

SOA ESB jBPM Integration Guide

4.2

JBoss Enterprise SOA Platform



ISBN:

Publication date: February, 2008

Guide to ESB / jBPM Integration in the JBoss Enterprise SOA Platform

SOA ESB jBPM Integration Guide: JBoss Enterprise SOA Platform

Copyright © 2008 Red Hat, Inc.

Copyright © 2008 Red Hat, Inc.. This material may only be distributed subject to the terms and conditions set forth in the Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License (which is presently available at <http://creativecommons.org/licenses/by-nc-sa/3.0/>).

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

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

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

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

Preface	vii
1. Document Conventions	vii
2. We Need Feedback	viii
1. jBPM Integration	1
1. Introduction	1
2. Integration Configuration	1
3. jBPM configuration	4
4. Creation and Deployment of a Process Definition	5
5. JBossESB to jBPM	8
6. Asynchronous Signaling	10
7. Exception Handling JBossESB to jBPM	11
8. jBPM to JBossESB	11
9. Exception Handling jBPM -> JBossESB	14
2. Service Orchestration	19
1. Introduction	19
2. Orchestration Diagram	19
3. Process Deployment and Instantiation	28
4. Conclusion	29
A. Revision History	31

Preface

1. Document Conventions

Certain words in this manual are represented in different fonts, styles, and weights. This highlighting indicates that the word is part of a specific category. The categories include the following:

Courier font

Courier font represents `commands, file names and paths, and prompts`.

When shown as below, it indicates computer output:

```
Desktop      about.html    logs          paulwesterberg.png
Mail         backupfiles  mail          reports
```

Courier font

Bold Courier font represents text that you are to type, such as: `service jonas start`

If you have to run a command as root, the root prompt (`#`) precedes the command:

```
# gconftool-2
```

italic Courier font

Italic Courier font represents a variable, such as an installation directory:

```
install_dir/bin/
```

font

Bold font represents **application programs** and **text found on a graphical interface**.

When shown like this: **OK**, it indicates a button on a graphical application interface.

Additionally, the manual uses different strategies to draw your attention to pieces of information. In order of how critical the information is to you, these items are marked as follows:



Note

A note is typically information that you need to understand the behavior of the system.



Tip

A tip is typically an alternative way of performing a task.



Important

Important information is necessary, but possibly unexpected, such as a configuration change that will not persist after a reboot.



Caution

A caution indicates an act that would violate your support agreement, such as recompiling the kernel.



Warning

A warning indicates potential data loss, as may happen when tuning hardware for maximum performance.

2. We Need Feedback

If you find a typographical error in the *SOA ESB JBPM Integration Guide*, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in JIRA: <http://jira.jboss.com/jira/> against the component *FIXME*.

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

ESB_JBPM_INT

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

jBPM Integration

1. 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 first.

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

- *Service Orchestration:* ESB services can be orchestrated using jBPM. You can create a jBPM process definition which makes calls into ESB services.
- *Human Task Management:* jBPM allows you to incorporate human task management integrated with machine based services.

2. Integration Configuration

The jbpms.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 jbpms.esb/jbpms-service.xml.

```
<classpath codebase="deploy" archives="jbpms.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:/JbpmsDS</attribute>
  <attribute name="ExistsSql">
    select * from JBPM_ID_USER</attribute>
  <attribute name="SqlFiles">
    jbpms-sql/jbpms.jpdl.hsqldb.sql,jbpms-sql/import.sql
  </attribute>
  <depends>
    jboss.jca:service=DataSourceBinding,name=JbpmsDS
  </depends>
</mbean>
<mbean code=
  "org.jboss.soa.esb.services.jbpms.configuration.JbpmsService"
  name="jboss.esb:service=JbpmsService">
</mbean>
```

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

- *Datasource*: se 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 [Figure 1.1, "The jBPM Console"](#).

User Name	Password	Group(s)
manager	manager	user manager admin
user	user	user
shipper	shipper	user
admin	admin	user admin

Figure 1.1. The jBPM Console

Please check the jBPM documentation to change the security settings for this application, which will involve changing some settings in the `conf/login-config.xml`. The console can be used for deploying and monitoring jBPM processes, but it can also be used for human task

management. For the different users a customized task list will be shown and they can work on these tasks. The quickstart `bpm_orchestration4` demonstrates this feature. This quickstart does not ship as part of the JBoss Enterprise SOA Platform, but can be obtained via the web from:

```
http://anonsvn.labs.jboss.com/labs/jbossesb/branches/JBESB_4_2_1_GA_CP
/product/samples/quickstarts/bpm_orchestration4/
```



Note

The URL above has been split across two lines for readability. It should be entered as one continuous line with no breaks or spaces.

