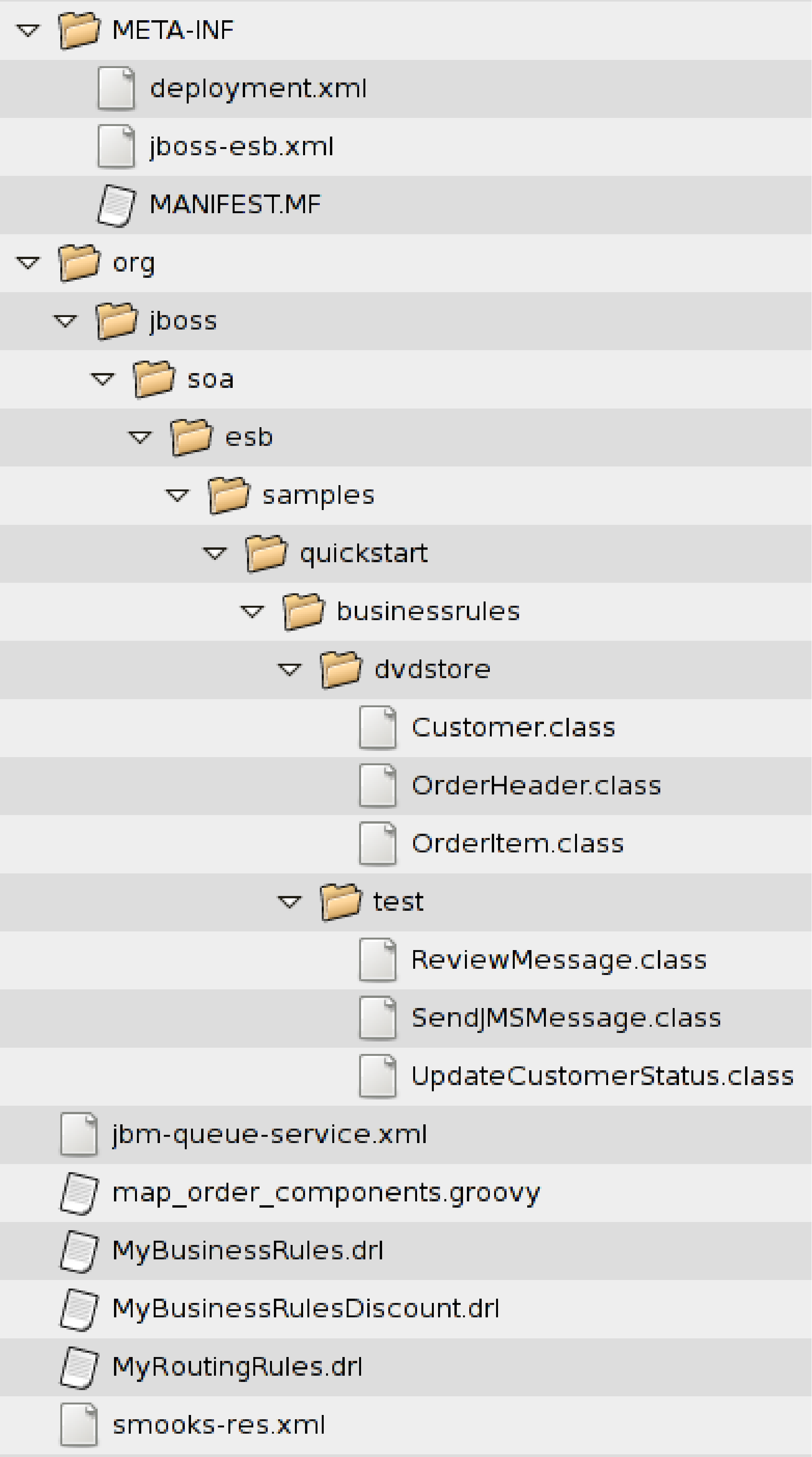
Expression	Result
`properties.Order`	gets the property object named **`Order`**
`attachment.1`	gets the first attachment Object
`attachment.AttachmentOne`	gets the attachment named **`AttachmentOne`**
`attachment.1.Order`	gets `getOrder()` return object on the attached Object.
`body.Order1.lineitem`	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 to traverse the bean graph.

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

The Object Mapper cannot flatten out entire collections. If you need to do that you have to use a transformation on the message first, to unroll the collection.

7.6. Deploying and Packaging

It is recommended that you package up your code into units of functionality, using .esb packages. The idea is to package up your routing rules alongside the rule services that use the rule sets. The figure below shows a layout of the business_rules_service quickstart to demonstrate a typical package.



- META-INF
  - deployment.xml
  - jboss-esb.xml
  - MANIFEST.MF
- org
  - jboss
    - soa
      - esb
        - samples
          - quickstart
            - businessrules
              - dvdstore
                - Customer.class
                - OrderHeader.class
                - OrderItem.class
              - test
                - ReviewMessage.class
                - SendJMSMessage.class
                - UpdateCustomerStatus.class
- jbm-queue-service.xml
- map_order_components.groovy
- MyBusinessRules.drl
- MyBusinessRulesDiscount.drl
- MyRoutingRules.drl
- smooks-res.xml

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

Typically, information within the ESB is conveniently packaged, transferred, and stored in the form of a message. Messages are addressed to Endpoint References (services or clients), that identify the machine/process/object that will ultimately deal with the content of the message. However, what happens if the specified address is no longer valid? For example, the service has failed or been removed from service? It is also possible that the service no longer deals with messages of that particular type; in which case, presumably some other service still handles the original function, but how should the message be handled? What if other services besides the intended recipient are interested in the message's content? What if no destination is specified?

One possible solution to these problems is Content-Based Routing. 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 opening it up and applying a set of rules to its content to determine the parties interested in its content.

The ESB can determine the destination of a given message based on the content of that message, freeing the sending application from having to know anything about where a message is going to end up.

Content-based routing and filtering networks are 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.

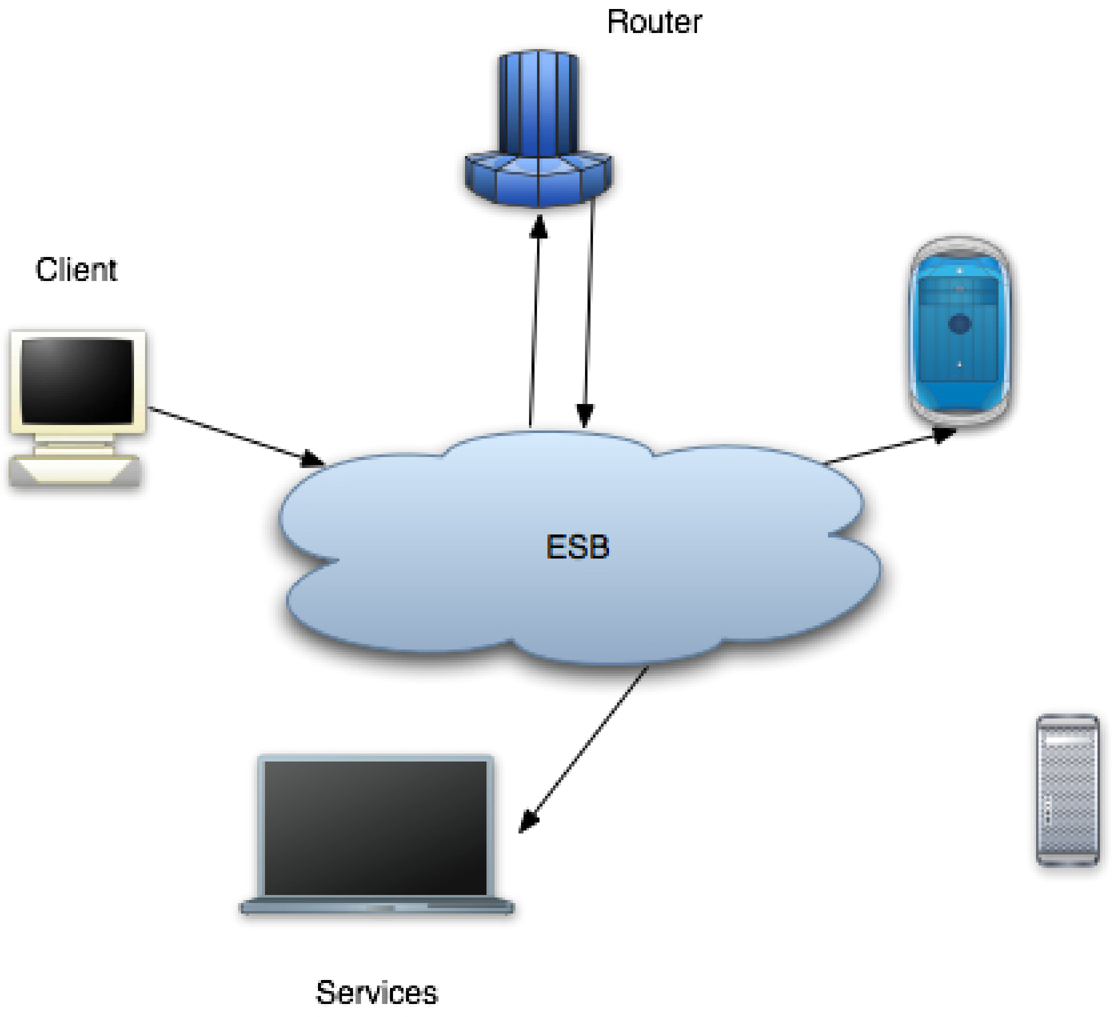
8.2. Simple Example

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 the messages they receive, apply rules to that content, and forward the messages as the rules dictate.

In addition to routers and services, some systems may also include harvesters, which specialize in finding interesting information, packaging it up as a formatted message before 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 typical CBR architecture using an ESB. At the heart of the system, represented by the cloud, is the ESB. Messages are sent from the client into the ESB, which directs them to the Router. This is then responsible for sending the messages to their ultimate destination (or destinations, as shown in this example).


