



JBoss Enterprise BRMS Platform 5

BRMS Business Process Management Guide

For JBoss Developers and Rules Authors

Edition 5.3.1

JBoss Enterprise BRMS Platform 5 BRMS Business Process Management Guide

For JBoss Developers and Rules Authors
Edition 5.3.1

Red Hat Content Services

Legal Notice

Copyright © 2012 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

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

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

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

A guide to using the Business Process Management capabilities of JBoss Enterprise Business Rules Management System 5.3.1

Table of Contents

PREFACE	3
CHAPTER 1. INTRODUCTION	4
1.1. INTRODUCTION	4
CHAPTER 2. BUSINESS PROCESS MANAGEMENT API	5
2.1. THE API	5
2.2. CREATE THE KNOWLEDGE BASE	5
2.3. CREATE A SESSION	6
2.4. EVENTS LISTENERS	9
CHAPTER 3. PROCESS OVERVIEW	12
3.1. PROCESS OVERVIEW	12
3.2. PROCESS NODES	12
3.3. PROCESS PROPERTIES	13
3.4. EVENTS	13
3.5. ACTIVITIES	16
3.6. GATEWAYS	23
3.7. DATA	25
3.8. CONSTRAINTS	26
3.9. TIMERS	27
3.10. UPDATING PROCESSES	27
3.11. PROCESS INSTANCE MIGRATION	28
3.12. MULTI-THREADING	29
3.13. ASYNCHRONOUS HANDLERS	30
CHAPTER 4. BPMN 2.0 NOTATION	32
4.1. BUSINESS PROCESS MODEL AND NOTATION (BPMN) 2.0 SPECIFICATION	32
4.2. AN EXAMPLE BPMN 2.0 PROCESS	36
CHAPTER 5. PROCESS DESIGNER	39
5.1. PROCESS DESIGNER	39
5.2. CONFIGURING THE PROCESS DESIGNER	39
5.3. PROCESS CREATION AND VALIDATION	40
5.4. IMPORTING EXISTING PROCESSES	41
5.5. VIEW AND SHARE PROCESSES	42
5.6. DEFINING DOMAIN-SPECIFIC SERVICE NODES	43
5.7. CONNECTING TO A SERVICE REPOSITORY	44
5.8. GENERATE PROCESS AND TASK FORMS	44
CHAPTER 6. JBOSS DEVELOPER STUDIO	47
6.1. JBOSS DEVELOPER STUDIO	47
6.2. PROJECT CREATION	47
6.3. PROCESS CREATION	47
6.4. VALIDATION AND DEBUGGING	48
CHAPTER 7. PERSISTENCE	49
7.1. PERSISTENT	49
7.2. RUNTIME STATE	49
7.3. CONFIGURING PERSISTENCE	52
7.4. HISTORY LOG	55
CHAPTER 8. BUSINESS CENTRAL CONSOLE	60
8.1. BUSINESS CENTRAL CONSOLE	60

8.2. BUSINESS CENTRAL CONSOLE AND BRMS INTEGRATION	60
8.3. LOG ON TO THE BUSINESS CENTRAL CONSOLE	62
8.4. MANAGING PROCESS INSTANCES	62
8.5. HUMAN TASK LISTS	63
8.6. REGISTERING SERVICE HANDLERS	63
8.7. ADDING NEW PROCESS AND TASK FORMS	64
8.8. REST INTERFACE	65
CHAPTER 9. DOMAIN-SPECIFIC PROCESSES	67
9.1. DOMAIN-SPECIFIC SERVICE NODES	67
9.2. DEFINE A WORK ITEM	67
9.3. REGISTER THE WORK DEFINITION	69
9.4. EXECUTING SERVICE NODES	70
9.5. SERVICE REPOSITORY	71
CHAPTER 10. HUMAN TASKS	73
10.1. HUMAN TASKS	73
10.2. ADDING HUMAN TASKS TO PROCESSES	73
10.3. HUMAN TASK SERVICE	80
10.4. HUMAN TASK PERSISTENCE	87
CHAPTER 11. TESTING AND DEBUGGING	101
11.1. UNIT TESTING	101
11.2. DEBUGGING	104
CHAPTER 12. BUSINESS ACTIVITY MONITORING	107
12.1. BUSINESS ACTIVITY MONITORING	107
CHAPTER 13. INTEGRATION	108
13.1. INTEGRATION	108
13.2. OSGI	108
13.3. SPRING	109
13.4. MAVEN	110
APPENDIX A. REVISION HISTORY	114

PREFACE

CHAPTER 1. INTRODUCTION

1.1. INTRODUCTION

The Business Process Management capabilities of JBoss BRMS 5.3 enable users to model business processes as flow charts that describe the steps necessary to achieve business goals.



Figure 1.1. Example Process

JBoss BRMS and the Business Process Management (BPM) engine can deploy as standalone installations running in JBoss Enterprise Application Server 5.1.2, or they can be deployed to an existing application server. A full list of certified and compatible configurations can be found at <http://www.redhat.com/resourcelibrary/articles/jboss-enterprise-brms-supported-configurations>.

For installation instructions, refer to the *JBoss BRMS 5.3 Getting Started Guide*.

The Business Process Management capabilities included with JBoss BRMS 5.3 are written in Java and implement the BPMN 2.0 specification (for full details see *Business Process Model and Notation (BPMN) 2.0 Specification*).

The following tools are included to support business processes throughout their entire life cycle:

- A graphical editor embedded in the JBoss BRMS user interface to create and edit business processes
- A graphical editor plug-in for JBoss Developer studio to create and edit business processes
- A management console that provides process monitoring, Human Task management, and the ability to add reporting
- Integration with the JBoss BRMS repository for storing, versioning, and managing processes
- Integration with an external Human Tasks service

See Also:

- [Section 4.1, “Business Process Model and Notation \(BPMN\) 2.0 Specification”](#)

[Report a bug](#)

CHAPTER 2. BUSINESS PROCESS MANAGEMENT API

2.1. THE API

The JBoss BRMS Platform knowledge API is used to load and execute processes, business rules, and complex event processing.

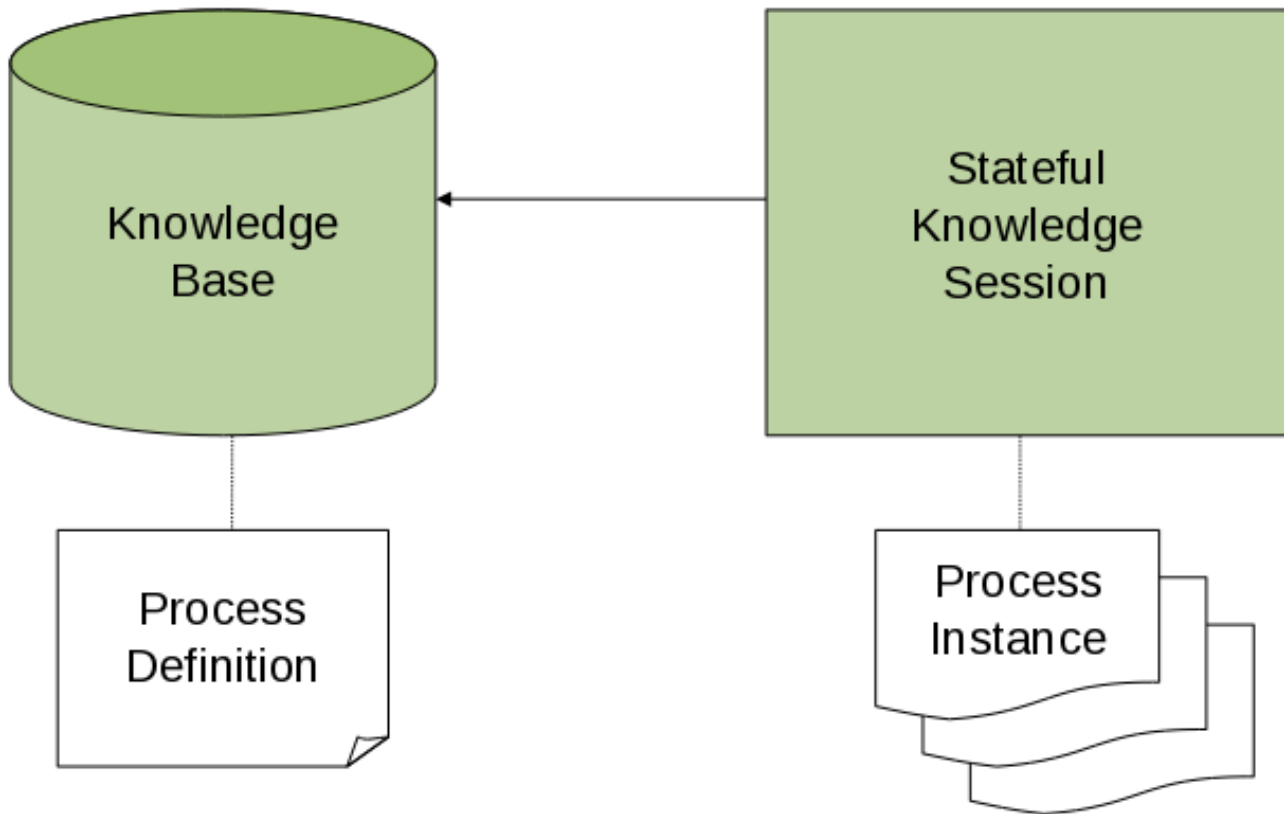


Figure 2.1. Knowledge Base and Knowledge Session

The API is intended for the following:

1. Creating a knowledge base that contains the process definitions
2. Creating a session to start new process instances, signal existing process instances, and register listeners

A knowledge base provides the application's process definitions. Definitions are either contained in the knowledge base or referenced by the knowledge base and loaded from different resources, such as, the classpath and file system.

A session which has access to the knowledge base must be instantiated to communicate with the process engine and execute the processes. Whenever a process is started, a new instance of that process definition is created and is used to maintain the state of the specific instance of the process.