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-DCQLListener"
                    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` (`jbpm-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 simply 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 [Section 8, “jBPM to JBossESB”](#) later on in this chapter.

3. 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 transaction 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.

4. Creation and Deployment of a Process Definition

To create a Process Definition we recommend using a graphical editor such as JBoss Developer Studio. [Figure 1.2, “jBPM Graphical Editor”](#) shows a graphical editor.

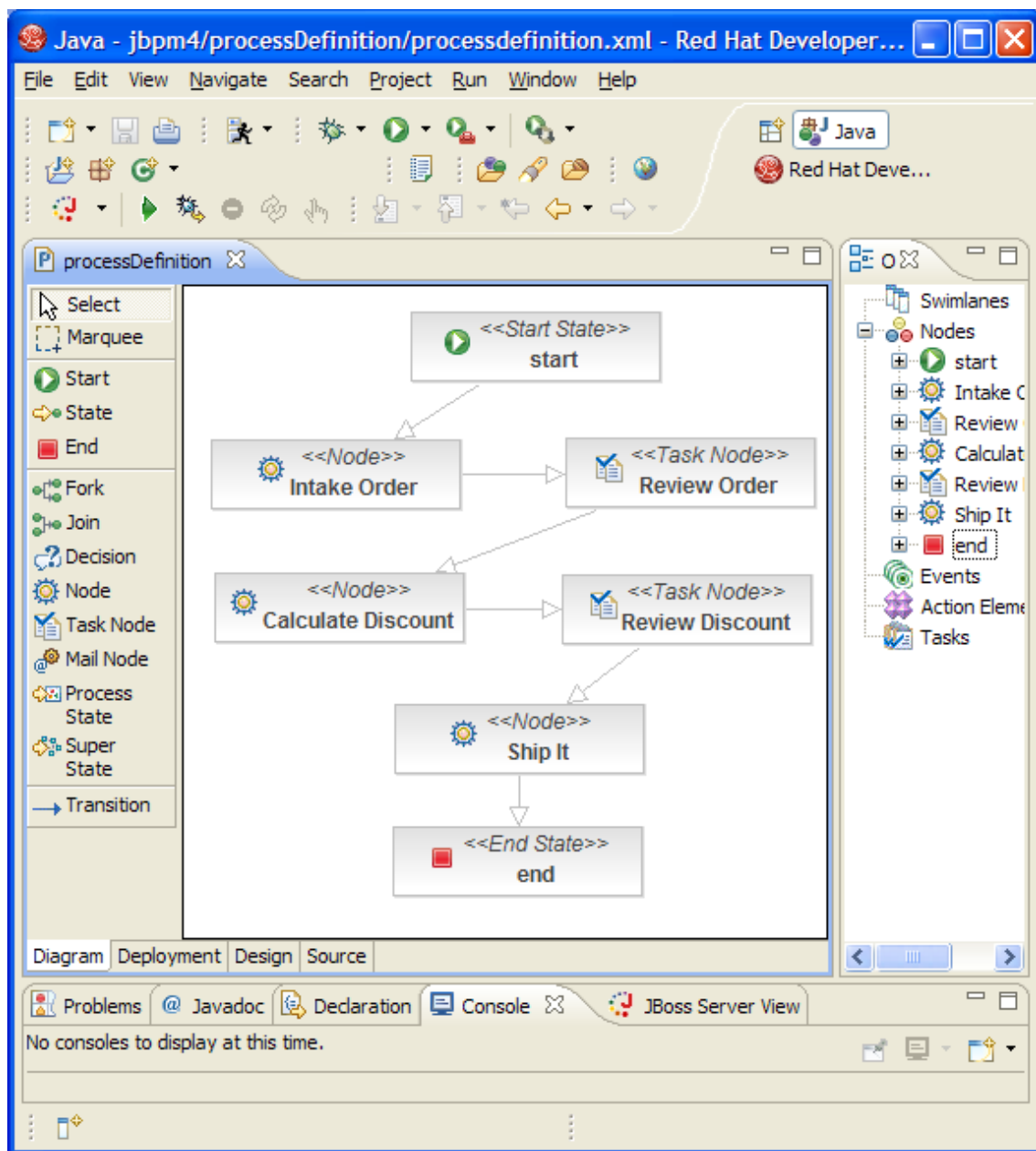


Figure 1.2. jBPM Graphical Editor

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

To deploy a process definition the server needs to be up and running. Only then can you go to the 'Deployment' tab in the graphical designer to deploy a process archive (par). [Figure 1.3,](#)

“The Deployment View” shows the “Deployment” tab view.