Router
Client
ESB
Services

Content Based Routing Using Drools

9.1. Introduction

The Content Based Router (CBR) in the JBossESB uses JBossRules/Drools as its evaluation engine. JBossESB integrates with Drools through three different routing action classes,

- a routing rule set, written in Drools drl (and optionally dsl) language.
- The EsbMessage content, either the serialized XML, or objects in the message, which is the data going into the rules engine.
- destination(s) which is the result coming out of the rules engine.

When a message gets send to the CBR, a certain rule set will evaluate the message content and return a set of Service destinations. We discuss how a target rule set can be targeted, how the message content is evaluated and what is done with the destination results.

9.2. Three Different Routing Action Classes

JBossESB ships with three slightly different routing action classes. Each of these action classes implements an Enterprise Integration Pattern. The JBossESB Wiki contains more information about the Enterprise Integration Pattern. The following actions are supported:

org.jboss.soa.esb.actions.ContentBasedRouter

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 it is evaluating it against. The CBR throws an exception when no destinations are matched for a given rule set/message combination. This action will terminate any further pipeline processing, so it should be the last action of your pipeline.

org.jboss.soa.esb.actions.ContentBasedWireTap

Implements the WireTap pattern. The WireTap is an Enterprise Integration Pattern (EIP) where a copy of the message is send to a control channel. The CBR-WT is identical in functionality to the ContentBasedRouter, however it does not terminate the pipeline which makes it suitable to be used as a WireTap.

rg.jboss.soa.esb.actions.MessageFilter

Implements the Message-Filter pattern. The Message Filter pattern represents the case where messages can simply be dropped if certain content requirements are not met. The CBR-MF is identical in functionality to the ContentBasedRouter, but it does not throw an exception if the rule set does not match any destinations. In this case the message is simply filter out.

9.3. Rule Set Creation

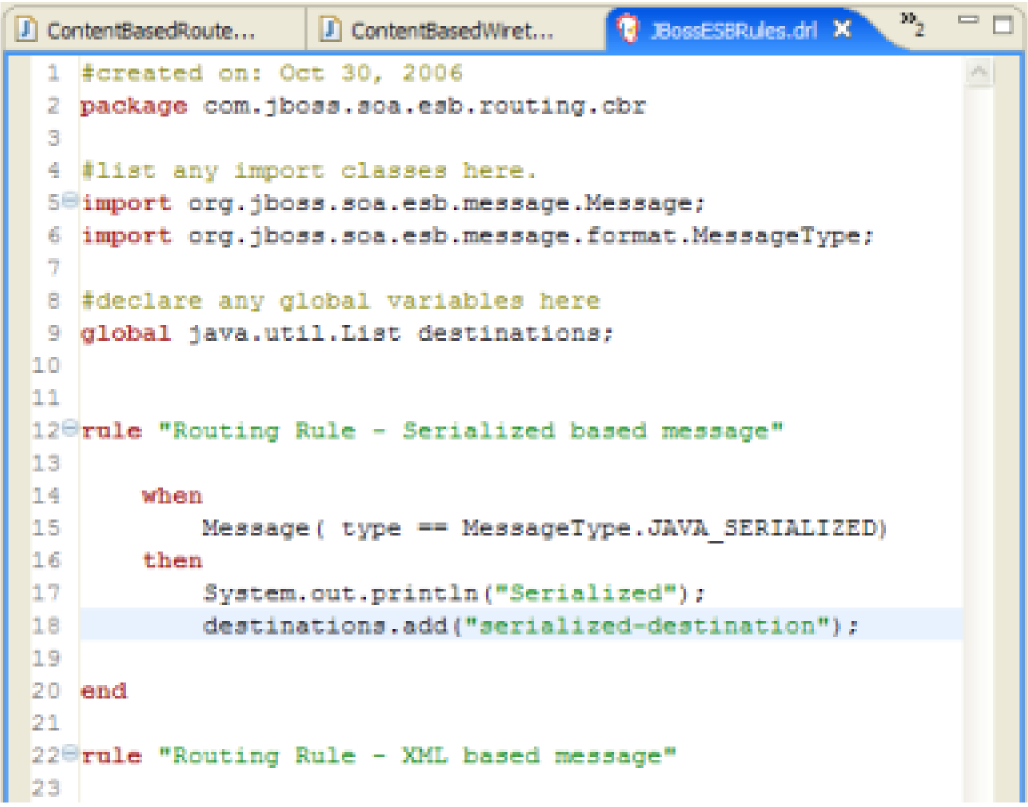
A rule set can be created using the JBossIDE or Red Hat Developer Studio which includes a plug-in for JBossRules. *Figure 9.1, “Create a new ruleSet using JBossIDE or JBoss Developer Studio”* shows a screen shot of the plug-in. For a detailed discussion on rule creation and the Drools language itself please see the Drools documention. To turn a regular ruleSet into a Countent Based Routing RuleSet you must be evaluating an EsbMessage and the rule match should result in a List of Strings containing the service destination names. To do this you need to make sure you remember two things:

- your rule set imports the EsbMessage

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

- and your rule set defines the following global variable which will make the list of destinations available to the ESB

  ```
  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 ruleSet using JBossIDE or JBoss Developer Studio

The message will be added to the working memory of the rules engine. Figure 2 shows an example where the MessageType is used to determine to which destination the Message is going to be send. This particular ruleSet is shipped in the JBossESBRules.drl file and the rule checks if the type is XML or Serializable.

9.4. XPath Domain Specific Language

For XML-based messages it is convenient to do XPath based evaluation. To support this we ship a “Domain Specific Language” implementation which allows us to use XPath expressions in the rule file. defined in the XPathLanguage.dsl. To use it you need to reference it in your ruleSet with: `expander XPathLanguage.dsl`

Currently the XPath Language makes sure the message is of the type JBOSS_XML and it defines

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 it's value equals the value.

3. **`xpathGreaterThan`** *`<element>`*, *`<value>`* : yields true if the element is found and it's value is greater than the value.

4. **`xpathLessThan`** *`<element>`*, *`<value>`* : yields true if the element is found and it's value is lower then the value.

The **`XPathLanguage.dsl`** is defined in a file called **`XPathLanguage.dsl`**, and can be customized if needed, or you can define your own DSL altogether. The Quickstart called fun_cbr demonstrates this use of XPath.

9.5. Configuration

Now that we have seen all the individual pieces how does it all tie together? It basically all comes down to configuration at this point, which is all done in your **`jboss-esb.xml`**. The Service Configuration below shows a service configuration fragment. In this fragment the service is listening on a JMS queue.

Each ESBMessage is passed on to in this case the ContentBasedRouter action class which is loaded with a certain rule set. It sets the ESBMessage into Working Memory, fires the rules, obtains the list of destinations and routes copies of the ESBMessage to these services. It uses the rule set JbossESBRules.drl, which matches two destinations, name 'xml-destination' and 'serialized-destination'. These names are mapped to real service names in the 'route-to' section.

```
<service category="MessageRouting"
 name="YourServiceName"
 description="CBR Service">
 <listeners>
  <jms-listener name="CBR-Listener"
   busidref="QueueA" maxThreads="1">
  </jms-listener>
 </listeners>
 <actions>
  <action class="org.jboss.soa.esb.actions.ContentBasedRouter"
   name="YourActionName">
   <property name="ruleSet" value="JBossESBRules.drl"/>
   <property name="ruleReload" value="true"/>
   <property name="destinations">
    <route-to destination-name="xml-destination"
     service-category="category01"
     service-name="jbossesbtest1" />
    <route-to destination-name="serialized-destination"
     service-category="category02"
     service-name="jbossesbtest2" />
   </property>
   <property name="object-paths">
    <object-path esb="body.test1" />
    <object-path esb="body.test2" />
   </property>
  </action>
 </actions>
</service>
```

Figure 9.2. Example Content Based Routing Service Configuration

The action attributes to the action tag are shown in the following table. The attributes specify which action is to be used and which name this action is to be given.

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

Table 9.1. CBR Action Configuration Attributes

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

Property	Description
ruleSet	Name of the filename containing the Drools ruleSet. The set of rules that is used to evaluate the content. Only 1 ruleSet can be given for each CBR instance.

Property	Description
ruleLanguage	Optional reference to a file containing the definition of a Domain Specific Language to be used for evaluating the rule set.
ruleAgentProperties	This property points to a rule agent properties file located on the classpath. The properties file can contain a property that points to precompiled rules packages on the file system, in a directory, or identified by an URL for integration with the BRMS. See the “RuleAgent” section below for more information.
ruleReload	Optional property which can be set to true to enable 'hot' redeployment of rule sets. Note that this feature will cause some overhead on the rules processing. Note that rules will also reload if the .esb archive in which they live is redeployed.
stateful	Optional property which tells the RuleService to use a stateful session where facts will be remembered between invokations. See the “Stateful Rules” section for more information about stateful rules.
destinations	A set of route-to properties each containing 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	Optional property to pass Message objects into Drools WorkingMemory.

Table 9.2. CBR Action Configuration Properties

9.6. Object Paths

Note that JBossRules treats objects as shallow objects to achieve highly optimized performance. To evaluate an object deeper than the object tree the optional **`object-paths`** property can be used, which results in the extraction of objects from the message, using an “ESB Message Object Path”. MVEL is used to extract the object and the path used should follow the syntax:

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

where,

location
    one of {body, header, properties, attachment}

objectname
    name of the object name, attachments can be named or numbered, so for attachments this can be a number too.

beannames
    optionally you traverse a bean graph by specifying bean names

Examples:

- **`properties.Order`** - gets the property object named **`Order`**
- **`attachment.1`** - gets the first attachment Object
- **`attachment.FirstAttachment`** - gets the attachment named **`FirstAttachment`**
- **`attachment.1.Order`** - gets `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 to traverse the bean graph.

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

Finally, the Object Mapper cannot flatten out entire collections, so if you need to do that you have to do a (Smooks-) transformation on the message first, to unroll the collection.

9.7. 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 disposed.

Stateful rule services must be told via messge properties when to continue with a current stateful session and when to dispose of it. To signal that you want to continue an existing stateful session two message properties must be set :

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

When you invoke the rules for the last time you must set “dispose” to true so that the working memory is disposed:

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

For more details about the RuleService please see *Chapter 7, Rule Services Using Drools*.

For an example of using stateful rules take a look at the business_ruleservice_stateful quickstart

9.8. RuleAgent

By using the rule agent property you can use precompiled rules packages that can be located on the local file system, in a local directory, or point to an URL. For information about the configuration options that exist for the properties file please refer to section *9.4.4.1. The Rule Agent*[1] of the Drools manual.

For more details about the RuleService please see *Chapter 7, Rule Services Using Drools*.

For an example of using a rule agent take a look at the business_ruleservice_ruleAgent quickstart.

9.9. RuleAgent and Business Rule Management System

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

---

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

For more details about the RuleService please see *Chapter 7, Rule Services Using Drools*.

For information about the how to install and configure the BRMS please refer to the chapter *Chapter 9*[3] of the Drools manual.

9.10. Executing Business Rules

Rule execution for modifying data in the message according to business rules is closely related to rule execution for routing. An example Quickstart called business_rule_service demonstrates this use case. This quickstart uses the action class **`org.jboss.soa.esb.actions.BusinessRulesProcessor`**.

The functionality of the Business Rule Processor (BRP) is identical to the Content Based Router, but it does not do any routing, instead it returns the modified EsbMessage for further action pipeline processing. You may mix business and routing rules in one rule set if you wish to do so, but routing will only occur if you use one of the three routing action classes mentioned earlier.

9.11. Changing RuleService Implementations

If you would like to use a different RuleService than the default one that is shipped with JBossESB, then this is possible by specifying the class you would like to use in the action configuration:

```

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

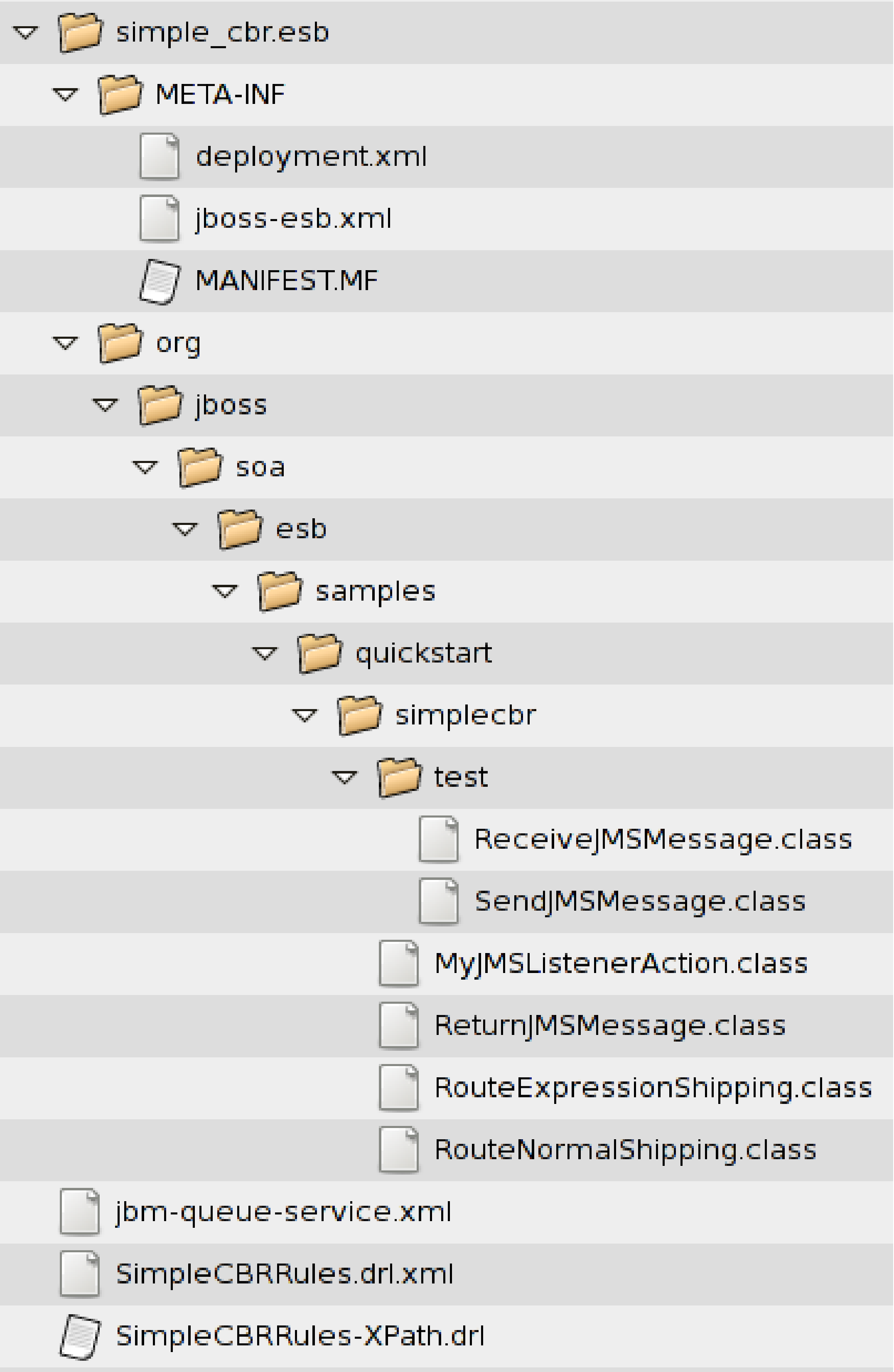
The requirement is that your rule service implements the interface: `org.jboss.soa.esb.services.rules.RuleService`.

9.12. Deployment and Packaging

It is recommended that you package up your code into units of functionality, using .esb packages. The idea is to package up your routing rules alongside the rule services that use the rule sets. *Figure 9.3, “Typical .esb archive which uses Drools.”* below shows a layout of the simple_cbr quickstart to demonstrate a typical package.

---

[3] http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html/ch09.html



```
simple_cbr.esb
    META-INF
        deployment.xml
        jboss-esb.xml
        MANIFEST.MF
    org
        jboss
            soa
                esb
                    samples
                        quickstart
                            simplecbr
                                test
                                    ReceiveJMSMessage.class
                                    SendJMSMessage.class
                                MyJMSListenerAction.class
                                ReturnJMSMessage.class
                                RouteExpressionShipping.class
                                RouteNormalShipping.class
    jbm-queue-service.xml
    SimpleCBRRules.drl.xml
    SimpleCBRRules-XPath.drl
```

Figure 9.3. Typical .esb archive which uses Drools.

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>
```

Content Based Routing Using Smooks

10.1. Introduction

The SmooksAction can be used for splitting HUGE messages into split fragments and performing Content-Based Routing on these split fragments.

An example of this might be a huge order message with thousands/millions of order items per message. You might need to split the order up by order item and route each order item split fragment to one or more destinations based on the fragment content. This example can be illustrated as follows:

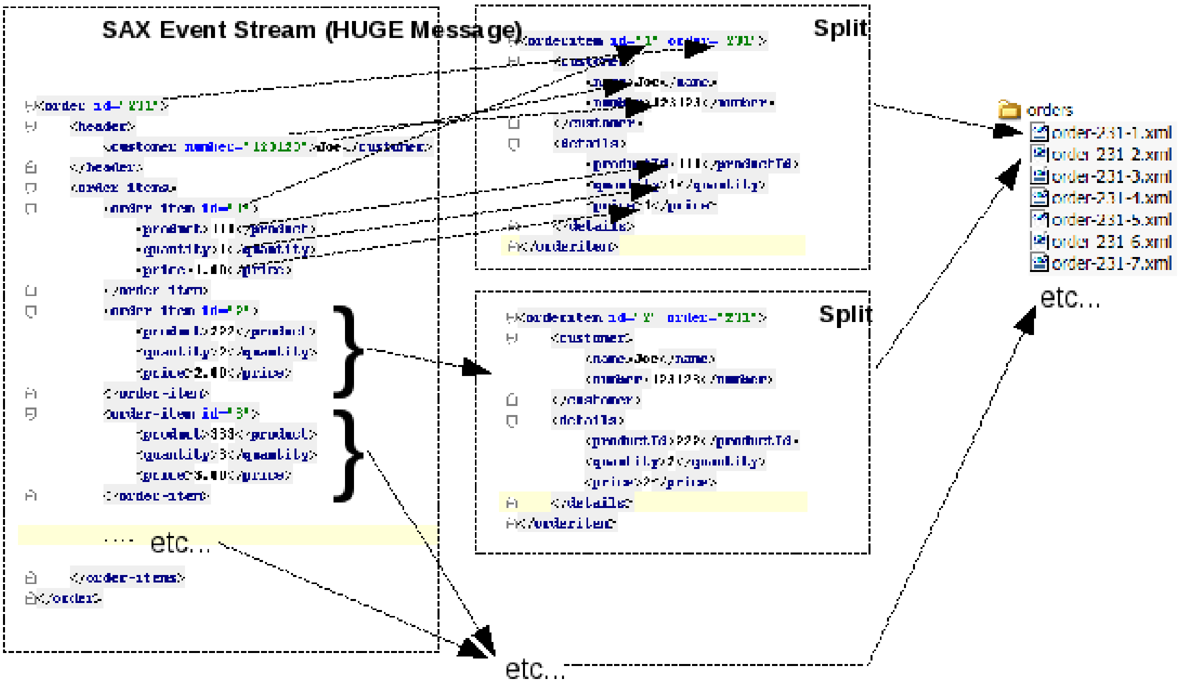


Figure 10.1. Message Splitting

The above illustration shows how we would like to perform the by-order-item splitting operation and route the split messages to file. The split messages contain a full XML document with data merged from the order header and the order item in question i.e. not just a dumb split. In this illustration, we simply route all the message fragments to file, but with the Smooks Action, we can also route the fragment messages to JMS and to a Database and in different formats (EDI, populated Java Objects, etc).

The Smooks configuration for the above example would look as follows. Resource configurations #1 and #2 are there to bind data from the source message into Java Object in the Smooks bean context. In this case, we're just binding the data into HahMaps. The Map being populated in configuration #2 is recreated and repopulated for every order item as the message is being filtered. The populated Java Objects (from resources #1 and #2) are use to populate a FreeMarker template (resource #4), which gets applied on every order item, with the result of the templating operation being output to a FileResourceStream (resource #3). The FileResourceStream (resource #3) also gets applied on every order item, managing the file output for the split messages.

This functionality is available in JBoss ESB 4.3 GA as a Technical Preview and we would greatly appreciate your feedback. What the above 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 endpoints. We will be adding a quickstart dedicated to demoing these features of the ESB. Check the User Forum for details.

Message Transformation

The JBoss ESB supports message data transformation through several of mechanisms.

11.1. Smooks

Smooks is, among other things, a Fragment based Data Transformation and Analysis tool (XML, EID, CSV, Java etc). It supports a wide range of data processing and manipulation features.

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

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

Samples and Tutorials

A number of Transformation Quickstart samples are included in the JBoss SOA Platform distribution. These can be found in the **`samples/quickstarts`** directory. The directory name of each transformation quickstart begins with **`transform_`**.

The JBoss SOA Platform Programmers Guide also contains more detailed material on this topic as well as references to additional information on the Smooks website.

Note

Some of the Quickstarts use the older *SmooksTransformer* action class instead of the newer *SmooksAction*. The *SmooksTransformer* will be deprecated in a future release.

11.2. XSL Transformations

JBoss ESB supports message transformation through the standard XSLT usage model, as well as through the Smooks. Native XSLT may be added in future releases. Support for XSLT can be provided by creating a custom **`org.jboss.soa.esb.actions.ActionProcessor`** implementation.

11.3. ActionProcessor Data Transformation

Where Smooks can not handle a specific transformation usecase, you can implement a custom transformation solution through implementation of the **`org.jboss.soa.esb.actions.ActionProcessor`** interface.

jBPM Integration

JBoss jBPM is a powerful workflow and Business Process Management (BPM) engine. It enables the creation of business processes that coordinate between people, applications and services. JBoss jBPM uses a modular architecture combining easy development of workflow applications with a flexible and scalable process engine. The JBoss jBPM process designer visually represents the business process steps to facilitate a strong link between the business analyst and the technical developer. This document assumes that you are familiar with jBPM. If you are not you should read the included jBPM Reference Guide first.

JBossESB integrates the jBPM so that it can be used for two purposes.

1. Service Orchestration

   ESB services can be orchestrated using jBPM. You can create a jBPM process definition which makes calls into ESB services.

2. Human Task Management

   jBPM allows you to incorporate human task management integrated with machine based services.

12.1. Integration Configuration

The jbpm.esb deployment that ships with the ESB includes the full jBPM runtime and the jBPM console. The runtime and the console share a common jBPM database. The ESB DatabaseInitializer mbean creates this database on startup. The configuration for this mbean is found in the file **`jbpm.esb/jbpm-service.xml`**.