[Report a bug](#)

2.2. CREATE THE KNOWLEDGE BASE

A knowledge base needs to contain all of the process definitions, or references to the process definitions, that the session might need to execute.

Use a knowledge builder to load the processes from the required resources (for example, the classpath or file system) and then create a new knowledge base from the knowledge builder. The following code snippet creates a knowledge base consisting of one process definition using a resource from the classpath.

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(ResourceFactory.newClassPathResource("MyProcess.bpmn"),
ResourceType.BPMN2);
KnowledgeBase kbase = kbuilder.newKnowledgeBase();
```

The **ResourceFactory** has similar methods to load files from file system, URL, InputStream, and Reader.

A knowledge base can be shared across sessions and is usually created once at the start of the application. Knowledge bases can be changed dynamically, allowing processes to be added or removed at runtime.

[Report a bug](#)

2.3. CREATE A SESSION

Sessions are created to interact with the process engine and execute the processes; that is, start new processes and signal events. As many sessions as are required can be started; however, depending on the requirements of the application, a single session, which can be called from multiple places in the application, may be sufficient.

The following code snippet creates a session based on the previously created knowledge base and starts a process with its ID.

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
ProcessInstance processInstance =
ksession.startProcess("com.sample.MyProcess");
```

The **ProcessRuntime** interface defines all the session methods for interacting with processes, as shown in [Example 2.1, "ProcessRuntime Interface"](#).

Example 2.1. ProcessRuntime Interface

```
/**
 * Start a new process instance. The process (definition) that should
 * be used is referenced by the given process id.
 *
 * @param processId The id of the process that should be started
 * @return the ProcessInstance that represents the instance of the
 * process that was started
 */
ProcessInstance startProcess(String processId);

/**
 * Start a new process instance. The process (definition) that should
```

```

    * be used is referenced by the given process id. Parameters can be
    passed
    * to the process instance (as name-value pairs), and these will be set
    * as variables of the process instance.
    *
    * @param processId the id of the process that should be started
    * @param parameters the process variables that should be set when
    starting the process instance
    * @return the ProcessInstance that represents the instance of the
    process that was started
    */
    ProcessInstance startProcess(String processId,
                                Map<String, Object> parameters);

    /**
     * Signals the engine that an event has occurred. The type parameter
     defines
     * which type of event and the event parameter can contain
     additional information
     * related to the event. All process instances that are listening
     to this type
     * of (external) event will be notified. For performance reasons,
     this type of event
     * signaling should only be used if one process instance should be
     able to notify
     * other process instances. For internal event within one process
     instance, use the
     * signalEvent method that also include the processInstanceId of the
     process instance
     * in question.
     *
     * @param type the type of event
     * @param event the data associated with this event
     */
    void signalEvent(String type,
                    Object event);

    /**
     * Signals the process instance that an event has occurred. The type
     parameter defines
     * which type of event and the event parameter can contain
     additional information
     * related to the event. All node instances inside the given
     process instance that
     * are listening to this type of (internal) event will be notified.
     Note that the event
     * will only be processed inside the given process instance. All
     other process instances
     * waiting for this type of event will not be notified.
     *
     * @param type the type of event
     * @param event the data associated with this event
     * @param processInstanceId the id of the process instance that
     should be signaled
     */
    void signalEvent(String type,

```

```

        Object event,
        long processInstanceId);

    /**
     * Returns a collection of currently active process instances.
     Note that only process
     * instances that are currently loaded and active inside the engine
     will be returned.
     * When using persistence, it is likely not all running process
     instances will be loaded
     * as their state will be stored persistently. It is recommended
     not to use this
     * method to collect information about the state of your process
     instances but to use
     * a history log for that purpose.
     *
     * @return a collection of process instances currently active in the
     session
     */
    Collection<ProcessInstance> getProcessInstances();

    /**
     * Returns the process instance with the given id. Note that only
     active process instances
     * will be returned. If a process instance has been completed
     already, this method will return
     * null.
     *
     * @param id the id of the process instance
     * @return the process instance with the given id or null if it
     cannot be found
     */
    ProcessInstance getProcessInstance(long processInstanceId);

    /**
     * Aborts the process instance with the given id. If the process
     instance has been completed
     * (or aborted), or the process instance cannot be found, this
     method will throw an
     * IllegalArgumentException.
     *
     * @param id the id of the process instance
     */
    void abortProcessInstance(long processInstanceId);

    /**
     * Returns the WorkItemManager related to this session. This can be
     used to
     * register new WorkItemHandlers or to complete (or abort)
     WorkItems.
     *
     * @return the WorkItemManager related to this session
     */
    WorkItemManager getWorkItemManager();

```

[Report a bug](#)

2.4. EVENTS LISTENERS

The session provides methods for registering and removing listeners. A **ProcessEventListener** can be used to listen to process-related events, such as starting or completing a process, entering or leaving a node, etc. The different methods of the **ProcessEventListener** class are shown in the following example. An event object provides access to related information, such as the process instance and node instance linked to the event. This API can be used to register event listeners.

Example 2.2. ProcessEventListener Class

```
public interface ProcessEventListener {

    void beforeProcessStarted( ProcessStartedEvent event );
    void afterProcessStarted( ProcessStartedEvent event );
    void beforeProcessCompleted( ProcessCompletedEvent event );
    void afterProcessCompleted( ProcessCompletedEvent event );
    void beforeNodeTriggered( ProcessNodeTriggeredEvent event );
    void afterNodeTriggered( ProcessNodeTriggeredEvent event );
    void beforeNodeLeft( ProcessNodeLeftEvent event );
    void afterNodeLeft( ProcessNodeLeftEvent event );
    void beforeVariableChanged( ProcessVariableChangedEvent event );
    void afterVariableChanged( ProcessVariableChangedEvent event );

}
```

'Before' and 'after' events typically act like a stack, which means that any events that occur as a direct result of the previous event will occur between the 'before' and the 'after' of that event. For example, if a subsequent node is triggered as result of leaving a node, the 'NodeTriggered' events will occur in between the 'beforeNodeLeftEvent' and the 'afterNodeLeftEvent' of the node that is left (as the triggering of the second node is a direct result of leaving the first node). This triggering allows us to derive cause relationships between events more easily. Similarly, all 'NodeTriggered' and 'NodeLeft' events that are the direct result of starting a process will occur between the 'beforeProcessStarted' and 'afterProcessStarted' events. In general, if you just want to be notified when a particular event occurs, you should be looking at the 'before' events only (as they occur immediately before the event actually occurs). When only looking at the 'after' events, one might get the impression that the events are fired in the wrong order; however, this only occurs because the 'after' events are triggered as a stack ('after' events will only fire when all events that were triggered as a result of this event have already fired). 'After' events should only be used if you want to make sure that all processing related to this event has ended; for example, when you want to be notified when the starting of a particular process instance has ended, an 'after' event would be ideal.

Also note that not all nodes always generate 'NodeTriggered' or 'NodeLeft' events. Depending on the type of node, some nodes might only generate 'NodeLeft' events; other events might only generate 'NodeTriggered' events. 'Catching' intermediate events, for example, are not generating 'triggered' events (they are only generating 'left' events as they are not really triggered by another node, rather activated from outside). Similarly, 'throwing' intermediate events do not generate 'left' events (they only generate 'triggered' events, as they have not really left because they have no outgoing connection).

An event listener is provided that can be used to create an audit log (either to the console or a file on the file system). This audit log contains all the events that occurred at runtime. Note that these loggers should only be used for debugging purposes.

The following logger implementations are supported by default:

Table 2.1. Supported Loggers






Type	Description	Required Arguments
Console	Output is written to the console when the logger is closed or the number of events reaches a predefined level.	The knowledge session to be logged
File	Output is written to a file in XML.	The knowledge session to be logged The name of the log file to be created
Threaded File	Output is written to a file after a specified interval; this is useful to visualize the progress in realtime during debugging.	The knowledge session to be logged The name of the log file to be created The interval (in milliseconds) to save the events.

The **KnowledgeRuntimeLoggerFactory** lets you add a logger to your session, as shown in the following example. You should always close the logger at the end of your application.

Example 2.3. KnowledgeRuntimeLogger

```
KnowledgeRuntimeLogger logger =
    KnowledgeRuntimeLoggerFactory.newFileLogger( ksession, "test" );
// add invocations to the process engine here,
// e.g. ksession.startProcess(processId);
...
logger.close();
```

The log file that is created by the file-based loggers contains an XML-based overview of all the events that occurred at runtime. It can be opened in JBoss Developer Studio using the Audit View in the Drools plug-in; through Audit View the events are visualized as a tree. Events that occur between the 'before' and 'after' events are shown as children of that event. The following screenshot shows a simple example where a process is started; this results in the activation of the Start node, an Action node and an End node, after which the process completes.

- ▼  RuleFlow started: ruleflow[com.sample.ruleflow]
 - ▼  RuleFlow node triggered: Start in process ruleflow[com.sample.ruleflow]
 - ▼  RuleFlow node triggered: Hello in process ruleflow[com.sample.ruleflow]
 - ▼  RuleFlow node triggered: End in process ruleflow[com.sample.ruleflow]
 -  RuleFlow completed: ruleflow[com.sample.ruleflow]

[Report a bug](#)

CHAPTER 3. PROCESS OVERVIEW

3.1. PROCESS OVERVIEW

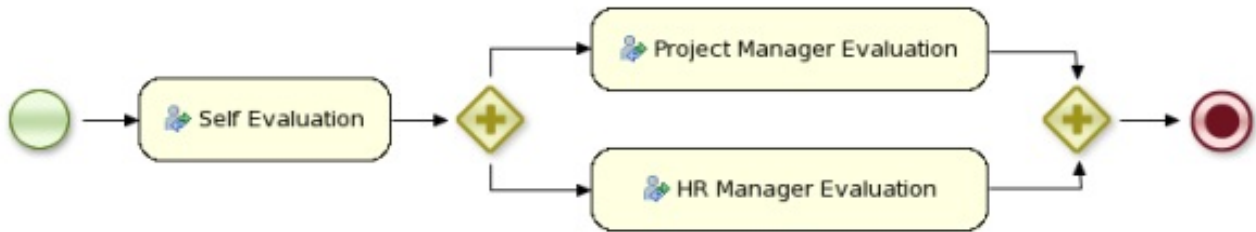


Figure 3.1. A Business Process

A business process is a flow chart that describes the order in which a series of steps need to be executed. A process consists of a collection of nodes that are linked to each other using connections. Each of the nodes represents one step in the overall process and the connections specify how to transition from one node to the next. A large selection of predefined node types have been defined.