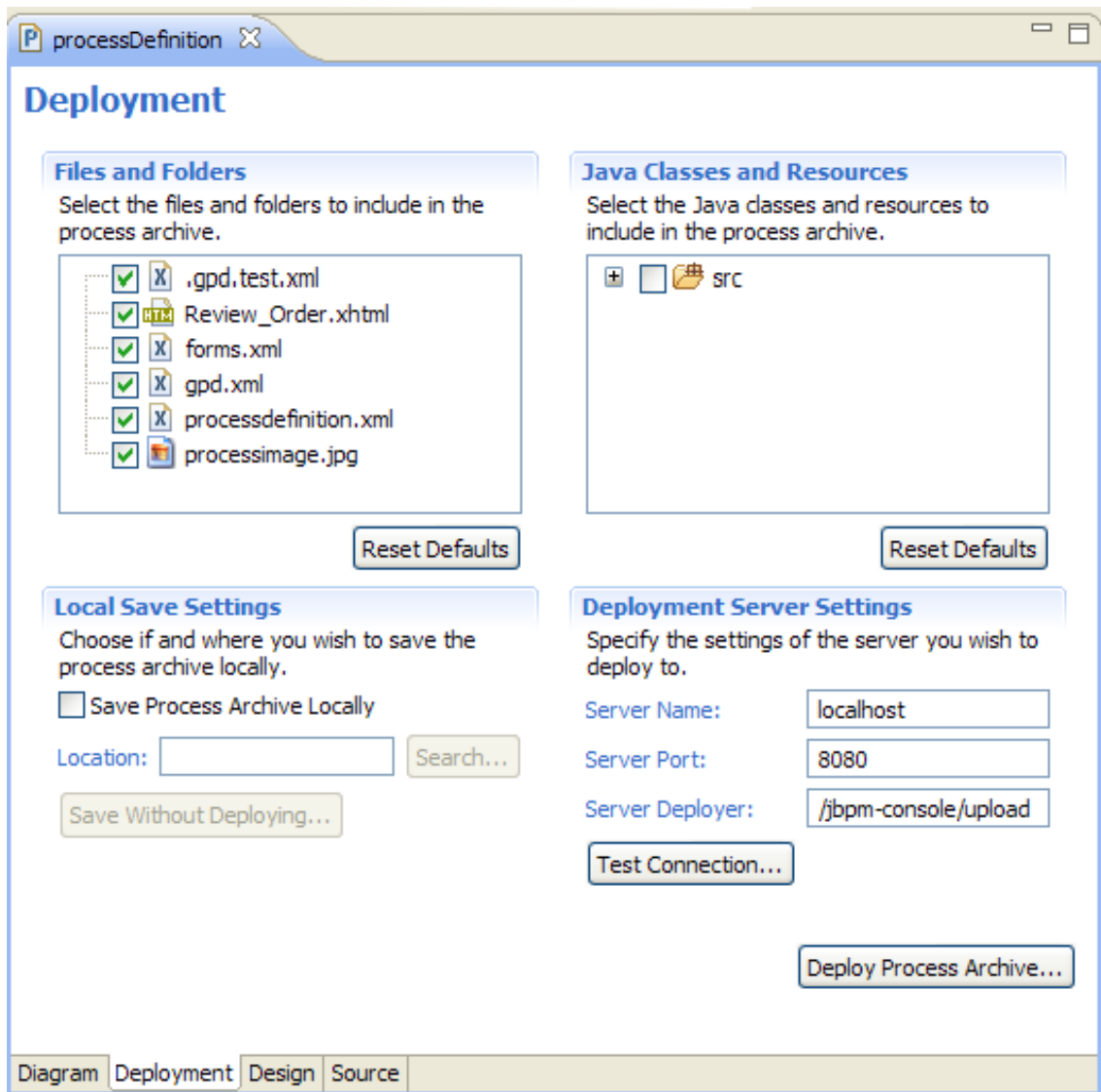


Figure 1.3. The Deployment View

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 jar, which means that they will be stored and versioned in the database. 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 (requires administrative privileges), see [Figure 1.4, “Deploying a new process definition”](#)
- by saving the deployment to a par file and using the jBPM ant task to deploy the archive.