```
<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

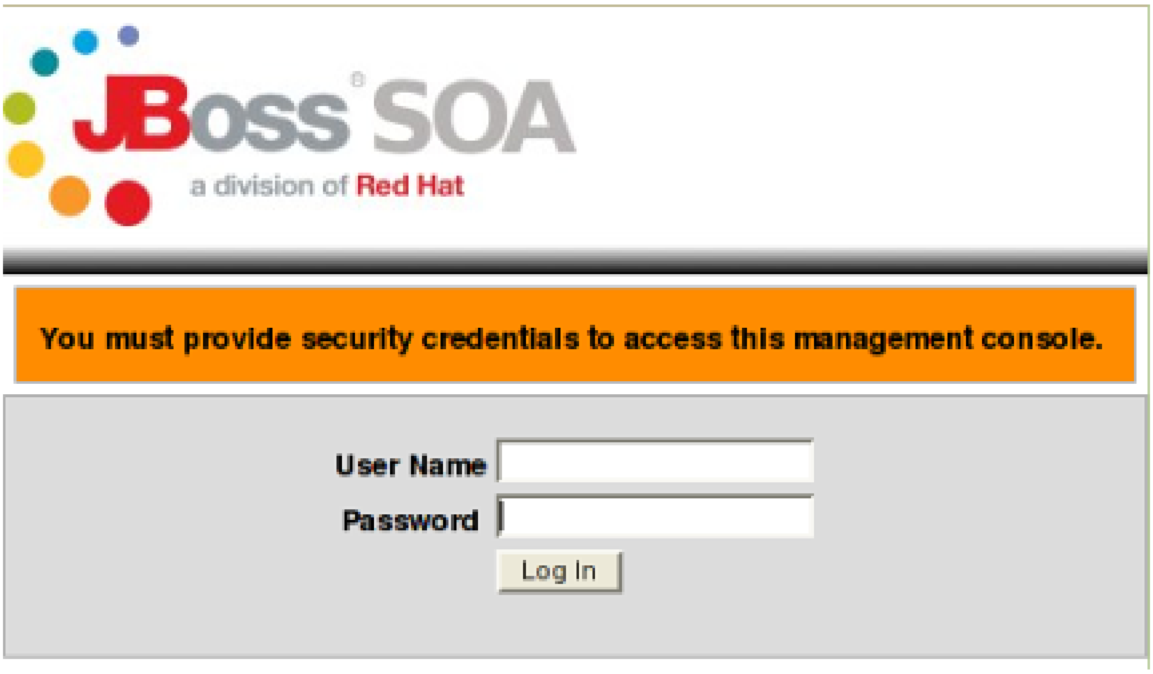
The first Mbean configuration element contains the configuration for the DatabaseInitializer.

Property	description	Default
`Datasource`	The datasource for the jBPM database	java:/JbpmDS
`ExistsSql`	The SQL command that is used to confirm the existance of the database.	Select * from JBPM_ID_USER
`SqlFiles`	The files containing the SQL commands to create the jBPM database if it is not found.	jbpm-sql/jbpm.jpdl.hsqldb.sql, jbpm-sql/import.sql

Table 12.1. ESB DatabaseInitializer mbean default values

The DatabaseInitializer mbean is configured in jbpm-service.xml to wait for the JbpmDS to be deployed, before deploying itself. The second mbean “JbpmService” ties the lifecycle of the jBPM job executor to the jbpm.esb lifecycle, it starts a job executor instance on startup and stops it on shutdown. The JbpmDS datasource is defined in the jbpm-ds.xml and by default it uses a HSQL database. In production you will want change to a production strength database. All jbpm.esb deployments should share the same database instance so that the various ESB nodes have access to the same processes definitions and instances.

The jBPM console is a web application accessible at *http://localhost:8080/jbpm-console* when you start the server.



You must provide security credentials to access this management console.

User Name

Password

Figure 12.2. jBPM Console Login

Please check the jBPM Reference Guide to change the security settings for this application. This will involve changing some settings in the **`conf/login-config.xml`**. The console can be used for deploying and monitoring jBPM processes, but can also be used for human task management. For the different users a customized task list will be shown and they can work on these tasks. The quickstart **`bpm_orchestration4`** demonstrates this feature. The **`jbpm.esb/META-INF`** directory contains the **`deployment.xml`** and the **`jboss-esb.xml`**.

The **`deployment.xml`** specifies the resources this esb archive depends on which are the **`jbossesb.esb`** and the JbpmDS datasource. 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 12.3. deployment.xml dependancy declarations

The jboss-esb.xml deploys one internal service called “JBpmCallbackService”