Processes can be created with the following methods:

- As an XML file that follows the XML schema defined in the BPMN 2.0 specification.
For details see [Section 4.1, “Business Process Model and Notation \(BPMN\) 2.0 Specification”](#)
- With the graphical web-based designer included in the BRMS user interface.
For details see [Section 5.1, “Process Designer”](#)
- With the graphical process editor in the JBoss Developer Studio plug-in.
For details see [Section 6.1, “JBoss Developer Studio”](#)

[Report a bug](#)

3.2. PROCESS NODES

Executable processes consist of different types of nodes which are connected to each other. The BPMN 2.0 specification defines three main types of nodes:

Events

The start of a process and the end of a process are both types of events. Intermediate events indicate events that could occur during the execution of the process.

Activities

Activities are actions that need to be performed during the execution of the process.

Gateways

Gateways are used to define paths of execution through a process.

[Report a bug](#)

3.3. PROCESS PROPERTIES

Every process has the following properties:

- ID: The unique ID of the process
- Name: The display name of the process
- Version: The version number of the process
- Package: The package (namespace) the process is defined in
- Variables (optional): Variables to store data during the execution of your process
- Swimlanes: Swimlanes used in the process for assigning human tasks

See Also:

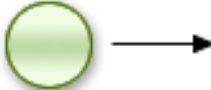
- [Section 10.1, “Human Tasks”](#)




[Report a bug](#)


3.4. EVENTS

Processes include start events, intermediate events, and end events. Every event has an ID, which identifies the node, and a name, which is the display name of the node. Additional properties are listed in the table below.

Table 3.1. Events

Event Type	Additional Properties	Usage
Start Event 		Processes have one start node with one outgoing connection. Execution of a process always starts at the start node.

Event Type	Additional Properties	Usage
<p>End Event</p> 	<ul style="list-style-type: none"> • Terminate: An end event terminates either the entire process or just the current path of execution. 	<p>Processes have one or more end events. Each end event has one incoming connection and no outgoing connections.</p> <p>If the process is terminated, all active nodes (on parallel paths of execution) are cancelled. Non-terminating end events end the current path of execution but allow other paths to continue. Terminating end events are visualized with a full circle inside the event node; non-terminating event nodes are empty. Note that if you use a terminating event node inside a sub-process, you are terminating the top-level process instance, not just that sub-process.</p>
<p>Throwing Error Event</p> 	<ul style="list-style-type: none"> • FaultName: Provides the name of the fault, which is used to search for appropriate exception handlers that are capable of handling this kind of fault. • FaultVariable: Provides the name of the variable where the data associated with this fault is stored. This data is also passed on to the exception handler (if one is found). 	<p>Error events are used to signal an exceptional condition in the process. It should have one incoming connection and no outgoing connections. When an Error Event is reached in the process, it will throw an error with the given name. The process will search for an appropriate error handler that is capable of handling this kind of fault. If no error handler is found, the process instance will be aborted.</p> <p>Error handlers can be specified using boundary events when working with XML.</p>
<p>Catching Timer Event</p> 	<ul style="list-style-type: none"> • Timer period: The period between two subsequent triggers. If the period is 0, the timer should only be triggered once. 	<p>Catching timer events represent a timer that can trigger one or multiple times after a given period of time. A Timer Event should have one incoming connection and one outgoing connection. When a Timer Event is reached in the process, it will start the associated timer.</p> <p>See Section 3.9, "Timers" for expression syntax and further details.</p>

Event Type	Additional Properties	Usage
<p>Catching Signal Event</p> 	<ul style="list-style-type: none"> • EventType: The type of event that is expected. • VariableName: The name of the variable where the data associated with this event will be stored. 	<p>A Signal Event can be used to respond to internal or external events during the execution of the process. A Signal Event should have no incoming connections and one outgoing connection. It specifies the type of event that is expected. Whenever that type of event is detected, the node connected to this event node will be triggered.</p> <p>A process instance can be signaled that a specific event occurred using:</p> <pre>ksession.signalEvent(eventType, data, processInstanceId)</pre> <p>This triggers all (active) signal event nodes in the process instance that are waiting for that event type. Data related to the event can be passed using the data parameter. If the event node specifies a variable name, this data will be copied to that variable when the event occurs.</p> <p>Events can be used inside sub-processes; however, these event nodes will only be active when the sub-process is active.</p> <p>A signal can be generated from inside a process instance with a script, for instance:</p> <pre>kcontext.getKnowledgeRuntime().signalEvent(eventType, data, kcontext.getProcessInstance().getId());</pre>

In addition to ensuring all of the process tasks are executed in the correct order, the process engine can be instructed to respond to events that occur outside the process. By explicitly representing events that occur outside the process, the process author can specify how the process should react to the events.

Events have a type and can have data associated with them, and users can define their own event types and associated data.

A process can specify how to respond to events by using a message event. An event node needs to specify the type of event the node is interested in. It can also define the name of a variable, which will receive the data that is associated with the event. This allows subsequent nodes in the process to

access the event data and take appropriate action based on this data.

An event can be signaled to a running instance of a process in a number of ways:

- Internal event: Any action inside a process (e.g., the action of an action node or an on-entry or on-exit action of some node) can signal the occurrence of an internal event to the surrounding process instance. Example code of an internal event is demonstrated below.

```
kcontext.getProcessInstance().signalEvent(type, eventData);
```

- External event: A process instance can be notified of an event from outside using code such as the following:

```
processInstance.signalEvent(type, eventData);
```

- External event using event correlation: Instead of notifying a process instance directly, it is possible to have the engine automatically determine which process instances might be interested in an event using event correlation, which is based on the event type. A process instance that contains an event node listening to external events of some type is notified whenever such an event occurs. To signal such an event to the process engine, use the following code:

```
ksession.signalEvent(type, eventData);
```

[Report a bug](#)

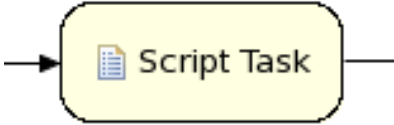

3.5. ACTIVITIES

Activities are actions that need to be performed during the execution of the process. Each activity has one incoming connection and one outgoing connection.


Every activity has an ID, which identifies the node, and a name, which is the display name of the node. Additional properties are listed in the table below.


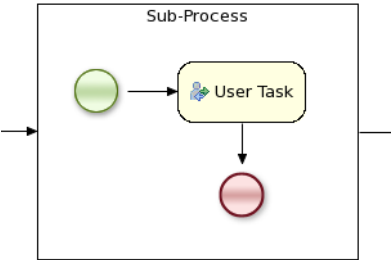
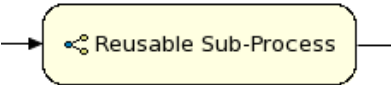
Table 3.2. Activities

Activity Type	Additional Properties	Usage
---------------	-----------------------	-------

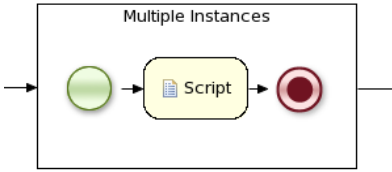
Activity Type	Additional Properties	Usage
<p>Script Tasks</p> 	<ul style="list-style-type: none"> • Action: The action script associated with the node. 	<p>The Script Task represents a script that should be executed in this process. The associated action specifies what should be executed, the dialect used for coding the action (i.e., Java or MVEL), and the actual action code. This code can access any variables and globals. There is also a predefined variable 'kcontext' that references the 'ProcessContext' object (which can be used to access the current ProcessInstance or NodeInstance; this object can also be used to get and set variables, or for it to get access to the ksession using:</p> <pre>kcontext.getKnowledgeRuntime()</pre> <p>When a Script Task is reached in the process, it will execute the action and then continue with the next node.</p>
<p>User Task</p> 	<ul style="list-style-type: none"> • TaskName: The name of the human task. • Priority: An integer indicating the priority of the human task. • Comment: A comment associated with the human task. • ActorId: The actor ID responsible for executing the human task. A list of actor ID's can be specified using a comma (',') as a separator. • GroupId: The group ID responsible 	<p>Processes can involve tasks that need to be executed by human actors. A user task represents an atomic task to be completed by a human actor. User tasks can be used in combination with swim lanes to assign multiple human tasks to similar actors. For more information about swim lanes and human tasks, see Human Tasks chapter.</p>

Activity Type	Additional Properties	Usage
	<p>for executing the human task. Actor group ID's can be specified using a comma (',') as a separator.</p> <ul style="list-style-type: none"> • Skippable: Specifies whether the human task is optional. • Content: The data associated with this task. • Swimlane: The swimlane this human task node is part of. • On entry and on exit actions: Action scripts that are executed upon entry and exit of this node. • Parameter mapping: Copies the value of process variables to the human task parameters. The values are copied when the human task is created. • Result mapping: Copies the result parameters from a human task to a process variable. A human task with a variable called result will contain the data return by the human actor; the result variable ActorId will contain the ID of the actor who completed the task. 	

