JBoss ESB 4.7

Services Guide

JBESB-SG-

Legal Notices

JBESB-SG-

Software Version

JBoss ESB 4.7

Contents

About This Guide

What This Guide Contains

The Services Guide contains important information on changes to JBoss ESB 4.7 since the last release and information on any outstanding issues.

Audience

This guide is most relevant to engineers who are responsible for administering JBoss ESB 4.7 installations.

Prerequisites

None.

Documentation Conventions

The following conventions are used in this guide:

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBoss ESB 4.7 documentation set:

1. **JBoss ESB 4.7** *Getting Started Guide*: Provides a quick start reference to configuring and using the ESB.
2. **JBoss ESB 4.7** *Programmers Guide*: How to use JBossESB.
3. **JBoss ESB 4.7** *Release Notes*: Information on the differences between this release and previous releases.
4. **JBoss ESB 4.7** *Administration Guide*: How to manage the ESB.

Contacting Us

Questions or comments about JBoss ESB 4.7 should be directed to our support team.

What is the Registry?

Introduction

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

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 as the number of services, the frequency of interactions, the physical distributed nature of the environment, increases. 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 concrete 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 (e.g., 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.

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

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.

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 maintaines those items.

SOA components

As the W3C puts it: *An SOA is a specific type of distributed system in which the agents are "services" (http://www.w3.org/TR/2003/WD-ws-arch-20030808/#id2617708).*

The key components of a Service Oriented Architecture are the messages that are exchanged, agents that act as service requesters and service providers, and 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 message exchange patter between itself and its users. Within an SOA there are thus three critical roles: requester, provider, and broker.

- *Service provider*: allows access to services, creates a description of a service and publishes it to the service broker.
- *Service broker*: hosts a registry of service descriptions. It is responsible for linking a requestor to a service provider.
- *Service 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.

ServiceBrokerServiceRequestorServiceProvider

UDDI

The Universal Description, Discovery and Integration 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 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.

The Registry and JBossESB

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

As with all environments within which registries reside, it is not possible for the registry to determine the liveness of the entities its data represents, e.g., 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 services that is no longer active. At present JBossESB does not perform life-cycle monitoring of the services that are deployed within it. As such, 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 removing any out-of-date items.

In JBoss ESB versions 4.6 and before, the ESB used a jUDDI release that implemented the UDDI v2 specification as the backing registry. In JBoss ESB 4.7, the ESB uses a jUDDI release that supports the v3 specification as the backing registry and a Scout release that provides support for UDDI v3..

Configuring the Registry

Introduction

The JBossESB Registry architecture allows for many ways to configure the ESB to use 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 JBossESB 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.v3.client.transport.wrapper.UDDIInquiryService#inquire"/>

            <property name="org.jboss.soa.esb.registry.lifeCycleManagerURI"


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


value="org.apache.juddi.v3.client.transport.wrapper.UDDISecurityService#secure"/>

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

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

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

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

Property	Description

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.securityManagerURI	The URI used by JAXR to authenticate.
org.jboss.soa.esb.registry.user	The user used for edits.
org.jboss.soa.esb.registry.password	The password to go along with the user.
org.jboss.soa.esb.scout.proxy.uddiVersion	The UDDI Version of the query.
org.jboss.soa.esb.scout.proxy.uddiNameSpace	The UDDI namespace.
org.jboss.soa.esb.scout.proxy.transportClass	The name of the class used by scout to do the transport from scout to the UDDI.
org.jboss.soa.esb.registry.interceptors	The list of interceptors that are applied to the configured registry. The codebase currently provides two interceptors, one for handling InVM registration and a second for applying a cache over the registry. The default interceptor list consists solely the InVM interceptor.
org.jboss.soa.esb.registry.cache.maxSize	The maximum number of server entries allowed in the cache. If this value is exceeded then entries will be evicted on a LRU basis. The default value is 100 entries.
org.jboss.soa.esb.registry.cache.validityPeriod	The validity period of the caching interceptor. This is specified in milliseconds and defaults to 600000 (ten minutes). If this value is zero (or less) then there is no expiry specified on the cache.

org.jboss.soa.esb.registry.orgCategory	The Organization Category name for the ESB instance. Default is "org.jboss.soa.esb.:category".

The components involved

The registry can be configured in many ways. Figure 1 shows a blue print of all the registry components. From the top down we can see that the JBossESB 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.

JAXR(JAXR-Implementation)UDDIebXMLOther XMLRegistryJBossESBOther Java APIRegistry Interface

Figure 1. Blue print of the Registry component architecture.

The Registry Implementation Class

Property: `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 SystinetRegistryImplemtation class and referencing it in this property.

Using JAXR

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

If you decided to use JAXR then you will have to pick which JAXR implementation to use. This property is used to configure that class. By default we use Scout and therefore it is set to the scout factory '`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, which is done by setting the `org.jboss.soa.esb.registry.queryManagerURI,` and `org.jboss.soa.esb.registry.lifeCycleManagerURI and` `org.jboss.soa.esb.registry.securityManagerURI` respectively, along with the username (`org.jboss.soa.esb.registry.user)` and password (`org.jboss.soa.esb.registry.password)` for the UDDI.

Using jUDDI Transports

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

When using Scout with a UDDI implementation there is an additional parameter that one can set - the transport class that is used for communication between Scout and the UDDI registry. If you are using Scout to communicate with jUDDI v3, we suggest leaving the transportClass as LocalTransport and using jUDDI's uddi.xml to use jUDDI's transports (InVM, RMI, WS).