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

Figure 12.4. JBpmCallbackService

This service listens to the jBPMCallbackBus, which by default is a JMS Queue on either a JBossMQ (**`jbmq-queue-service.xml`**) or a JbossMessaging (**`jbm-queue-service.xml`**) messaging provider. Make sure only one of these files gets deployed in your **`jbpm.esb`** archive. If you want to use your own provider simple modify the provider section in the **`jboss-esb.xml`** to reference your JMS provider.

```
<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 12.5. Modifying the provider section in the **`jboss-esb.xml`** for your own JMS

Section 12.5, “jBPM to JBossESB” contains more details about JbpmCallbackService.

12.2. jBPM Configuration

The configuration of jBPM itself is managed by three files: **`jbpm.cfg.xml`** , **`hibernate.cfg.xml`** and **`jbpm.mail.templates.xml`**.

`jbpm.cfg.xml` is set 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 12.6. Default values in `jbpm.cfg.xml`

Other settings are left to the default jBPM settings.

The `hibernate.cfg.xml` file is also slightly 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 12.7. Default values in `hibernate.cfg.xml`

Hibernate is not used to create the database schema. The DatabaseInitiazer mbean refered to in *Section 12.1, “Integration Configuration”* is used for this.

The `jbpm.mail.templates.xml` is left empty by default. For more details on each of these configuration files please see the jBPM documentation.

Important

The configuration files that usually ship with `jbpm-console.war` have been removed. This is done to centralized all the configuration in the files in the root of the `jbpm.esb` archive.

12.3. Creation and Deployment of a Process Definition

To create a Process Definition we recommend using JBoss Developer Studio.

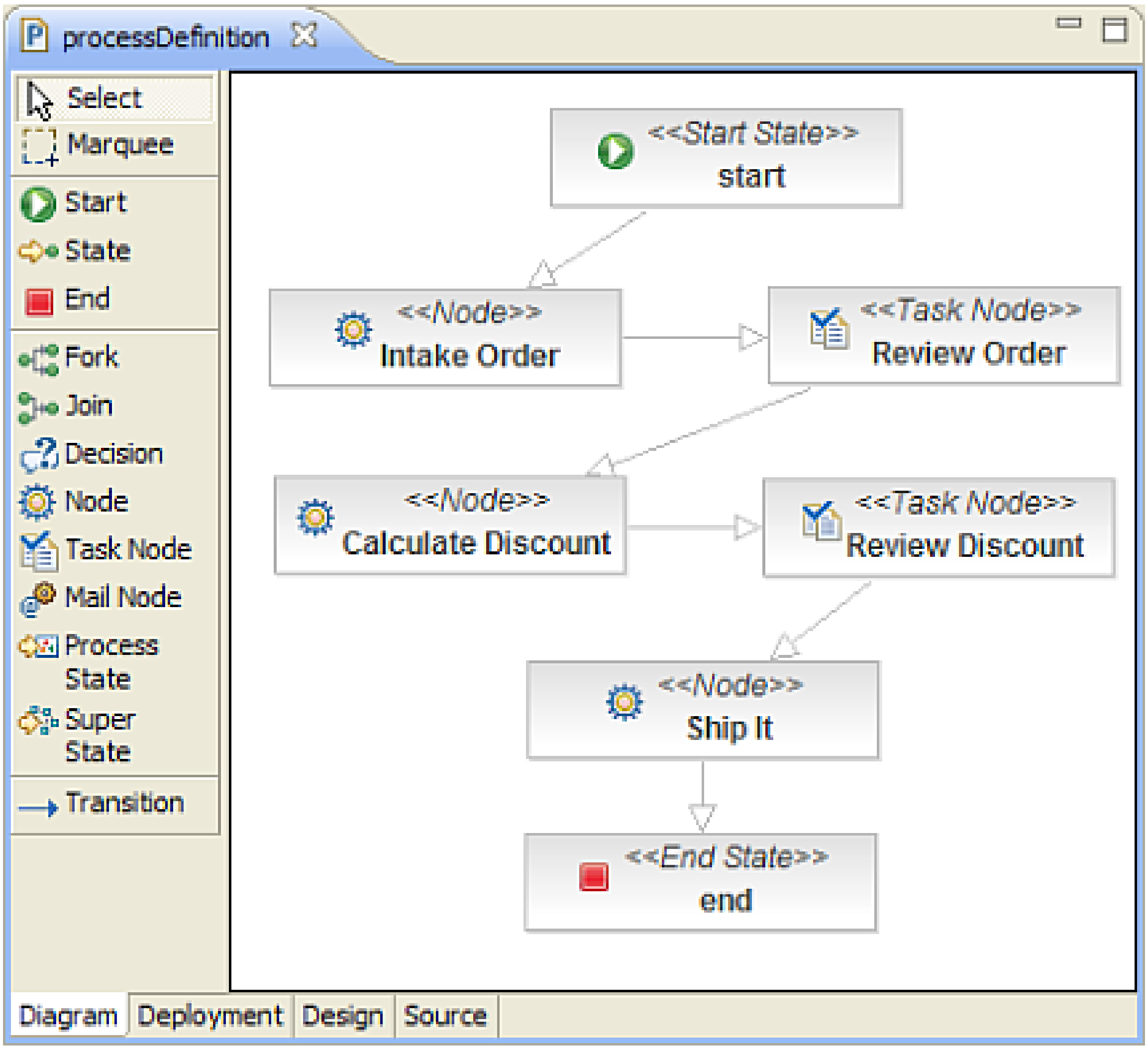
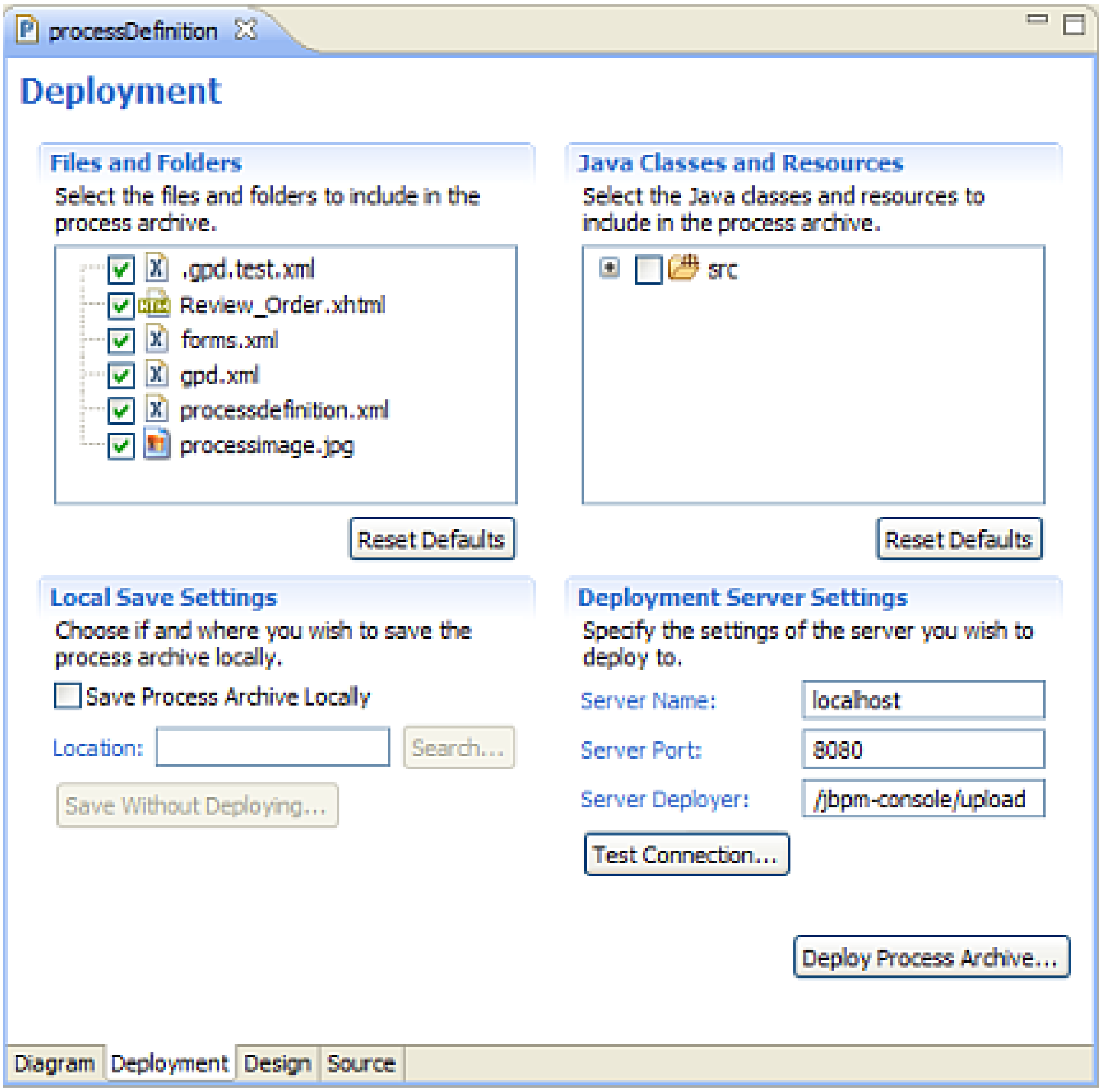


Figure 12.8. JBoss Developer Studio - jBPM Graphical Editor

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

To deploy a process definition the server needs to be up and running. Only then can you go to the 'Deployment' tab in the graphical designer to deploy a process archive (par).



Files and Folders

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

- ☑ .gpd.test.xml
- ☑ Review_Order.xhtml
- ☑ forms.xml
- ☑ gpd.xml
- ☑ processdefinition.xml
- ☑ processimage.jpg

Java Classes and Resources

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

- ☐ src

Local Save Settings

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

☐ Save Process Archive Locally

Location:

Deployment Server Settings

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

Server Name: localhost

Server Port: 8080

Server Deployer: /jbpm-console/upload

Figure 12.9. JBoss Developer Studio - jBPM Deployment View

In some cases it would suffice to deploy just the `processdefinition.xml`, but in most cases you will be deploying other type of artifacts as well, such as task forms.

It is also possible to deploy Java classes in a process archive, which means that they end up in the database where they will be stored and versioned. This is not recommended to do this in the ESB environment as it can lead to class loading issues. The recommended practice is to deploy your classes in the lib directory of the server.

There are three mechanisms that you can use to deploy a process defination.

1. You can deploy directly from JBoss Developer Studio, by clicking on the **Deploy Process Archive** button in the Deployment view
2. You can saving the deployment to a local `.par` file from the Deployment view and then use the jBPM console to deploy the archive. You need to be able to login to the console with administrative privileges to do this.
3. You can also deploy by using the DeployProcessToServer jBPM ant task.

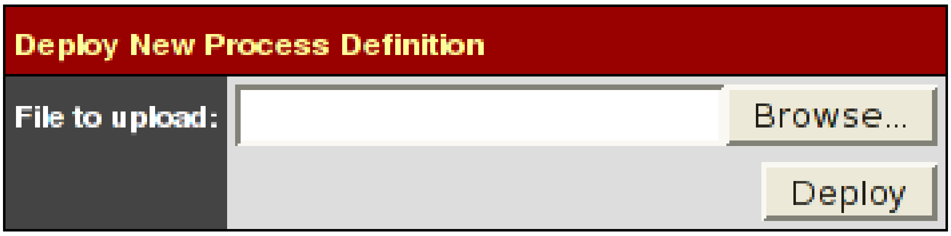


Figure 12.10. jBPM Console - uploading a new Process Definition

12.4. JBossESB to jBPM

The JBoss ESB can make calls into jBPM using the BpmProcessor action. This action uses the jBPM command API to make calls into jBPM. The following jBPM commands have been implemented.

NewProcessInstanceCommand

This command starts a new ProcessInstance with a process definition that was already deployed to jBPM. The NewProcessInstanceCommand leaves the Process Instance in the start state, which is needed if there is an task associated to the Start node such as a task on an actor's tasklist.

StartProcessInstanceCommand

This command is identical to the NewProcessInstance-Command except that the new Process Instance is automatically moved from the Start position into the first Node.

CancelProcessInstanceCommand

This command cancels a ProcessInstance. You can use this in situation such as when event comes in which should result in the cancellation of the entire ProcessInstance. This action requires some jBPM context variables to be set on the message, in particular the ProcessInstance Id.

```
<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 12.11. BpmProcessor action configuration in `jboss-esb.xml`

There are two required action attributes.

1. name

   You are free to use any value for the name attribute as long as it is unique in the action pipeline.

2. class

   This attributes must be set 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 or CancelProcessInstanceCommand.

2. processdefinition

   This is a required property 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 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.