Activity Type	Additional Properties	Usage
<p>Service Tasks</p> 	<ul style="list-style-type: none"> • Parameter mapping: Copies the value of process variables to the work item parameters. The values are copied when the work item is created. • Result mapping: Copies the result parameters from work items to a process variable when the work item has completed. For example, the FileFinder work item has the parameter Files, which returns a list of files that match a search criteria. This list of files can then be bound to a process variable for use within the process. • On-entry and on-exit actions: Actions that are executed upon entry or exit of the node. • Additional parameters: Each type of work item can define additional parameters that are relevant for that type of work item. The user can either provide these values directly or define a parameter mapping. If both methods are used, the parameter mapping will override parameters provided by the user. The value will be retrieved when creating the work item, and the substitution expression will be replaced by the result of calling toString() on the variable. The expression could be the name of a variable (so that it resolves to the value of the variable) but more advanced MVEL 	<p>Service Tasks represent an abstract unit of work that should be executed in this process. All work that is executed outside the process engine should be represented (in a declarative way) using a Service Task. Different types of services are predefined, e.g., sending an email, logging a message, etc. however, users can define domain-specific services or work items. For further details, see Domain-Specific Processes chapter.</p>

Activity Type	Additional Properties	Usage
<p>Business Rule Task</p> 	<p>expressions are possible {person.name.firstname},</p> <ul style="list-style-type: none"> RuleFlowGroup <p>The name of the ruleflow group that represents the set of rules of this RuleFlowGroup node.</p>	<p>Business Rule Task represents a set of rules that need to be evaluated. Rules are defined in separate files using the Drools rule format. Rules can become part of a specific ruleflow group using the ruleflow-group attribute in the header of the rule. When a Rule Task is reached in the process, the engine will start executing rules that are part of the corresponding ruleflow-group (if any). Execution will automatically continue to the next node if there are no more active rules in this ruleflow group. This means that during the execution of a ruleflow group, it is possible that new activations belonging to the currently active ruleflow group are added to the Agenda; this occurs because of changes made to the facts by the other rules. If the ruleflow group was already active, the ruleflow group will remain active and execution will only continue if all active rules of the ruleflow group have been completed.</p>
<p>Embedded Sub-Process</p> 	<ul style="list-style-type: none"> VariableID <p>Unlike other variables which have a variable ID which is derived from variable_name, embedded sub-process have a variable ID in the form:</p> <pre>subprocess_node_id:variable_name</pre> <ul style="list-style-type: none"> Variables <p>Additional variables can be defined to store data during the execution of this node. For more details see Section 3.7, "Data"</p>	<p>A sub-process is a node that can contain other nodes so that it acts as a node container. This allows not only the embedding of a part of the process within such a sub-process node, but also the definition of additional variables that are accessible for all nodes inside this container. A sub-process needs a start event and one or more end events.</p>
<p>Reusable Sub-process</p> 	<ul style="list-style-type: none"> ProcessId: <p>The ID of the process that should be executed.</p>	<p>Reusable sub-processes represent the invocation of another process from within a process. When a reusable sub-</p>

Activity Type	Additional Properties	Usage
	<ul style="list-style-type: none"> • Wait for Completion: If false, the process will continue as soon as the sub-process has been started. If true, it will wait for the sub-process to finish (complete or abort). • Independent: If true, the child process is started as an independent process and will not be terminated if the parent process completes. Independent can only be set to false when Wait for Completion is set to true. If set to false, the sub-process will be canceled on termination of the parent process. • On-entry and on-exit actions: Actions that are executed upon entry or exit of this node. • Parameter in/out mapping: A sub-process node can define "in" and "out" mappings for variables. The variables given in the "in" mapping will be used as parameters (with the associated parameter name) when starting the process. The variables of the child process that are defined for the "out" mappings will be copied to the variables of this process when the child process has been completed. Note that "out" mappings can only be used when "Wait for completion" is set to true. 	<p>process node is reached in the process, the engine will start the process with the given ID.</p>

Activity Type	Additional Properties	Usage
<p>Multi-Instance Sub-Process</p> 	<ul style="list-style-type: none"> • CollectionExpression: A collection of elements either in the form of an array or the type <code>java.util.Collection</code>. • VariableName The name of the variable to contain the current element from the collection. This gives nodes within the composite node access to the selected element. 	<p>A Multiple Instance sub-process is a special kind of sub-process that allows you to execute the contained process segment multiple times, once for each element in a collection. It waits until the embedded process fragment is completed for each of the elements in the given collection before continuing. If the collection expression evaluates to null or an empty collection, the multiple instances sub-process will be completed immediately and follow its outgoing connection.</p>

Any valid Java code can be used inside a script node. Some points to consider when writing code for a script node:

- Avoid low level implementation details inside the process when defining high-level business processes that need to be understood by business users. A Script Task could be used to manipulate variables, but other concepts like a service task could be used to model more complex behavior in a higher-level manner.
- Scripts should be immediate as they use the engine thread to execute the script. Scripts that take some time to execute should be modeled as an asynchronous Service Task.
- Avoid contacting external services through a script node; instead, model communication with an external service using a service task.
- Scripts should not throw exceptions. Runtime exceptions should be caught and managed inside the script or transformed into signals or errors that can then be handled inside the process.

kcontext variable

This variable is of type `org.drools.runtime.process.ProcessContext` and can be used for several tasks:

- Getting the current node instance (if applicable). The node instance could be queried for data, such as its name and type. You can also cancel the current node instance.

```
NodeInstance node = kcontext.getNodeInstance();
String name = node.getNodeName();
```

- Getting the current process instance. A process instance can be queried for data (name, id, processId, etc.), aborted or signaled an internal event.

```
ksession.signalEvent(eventType, data, processInstanceId)
```

- Getting or setting the value of variables.

- Accessing the Knowledge Runtime allows you do things like starting a process, signaling (external) events, inserting data, etc.

Dialects

Both Java and MVEL can be used. Java actions should be valid Java code. MVEL actions can use the business scripting language MVEL to express the action. MVEL accepts any valid Java code but additionally provides support for nested accesses of parameters (e.g., `person.name` instead of `person.getName()`), and it provides many other scripting improvements. Thus, MVEL expressions are more convenient for the business user. For example, an action that prints out the name of the person in the "requester" variable of the process would look like this:

```
// Java dialect
System.out.println( person.getName() );

// MVEL dialect
System.out.println( person.name );
```

See Also:

- [Section 10.1, "Human Tasks"](#)
- [Section 9.1, "Domain-Specific Service Nodes"](#)

[Report a bug](#)

3.6. GATEWAYS

Gateways make it possible to provide multiple paths of execution through a process. There are two types of gateways: diverging gateways and converging gateways. Diverging gateways create branches in processes, and converging gateways close the branches by merging them back together.

It is possible to branch processes in the following ways:

- AND (Parallel)

The flow of control continues along all outgoing connections (branches) simultaneously.

- XOR (Exclusive)

The flow of control continues along only one of the outgoing connections. The connection with the constraint with the lowest priority number that evaluates to true is selected. For information about defining constraints see [Section 3.8, "Constraints"](#). At least one of the outgoing connections must evaluate to true at runtime, or the process will throw an exception at runtime.

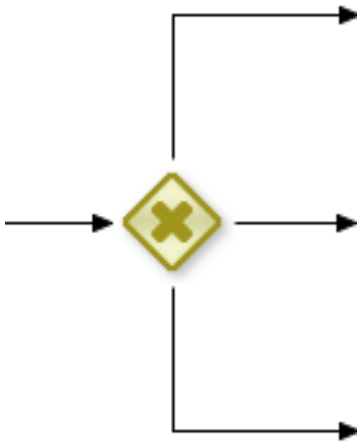


Figure 3.2. Diverging Gateway

A diverging gateway has one incoming connection and two or more outgoing connections. A diverging gateway contains the following properties:

- ID: The ID of the node (which is unique within one node container).
- Name: The display name of the node.
- Type: AND or XOR.
- Constraints: The constraints linked to each of the outgoing connections (exclusive and inclusive gateways).

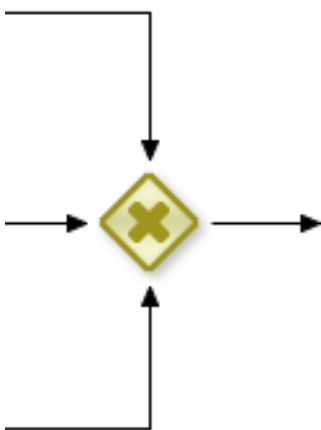


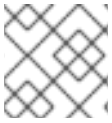
Figure 3.3. Converging Gateway

A converging gateway has two or more incoming connections and one outgoing connection. A converging gateway contains the following properties:

- ID: The ID of the node (which is unique within one node container).
- Name: The display name of the node.
- Type: AND or XOR.

Depending on the type of diverging gateway used, the converging gateway will behave in different ways:

- When a parallel (AND) split was used, the converging gateway will wait for all incoming branches to be completed before continuing.
- When an exclusive (XOR) split was used, the converging gateway will continue as soon as the one incoming branch has completed.



NOTE

Inclusive (OR) gateways are not currently supported.

[Report a bug](#)

3.7. DATA

Throughout the execution of a process, data can be retrieved, stored, passed on, and used. To store runtime data during the execution of the process, process variables are used. A variable is defined with a name and a data type. A basic data type could include the following: boolean, int, String, or any kind of object subclass.

Variables can be defined inside a variable scope. The top-level scope is the variable scope of the process itself. Sub-scopes can be defined using a sub-process. Variables that are defined in a sub-scope are only accessible for nodes within that scope.

Whenever a variable is accessed, the process will search for the appropriate variable scope that defines the variable. Nesting variable scopes are allowed. A node will always search for a variable in its parent container; if the variable cannot be found, the node will look in the parent's parent container, and so on, until the process instance itself is reached. If the variable cannot be found, a read access yields null, and a write access produces an error message. All of this occurs with the process continuing execution.