jUDDI's uddi.xml resides in the *server/<config>/deploy/jbossesb.sar/META-INF* directory and contain the concept of a “node”, which is a jUDDI registry location. Use the node settings to determine whether you want to use JAX-WS, InVM, or RMI as your transport :

```
<node>
    <!-- required 'default' node -->
    <name>default</name>
    <description>Main jUDDI node</description>
    <properties>
                <property name="serverName" value="localhost" />
                 <property name="serverPort" value="8880" />
    </properties>
    <!-- JAX-WS Transport
     <proxyTransport>org.apache.juddi.v3.client.transport.JAXWSTransport</proxyTra
nsport>
    <custodyTransferUrl>http://${serverName}:${serverPort}/juddiv3/services/custody-
transfer?wsdl</custodyTransferUrl>
   <inquiryUrl>http://${serverName}:${serverPort}/juddiv3/services/inquiry?wsdl</inq
uiryUrl>
   <publishUrl>http://${serverName}:${serverPort}/juddiv3/services/publish?wsdl</pu
blishUrl>
   <securityUrl>http://${serverName}:${serverPort}/juddiv3/services/security?wsdl</s
ecurityUrl>
   <subscriptionUrl>http://${serverName}:${serverPort}/juddiv3/services/subscription
?wsdl</subscriptionUrl>
   <subscriptionListenerUrl>http://${serverName}:${serverPort}/juddiv3/services/subs
cription-listener?wsdl</subscriptionListenerUrl>
   <juddiApiUrl>http://${serverName}:${serverPort}/juddiv3/services/juddi-
api?wsdl</juddiApiUrl>
    -->
   <!-- In VM Transport Settings
   <proxyTransport>org.apache.juddi.v3.client.transport.InVMTransport</proxyTrans
port>
   <custodyTransferUrl>org.apache.juddi.api.impl.UDDICustodyTransferImpl</custod
yTransferUrl>
    <inquiryUrl>org.apache.juddi.api.impl.UDDIInquiryImpl</inquiryUrl>
    <publishUrl>org.apache.juddi.api.impl.UDDIPublicationImpl</publishUrl>
   <securityUrl>org.apache.juddi.api.impl.UDDISecurityImpl</securityUrl>
   <subscriptionUrl>org.apache.juddi.api.impl.UDDISubscriptionImpl</subscriptionUrl
>
   <subscriptionListenerUrl>org.apache.juddi.api.impl.UDDISubscriptionListenerImpl
</subscriptionListenerUrl>
   <juddiApiUrl>org.apache.juddi.api.impl.JUDDIApiImpl</juddiApiUrl>
-->
<!-- RMI Transport Settings -->
<proxyTransport>org.apache.juddi.v3.client.transport.RMITransport</proxyTransport
>
<custodyTransferUrl>/juddiv3/UDDICustodyTransferService</custodyTransferUrl>
<inquiryUrl>/juddiv3/UDDIInquiryService</inquiryUrl>
<publishUrl>/juddiv3/UDDIPublicationService</publishUrl>
<securityUrl>/juddiv3/UDDISecurityService</securityUrl>
<subscriptionUrl>/juddiv3/UDDISubscriptionService</subscriptionUrl>
<subscriptionListenerUrl>/juddiv3/UDDISubscriptionListenerService</subscriptionLis
tenerUrl>
<juddiApiUrl>/juddiv3/JUDDIApiService</juddiApiUrl>
<javaNamingFactoryInitial>org.jnp.interfaces.NamingContextFactory</javaNamingFa
ctoryInitial>
<javaNamingFactoryUrlPkgs>org.jboss.naming</javaNamingFactoryUrlPkgs>
<javaNamingProviderUrl>jnp://localhost:1099</javaNamingProviderUrl>
</node>
```

As seen above, a transport should specify a proxyTransport, a URL for all of the supported UDDI API (inquiry, publish, security, subscription, subscription-listener, custodytransfer), and a jUDDI API URL.   The RMI transport also includes JNDI settings. By default, the RMI settings are enabled – to switch transports you can comment them out and enable whichever of the transports you choose.

Using Scout and jUDDI

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

When using Scout with a UDDI implementation there is an additional parameter that one can set - the transport class that is used for communication between Scout and the UDDI registry. If you are using Scout to communicate with jUDDI v3, we suggest leaving the transportClass as LocalTransport and using jUDDI's uddi.xml to use jUDDI's transports (InVM, RMI, WS).

However, when communicating with another UDDI registry (Systinet, SOA Software, etc), it is preferable to use scout's JAXR transports. There are 4 implementations of this class which are based on SOAP, SAAJ, RMI and Local (embedded java). Note that when you change the transport, you will also have to change the query and lifecycle URIs. For example:

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

RMI
queryManagerURI

jnp://localhost:1099/InquiryService?org.apache.juddi.registry.rmi.Inquiry#inquire
lifeCycleManagerURI

jnp://localhost:1099/PublishService?org.apache.juddi.registry.rmi.Publish#publish
transportClass                    org.apache.ws.scout.transport.RMITransport

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

For jUDDI we have two requirements that need to be fulfilled:

1 access to the jUDDI database. juddi-ds.xml in jbossesb.sar specifies the JUDDI datasource, and persistence.xml refers to that datasource and specifies the corresponding hibernate database dialect.

2 juddiv3.properties and esb.juddi.xml. The configuration of jUDDI itself.

The database can be automatically created if the user you have created has enough rights to create tables. jUDDI should be able to create a database for any database that has a hibernate dialect associated with it.

Chapter 3

Registry Configuration Examples

Introduction

As mentioned before, by default the JBossESB is configured to use the JAXR API using Scout as its implementation and jUDDI as the registry. Here are some examples of how you can deploy this combo.

Embedded

All ESB components (with components we really mean JVMs in this case) can embed the registry and they all can connect to the same database (or different once if that makes sense).

JAXRScoutjUDDIJava Application1LocalJAXRScoutjUDDIJava Application2Local

Figure 2. Embedded jUDDI.

Properties example:

```
<properties name="registry">
            <property name="org.jboss.soa.esb.registry.implementationClass"

            value="org.jboss.internal.soa.esb.services.registry.JAXRRegistryImpl"/>

            <property name="org.jboss.soa.esb.registry.factoryClass"

            value="org.apache.ws.scout.registry.ConnectionFactoryImpl"/>

            <property name="org.jboss.soa.esb.registry.queryManagerURI"

            value="org.apache.juddi.v3.client.transport.wrapper.InquiryService#inquire"/>

            <property name="org.jboss.soa.esb.registry.securityManagerURI"

            value="org.apache.juddi.v3.client.transport.wrapper.SecurityService#secure"/>

            <property name="org.jboss.soa.esb.registry.lifeCycleManagerURI"
```

```
value="org.apache.juddi.v3.client.transport.wrapper.PublicationService#publish"/>

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

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

RMI using the juddi.war or jbossesb.sar

Deploy a version of the jUDDI that brings up an RMI service. The JBossESB deploys the RMI service by default – it starts the registry within the jbossesb.sar.

The jbossesb.sar also registers a RMI service.

RMI using another Service

If you want to use RMI to connect to a jUDDI v3 instance, it is suggested that you change the jbossesb.sar/META-INF/uddi.xml settings and specify that RMI instance. However, if you wish to connect to another UDDI instance, use the jbossesb-properties.xml settings and the scout JAXR RMI transport as seen below.

JAXRScoutjUDDIJava Application1LocalJAXRScoutJava Application2RMIRMI-ServiceJNDI-Registration
Figure 4. RMI using your own JNDI registration

Properties example: For application 1 you need the Local 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>
```

while for application2 you 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="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>
```

Where the hostname of the queryManagerURI and lifeCycleManagerURI need to point to the hostname on which jUDDI is running (which would be where application1 is running). Obviously application1 needs to have access to a naming service. To do the registration process you need to do something like:

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

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

context.bind(PUBLISH_SERVICE, publish);
```

2.4 SOAP

Finally, you can make the communication between Scout and jUDDI SOAP based. JBossESB does not ship with the jUDDI v 3.0.0 WAR, so you would have to obtain the WAR from the jUDDI project distribution and copy it into the jbossesb.sar. Make sure to leave the jbossesb.sar persistence.xml intact and remove all persistence from the WAR.