4. actor

   This is a optional property to specify the jBPM actor id. This 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 you have a named parameter called *businessKey* in the body of your message you would use *body.businessKey*. This property is used for NewProcessInstanceCommand and StartProcessInstanceCommand only.

6. transition-name

   This is a optional property. This property 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 EsbMessage and set into jBPM context for the particular process instance. The list consists of mapping elements. Each mapping element can have the following attributes:

- esb

  This is a required attribute which can contain an MVEL expression to extract a value anywhere from the EsbMessage.

- bpm

  This is a optional attribute containing the name which be used on the jBPM side. If omitted the esb name is used.

- default

  This is a optional attribute which can hold a default value if the esb MVEL expression does not find a value set in the EsbMessage.

- reply-to-originator

  This is a optional property for the New- and StartProcessInstanceCommands. If this property is specified, with a value of true, then the creation of the process instance will store the ReplyTo/FaultTo EPRs of the invoking message 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 parameter for CancelProcessInstanceCommand only. It is up to the user make sure this value is set as a named parameter on the EsbMessage body.

12.4.1. ESB to jBPM Exception Handling

For ESB calls into jBPM an exception of the type JbpmException can be thrown from the jBPM Command API. This exception is not handled by the integration and we let it propagate into the ESB Action Pipeline code. The action pipeline will log the error, send the message to the DeadLetterService (DLS), and send the an error message to the faultTo EPR, if a faultTo EPR is set on the message.

12.5. jBPM to JBossESB

jBPM to JBossESB communication provides us with the capability to use jBPM for service orchestration. Service Orchestration itself will be discussed in more detail in the next chapter but we will focus on the details of the integration here first.

The integration implements two jBPM action handler classes, **`EsbActionHandler`** and **`EsbNotifier`**. The **`EsbActionHandler`** is a request-reply type action, which sends a message to a Service and then waits for a response. **`EsbNotifier`** does not wait for a response. The interaction with JBossESB is asynchronous in nature and does not block the process instance while the Service executes.

First we'll discuss **`EsbNotifier`** as it implements a subset of the configuration of **`EsbActionHandler`**.

12.5.1. ESBNotifier

The **`EsbNotifier`** action should be attached to an outgoing transition so the jBPM processing can continue while the request to the ESB service is processed in the background. In the jBPM **`processdefinition.xml`** we need attach the **`EsbNotifier`** 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>
```

Figure 12.12. *Ship It* node with **`EsbNotifier`** attached

The following attributes can be specified.

- name

  This is required attribute. User specified name of the action

- class

  This is a required attribute. Required to be set to **`org.jboss.soa.esb.services.jbpm.actionhandlers.EsbNotifier`**.

The following subelements 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*.

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

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

    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 in jBPM context the value of this field is taken instead.

  - process-scope

    This attribute is optional. This boolean valued parameter can override the setting of the setting of the globalProcessScope for this mapping.

Note

When working on variable mapping configuration it is recommended to turn on debug level logging.

12.5.2. ESBActionHandler

The EsbActionHandler is designed to work as a reply-response type call into JBossESB. The EsbActionHandler should be attached to the node. When this node is entered this action will be called. The EsbActionHandler executes and leaves the node waiting for a transition signal. The signal can come from any other thread of execution, but under normal processing the signal will be send by the JBossESB callback Service.

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

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


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

</action>
```

Example 12.2. Configuration for the EsbActionHandler

The configuration for the EsbActionHandler action extends the EsbNotifier configuration. The extensions are the following subelements.

- esbToBpmVars

  This element is optional. This subelement is identical to the esbToBpmVars property mentioned in *Section 12.4, "JBossESB to jBPM"* for the BpmProcessor configuration. The element defines a list of variables that need to be extracted from the EsbMessage and set into jBPM context for the particular process instance. If unspecified the globalProcessScope value defaults to true when setting variables. The list consists of mapping elements. Each mapping element can have the following attributes.

  - esb

    A required attribute which can contain an MVEL expression to extract a value anywhere from the EsbMessage.

  - bpm

    An optional attribute containing the name which be used on the jBPM side. If this is not supplied then the esb name is used.

  - default

    An optional attribute which can hold a default value if the esb MVEL expression does not find a value set in the EsbMessage.

  - process-scope

    An optional attribute. This boolean valued parameter can override the setting of the setting of the globalProcessScope for this mapping.

- exceptionTransition

An optional element. The name of the transition that should be taken if an exception occurs while processing the Service. This requires the current node to have more then one outgoing transition where one of the transition handles *exception processing*.

Optionally you may want to specify a timeout value for this action. For this you can use a jBPM native Timer on the node. *Example 12.3, “Specifying a timeout value for an action”* demonstrates adding a timeout so if no signal is received within 10 seconds of entering this node the transition called **`time-out`** is taken.

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

Example 12.3. Specifying a timeout value for an action

12.5.3. jBPM to ESB Exception Handling

There are two types of scenarios where exceptions can arise.

1. MessageDeliveryException is thrown by the ServiceInvoker when delivery of the message to the ESB failed. If this happens things are pretty bad and you have probably misspelled the name of the Service you are trying to reach. This type of exception can be thrown from both the EsbNotifier as well as the EsbActionHandler. In the jBPM node you can add an ExceptionHandler to handle this exception.

2. The second type of exception is when the Service received the request, but something goes wrong during processing. Only if the call was made from the EsbActionHandler it makes sense to report back the exception to jBPM. If the call was made from the EsbNotifier jBPM processing has already moved on, and it is of little value to notify the process instance of the exception. This is why the exception-transition can only be specified for EsbAction-Handler.

To illustrate the type of error handling that is now possible using standard jBPM features we will discuss some scenarios illustrated in *Figure 12.13, “Three exception handling scenarios: time-out, exception-transition and exception-decision.”*.

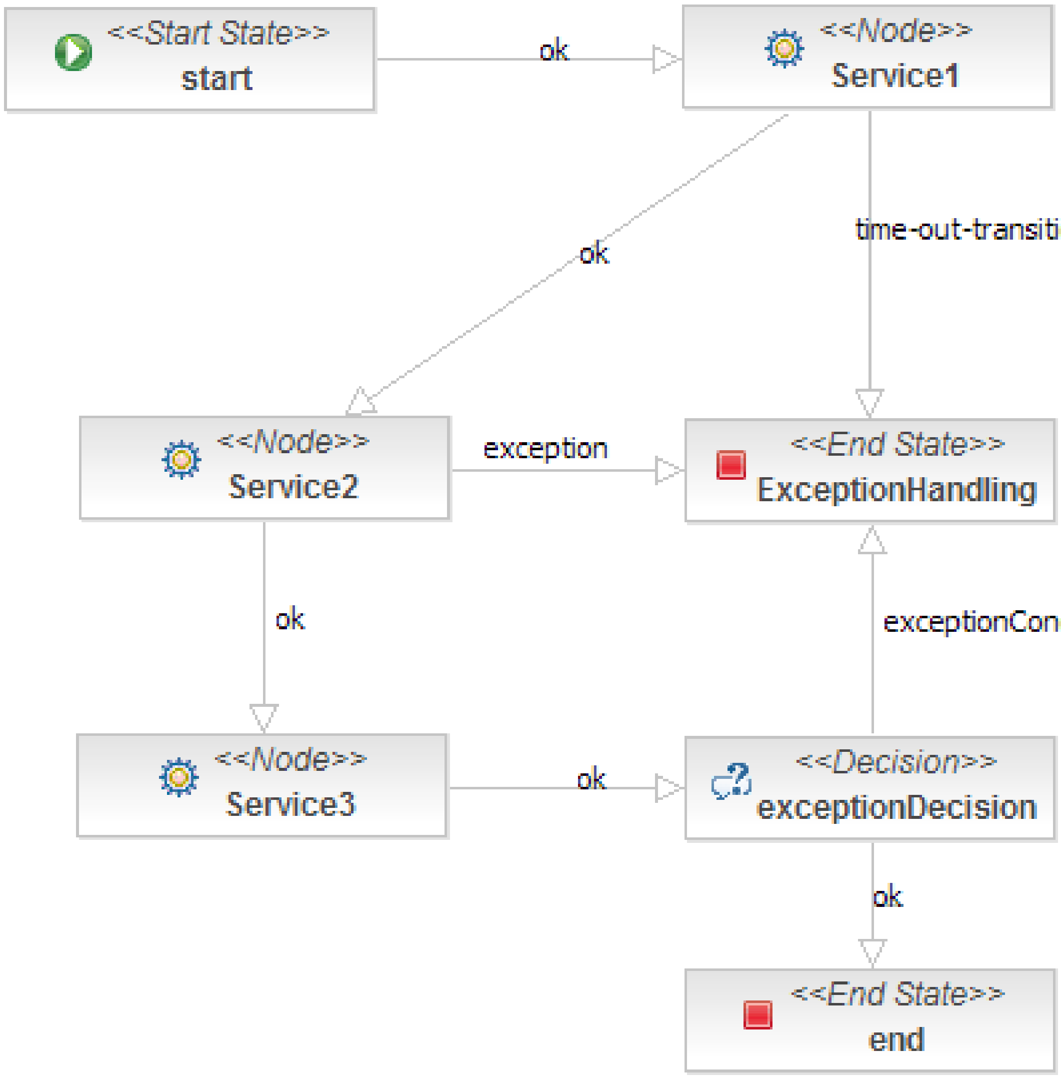


Figure 12.13. Three exception handling scenarios: time-out, exception-transition and exception-decision.