Variables can be used in the following ways:

- Process-level variables can be set when starting a process by providing a map of parameters to the invocation of the `startProcess` method. These parameters will be set as variables on the process scope.
- Script actions can access variables directly simply by using the name of the variable as a local parameter in their script. For example, if the process defines a variable of type "org.jbpm.Person" in the process, a script in the process could access this directly:

```
// call method on the process variable "person"
person.setAge(10);
```

Changing the value of a variable in a script can be done through the knowledge context:

```
kcontext.setVariable(variableName, value);
```

- Service tasks (and reusable sub-processes) can pass the value of process variables to the outside world (or another process instance) by mapping the variable to an outgoing parameter. For example, the parameter mapping of a service task could define that the value of the process variable `x` should be mapped to a task parameter `y` just before the service is invoked. You can also inject the value of the process variable into a hard-coded parameter String using `# {expression}`. For example, the description of a human task could be defined as the following:

```
You need to contact person #{person.getName() }
```

Where person is a process variable. This will replace this expression with the actual name of the person when the service needs to be invoked. Similar results of a service (or reusable sub-process) can also be copied back to a variable using result mapping.

- Various other nodes can also access data. Event nodes, for example, can store the data associated to the event in a variable. Check the properties of the different node types for more information.

Finally, processes (and rules) have access to globals, i.e., globally defined variables and data in the Knowledge Session. Globals are directly accessible in actions like variables. Globals need to be defined as part of the process before they can be used. Globals can be set using the following:

```
ksession.setGlobal(name, value)
```

Globals can also be set from inside process scripts using:

```
kcontext.getKnowledgeRuntime().setGlobal(name, value);
```

[Report a bug](#)

3.8. CONSTRAINTS

There are two types of constraints in business processes: code constraints and rule constraints.

- Code constraints are boolean expressions evaluated directly whenever they are reached; these constraints are written in either Java or MVEL. Both Java and MVEL code constraints have direct access to the globals and variables defined in the process.

Here is an example of a valid Java code constraint, person being a variable in the process:

```
return person.getAge() > 20;
```

Here is an example of a valid MVEL code constraint, person being a variable in the process:

```
return person.age > 20;
```

- Rule constraints are equal to normal Drools rule conditions. They use the Drools Rule Language syntax to express complex constraints. These rules can, like any other rule, refer to data in the working memory. They can also refer to globals directly. Here is an example of a valid rule constraint:

```
Person( age > 20 )
```

This tests for a person older than 20 in the working memory.

Rule constraints do not have direct access to variables defined inside the process. However, it is possible to refer to the current process instance inside a rule constraint by adding the process instance to the working memory and matching for the process instance in your rule constraint. Logic is included to make sure that a variable processInstance of type WorkflowProcessInstance will only match the current process instance and not other process instances in the working memory. Note, it is necessary to insert

the process instance into the session. If it is necessary to update the process instance, use Java code or an on-entry, on-exit, or explicit action in the process. The following example of a rule constraint will search for a person with the same name as the value stored in the variable *name* of the process:

```
processInstance : WorkflowProcessInstance()
Person( name == ( processInstance.getVariable("name") ) )
# add more constraints here ...
```

[Report a bug](#)

3.9. TIMERS

A timer node has a delay and a period. The delay specifies the amount of time to wait after node activation before triggering the timer the first time. The period defines the time between subsequent trigger activations; a period of 0 results in a timer that does not repeat.

The syntax for defining the delay and period is the following:

```
[#d][#h][#m][#s][#[ms]]
```

With this syntax, it is possible to specify the amount of days, hours, minutes, seconds, and milliseconds (which is the default if nothing is specified). For example, the expression "1h" will wait one hour before triggering the timer.

The expression could also use `#{expr}` to dynamically derive the period based on a process variable or a more complex expression based on a process variable (e.g. `myVariable.getValue()`).

The timer service is responsible for making sure that timers get triggered at the correct time. Timers can also be cancelled, which means the timer will no longer be triggered.

A timer event may be added to the process flow. Its activation starts the timer, and when it triggers, once or repeatedly, it activates the timer node's successor. This means that the outgoing connection of a timer with a positive period is triggered multiple times. Canceling a timer node also cancels the associated timer, after which no more triggers will occur.

[Report a bug](#)

3.10. UPDATING PROCESSES

Over time, the requirements for a process might change. In order to update a process, it is necessary to deploy an updated version of the process. The updated process must have an updated version number to distinguish it from the old process; this is necessary because the old process may still be required by an existing process instance.

Whenever a process is updated, it is important to determine what should happen to the already running process instances. There are three possibilities:

- **Proceed:** The running process instance proceeds as normal, following the process definition as it was defined when the process instance was started. As a result, the already running instance will proceed as if the process was never updated. New instances can be started using the updated process.

- Abort (and restart): The already running instance is aborted. If necessary, the process instance can be restarted using the new process definition.
- Transfer: The process instance is migrated to the new process definition; that is, once it has been migrated successfully, it will continue executing based on the updated process logic. For further details see [Section 3.11, "Process Instance Migration"](#).

The default behavior follows the proceed approach, which results in multiple versions of the same process being deployed. Existing process instances continue executing based on the process definition that was used when starting the process instance.

The version number should be tracked to determine which version of a process definition a process instance is using. The current version of the process can be retrieved:

```
processInstance.getProcess().getVersion()
```

[Report a bug](#)

3.11. PROCESS INSTANCE MIGRATION

A process instance contains all the runtime information needed to continue execution at some later point in time. This includes all the data linked to this process instance (such as variables); it also includes the current state in the process diagram. For each node that is currently active, a node instance contains the state of the node. This node instance can also contain additional state information linked to the execution of that specific node. There are different types of node instances, one for each type of node.

A process instance only contains the runtime state and is linked to a particular process (indirectly, using ID references) that represents the process logic that needs to be followed when executing this process instance. The clear separation of definition and runtime state allows reuse of the definition across all process instances based on this process, and it minimizes runtime state. As a result, updating a running process instance to a newer version so it uses the new process logic instead of the old one is simply a matter of changing the referenced process ID from the old ID to the new ID.

However, this does not include the state of the process instance (the variable instances and the node instances) which might need to be migrated as well. In cases where the process is only extended and all existing wait states are kept, the runtime state of the process instance does not need to change at all. However, it is also possible that a more sophisticated mapping is necessary. For example, when an existing wait state is removed or split into multiple wait states, an existing process instance in a wait state cannot simply be updated. When a new process variable is introduced, the variable might need to be initiated correctly so it can be used in the remainder of the (updated) process.

The `WorkflowProcessInstanceUpgrader` can be used to upgrade a workflow process instance to a newer process instance. The process instance must be provided with the new process ID. By default, the old node instances will be automatically mapped to the new node instances with the same ID. But you can provide a mapping of the old (unique) node ID to the new node ID. The unique node ID is the node ID, preceded by the node IDs of its parents (with a colon in-between), to uniquely identify a node when composite nodes are used (as a node ID is only unique within its node container). The new node ID is just the new node ID in the node container with no need to reference unique node IDs. The following code snippet shows an example.

```
// create the session and start the process "com.sample.process"  
KnowledgeBuilder kbuilder = ...  
StatefulKnowledgeSession ksession = ...  
ProcessInstance processInstance =
```



```

ksession.startProcess("com.sample.process");

// add a new version of the process "com.sample.process2"
kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(..., ResourceType.BPMN2);
kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());

// migrate process instance to new version
Map<String, Long> mapping = new HashMap<String, Long>();
// top level node 2 is mapped to a new node with id 3
mapping.put("2", 3L);
// node 2, which is part of composite node 5, is mapped to a new node with
id 4
mapping.put("5.2", 4L);
WorkflowProcessInstanceUpgrader.upgradeProcessInstance(
    ksession, processInstance.getId(),
    "com.sample.process2", mapping);

```

If this kind of mapping is insufficient, you can describe your own custom mappers for specific situations. First disconnect the process instance, change the state accordingly, and then reconnect the process instance. This is similar to how the `WorkflowProcessInstanceUpgrader` does it.

[Report a bug](#)

3.12. MULTI-THREADING

3.12.1. Multi-threading

In the following text, we will refer to two types of "multi-threading": *logical* and *technical*. *Technical multi-threading* is what happens when multiple threads or processes are started on a computer, for example by a Java or C program. *Logical multi-threading* is what we see in a BPM process after the process reaches a parallel gateway. From a functional standpoint, the original process will then split into two processes that are executed in a parallel fashion.

The BPM engine supports logical multi-threading; for example, processes that include a parallel gateway are supported. We've chosen to implement logical multi-threading using one thread; accordingly, a BPM process that includes logical multi-threading will only be executed in one technical thread. The main reason for doing this is that multiple (technical) threads need to be able to communicate state information with each other if they are working on the same process. This requirement brings with it a number of complications. While it might seem that multi-threading would bring performance benefits with it, the extra logic needed to make sure the different threads work together well means that this is not guaranteed. There is also the extra overhead incurred because we need to avoid race conditions and deadlocks.

[Report a bug](#)

3.12.2. Engine Execution

In general, the BPM engine executes actions in serial. For example, when the engine encounters a script task in a process, it will synchronously execute that script and wait for it to complete before continuing execution. Similarly, if a process encounters a parallel gateway, it will sequentially trigger each of the outgoing branches, one after the other. This is possible since execution is almost always instantaneous,

meaning that it is extremely fast and produces almost no overhead. As a result, the user will usually not even notice this. Similarly, action scripts in a process are also synchronously executed, and the engine will wait for them to finish before continuing the process. For example, doing a `Thread.sleep(...)` as part of a script will not make the engine continue execution elsewhere but will block the engine thread during that period.

The same principle applies to service tasks. When a service task is reached in a process, the engine will also invoke the handler of this service synchronously. The engine will wait for the `completeWorkItem(...)` method to return before continuing execution. It is important that your service handler executes your service asynchronously if its execution is not instantaneous.

An example of this would be a service task that invokes an external service. Since the delay in invoking this service remotely and waiting for the results might be too long, it might be a good idea to invoke this service asynchronously. This means that the handler will only invoke the service and will notify the engine later when the results are available. In the mean time, the process engine then continues execution of the process.