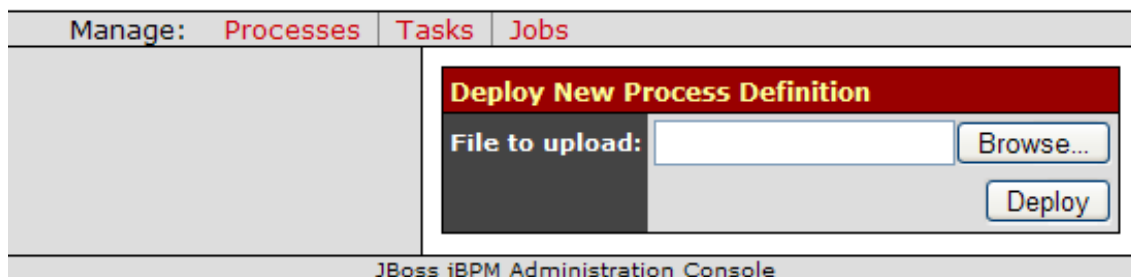


Figure 1.4. Deploying a new process definition

5. 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 `NewProcessInstanceCommand` 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 `NewProcessInstanceCommand`, but additionally the new `Process Instance` is moved from the Start position into the first Node.
- `SignalCommand` - Signal a Node with a certain transition to go to the next Node. This action requires some jBPM context variables to be set on the message, in particular the Node Id. Details on that are discussed later, but it basically means that the message needs to pass through jBPM first to add this context information.
- `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 this:

```
<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 attribute 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`, `SignalCommand` or `CancelProcessInstanceCommand`.
- *processdefinition* – required property for `NewProcessInstanceCommand` and `StartProcessInstanceCommand` 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 wish to create a new instance. This property does not apply to `SignalProcessInstanceCommand` and `CancelProcessInstanceCommand`.
- *process-definition-id* – required property for the `NewProcessInstanceCommand` and `StartProcessInstanceCommand` if the `processdefinition` property is not used. The value of this property should reference a processdefinition id in jBPM of which you want to create a new instance. This property does not apply to the `SignalProcessInstanceCommand` and `CancelProcessInstanceCommand`.
- *actor* – optional property to specify the jBPM actor id, which applies to the `NewProcessInstanceCommand` and `StartProcessInstanceCommand` 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 `NewStartProcessInstanceCommand` and `StartProcessInstanceCommand` only.
- *transition-name* – optional property. This property only applies to the

`StartProcessInstanceCommand` and `SignalCommand`, and is of use only if there are more than 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 `NewProcessInstanceCommand` and `StartProcessInstanceCommand` and the `SignalCommand`. 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`.

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

- *jbpmProcessInstId* – required parameter which applies to the `CancelProcessInstanceCommand` only. It is up to the user make sure this value is set as a named parameter on the `EsbMessage` body.
- *jbpmTokenId* or *jbpmProcessInstId* – either one is a required parameter and applies to the `SignalCommand` only. The `SignalCommand` first looks for the value of the token id to which it will send a signal. If this is not set it will try to obtain the process instance id and get the root token. It is up to the user make sure either the *jbpmTokenId* or the *jbpmProcessInstId* is set on the `EsbMessage` body.

6. Asynchronous Signaling

In most cases you will want your signal commands from JBossESB to jBPM to be asynchronous. The advantages of making the signal asynchronous are:

- a potential performance increase as you don't have to wait for the signal command to return, as it returns immediately. Note that in synchronous mode it won't return until the process enters a new wait state.
- transaction demarcation. Each node will run in its own thread and its own transaction.

You can achieve this simply by adding the attribute `async="true"` to the node in your jPDL. Asynchronous nodes will not be executed in the thread of the client. Instead, a message is sent over the asynchronous messaging system and the thread is returned to the client (meaning

that the `token.signal()` or `taskInstance.end()` will return). For more details you may want to check chapter 9 and chapter 14 of the jBPM Userguide.

Just to clarify, let's say we have a transition from node1 to node2. Now if you want the signal to node1 (to transition to node2) to return immediately, you need to set `async="true"` in jPDL of node2 to make the signal asynchronous.

7. 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.

8. jBPM to JBossESB

The JBossESB to jBPM may be 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 - here we'll focus first on the details of the integration. 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" async="true">
  <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* – required element. The category name of the ESB service
- *esbServiceName* – required element. The name of the ESB service.
- *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.
- *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 send by the JBossESB callback Service. An example configuration for the `EsbActionHandler` could look like:

```
<node name="Intake Order" async="true">
  <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 [Section 5, “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. 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*.
- *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.

9. 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* 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* 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 *EsbActionHandler*.

To illustrate the type of error handling that is now possible using standard jBPM features we will discuss some scenarios illustrated in [Figure 1.5, “Three exception handling scenarios”](#).

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 1.5, "Three exception handling scenarios"](#). 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" async="true">
  <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 1.5, "Three exception handling scenarios"](#) this means that the processing ends up in the `ExceptionHandling` node in which one can perform compensating work.

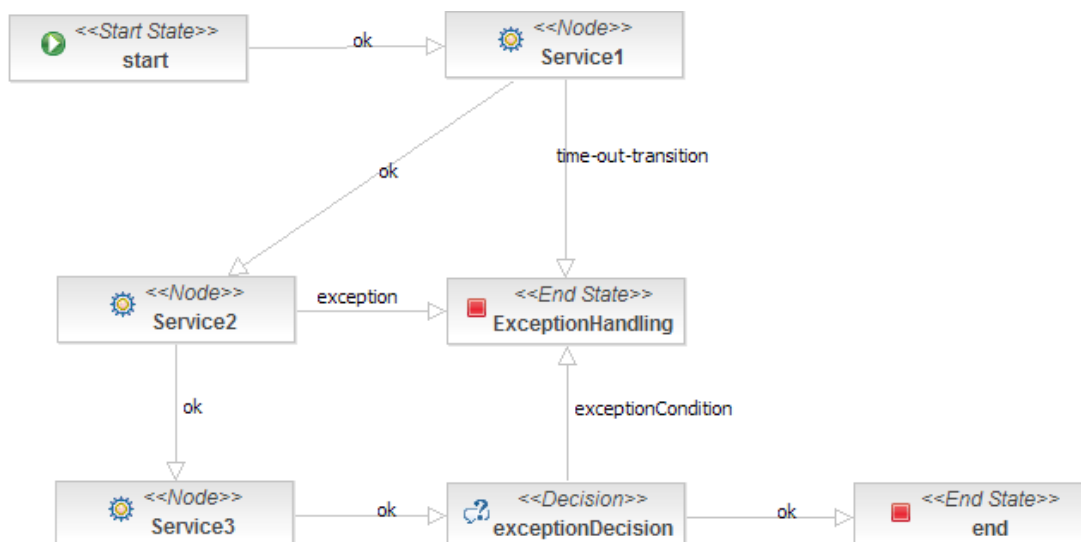


Figure 1.5. Three exception handling scenarios

Scenario 2. Exception Transition.

To handle an 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" async="true">
  <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.

Service Orchestration

1. Introduction

Service Orchestration is the arrangement of business processes. Traditionally BPEL is used to execute SOAP based Web Services. In the Guide 'Service Orchestration' you can obtain more details on how to use ActiveBPEL with JBossESB. If you want to orchestrate JBossESB regardless of the end point type, then it makes more sense to use jBPM. This chapter explains how to use the integration discussed in [Chapter 1, jBPM Integration](#) to do Service Orchestration using jBPM.

2. Orchestration Diagram

A key component of Service Orchestration is to use a flow-chart like design tool to design and deploy processes. JBoss Developer Studio can be used for just this. [Figure 2.1, ““Order Process” Service Orchestration using jBPM”](#) shows an example of such a flow-chart, which represents a simplified order process. This example is taken from the bpm_orchestration4 quick start which ships with the JBoss Enterprise SOA Platform.