It is suggested that you use the JAXR LocalTransport and edit uddi.xml so that you use the JAX-WS transport for use with jUDDI v3. Uncomment the section that shows the JAX-WS settings and comment out the current transport (which should be RMI). However, if you are connecting to another UDDI registry, you may use the JAXR settings below.

JAXRScoutjuddi.warJava Application1JAXRScoutJava Application2Not Java Application3SOAPSOAPSOAP
Figure 5. SOAP.

Properties example:

```
<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>
```

1 JBossAS 4.2 ships with older versions of Scout and jUDDI. It is recommended to remove the juddi.sar to prevent versioning issues.

Chapter 4

Registry Troubleshooting

Scout and jUDDI pitfalls

- If you use RMI you need the juddi-client.jar, which can be found as part of the jUDDI distribution.
- Make sure the jbossesb-properties.xml file is on the classpath and read or else the registry will try to instantiate classes with the name of 'null'.
- Make sure that you have META-INF/uddi.xml that specifies a valid transport.
- Make sure the settings in your persistence.xml file are valid and that you are using the Hibernate dialect that matches your datasource.
- Make sure you have a juddiv3.properties file on your classpath for jUDDI to configure itself.
- In the event that a service fails or does not shut down cleanly, it is possible that stale entries may persist within a registry. You will have to remove these manually.

More Information

- For more information see the 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?

Introduction

The JBoss ESB Rule Service allows you to deploy rules created in JBoss Drools as services on the ESB. This is beneficial, because it means you don't have to develop as much client code to integrate rules into your application environment, and rules can be accessed as part of an action chain or orchestrated business process. To understand these types of services, you should first learn about JBoss Drools.

Rule Services are supported by the BusinessRulesProcessor action class and the DroolsRuleService, which implement the RuleService interface. While it is possible to use rule engines other than JBoss Drools, only JBoss Drools is supported out the the box. The BusinessRulesProcessor supports rules loaded from the classpath that are defined in .drl files, .dsl files (domain specific language support), and .xls (decision table support) files. These are primarily for testing, prototypes, and very simple rule services. There is no way to specify multiple rule files in the jboss-esb.xml file, so 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, the source of which can be:

1 Drools BRMS.

2 Imported DRL files.

3 Rules written in a Domain Specific Language.

4 Rules from Decision Tables.

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

The BusinessRulesProcessor action supports both Drools stateless and stateful execution models. Most rule services will be stateless. That is, a message will be sent to the rule service that includes all the facts to be inserted into the rule engine in the message body, the rules will execute, updating either the message and / or the facts. Stateful execution takes place over time, with several messages being sent to the rule service, the rules being executed each time, the message and / or facts being updated each time, and a final message that tells the rule service to dispose of the stateful session working memory of the rule engine. There are limitations in this configuration, namely that there can only be a single (stateful) rule service in the message flow. This may change in the future, when there are better ways to identify a stateful conversation over the ESB.

Rule Services using Drools

Introduction

The Rule Service support in the JBossESB uses JBossRules/Drools as its rule engine. JBossESB integrates with Drools through

- The BusinessRulesProcessor action class
- Rules written in Drools drl, dsl, decision table, or business rule editor.
- The ESB Message
- The ESB Message content, i.e., the objects in the message, which is the data going into the rules engine (the “facts”).

When a message is sent to the BusinessRulesProcessor, a certain rule set will execute over the objects in the message, and update those objects and / or the message.

Rule Set Creation

A rule set can be created using the Red Hat Developer Studio which includes a plug-in for JBoss Drools, or with Eclispe 3.3 and the plugin installed (see Drools download site for the plugin). Since the message is added as a global, you need to add jbossesb-rosetta.jar to your Drools project.

You can also write your rules using Drools BRMS business rule editor. When using the Drools BRMS, it is not necessary to add the ESB Message class to the imports, as long as jbossesb-rosetta.jar is somewhere on the classpath of the BRMS web application.

For a detailed discussion on rule creation and the Drools language itself please see the Drools documention.

For the most part, rules can be written without regard to their deployment on the ESB as a service. There are a few caveats however:

1) All rules deployed as a rule service must define the ESBMessage as a global, i.e.,

#declare any global variables here

```
global org.jboss.soa.esb.message.Message;
```

The rationale for this is that most rule services will want to update the message as a way of communicating results to other services in the flow, so the BusinessRulesProcessor / DroolsRuleService will always set the message as a global.

2) The BusinessRulesProcessor / DroolsRuleService does not provide a means to set globals in the jboss-esb.xml and have them set in working memory. This would have made for additional configuration support in the jboss-esb.xml, and could be supported in the future. For now, if additional globals (other than the ESB Message) need to be set, they can be done in higher salience rule. E.g.,

```
rule "Set a global"
```

```
          salience 100
          when
          then

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

3) The DroolsRuleService does not provide a means to start a RuleFlow from the rule service. This also would have made for additional configuration support in the jboss-esb.xml, and could be supported in the future. For now, if a RuleFlow needs to be started, this can be done in higher salience rule. E.g.,

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

Rule Service Consumers

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

In some cases the client is ESB 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(0);
order.setQuantity(20);
order.setUnitPrice(new Float("20.0"));
message.getBody().add("Order", order);
```

In other cases the data may be in an XML message, and a transformation service will be added to the message flow to transform the XML to POJOs before the rule service is invoked.

Using 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);
```

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);
```

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

These can be added directly by an ESB aware client. A non-ESB aware client would have to communicate the position of the message (first, ongoing, last) in the data, and an action class would need to be added to the pipeline to add the properties to the ESB message (see quickstarts/business_ruleservice_stateful for an example of this type of service).

N.B. In ESB 4.6 and prior releases, the "continue" functionality for stateful rule execution did not dispose of working memories if the value of the property was false or it was absent. This has now been fixed through the work for JBESB-2900. The previous behavior can be re-enabled by changing the value of the configuration property "org.jboss.soa.esb.services.rules.continueState", found within jbossesb-properties.xml, to true.

Configuration

Configuration of a rule service is in the jboss-esb action element for the service. Several configuration parameters are required or optional

The action class and name is required:

```
  <action

 class="org.jboss.soa.esb.actions.BusinessRulesProcessor"
           name="OrderDiscountRuleService">
```

This configures the action class and its name. The name is user defined.

One of the following is also required

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

for specifying a drl file, or

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

for specifying a dsl and dslr (domain specific language) files , or

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

for specifying a decisionTable on the classpath, or

```
   <property name="ruleAgentProperties"

 value="brmsdeployedrules.properties" />
```

for specifying 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 system.

Several example configurations follow:

Example 1: Rules are in a drl, execution is stateless.

```
<action
```

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

Example 2: Rules are in a drl, execution is stateful.

In this scenario, a service may receive multiple messages over time and wish to use rules to accumulate data across this message set. Each time a message is received, the rules will be fired within the context of a single Stateful Session. The active Session can be disposed of (reset) via the "dispose" property.