Human tasks are a typical example of a service that needs to be invoked asynchronously, as we don't want the engine to wait until a human actor has responded to the request. The human task handler will only create a new task (on the task list of the assigned actor) when the human task node is triggered. The engine will then be able to continue execution on the rest of the process (if necessary), and the handler will notify the engine asynchronously when the user has completed the task.

[Report a bug](#)

3.13. ASYNCHRONOUS HANDLERS

To implement an asynchronous service handler using simple Java, execute the actual service in a new thread:

```
public class MyServiceTaskHandler implements WorkItemHandler {
    public void executeWorkItem(WorkItem workItem, WorkItemManager manager)
    {
        new Thread(new Runnable() {
            public void run() {
                // Do the heavy lifting here ...
            }
        }).start();
    }

    public void abortWorkItem(WorkItem workItem, WorkItemManager manager) {
    }
}
```

In general, a handler usually will not implement the business logic to perform the work item but will contact an existing service to do the work. For example, the human task handler simply invokes the human task service to add a task there. To implement an asynchronous handler, you usually have to do an asynchronous invocation of this service. This usually depends on the technology you use to do the communication, but this might be as simple as asynchronously invoking a web service, sending a JMS message to the external service, etc.

[Report a bug](#)

CHAPTER 4. BPMN 2.0 NOTATION

4.1. BUSINESS PROCESS MODEL AND NOTATION (BPMN) 2.0 SPECIFICATION

The Business Process Model and Notation (BPMN) 2.0 specification defines a standard for graphically representing a business process; it includes execution semantics for the defined elements and an XML format to store and share process definitions.

The table below shows the supported elements of the BPMN 2.0 specification and includes some additional elements and attributes.

Table 4.1. BPMN 2.0 Supported Elements and Attributes

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
definitions		rootElement BPMNDiagram		
process	processType isExecutable name id	property laneSet flowElement	packageName adHoc version	import global
sequenceFlow	sourceRef targetRef isImmediate name id	conditionExpressio n	priority	
interface	name id	operation		
operation	name id	inMessageRef		
laneSet		lane		
lane	name id	flowNodeRef		
import*		name		
global*		identifier type		
Events				
startEvent	name id	dataOutput dataOutputAssocia tion outputSet eventDefinition	x y width height	

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
endEvent	name id	dataInput dataInputAssociati on inputSet eventDefinition	x y width height	
intermediateCatch Event	name id	dataOutput dataOutputAssocia tion outputSet eventDefinition	x y width height	
intermediateThrow Event	name id	dataInput dataInputAssociati on inputSet eventDefinition	x y width height	
boundaryEvent	cancelActivity attachedToRef name id	eventDefinition	x y width height	
terminateEventDef inition				
compensateEvent Definition	activityRef	documentation extensionElements		
conditionalEventD efinition		condition		
errorEventDefinitio n	errorRef			
error	errorCode id			
escalationEventDe finition	escalationRef			
escalation	escalationCode id			
messageEventDefi nition	messageRef			
message	itemRef id			
signalEventDefiniti on	signalRef			

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
timerEventDefinition		timeCycle timeDuration		
Activities				
task	name id	ioSpecification dataInputAssociation dataOutputAssociation	taskName x y width height	
scriptTask	scriptFormat name id	script	x y width height	
script		text[mixed content]		
userTask	name id	ioSpecification dataInputAssociation dataOutputAssociation resourceRole	x y width height	onEntry-script onExit-script
potentialOwner		resourceAssignmentExpression		
resourceAssignmentExpression		expression		
businessRuleTask	name id		x y width height ruleFlowGroup	onEntry-script onExit-script
manualTask	name id		x y width height	onEntry-script onExit-script
sendTask	messageRef name id	ioSpecification dataInputAssociation	x y width height	onEntry-script onExit-script
receiveTask	messageRef name id	ioSpecification dataOutputAssociation	x y width height	onEntry-script onExit-script
serviceTask	operationRef name id	ioSpecification dataInputAssociation dataOutputAssociation	x y width height	onEntry-script onExit-script

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
subProcess	name id	flowElement property loopCharacteristics	x y width height	
adHocSubProcess	cancelRemainingI nstances name id	completionConditio n flowElement property	x y width height	
callActivity	calledElement name id	ioSpecification dataInputAssociati on dataOutputAssocia tion	x y width height waitForCompletion independent	onEntry-script onExit-script
multiInstanceLoop Characteristics		loopDataInputRef inputDataItem		
onEntry-script*	scriptFormat		script	
onExit-script*	scriptFormat		script	
Gateways				
parallelGateway	gatewayDirection name id		x y width height	
eventBasedGatew ay	gatewayDirection name id		x y width height	
exclusiveGateway	default gatewayDirection name id		x y width height	
inclusiveGateway	default gatewayDirection name id		x y width height	
Data				
property	itemSubjectRef id			
dataObject	itemSubjectRef id			
itemDefinition	structureRef id			

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
ioSpecification		dataInput dataOutput inputSet outputSet		
dataInput	name id			
dataInputAssociation		sourceRef targetRef assignment		
dataOutput	name id			
dataOutputAssociation		sourceRef targetRef assignment		
inputSet		dataInputRefs		
outputSet		dataOutputRefs		
assignment		from to		
formalExpression	language	text[mixed content]		
BPMNDI				
BPMNDiagram		BPMNPlane		
BPMNPlane	bpmnElement	BPMNEdge BPMNShape		
BPMNShape	bpmnElement	Bounds		
BPMNEdge	bpmnElement	waypoint		
Bounds	x y width height			
waypoint	x y			

[Report a bug](#)

4.2. AN EXAMPLE BPMN 2.0 PROCESS

The process XML file consists of two parts. The top part (the "process" element) contains the definition of

the different nodes and their properties; the lower part (the "BPMNDiagram" element) contains all graphical information, like the location of the nodes. The process XML consist of exactly one <process> element. This element contains parameters related to the process (its type, name, id and package name), and it consists of three subsections: a header section (where process-level information like variables, globals, imports and lanes can be defined), a nodes section that defines each of the nodes in the process, and a connections section that contains the connections between all the nodes in the process. In the nodes section, there is a specific element for each node. Each element defines the various parameters and, possibly, sub-elements for that node type.

A simple hello world process that prints out the phrase 'Hello World' is represented graphically in the following image.

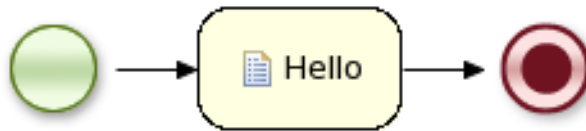


Figure 4.1. Graphical Hello World Process

An executable version of this process expressed using BPMN 2.0 XML is provided in the following example.

Example 4.1. BPMN 2.0 XML Hello World Process

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"
  targetNamespace="http://www.example.org/MinimalExample"
  typeLanguage="http://www.java.com/javaTypes"
  expressionLanguage="http://www.mvel.org/2.0"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"

  xs:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
  BPMN20.xsd"

  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
  xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
  xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
  xmlns:tns="http://www.jboss.org/drools">

  <process processType="Private" isExecutable="true"
  id="com.sample.HelloWorld" name="Hello World" >

    <!-- nodes -->
    <startEvent id="_1" name="StartProcess" />
    <scriptTask id="_2" name="Hello" >
      <script>System.out.println("Hello World");</script>
    </scriptTask>
    <endEvent id="_3" name="EndProcess" >
      <terminateEventDefinition/>
    </endEvent>

    <!-- connections -->
    <sequenceFlow id="_1-_2" sourceRef="_1" targetRef="_2" />
    <sequenceFlow id="_2-_3" sourceRef="_2" targetRef="_3" />

  </process>
  
```

```
<bpmndi:BPMNDiagram>
  <bpmndi:BPMNPlane bpmnElement="Minimal" >
    <bpmndi:BPMNShape bpmnElement="_1" >
      <dc:Bounds x="15" y="91" width="48" height="48" />
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="_2" >
      <dc:Bounds x="95" y="88" width="83" height="48" />
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="_3" >
      <dc:Bounds x="258" y="86" width="48" height="48" />
    </bpmndi:BPMNShape>
    <bpmndi:BPMNEdge bpmnElement="_1-_2" >
      <di:waypoint x="39" y="115" />
      <di:waypoint x="75" y="46" />
      <di:waypoint x="136" y="112" />
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge bpmnElement="_2-_3" >
      <di:waypoint x="136" y="112" />
      <di:waypoint x="240" y="240" />
      <di:waypoint x="282" y="110" />
    </bpmndi:BPMNEdge>
  </bpmndi:BPMNPlane>
</bpmndi:BPMNDiagram>