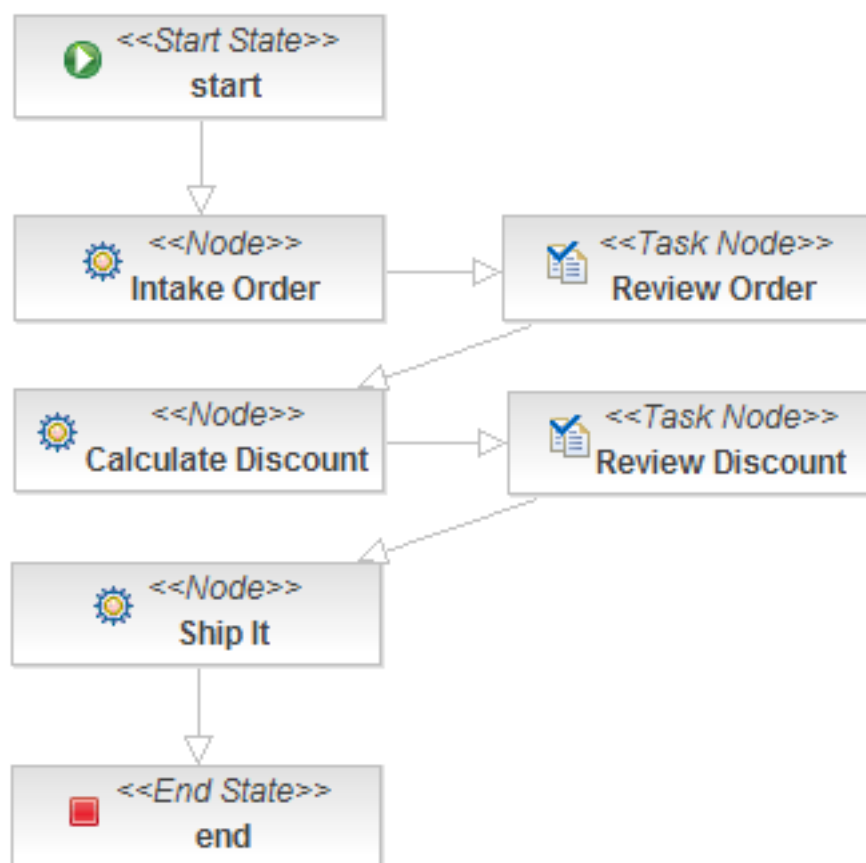


Figure 2.1. “Order Process” Service Orchestration using jBPM

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 2.1, ““Order Process” Service Orchestration using jBPM”](#) using JBoss Developer Studio:

1. Open JBoss Developer Studio
2. Click on "Create New...".
3. Select "Create jBPM Process Project".
4. Enter a project name.
5. Click Next.
6. Locate the jBPM runtime by pointing to the `jbp- jpd1` directory of your SOA installation.
7. Access your new jBPM project by clicking on "Workbench"
8. Bring up the Graphical Process Designer by expanding the `src/main/ jpd1` node, then expanding the "simple" node and clicking on `gpd.xml`

After creating a new process definition you can drag and drop any item from the menu, shown in [Figure 2.2, “jBPM IDE menu palette”](#), into the process design view. You can switch between the design and source modes if needed to check the XML elements that are being added, or to add XML fragments that are needed for the integration.

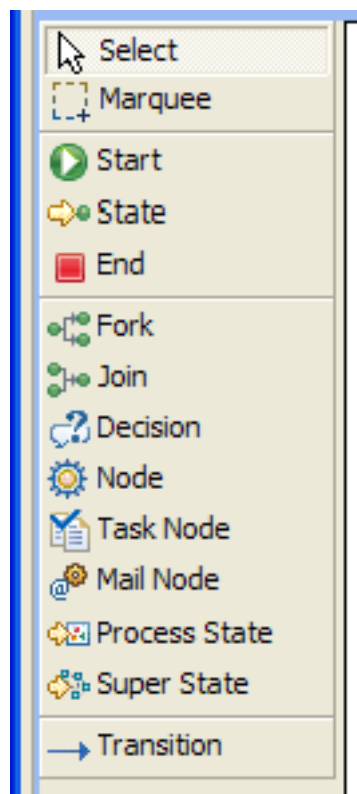


Figure 2.2. jBPM IDE menu palette

Before building the “Order Process” diagram of [Figure 2.1, ““Order Process” Service Orchestration using jBPM”](#), 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 if you want details on them, but the 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, jBPM Integration](#). 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" async="true">
    <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" async="true">

  <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`.