NOTES:

1. A single, synchronized Session instance is shared across all concurrent executions of a Stateful Session deployments. This greatly limits the type of usecase for which the Stateful deployment model is applicable. If multiple, client oriented sessions are required per Service deployment, consider using a jBPM/BPEL solution.
2. Stateful Sessions are not persistent and are therefore volatile in nature.
3. Stateful Sessions are not clustered.

```
<action

class="org.jboss.soa.esb.actions.BusinessRulesProcessor"
          name="OrderDiscountMultipleRuleServiceStateful">
          <property name="ruleSet"

          value="drl/OrderDiscountOnMultipleOrders.drl" />
          <property name="ruleReload" value="false" />
          <property name="stateful" value="true" >
          <property name="object-paths">
                    <object-path esb="body.Customer" />
                    <object-path esb="body.Order" />
          </property>
</action>
```

Example 3: Rules are in a Domain Specific Language, execution is stateless.

```
<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 4: Rules are in a DecisionTable, execution is stateless.

```
<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 5: Rules are in the BRMS, execution is stateless.

```
<action

class="org.jboss.soa.esb.actions.BusinessRulesProcessor"
          name="RuleAgentPolicyService"
          <property name="ruleAgentProperties"

          value="ruleAgent/brmsdeployedrules.properties" />
          <property name="object-paths">
                    <object-path esb="body.Driver" />
                    <object-path esb="body.Policy" />
              </property>
</action>
```

The action attributes to the action tag are show in Table 1. The attributes specify which action is to be used and which name this action is to be given.

Attribute	***Description***
Class	Action class, :

	`org.jboss.soa.esb.actions.BusinessRulesProcessor`
Name	Custom action name

Table 1. BusinessRulesProcessor action configuration attributes.

The action properties are shown in Table 2. The properties specify the set of rules (ruleSet) to be used in this action.

Property	***Description***
ruleSet	Optional reference to a file containing the Drools 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 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.
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 package(s). Note there is no need to specify ruleReload with a ruleAgent, since this is controlled through the properties file.
Stateful	Optional property which can be to true to specify that the rule service will receive multiple messages over time that will add fact to the rule engine working memory and re-execute the rules. ***NOTE:*** *A single shared session is shared across all Service executions.*
object-paths	Optional property to pass Message objects into Drools WorkingMemory.

Table 2. BusinessRulesProcessor action configuration properties.

Object Paths

Note that Drools treats objects as shallow objects to achieve highly optimized performance, so what if you want to evaluate an object deeper in the object tree? An 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 can flatten out entire collections. For example a collectoin Orders will be unrolled, and each order object inserted into working memory.

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 3 shows a layout of the business_rules_service quickstart to demonstrate a typical package.

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

Figure 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>
```

What is Content-Based Routing?

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.

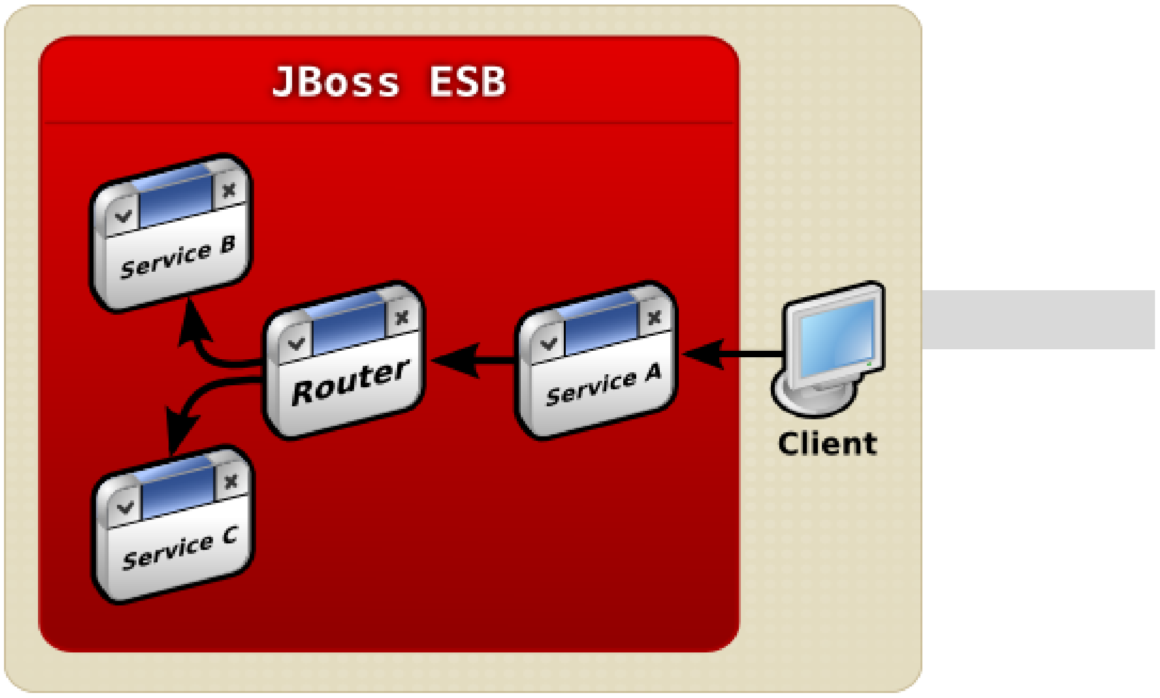
Simple example

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).



Content Based Routing using XPath

Introduction

An easy way of performing content based routing in JBoss ESB is is via the XPath rules provider on the ContentBasedRouter action.

This provider is very easy to use and supports both inline and external rule definitions.

Inline Rule Definitions

Defining inline XPath routing rules is trivial. You just need to configure the "**cbrAlias**" property to "**XPath**" and then define the routing rules in the <route-to> configurations in the container destinations property.

Example:

```
<action class="org.jboss.soa.esb.actions.ContentBasedRouter" name="ContentBasedRouter">
  <property name="cbrAlias" value="XPath"/>
  <property name="destinations">
    <route-to service-category="BlueTeam"  service-name="GoBlue"  expression="/Order[@statusCode='0']" />
    <route-to service-category="RedTeam"   service-name="GoRed"   expression="/Order[@statusCode='1']" />
    <route-to service-category="GreenTeam" service-name="GoGreen" expression="/Order[@statusCode='2']" />
  </property>
</action>
```

External Rule Definitions

Defining external XPath routing rules is also trivial. Again, you configure the "**cbrAlias**" property to "**XPath**" and then

1 Define the routing expressions in a .properties file, where the property keys are the destination names and the property values are the Xpath expressions for routing to the destination in question.

2 Define the routing rules in the <route-to> configurations in the container destinations property, with the "**destination-name**" attribute referring to the XPath rule key as defined in the external .properties file.

Example:

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

The XPath rules file is a simple .properties file as follows:

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

Namespaces

XML namespace prefix-to-uri mappings are defined in the <namespace> elements, contained within the "**namespaces**" container property. Namespaces prefix-to-uri mappings are define in exactly the same way for both inline and external rule definitions.

Example:

```
<action class="org.jboss.soa.esb.actions.ContentBasedRouter" name="ContentBasedRouter">
  <property name="cbrAlias" value="XPath"/>
  <property name="namespaces">
     <route-to prefix="ord" uri="http://www.acne.com/order" />
  </property>
  <property name="destinations">
     <route-to service-category="BlueTeam"  service-name="GoBlue"  expression="/ord:Order[@statusCode='0']" />
     <route-to service-category="RedTeam"   service-name="GoRed"   expression="/ord:Order[@statusCode='1']" />
     <route-to service-category="GreenTeam" service-name="GoGreen" expression="/ord:Order[@statusCode='2']" />
  </property>
</action>
```

Content Based Routing using Regex

Introduction

An easy way of performing content based routing in JBoss ESB is is via the Regex rules provider on the ContentBasedRouter action.

This provider is very easy to use and supports both inline and external rule definitions.

Inline Rule Definitions

Defining inline Regex routing rules is trivial. You just need to configure the "**cbrAlias**" property to "**Regex**" and then define the routing rules in the <route-to> configurations in the container destinations property.

Example:

```
<action class="org.jboss.soa.esb.actions.ContentBasedRouter" name="ContentBasedRouter">
  <property name="cbrAlias" value="Regex"/>
  <property name="destinations">
     <route-to service-category="BlueTeam"  service-name="GoBlue"  expression="#*111#*" />
     <route-to service-category="RedTeam"   service-name="GoRed"   expression="#*222#*" />
     <route-to service-category="GreenTeam" service-name="GoGreen" expression="#*333#*" />
  </property>
</action>
```

External Rule Definitions

Defining external XPath routing rules is also trivial. Again, you configure the "**cbrAlias**" property to "**Regex**" and then

1 Define the routing expressions in a .properties file, where the property keys are the destination names and the property values are the Regex expressions for routing to the destination in question.

2 Define the routing rules in the <route-to> configurations in the container destinations property, with the "**destination-name**" attribute referring to the Regex rule key as defined in the external .properties file.

Example:

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

The XPath rules file is a simple .properties file as follows:

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

Content Based Routing using Drools

Introduction

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

- a routing rule set, written in Drools drl (and optionally dsl) language.
- The ESB Message 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 sent 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.

Three different routing action classes

JBossESB ships with three slightly different routing action classes. Each of these action classes implements an Enterprise Integration Pattern. For more information of the Enterprise Integration Pattern you can check the JBossESB Wiki. 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.

org.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.

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 1 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 `global java.util.List destinations;` which will make the list of destinations available to the ESB

  Figure 1. Create a new ruleSet using JbossIDE or Red Hat Developer Studio

The message will be asserted into 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.

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. xpathLowerThan "<element>", "<value>": yields true if the element is found and it's value is lower then the value.

XPath and namespaces

To use namespaces with XPath, one needs to specify which namespace prefixes are to be used in the XPath expression. The namespace prefixes are specified as a comma separated list like this: "prefix=uri,prefix=uri". This can be accomplished for all the above types of XPath expressions:

1. xpathMatch expr "<expression>" use namespaces "<namepaces>"

2. xpathEquals expr "<expression>", "<value>" use namespaces "<namspaces>"

3. xpathGreaterThan "<expression>", "<value>" use namespaces "<namspaces>"

4. xpathLowerThan expr "<expression>, "<value> use namespaces "<namespaces>

Notice that the namespace aware statements differ in that they need the extra "**expr**" in front of the XPath expression. This is do avoid colliding with the non XPath aware statements in the dsl file.

Also note that the prefixes do not have to match those used in the xml to be evaluated, it only matters that the URI is the same.

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.

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. Figure 1 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 2. Example Content Based Routing Service configuration.

The action attributes to the action tag are show in Table 1. 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`

	`org.jboss.soa.esb.actions.MessageFilter`
Name	Custom action name

Table 1. CBR action configuration attributes.

The action properties are shown in Table 2. 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.
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 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 2. CBR action configuration properties.

Object Paths

Note that Drools treats objects as shallow objects to achieve highly optimized performance, so what if you want to evaluate an object deeper in the object tree? An 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 perform a (Smooks-) transformation on the message first, to unroll the collection.

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 message 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 RuleService chapter.

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

NOTES:

1. A single, synchronized Session instance is shared across all concurrent executions of a Stateful Session deployments. This greatly limits the type of usecase for which the Stateful deployment model is applicable. If multiple, client oriented sessions are required per Service deployment, consider using a jBPM/BPEL solution.
2. Stateful Sessions are not persistent and are therefore volatile in nature.
3. Stateful Sessions are not clustered.

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 of the Drools manual.

For more details about the RuleService please see RuleService chapter.

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

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 of the Drools manual.

For more details about the RuleService please see RuleService chapter.

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

Executing Business Rules

Related to rule execution for routing is the rule execution to simply *modifying* data in the message according to business rules. 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.

Changing RuleService implementations

If you would like to use a different RuleService then 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.

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 3 shows a layout of the simple_cbr quickstart to demonstrate a typical package.

```
simple_cbr.esb
 |   jbm-queue-service.xml
 |   SimpleCBRRules-XPath.drl
 |   SimpleCBRRules.drl
 |
 +---META-INF
 |       deployment.xml
 |       jboss-esb.xml
 |       MANIFEST.MF
 |
 \---org
    \---jboss
        \---soa
            \---esb
                \---samples
                    \---quickstart
                        \---simplecbr
                            |   MyJMSListenerAction.class
                            |   ReturnJMSMessage.class
                            |   RouteExpressShipping.class
                            |   RouteNormalShipping.class
                            |
                            \---test
                                    ReceiveJMSMessage$1.class
                                    ReceiveJMSMessage.class
                                    SendJMSMessage.class
```

Figure 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

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:

}}etc...SplitSplitetc...etc...SAX Event Stream (HUGE Message)

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.