12.5.4. Scenerio 1: Time-out

When using the EsbActionHandler action and the node is waiting for a callback, it maybe that you want to limit how long you want to wait for. For this scenario you can add a timer to the node. This is how Service1 is setup in Figure 5. The timer can be set to a certain due date. In this case it is set to 10 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>
```

Node *Service1* has 2 outgoing transitions. The first one is called **`ok`** while the second one is called **`time-out-transition`**. Under normal processing the call back would signal the default transition, which is the **`ok`** transition since it is defined first. However if the execution of the service takes more then 10 seconds the timer will fire. The transition attribute of the timer is set to **`time-out-transition`**, so this transition will be taken on time-out. In Figure 5 this means that the processing ends up in the **`ExceptionHandling`** node in which one can perform compensating work.

12.5.5. Scenerio 2: Exception Transition

To handle exceptions that may occur during processing of the Service, one can define an exceptionTransition. When doing so the faultTo EPR is set on the message so the ESB will make a callback to this node, signaling it with the exceptionTransition. Service2 has two outgoing transitions. Transition **`ok`** will be taken under normal processing, while the **`exception`** transition will be taken when the Service processing throws an exception.

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

Example 12.4. Definition of Service 2

In this definition of Service2, the exceptionTransition in the action is set to “exception”. In this scenario the process also ends in the **`ExceptionHandling`** node.

12.5.6. Scenerio 3: Exception Decision

Scenario 3 is illustrated in the configuration of Service3 and the **`exceptionDecision`** node that follows it. The idea is that processing of Service3 completes normally and the default transition out of node Service3 is taken. However, somewhere during the Service execution an errorCode was set, and the **`exceptionDecision`** node checks if a variable called **`errorCode`** was set.

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

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

Example 12.5. Definition of Service 3

where the esbToBpmVars mapping element extracts the errorCode called **`Some-ExceptionCode`** from the EsbMessage body and sets in the jBPM context, if this **`SomeExceptionCode`** is set that is. In the next node **`exceptionDecision`** the “ok” transition is taken under normal processing, but if a variable called **`errorCode`** is found in the jBPM context, the **`exceptionCondition`** transition is taken. This is using the decision node feature of jBPM where transition can nest a condition.

For more details on conditional transitions please refer to the included jBPM Reference Guide.

Chapter 13.

Service Orchestration

Service Orchestration is the arrangement of business processes. Traditionally BPEL is used to execute SOAP based WebServices. If you want to orchestrate JBossESB regardless of their end point type, then it makes more sense to use jBPM. This chapter explains how to use the integration discussed earlier to do Service Orchestration using jBPM.

13.1. 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, see the Message Action Guide.

JBoss and JBossESB also have a special support agreement with *ActiveEndpoints* [1] for their award wining *ActiveBPEL* [2] WS-BPEL Engine. JBoss ESB includes with the **`webservice_bpel`** QuickStart which demonstrates how the JBoss ESB and *ActiveBPEL* [3] 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* [4] provides the Process Orchestration. A number of flash based walk-thrus of this Quickstart are also *available online* [5].

> **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.

13.2. 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.

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.

---

[1] http://www.active-endpoints.com/
[2] http://www.active-endpoints.com/
[3] http://www.active-endpoints.com/
[4] http://www.active-endpoints.com/
[5] http://labs.jboss.com/jbossesb/resources/tutorials/bpel-demos/bpel-demos.html

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.

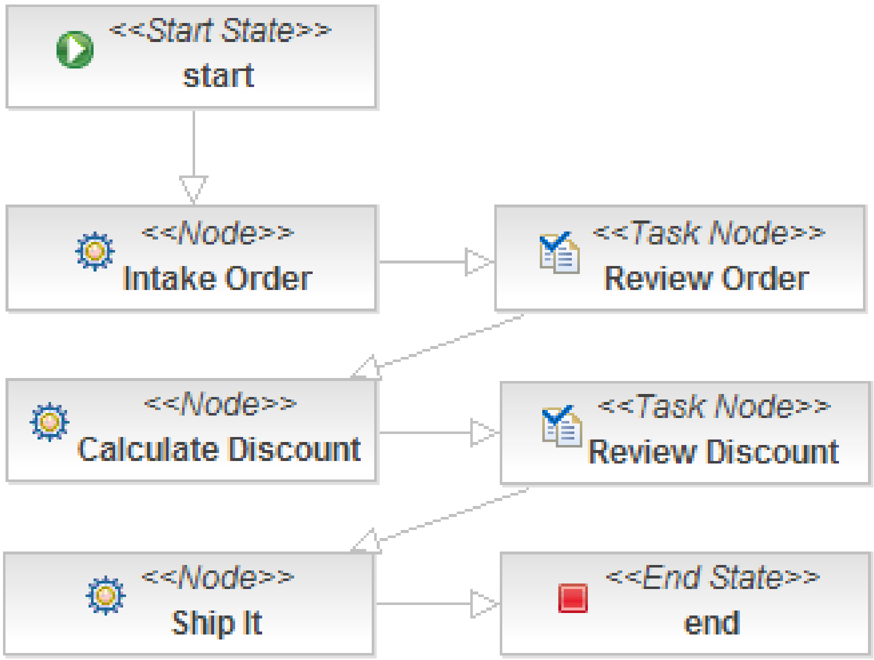


Figure 13.1. Orchestration diagram for the **`bpm_orchestration4`** QuickStart

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.

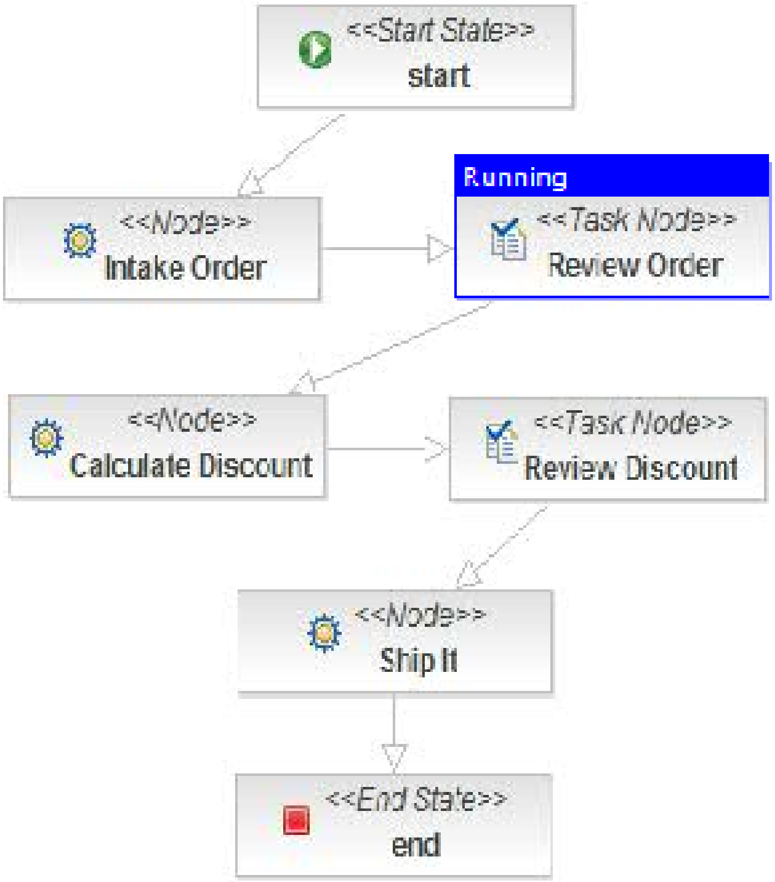


Figure 13.2. The Order process reached the “Review Order” node

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>
```