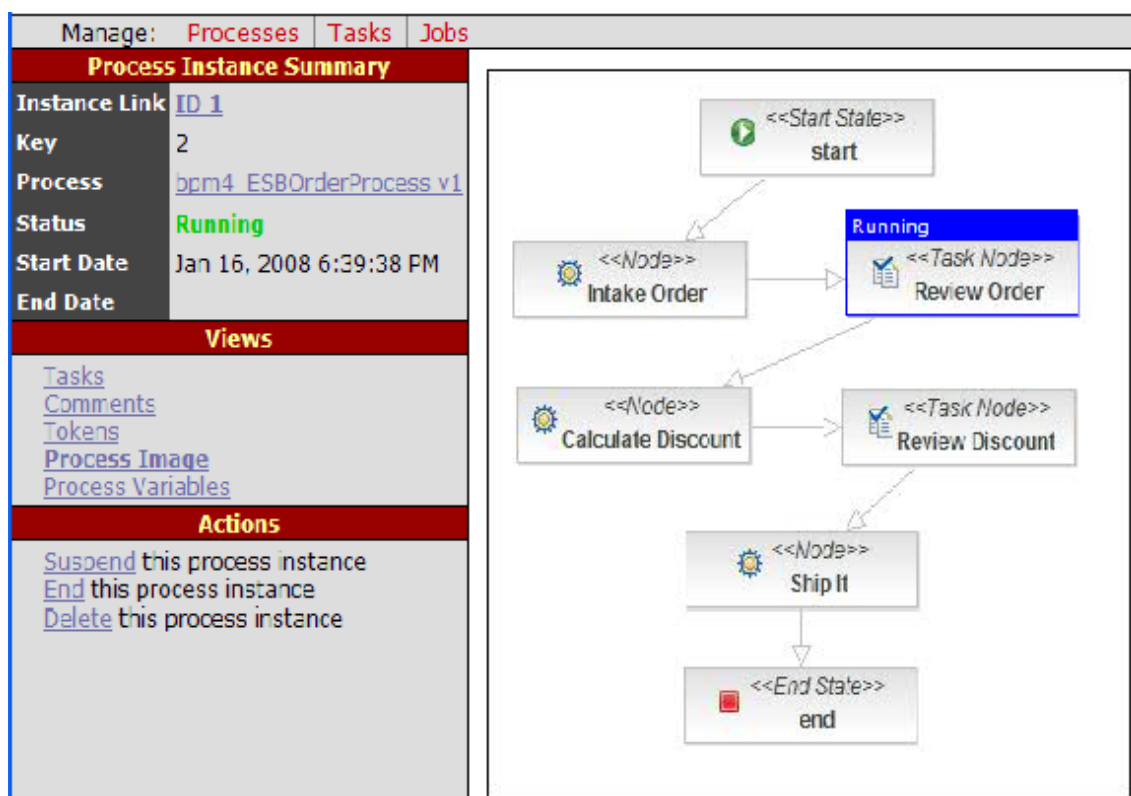


Figure 2.3. The Order process reached the “Review Order” node

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" async="true">
  <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 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 2.3, “The Order process reached the “Review Order” node”](#). When the 'user' user logs into the jbpm-console the user can click on 'Tasks' to see a list of tasks, as shown in [Figure 2.4, “The task list for user 'user'”](#). The user can 'examine' the task by clicking on it and the user will be presented with the a form as shown in [Figure 2.5, “The “Order Review” form”](#). The user can update some of the values and click “Save and Close” to let the process move to the next Node.

Manage: Processes		Tasks					
Tasks					First Prev - Page 1 of 1 - Next Last		
ID	Name	Pooled Actors	Assigned To	Status	Start Date	End Date	Actions
	<input type="text"/>		<input type="text"/>	<input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> R <input checked="" type="checkbox"/> G <input type="checkbox"/> C			Apply filter Clear filter
1	Order Review		user	Not Started			Examine Suspend Start

Figure 2.4. The task list for user 'user'

Task Summary	
Task Link	ID 1
Name	Order Review
Status	Not Started
Assigned To	user
Token	ID 1
Process Instance	ID 1
Process	bpm4_ESBOrderProcess v1
Created Date	Jan 16, 2008 6:39:48 PM