```
(1) <jb:bindings beanId="order" class="java.util.HashMap" createOnElement="order">
        <jb:value property="orderId" decoder="Integer" data="order/@id"/>
```

```
       <jb:value property="customerNumber" decoder="Long" data="header/customer/@number"/>
       <jb:value property="customerName" data="header/customer"/>
       <jb:wiring property="orderItem" beanIdRef="orderItem"/>
   </jb:bindings>
(2) <jb:bindings beanId="orderItem" class="java.util.HashMap" createOnElement="order-item">
       <jb:value property="itemId" decoder="Integer" data="order-item/@id"/>
       <jb:value property="productId" decoder="Long" data="order-item/product"/>
       <jb:value property="quantity" decoder="Integer" data="order-item/quantity"/>
       <jb:value property="price" decoder="Double" data="order-item/price"/>
   </jb:bindings>

(3) <file:outputStream openOnElement="order-item" resourceName="orderItemSplitStream">
       <file:fileNamePattern>
           order-${order.orderId}-${order.orderItem.itemId}.xml
       </file:fileNamePattern>
       <file:destinationDirectoryPattern>target/orders</file:destinationDirectoryPattern>
       <file:listFileNamePattern>order-${order.orderId}.lst</file:listFileNamePattern>

       <file:highWaterMark mark="3"/>
   </file:outputStream>

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

Resource configurations #1 and #2 are there to bind data from the source message into Java Objects in the Smooks bean context. In this case, we're just binding the data into HashMaps. 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 used 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 File OutputStream (resource #3). The File OutputStream (resource #3) also gets applied on every order item, managing the file output for the split messages.

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 to message aware endpoints. We will be adding a quickstart dedicated to demoing these features of the ESB. Check the User Forum for details.

JBoss ESB 4.7 upgrades to Smooks v1.1, which means that the split described above can be done without having to define the binding configurations (#1 and #2). For more on how to do this, see the documentation on “FreeMarker Transforms using NodeModels” in section 4 of the Smooks User Guide.

Message Transformation

Overview

JBoss ESB supports message data transformation through a number of mechanisms:

- **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
- **XSLT**: JBoss ESB supports message transformation through the standard XSLT usage model, as well as through the Smooks.
- **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.

Smooks

Message Transformation on JBossESB 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. See the Message Action Guide for more details.

Samples & Tutorials

1. A number of Transformation Quickstart samples accompany the JBossESB distribution. Check out the "transform_*" Quickstarts[1].
2. A number of tutorials are available online on the Smooks website. Any of these samples can be easily ported to JBossESB.

1Note that some of the ESB Quickstarts are still using the older “SmooksTransformer” action class. The SmooksAction is a more flexible and easier to use alternative to the SmooksTransformer. The SmooksTransformer will be deprecated in a future release (and removed later again).

XSL Transformations

XSLT transformation are supported by the XstlAction. Please see the section “XSLTAction” in the ProgrammersGuide for more information.

jBPM Integration

Introduction

JBoss jBPM is a powerful workflow and BPM (Business Process Management) engine. It enables the creation of business processes that coordinate between people, applications and services. With its modular architecture, JBoss jBPM combines easy development of workflow applications with a flexible and scalable process engine. The JBoss jBPM process designer graphically 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 jBPM documentation [TB-JBPM-USER] 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.

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>
```

The first Mbean configuration element contains the configuration for the DatabaseInitializer. By default the attributes are configured as follows:

- "Datasource" - use a datasource called JbpmDS,

- “ExistsSql” - check if the database exists by running the sql: “Select * from JBPM_ID_USER”
- “SqlFiles” - if the database does not exist it will attempt to run the files jbpm.jpdl.hsqldb.sql and import.sql. These files reside in the jbpm.esb/jbpm-sql directory and can be modified if needed. Note that slightly different ddl files are provided for the various databases.

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. The login screen is shown in Fig. 1.

Figure 1. The jBPM Console

Please check the jBPM documentation [TB-JBPM-USER] to change the security settings for this application, which will involve change some settings in the conf/login-config.xml. The console can be used for deploying and monitoring jBPM processes, but is 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 [JBESB-QS] 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:

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

which are the jbossesb.esb and the JbpmDS datasource. This information is used to determine the deployment order.

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>
```

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>
   <!-- change the following element to jms-jca-provider to
           enable transactional context  -->
   <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>
```

For more details on what the JbpmCallbackService does, please see the “jBPM to ESB” section later on in this chapter.

jBPM configuration

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

By default the jbpm.cfg.xml is set to use the JTA transacion manager, as defined in the section:

```
<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>
```

Other settings are left to the default jBPM settings.

The hibernate.cfg.xml 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>
```

Hibernate is not used to create the database schema, instead we use our own DatabaseInitiazer mbean, as mentioned in the previous section.

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

Note that the configuration files that usually ship with the jbpm-console.war have been removed so that all configuration is centralized in the configuration files in the root of the jbpm.esb archive.

Creation and Deployment of a Process Definition

To create a Process Definition we recommend using the eclipse based jBPM Process Designer Plugin [KA-JBPM-GPD]. You can either download and install it to eclipse yourself, or use JBoss Developer Studio. Figure 2 shows the graphical editor.

Figure 2. jBPM Grapical 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). Figure 3 shows the “Deployment” tab view.

Figure 3. The 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 par, which means that they end up in the database where they will be stored and versioned. However it is strongly discouraged to do this in the ESB environment as you will risk running into class loading issues. Instead we recommend deploying your classes in the lib directory of the server. You can deploy a process definition

- straight from the eclipse plugin, by clicking on the “Test Connection..” button and, on success, by clicking on the “Deploy Process Archive” button,
- by saving the deployment to a par file and using the jBPM console to deploy the archive, see Figure 4, or finally,
- by using the DeployProcessToServer jBPM ant task.

Figure 4. Someone with administrative privileges can deploy new process definition.

JBossESB to jBPM

JBossESB 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 - Start a new ProcessInstance given a process definition that was already deployed to jBPM. The NewProcessInstance-Command leaves the Process Instance in the start state, which would be needed if there is an task associated to the Start node (i.e. some task on some actor's tasklist). In most cases however you would like the new Process Instance to move to the first node, which is where the next command comes in.
- StartProcessInstanceCommand - Identical to the NewProcessInstance-Command, but additionally the new Process Instance is moved from the Start position into the first Node.

- GetProcessInstanceVariablesCommand – The the root node variables for a process instance, using the process instance ID.
- CancelProcessInstanceCommand - Cancel a ProcessInstance. i.e. when an 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. Details on that are discussed later.

The configuration for this action in the jboss-esb.xml looks like

```
<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>
```

There are two required action attributes:

- **name** - required attribute. You are free to use any value for the name attribute as long as it is unique in the action pipeline.
- **class** - required attribute. This attributes needs to be set to "org.jboss.soa.esb.services.jbpm.actions.BpmProcessor"

Furthermore one can configure the following configuration properties:

- **command** – required property. Needs to be one of: NewProcessInstanceCommand, StartProcessInstanceCommand, GetProcessInstanceVariablesCommand or CancelProcessInstanceCommand.
- **processdefinition** – required property for the New- and Start-ProcessInstanceCommands *if the process-definition-id property is not used.* The value of this property should reference a process definition that is already deployed to jBPM and of which you want to create a new instance. This property does not apply to the Signal- and CancelProcessInstance-Commands.
- process-definition-id – required property for the New- and Start-ProcessInstanceCommands *if the processdefinition property is not used*. The value of this property should reference a processdefintion id in jBPM of which you want to create a new instance. This property does not apply to the Signal- and CancelProcessInstanceCommands.
- **actor** – optional property to specify the jBPM actor id, which applies to the New- and StartProcessInstanceCommands only.
- **key** – optional property to specify the value of the jBPM key. For example one can pass a unique invoice id as the value for this key. On the jBPM side this key is as the "business" key id field. The key is a string based business key property on the process instance. The combination of business key + process definition must be unique if a business key is supplied. The key value can hold an MVEL expression to extract the desired value from the EsbMessage. For example if you have a named parameter called "businessKey" in the body of your message

you would use “body.businessKey”. Note that this property is used for the New- and StartProcessInstanceCommands only.