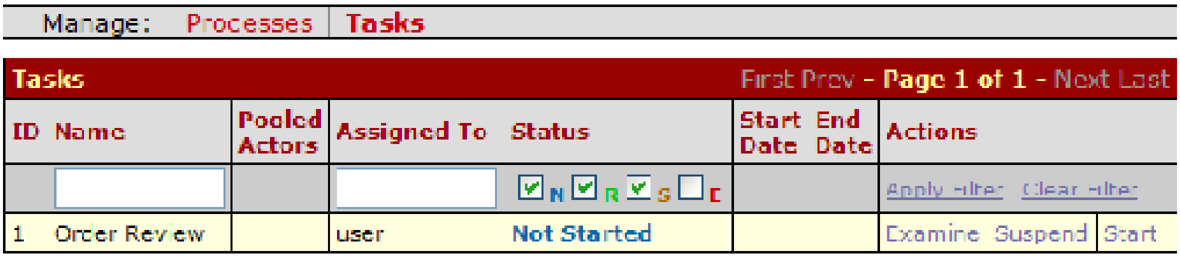
In order to display these variables in a form in the jbpm-console we need to create an xhtml dataform (see the Review_Order.xhtml file in the bpm_orchestration4 quick start [JBESB-QS] and tie this for this TaskNode using the forms.xml file:

```
<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:

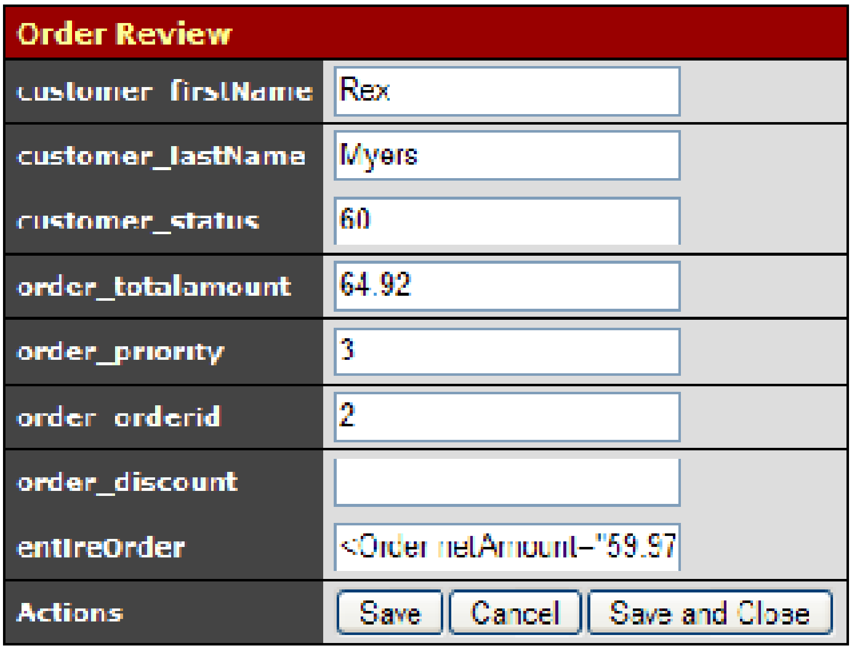
```
<jbpm:datacell>
<f:facet name="header">
<h:outputText value="customer_firstName"/>
</f:facet>
<h:inputText value="#{var['customer_firstName']}" />
</jbpm:datacell>
```

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.



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

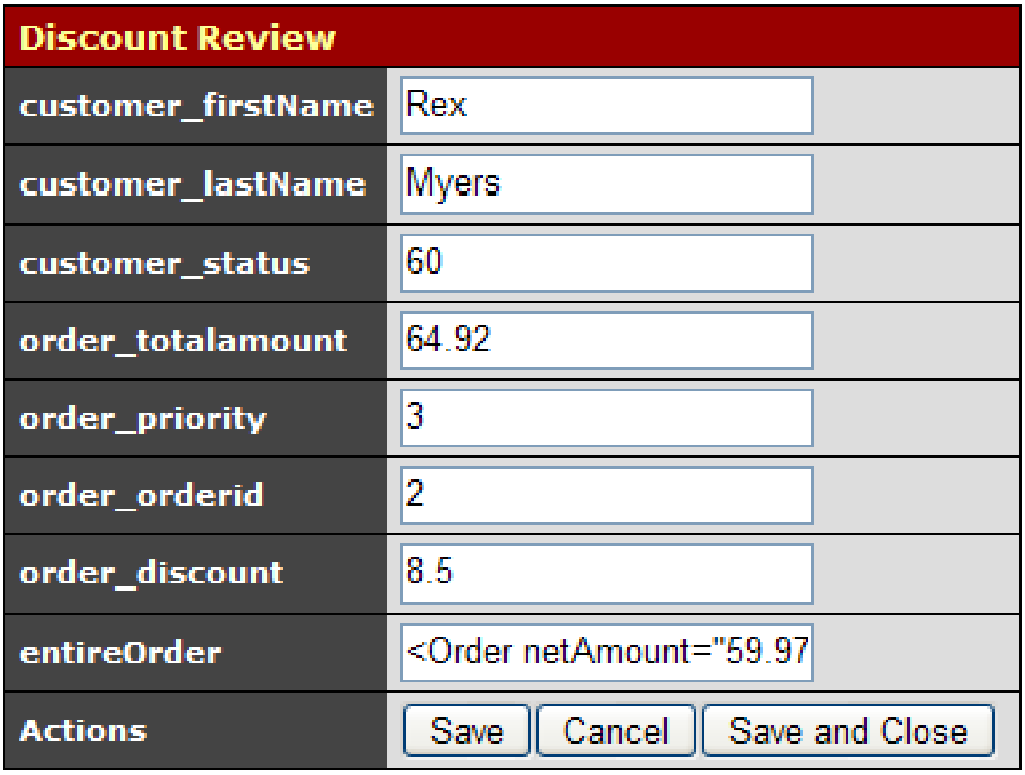
customer_firstName	Rex
customer_lastName	Myers
customer_status	60
order_totalamount	64.92
order_priority	3
order_orderid	2
order_discount	
entireOrder	<Order netAmount="59.97

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

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'.

13.3. 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 1). 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="Superman Returns"
price="29.98"/>
</OrderLine>
<OrderLine position="2" quantity="1">
<Product productId="299" title="Pulp Fiction" 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.

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

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

Important

In JBossESB 4.2 the Message Store is also used for storing messages that need to be redelivered in the event of failures. You can find additional details on this topic in the SOA 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 and each MessageStore implementation uses a URI to accomplish this identification. This URI is used as the “key” for that message 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

MessageStore implementations may use different formats for their URIs.

Messages can be stored within the store based upon classification using addMessage. If the classification is not defined then it is up to the implementation of the MessageStore how it will store the Message. Furthermore, the classification is only a hint: implementations are free to ignore this field if necessary.

It is implementation dependent as to whether or not the MessageStore imposes any kind of concurrency control on individual Messages. As such, you should use the removeMessage operation with care.

Because the current MessageStore interface is designed to support both audit trail and redelivery scenarios, you should not use the setUndelivered/setDelivered and associated operations unless they are applicable.

The default implementation of the MessageStore is provided by the **`org.jboss.internal.soa.esb.persistence.format.db.DBMessageStoreImpl`** class. The methods in this implementation make the required DB connections using a pooled Database Manager, **`DBConnectionManager`**.

To override the MessageStore implementation you should look at the MessageActionGuide and the MessagePersister Action.

The Message Store interface does not currently support transactions. Any use of the store within the scope of a global transaction will not be coordinated within the scope of any global transaction. This means that each message store update or read will be done as a separate, independent, transaction. Future versions of the Message Store will provide for control over whether or not specific interactions should be conducted within the scope of any enclosing transactional context.

14.2. Configuring the Message Store

To configure your Message Store you have to override the default service implementation 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.

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

Example 14.1. Sample SQL for message store table creation 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 Standalone manager uses C3PO to manage the connection pooling logic, and the J2eeConnectionManager uses a datasource to manage it's connection pool. This is intended for use when deploying your ESB endpoints inside a container such as Jboss AS or Tomcat. You can plug in your own connection pool manager by implementing the interface **`org.jboss.internal.soa.esb.persistence.manager.ConnectionManager`**.

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 by using a gateway or by using the ServiceInvoker to directly invoke the ESB service. When calling a service via a gateway, the gateway is responsible for extracting the security information needed to authenticate the caller. It does this by extracting the information from the transport that the gateway handles. Using this information the gateway creates an authentication request that is encrypted and then passed to the ESB.

When using the ServiceInvoker a gateway is not involved and it is the responsibility of the client to create the authentication request prior to invoking the service. Both of these situations will be looked at in the following sections.

The default security implementation is JAAS based but this is a configurable feature. The following sections describe the security components and how they can be configured.

15.1. Security Service Configuration

The Security Service is configured along with everything else in **`jbossesb-properties.xml`**.

```
<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. Default is JaasSecurityService.

org.jboss.soa.esb.services.security.callbackHandler
: Optional. A default CallbackHandler implementation when a JAAS based SecurityService is being used. See “Customizing security” for more information about the callbackHandler property.

org.jboss.soa.esb.services.security.sealAlgorithm
: The algorithm to use for sealing the SecurityContext.

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

org.jboss.soa.esb.services.security.contextTimeout
: The amount of time in milliseconds that a security context is valid for. This is a global setting that may be overridden on a per service basis by specifying this same property name on the security element in jboss-esb.xml.

org.jboss.soa.esb.services.security.contextPropagatorImplementationClass
: Optional property that configures a global SecurityContextPropagator. For more details on the SecurityContextPropagator please refer to the “Security Context Propagation”.

org.jboss.soa.esb.services.security.publicKeystore
: Path to the keystore that holds a keys used for encrypting and decrypting data external to the ESB. This is used to encrypt the AuthenticationRequest .

org.jboss.soa.esb.services.security.publicKeystorePassword
: Password to the public keystore.

org.jboss.soa.esb.services.security.publicKeyAlias
: Alias to use.

org.jboss.soa.esb.services.security.publicKeyPassword
: Password for the alias if one was specified upon creation.

org.jboss.soa.esb.services.security.publicKeyPassword
: Optional cipher transformation in the format: “algorithm/mode/padding”. If not specified this will default to the keys algorithm.

The JAAS login modules are configured in the way you would except using the login-config.xml file located in the conf directory of your JBoss Application Server. So you can use the ones that come pre-configured but also add you own login modules.

By default JBossESB ships with example keystore which should not be used in production. They are only provided as sample to help users get security working “out of the box”. The sample keystore can be updated with custom generate key pairs.

15.1.1. Configuring Security on Services

Security is configured per-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
: A named module that exist in conf/login-config.xml

runAs
: An optional runAs role.

rolesAllowed
: An optional comma separated list of roles that are allowed to execute the service. This is a check that is performed after a caller has been authenticated, to verfiy that the caller in a member 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.

```
<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>
```

Example 15.1. Overriding global configuration settings

15.2. Authentication

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

When a service needs to call another services. and that service requires authentication, another authentication process will be performed. So having a chain of services that are all configured for authentication will cause multiple authentications to be performed. To minimize such overhead the ESB will store an encrypted SecurityContext which will be propagated with the ESB Message object between services. If the ESB detects that a Message has a SecurityContext check that the SecurityContext is still valid and if so re-authentication is not performed. Note that the SecurityContext is only valid of a single ESB node. If the message is routed to a different ESB node a re-authentication will still be requried.

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);
```

Note that 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`** of overridden per-service by specifying the value in jboss-esb.xml. 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

Propagation in this case refers to propagating security context information in a way specific to an external system. For example, you might want to have the same credentials that were used to call the ESB, be used as the credentials when calling a EJB method. This can be accomplished by specifiying a SecurityContextPropagator that will perform the security context propagation specific to the destination environment.

A SecurityContextPropagator can be configured globally by specifying the **`org.jboss.soa.esb.services.security.contextPropagatorImplementationClass`** in **`jbossesb-properties.xml`**, or per-service by specifying the same property in **`jboss-esb.xml`**. *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.securityClass:`**

`JBossASContextPropagator`

This propagator will propagate security credentials to a JBoss Application Server. If you need to write your own implementation you only have to write a class that implements **`org.jboss.internal.soa.esb.services.security.SecurityContextPropagator`** and then either specify that implementation in **`jbossesb-properties.xml`** or **`jboss-esb.xml`** as 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.

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

Example 15.2. CertificateLoginModule configuration

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.

Example of fragment from **`login-config.xml`**

```
<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
: Path to the keystore that will be used to verify the certificates. This can be a file on the local file system or on the classpath.

keyStorePassword
: Password for the above keystore.

rolesPropertiesFile
: Optional. 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 JBossESB security. This interface 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&lt;String&gt; 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 this can be customized by implementing the above interface and configuring the custom SecurityService be used in **`jbossesb-properties.xml`**. For more details of the SecurityService interface's method please refer to the javadocs.

Appendix A. Revision History

Revision History		
Revision 1.0	Tue 9 Sep 2008	DarrinMison*`dmison@redhat.com`*
Created		