Order Review	
customer_firstName	<input type="text" value="Rex"/>
customer_lastName	<input type="text" value="Myers"/>
customer_status	<input type="text" value="60"/>
order_totalamount	<input type="text" value="64.92"/>
order_priority	<input type="text" value="3"/>
order_orderid	<input type="text" value="2"/>
order_discount	<input type="text"/>
entireOrder	<Order netAmount='59.97'
Actions	<input type="button" value="Save"/> <input type="button" value="Cancel"/> <input type="button" value="Save and Close"/>

JBoss BPM Administration Console

Figure 2.5. The “Order Review” form

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

```
<jbpm:datacell>
  <f:facet name="header">
    <h:outputText value="customer_firstName"/>
  </f:facet> <node name="Calculate Discount" async="true">
    <action name="esbAction" class="
org.jbpm.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>
  <h:inputText value="#{var['customer_firstName']}" />
</jbpm:datacell>
```

The Service receives the customer and orderHeader objects as well as the entireOrderAsXML, and computes a discount. The response maps the

`body.order_orderDiscount` value onto a jBPM context variable called `order_discount`, and the process is signaled to move to the “Review Discount” task node.

Discount Review	
customer_firstName	<input type="text" value="Rex"/>
customer_lastName	<input type="text" value="Myers"/>
customer_status	<input type="text" value="60"/>
order_totalamount	<input type="text" value="64.92"/>
order_priority	<input type="text" value="3"/>
order_orderid	<input type="text" value="2"/>
order_discount	<input type="text" value="8.5"/>
entireOrder	<input type="text" value='<Order netAmount="59.97'/>
Actions	<input type="button" value="Save"/> <input type="button" value="Cancel"/> <input type="button" value="Save and Close"/>

Figure 2.6. 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" async="true">
  <transition name="ProcessingComplete" to="end">
    <action name="ShipItAction" class=
      "org.jboss.soa.esb.services.jbpm.actionhandlers.EsbNotifier">
      <esbCategoryName>BPM_Orchestration4</esbCategoryName>
      <esbServiceName>ShippingService</esbServiceName>
      <bpmToEsbVars>
        <mapping bpm="entireCustomerAsObject" esb="customer" />
        <mapping bpm="entireOrderAsObject" esb="orderHeader" />
        <mapping bpm="entireOrderAsXML" esb="entireOrderAsXML" />
      </bpmToEsbVars>
    </action>
  </transition>
</node>
```

After notifying the `ShippingService` the Order process moves to the 'end' state and terminates.

The `ShippingService` itself may still be finishing up. In `bpm_orchestration4` it uses drools to determine whether this order should be shipped 'normal' or 'express'.

3. Process Deployment and Instantiation

In the previous paragraph we create the process definition and we quietly assumed we had an instance of it to explain the process flow. But now that we have created the `processdefinition.xml`, we can deploy it to jBPM using the IDE, ant or the `jbpm-console` (as explained in [Chapter 1, jBPM Integration](#)). In this example we use the IDE and deployed the files: `Review_Order.xhtml`, `forms.xml`, `gpd.xml`, `processdefinition.xml` and the `processimage.jpg`. On deployment the IDE creates a `par` archive 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.

The screenshot displays the jBPM IDE's deployment interface, organized into four main panels:

- Files and Folders:** A list of files to be included in the process archive. The files are: `.gpd.test.xml`, `Review_Order.xhtml`, `forms.xml`, `gpd.xml`, `processdefinition.xml`, and `processimage.jpg`. Each file has a checkbox and a small icon. A "Reset Defaults" button is located below the list.
- Java Classes and Resources:** A section for selecting Java classes and resources. It shows a folder icon and the text "src". A "Reset Defaults" button is located below.
- Local Save Settings:** A section for choosing where to save the process archive locally. It includes a checkbox for "Save Process Archive Locally" and a "Location:" field with a "Search..." button. A "Save Without Deploying..." button is at the bottom.
- Deployment Server Settings:** A section for specifying the settings of the server to deploy to. It includes fields for "Server Name:" (localhost), "Server Port:" (8080), and "Server Deployer:" (/jbpm-console/upload). A "Test Connection..." button is located below these fields.

A "Deploy Process Archive..." button is located at the bottom right of the interface.

Figure 2.7. 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 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.

4. Conclusion

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

Appendix A. Revision History

Revision History

Revision 1.0