- transition-name – optional property. This property only applies to the StartProcessInstance- and Signal Commands, and is of use only if there are *more then one* transition out of the current node. If this property is not specified the *default* transition out of the node is taken. The default transition is the *first* transition in the list of transition defined for that node in the jBPM processdefinition.xml.
- **esbToBpmVars** - 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** – required attribute which can contain an MVEL expression to extract a value anywhere from the EsbMessage.
- **bpm** – optional attribute containing the name which be used on the jBPM side. If omitted the esb name is used.
- **default** – optional attribute which can hold a default value if the esb MVEL expression does not find a value set in the EsbMessage.
- **bpmToEsbVars** - Structurally identical to the “esbToBpmVars” property (above). Used with the GetProcessInstanceVariablesCommand for mapping jBPM process instance variables (root token variables) onto the ESB message.
- **reply-to-originator** - 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.

Finally some variables can be set on the **body** of the EsbMessage:

- **jbpmProcessInstId** – required ESB message Body parameter that applies to the GetProcessInstanceVariablesCommand and CancelProcessInstanceCommand commands. It is up to the user make sure this value is set as a named parameter on the EsbMessage body.

Exception Handling JBossESB to jBPM

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.

jBPM to JBossESB

The JBossESB to jBPM maybe interesting but the other way around is probably far more interesting 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 and here we're focusing on the details of the integration first. The integration implements two jBPM action handler classes. The classes are “EsbActionHandler” and “EsbNotifier”. The EsbActionHandler is a request-reply type action, which drops a message on a Service and then waits for a response while the EsbNotifier only drops a message on a Service and continues its processing. The interaction with JBossESB is asynchronous in nature and does not block the process instance while the Service executes. First we'll discuss the EsbNotifier as it implements a subset of the configuration of EsbActionHandler class.

EsbNotifier

The EsbNotifier action should be attached to an outgoing transition. This way the jBPM processing can move along while the request to the ESB service is processed in the background. In the jBPM processdefinition.xml we would need attach the EsbNotifier to the outgoing transition. For example the configuration for a "Ship It" node could look like:

```
<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>
```

The following attributes can be specified:

- **name** – required attribute. User specified name of the action
- **class** – 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, required if not using the reply-to-originator functionality.
- **esbServiceName** – The name of the ESB service, required if not using the reply-to-originator functionality.
- **replyToOriginator** – Specify the 'reply' or 'fault' originator address previously stored in the process instance on creation.
- **globalProcessScope** - optional element. 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 given 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** – optional element. This element takes a list of mapping subelements to map a jBPM context variable to a location in the EsbMessage. Each mapping element can have the following attributes:
  - **bpm** – 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", so for example you can use mapping like:

```
<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" />
```

and one can reference jBPM context variable names directly.

- **esb** – optional attribute. 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** – optional attribute. If the variable is not found in jBPM context the value of this field is taken instead.
- **process-scope** – optional attribute. This boolean valued parameter can override the setting of the setting of the globalProcessScope for this mapping.

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

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 sent by the JBossESB callback Service. An example configuration for the EsbActionHandler could look like:

```
<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" />
   </esbToBpmVars>
 </action>

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

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

- **esbToBpmVars** – optional element. *This subelement is identical to the esbToBpmVars property mention in the previous section* "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, however, it should be noted that the effect of the globalProcessScope value, if not specified, will default to true when setting variables. The list consists of mapping elements. Each mapping element can have the following attributes:
    - **esb** – required attribute which can contain an MVEL expression to extract a value anywhere from the EsbMessage.
    - **bpm** – optional attribute containing the name which be used on the jBPM side. If omitted the esb name is used.
    - **default** – optional attribute which can hold a default value if the esb MVEL expression does not find a value set in the EsbMessage.
    - **process-scope** – optional attribute. This boolean valued parameter can override the setting of the setting of the globalProcessScope for this mapping.
- **exceptionTransition** – 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. If for example you only want to wait 10 seconds for the Service to complete you could add

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

to the node element. Now if no signal is received within 10 seconds of entering this node, the transition called "time-out" is taken.

Exception Handling jBPM -> JBossESB

There are two types of scenarios where exceptions can arise.

- The first type of exception is a MessageDeliveryException which is thrown by the ServiceInvoker. If this occurs it means that 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 one can add an ExceptionHandler [TB-JBPM-USER] to handle this exception.
- 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 does 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 5.

Scenario 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. The process definition configuration would look like

```
<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.

Figure 5. Three exception handling scenarios: time-out, exception-transition and exception-decision.

Scenario 2. Exception Transition

To handle exception that may occur during processing of the Service, one can define an exceptionTransition. When doing so the faultTo EPR is set on the message such that 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. The definition of Service2 looks like

```
<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>
```

where in the action, the exceptionTransition is set to “exception”. In this scenario the process also ends in the “ExceptionHandling” node.

Scenario 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. The configuration would look like

```
<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>
```

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. Here we check for the existence of the "errorCode" variable using the condition

```
<condition>#{ errorCode!=void }</condition>
```

For more details on conditional transitions please see the jBPM documentation [TB-JBPM-USER].

Service Orchestration

Introduction

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.

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 for their award wining ActiveBPEL WS-BPEL Engine. In support of this, JBossESB ships with a Quickstart dedicated to demonstrating how JBossESB and ActiveBPEL can collaborate effectively to provide a WS-BPEL based orchestration layer on top of a set of Services that don't expose Webservice Interfaces (the “webservice_bpel” Quickstart). JBossESB provides the Webservice Integration and ActiveBPEL provides the Process Orchestration. A number of flash based walk-thrus of this Quickstart are also available online.

1 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.

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 6 shows an example of such a flow-chart, which represents a simplified order process. This example is taken from the bpm_orchestration4 quick start [JBESB-QS] which ships with JBossESB.

Figure 6. “Order Process” Service Orchestration using jBPM

In the “Order Process” Diagram three of the nodes are JBossESB Services, the “Intake Order”, “Calculate Discount” and the “Ship It” nodes. For these nodes the regular “Node” type was used, which is why these are labeled with “<<Node>>”. Each of these nodes have the EsbActionHandler attached to the node itself. This means that the jBPM node will send a request to the Service and then it will remain in a wait state, waiting for the ESB to call back into the node with the response of the Service. The response of the service can then be used within jBPM context. For example when the Service of the “Intake Order” responds, the response is then used to populate the “Review Order” form. The “Review Order” node is a “Task Node”. Task Nodes are designed for human interaction. In this case someone is required to review the order before the Order Process can process.

To create the diagram in Figure 6, select File > New > Other, and from the Selection wizard select “JBoss jBPM “Process Definition” as shown in Figure 7. 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 7. Select new JBoss jBPM Process Definition

After creating a new process definition. You can drag and drop any item from menu, shown in Figure 8, 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" [KA-BLOG].

Figure 8. jBPM IDE menu palette.

Before building the “Order Process” diagram of Figure 6, we'd 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 quick start [JBESB-QS] 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 as discussed in Chapter 1. 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>
```

and 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 9. 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

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

which references the variables set in the jBPM context.

When the process reaches the “Review Node”, as shown in Figure 9. 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 10. The user can 'examine' the task by clicking on it and the user will be presented with a form as shown in Figure 11. The user can update some of the values and click “Save and Close” to let the process move to the next Node.

Figure 10. The task list for user 'user'

Figure 11. 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.

Figure 12. 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 [JBESB-QS] it uses drools to determine whether this order should be shipped 'normal' or 'express'.

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.

Figure 13. Deployment of the “Order Process”

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 [JBESB-QS] 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.

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 quick start bpm_orchestration2 [JBESB-QS] how to use the jBPM fork and join concepts.

The Message Store

Introduction

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.

1 In JBossESB 4.2 the Message Store is also used for storing messages that need to be redelivered in the event of failures. See the Programmers Guide around the ServiceInvoker for further details.

Message Store interface

The `org.jboss.soa.esb.services.persistence.MessageStore` interface is defined as follows:

```
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;
}
```

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.

1 `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.

1 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`.

Transactions

The Message Store interface does not currently support transactions. As such, any use of the store within the scope of a global transaction will not be coordinated within the scope of any global transaction, i.e., 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.

Configuring the Message Store

To configure your Message Store, you can change and override the default service implementation through the following settings found in the `jbossesb-properties.xml`:

```
<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"/>

                <!--  property name="org.jboss.soa.esb.persistence.db.conn.manager"
value="org.jboss.internal.soa.esb.persistence.manager.J2eeConnectionManager"/ -->

                <!-- this property is only used if using 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 again, very simple. They can be found under lib/jbossesb.esb/message-store-sql/<db_type>/create_database.sql of your JBossESB 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`)
);
```

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 new UUID random number generator in jdk 1.5. 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*. 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.StandaloneCo
nnectionManager"/>
<!--  property name="org.jboss.soa.esb.persistence.db.conn.manager"
value="org.jboss.soa.esb.persistence.manager.J2eeConnectionManager"/
-->
<!-- this property is only used if using 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
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, etc. 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

Introduction

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.

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>
```

Property	Description
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 your own login modules.

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

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:

```
<service category="Security" name="SimpleListenerSecured">
  <security moduleName="messaging" runAs="adminRole" rolesAllowed=”adminRole,
normalUsers”
callbackHandler="org.jboss.internal.soa.esb.services.security.UserPassCallbackH
andler">
              <property name="property1" value="value1"/>
              <property name="property2" value="value2"/>
  </security>
  ...
</service>
```

Security properties description:

Property	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:

Property	Description
org.jboss.soa.esb.services.security.contextTimeout	Optional property that lets the service override the global security context timeout (ms) specified in jbossesb-properties.xml.
org.jboss.soa.esb.services.security.contextPropagatorImplementationClass	Optional property that lets the service override the global security context propagator class implementation specified in jbossesb-properties.xml.

Example of overriding global configuration settings:

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

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

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

</security>
```

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, it checks that the SecurityContext is still valid, and if so, re-authentication is not performed. Note that the SecurityContext is only valid on a single ESB node. If the message is routed to a different ESB node, re-authentication will be required.

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 the “SecurityService Configuration” section 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.

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. Please see “Configuring Secuirtyon a Service” and “SecurityService Configuration” to see how this is done.

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 credentials that were used to call the ESB, be used as the credentials when calling an EJB method. This can be accomplished by specifying 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. Please see “Configuring Security on a Service” and “SecurityService Configuration” for examples of this.

Implementations of SecurityContextPropagator

Class	*Description*
Package: org.jboss.internal.soa.esb.services.security Class: JBossASContextPropagator	This propagator will propagate security credentials to a JBoss Application Server(AS). 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.

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.

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.

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. See “Role Mapping” for details on the role mapping functionality.

Example configuration:

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

Property	***Description***
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>
```

Property	Description
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. Please refer to the section “Role Mapping” for more details on this.

Role Mapping

This file is can be optionally specified in login-config.xml by using the '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.

Password encryption

Configuration files in JBossESB sometimes require passwords which up until now have been specified in clear text in the configuration files. This is a security risk and something that should be avoided. In JBossESB you have the ability to specify a path to a file that contains an encrypted password wherever a password is required.

Creating a encrypted password file

This can be achived by performing the following steps:

1. Go to the conf directory of your jboss server instance (eg: default/conf)

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

Option	Description
Salt	The salt used for the encryption. This is the “welcometojboss” string in the example above.
Iteration	The number of iterations. This is the number 13 in the example above.
Clear text password	The password you wish to encrypt. This is the string “testpass” in the example above.

Password file name	The name of the file where the encrypted password will be saved. This is the “esb.password” string in the example above.

Configuration of encrypted password files

Configuration is done by simply substituting the clear text password in your configuration file(s) with the path to the encrypted password file.

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<String> rolesAllowed,
                              final SecurityContext securityContext);

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

          void logout(final SecurityConfig securityConfig);

          void refreshSecurityConfig();
}
```

The default implementation is based on JAAS, but this can be customized by implementing the above interface and configuring the custom SecurityService to be used in jbossesb-properties.xml. For more details of the SecurityService interface methods, please refer to the javadocs.

References

[JBESB-QS], JBossESB QuickStarts, http://anonsvn.jboss.org/repos/labs/labs/jbossesb/tags/JBESB_4_4_GA/product/samples/quickstarts

[KA-BLOG] ESB Service Node, Koen Aers,
http://koentsje.blogspot.com/2008/01/esb-service-node-in-jbpm-jpdl-gpd-312.html

[KA-JBPM-GPD], JBoss jBPM Graphical Process Designer, Koen Aers,
http://docs.jboss.com/jbpm/v3/gpd/

[TB-JBPM-USER] jBPM User Documentation, Tom Baaijens
http://docs.jboss.com/jbpm/v3/userguide/

[TF-BPEL], Service Orchestration using ActiveBPEL, Tom Fennely,
http://anonsvn.jboss.org/repos/labs/labs/jbossesb/tags/JBESB_4_4_GA/product/docs/ServicesGuide.pdf

Index