</definitions>
```

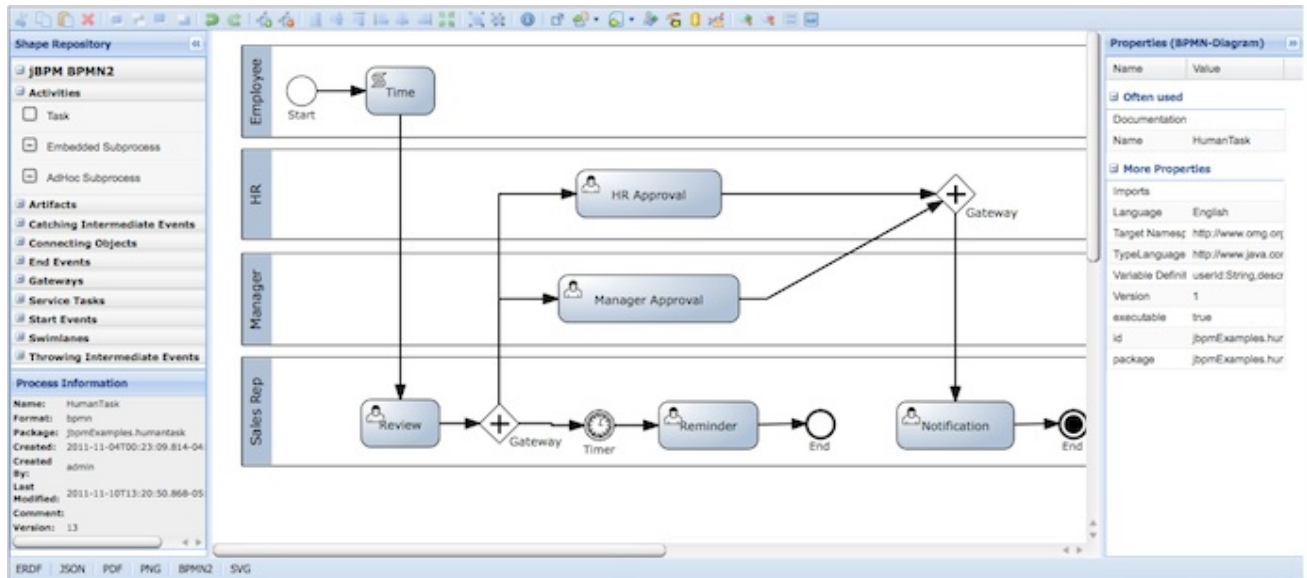
Processes can be defined with the BPMN 2.0 specification by using the designer included in the BRMS user interface (for more details see [Section 5.1, "Process Designer"](#)), in JBoss Developer Studio (for more details see [Section 6.1, "JBoss Developer Studio"](#)), or by manually writing the XML directly.

[Report a bug](#)

CHAPTER 5. PROCESS DESIGNER

5.1. PROCESS DESIGNER

The Process Designer is accessible through the BRMS user interface, and it can be used to create, view, edit, or update processes with a drag and drop interface.



[Report a bug](#)

5.2. CONFIGURING THE PROCESS DESIGNER

The process designer is a separate application that has been tightly integrated with the JBoss BRMS user interface and repository, and any configuration changes that are made to JBoss BRMS must also be applied to the process designer. The process designer is configured with the `jbpm.xml` file, which is located in the `jboss-as/server/production/deploy/designer.war/profiles/` directory.

The configuration attributes include:

- protocol: the protocol to use (http/https)
- host: localhost:8080
- subdomain: jboss-brms/org.drools.guvnor.Guvnor/oryxeditor
- usr: admin (default, must be a valid JBoss BRMS user)
- pwd: admin (default, must correspond to a valid JBoss BRMS user password)



NOTE

If the host and port are changed, the information also needs to be updated in `production/deploy/jboss-brms.war/WEB-INF/classes/preferences.properties`.

[Report a bug](#)

5.3. PROCESS CREATION AND VALIDATION

The following procedure explains how to create a simple process with the Process Designer in the BRMS user interface. Users must be logged on to the BRMS user interface to use the Process Designer.

Procedure 5.1. Create a Process

1. From the navigation panel select **Knowledge Bases** → **Create New** → **New BPMN2 Process**.
2. Enter a name for the process in the **Name**: dialogue box.
3. Select which package to create the process in from the drop down menu and provide a description.



NOTE

At this stage, Red Hat advises against creating processes in the global area as not all of the processes attributes can be successfully imported into other packages.

4. Processes are created by dragging and dropping the process elements from the shape repository panel on the left of the screen onto the canvas in the center of the screen.

Select the node to start the process with from the shape repository panel and drag it onto the canvas.

Name the node by clicking the value column next to **Name** in the properties panel on the right side of the Process Designer.
5. Add the next node in the process by selecting the required node type from the shape repository panel and dragging it onto the canvas.

Edit the node's properties in the properties panel.
6. Connect the nodes by clicking the first node; then by selecting and holding the arrow icon, drag it to the second node.
7. Continue to add nodes and connections until the process is finished.

Process Designer and Unicode Characters

When creating new processes it is important not to include unicode characters in either the process name or the process ID, as this is not currently supported and will result in unexpected behavior from the process designer when saving and retrieving the process assets.

Process Validation

The Process Designer can be used to check that processes are complete. If validation errors are encountered, an error message is displayed; however, validation errors should not be relied upon to check for overall correctness of the process.

To validate processes, select the validate process button on the top menu bar of the Process Designer.

If any validation errors are found, an error symbol is displayed to the left of the node that contains the error. A tool tip describing the validation error will be displayed when the cursor is positioned over the error symbol.

[Report a bug](#)

5.4. IMPORTING EXISTING PROCESSES

5.4.1. Importing Existing BPMN2 Processes

To import existing BPMN2 processes into the designer, select the import icon on the top menu in the Process Designer and select **Import from BPMN2**. The import menu provides two options:

- Upload an existing file from the local filesystem.
- Copy and paste the BPMN2 XML directly into the import dialogue box.

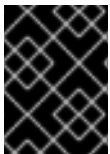
If a BPMN2 process has already been imported and a change to the process is required, the process should be edited in the process designer. Do not attempt to import the same process twice as the original process will not be overwritten.

When importing processes, it is important to note that the designer provides visual support for Data Objects, Lanes, and Groups. If the BPMN2 file being imported does not include positioning information for these nodes, they will need to be added manually.

(To import processes designed in JBoss Developer Studio, please refer to the *BRMS Getting Started Guide*).

[Report a bug](#)

5.4.2. Migrating jPDL 3.2 to BPMN2



IMPORTANT

The migration tool for jPDL 3.2 to BPMN2 is an experimental feature and must be enabled by the user. Red Hat does not support the jPDL 3.2 migration tool.

Procedure 5.2. Enabling the jPDL 3.2 Migration Tool

1. Stop the server.
2. Locate `jbpm.xml` in the `server/profile/deploy/designer.war/profiles/` directory.
3. Remove the comment tags around the JPDL migration plugin tool entry:

```
<!-- plugin name="ORYX.Plugins.JPDLMigration"/ -->
```

4. Save the file with the comment tags removed:

```
<plugin name="ORYX.Plugins.JPDLMigration"/>
```

5. Restart the server.
6. Log onto the JBoss BRMS user interface and navigate to the process designer. The jPDL migration tool button has now been added to the process designer user interface.

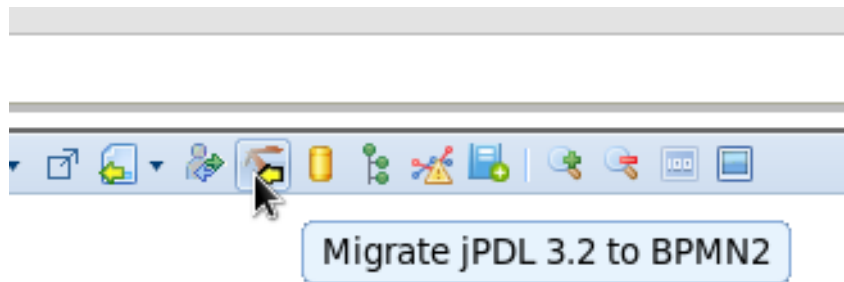


Figure 5.1. jPDL Migration Tool

[Report a bug](#)

5.5. VIEW AND SHARE PROCESSES

Processes designed, or accessed through, the Process Designer can be saved in a variety of formats making it possible for people who do not have access to the Process Designer to view the processes. The formats processes can be saved in are the following:

- ERDF
- JSON
- PDF
- PNG
- BPMN2
- SVG



NOTE

In order for users of the business process console to view process diagrams, it is necessary to save them as PNG files; this is done by clicking the PNG option at the bottom of the process designer window.

It is also possible to create code to share the process by selecting the share icon on the top menu of the Process Designer. The following formats are available:

- Process Image
- Process PDF
- Embeddable Process

To view the process BPMN2 source code during development of the process select **Source** → **View Source**.

[Report a bug](#)

5.6. DEFINING DOMAIN-SPECIFIC SERVICE NODES

The Process Designer supports domain-specific nodes. To include service nodes in the Process Designer's BPMN2 stencil set, the service node definitions can either be uploaded from JBoss Developer Studio (see [Section 6.1, “JBoss Developer Studio”](#)) or configured in the BRMS user interface.

Procedure 5.3. Define a Service Node

1. From the navigation panel, select **Knowledge Bases** → **Create New** → **New Work Item Definition**.

2. Enter a name for the definition in the **Name**: dialogue box.

Select which package to assign the service node to, or select **Create in Global Area**, provide a description and click **OK**.

3. Configure the service node as required.

For example, an email service node could be named **Email**, and parameters, **To**, **From**, **Subject**, and **Body**, all of type String. Edit the existing parameters and use the **Parameter** button to add new ones.

Edit the display name to show the required name for the work task.

Select **File** → **Save Changes**.

4. Upload an icon for the Service node.

Select **Create New** → **Create a file**.

Enter a name into the **Name** dialogue box.

Enter the file extension type.

Select which package the file should be created in and click **OK**.

5. Click **Choose File** to select an image file from the local filesystem and click **Upload**.

6. Add the icon to the service node.

Click the package name from the navigation panel; expand the WorkItemDefinition tab and select the work and click **open** for the service node. PNG and GIF formats are supported for icons.

The icon is now available for the **Select Icon to add** drop down box. Select the icon to use.

7. Click **File** → **Save Changes**.

8. To add the new service node to a process, open a process.

1. From the navigation panel select **Knowledge Bases** → **Packages** and select the package.
2. Expand the processes area under the **assets** tab and click **open** on the required process.

- From the **Shape Repository**, expand the **Service Tasks** tab and drag the new service node on to the process as required.

See Also:

- [Section 9.1, “Domain-Specific Service Nodes”](#)

[Report a bug](#)

5.7. CONNECTING TO A SERVICE REPOSITORY

Domain specific services and work items can be stored in a service repository. For instructions to set up a service repository, see [Section 9.5, “Service Repository”](#).

Procedure 5.4. Connection to the Service Repository

- Log onto the JBoss BRMS user interface.
- Access the process designer by either selecting an existing process to edit or by creating a new process.
- Select the service repository icon to enter the service repository URL.

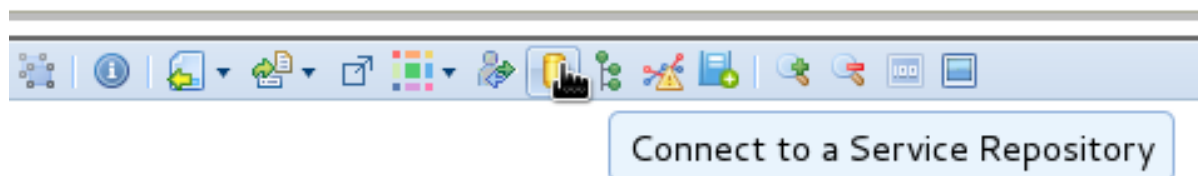


Figure 5.2. Service Repository Connection

- Enter the URL for the service repository and click **Connect**.
- Double click the required services to add them to the package and make them available in the **Shape Repository** of the process designer.

[Report a bug](#)

5.8. GENERATE PROCESS AND TASK FORMS

Designer allows users to generate process and task ftl forms. These forms are fully usable in the jBPM console. To start using this feature, locate the "Generate Task Form Templates" button in the designer toolbar:



Designer will iterate through your process BPMN2 and create forms for your process and each of the

human tasks in your process. It uses the defined process variables and human task data input/output parameters and associations to create form fields. The generated forms are stored in Guvnor, and the user is presented with a page which shows each of the forms created as well as a link to their sources in Guvnor:

Form Generation Results for Process demo1.demoprocess

Generated Templates

demoprocess-taskform	View Source
EnterInfo-taskform	View Source
<div style="border: 1px solid #ccc; padding: 5px; display: inline-block;">CLOSE</div>	

All forms are fully usable inside the jBPM console. In addition, each form includes basic JavaScript form validation which is determined based on the type of process variables and/or human task data input/output association definitions. Here is an example generated human task form.

User Task Form: DemoProcess.EnterInfo

Task Info	
Owners	tsurdilo
Actor ID	
Group	
Skippable	
Priority	
Comment	

Task Inputs	
input	<code>\${message}</code>

Task Outputs	
output	<input type="text"/> <input type="submit" value="SUBMIT"/>

In order for process and task forms to be generated, you have to make sure that your process has its ID parameter set and that each of the human tasks have the TaskName parameter set. Task forms contain pure HTML, CSS, and JavaScript, so they are easily editable in any HTML editor. Please note that there is no edit feature available in Designer, so each time a form is generated, existing forms (of the same name) will be overwritten.

[Report a bug](#)

CHAPTER 6. JBOSS DEVELOPER STUDIO

6.1. JBOSS DEVELOPER STUDIO

JBoss Developer Studio is the JBoss Integrated Development Environment available from the Red Hat customer support portal at <https://access.redhat.com>. JBoss Developer Studio provides tools and interfaces for developers working with JBoss Enterprise BRMS.

For installation and configuration instructions refer to the *JBoss BRMS Getting Started Guide*.



NOTE

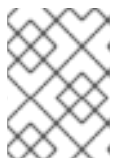
The JBoss BRMS plug-ins for JBoss Developer Studio are known by the names of the open-source projects they are based on. Guvnor is the project the BRMS user interface and repository are based on; Drools is the project the rules engine is based on; and jBPM is the project the Business Process Management functionality comes from.

[Report a bug](#)

6.2. PROJECT CREATION

Procedure 6.1. Create a New Project

1. Select **File** → **New** → **Project**.
2. Select **jBPM** → **jBPM project**.
3. Enter a name for the project into the **Project name**: text box and click **Next**.



NOTE

JBDS provides the option to add a sample HelloWorld Process to the project. Accept this default to test the sample project in the next step.

4. To test the project, right click the Java file that contains the main method and select **Run** → **run as** → **Java Application**.

The output will be displayed in the console tab.

[Report a bug](#)

6.3. PROCESS CREATION

Procedure 6.2. Create a New Process

1. To create a new process, select **File** → **New** → **Other** and then select **jBPM** → **BPMN2 Process**.
2. Select the parent folder for the process.

3. Enter a name in the **File name:** dialogue box and click **Finish**.
4. Open the graphical editor by right clicking the process .bpmn file, select **Open With** and then click the radio button next to **BPMN2 Process Editor**.
5. Add nodes to the process by clicking on the required node in the palette and clicking on the canvas where the node should be placed.
6. Connect the nodes with sequence flows. Select **Sequence Flow** from the palette, then click the nodes to connect them.
7. To edit a node's properties, click the node, open the properties tab in the bottom panel of the JBDS workspace, and click the values to be edited.

If the properties tab is not already open, right click the bpmn file in the package panel and select **Show in → Properties**.

8. Click the save icon to save the process.

[Report a bug](#)

6.4. VALIDATION AND DEBUGGING

JBoss Developer Studio can validate and debug processes.

Validation

To validate a process, right click the .bpmn file and select **Validate**.

If validation completes successfully, a dialogue box will appear stating there are no errors or warning.

If validation is unsuccessful, the errors will display in the **Problems** tab. Fix the problems and rerun the validation.

Debug

To debug a process, right click the .bpmn file and select **Debug As → Debug Configurations**; make any required changes to the test configuration and click **Debug**.

If no errors are found, the process will execute.

If errors are encountered, they will be described in the bottom window of JBoss Developer Studio. Fix the errors and rerun the debug process.

For further information about debugging processes, refer to [Section 11.2.1, “Debugging”](#).

[Report a bug](#)

CHAPTER 7. PERSISTENCE

7.1. PERSISTENT

Storing information persistently in a database makes it possible for processes to recover from unexpected interruptions. Information is stored persistently by default for JBoss BRMS 5.3 Standalone; however, customers deploying JBoss BRMS 5.3 as a deployable web app will need to configure persistence.

The types of information that can be stored persistently are the following:

- Runtime
- Process Definitions
- Historical information (logs)

This chapter describes the different types of persistence and how to configure them.

[Report a bug](#)

7.2. RUNTIME STATE

7.2.1. Runtime State

Whenever a process is started, an instance of that process is created. For instance, look at an example of a process representing a sales order. Each time a sales order is requested, one instance of the sales order process is created and contains only the information relevant to that sales order. When that instance is stored persistently, only the minimum set of information required to continue execution of the instance at some later time is stored. The engine automatically stores the runtime state in a database, ensuring that whenever the engine is invoked that any changes are stored at the end of the invocation and at safe points.

If it is ever necessary to restore the engine from a database, it is important not to reload the process instances and trigger them manually. Process instances will automatically resume execution if they are triggered, for example, by a timer expiring, the completion of a task that was requested by that process instance, or a signal being sent to the process instance. The engine will automatically reload process instances on demand. It is important to note that the snapshot of the process will be reloaded, and any changes that have occurred since the snapshot was taken will be lost.

The latest snapshot state can be retrieved using the following:

```
ksession.getProcessInstance(id)
```

The runtime persistence data should be considered internal, and the database tables where the data is stored should not be accessed directly. Where information about the current execution state of a process instance is required, refer to the history logs see [Section 7.4.1, “History Log”](#).

[Report a bug](#)

7.2.2. Safe Points

The state of a process instance is stored at safe points during execution of the process. A safe point occurs when the process instance has completed, aborted, or reached a wait state in all parallel paths of execution. When the process instance is stored persistently at safe points, the current state of the process instance and all other process instances that might have been affected are stored.

[Report a bug](#)

7.2.3. Binary Persistence

Binary persistence, also known as marshaling, converts the state of the process instance into a binary dataset. Binary persistence is the mechanism used to store and retrieve information persistently. The same mechanism is also applied to the session state and any work item states.

When the process instance state is persisted, two things happen:

- The process instance information is transformed into binary data. For performance reasons, a custom serialization mechanism is used and not normal Java serialization.
- The binary data is stored alongside other metadata about the process instance. This metadata includes the process instance ID, process ID, and the process start date.

Apart from the process instance state, the session itself can also store other forms of state, such as the state of timer jobs. The session can also store the data that any business rules would be evaluated over. This session state is stored separately as a binary dataset along with the ID of the session and some metadata. The session state can be restored by reloading the session with the given ID. The session ID can be retrieved using `ksession.getId()`.

Note that the process instance binary datasets are usually relatively small, as they only contain the minimal execution state of the process instance. For a simple process instance, the binary datasets usually contain one or a few node instances, i.e., any node that is currently executing and any existing variable values.

As a result of binary persistence, the data model is both simple and small:

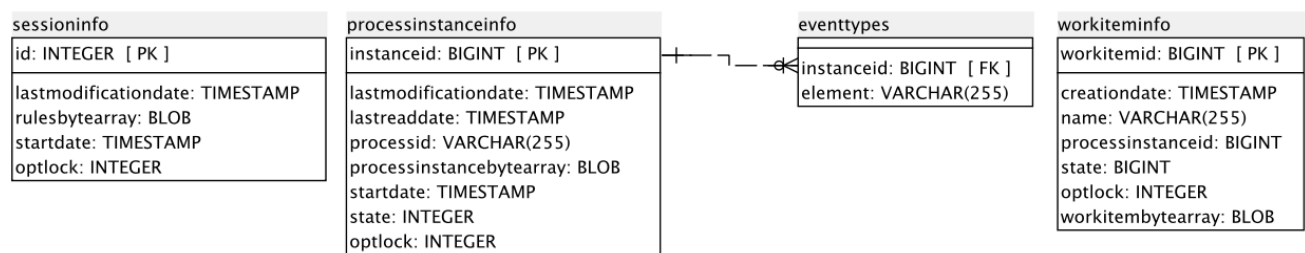


Figure 7.1. Data Model

The `sessioninfo` entity contains the state of the (knowledge) session in which the process instance is running.

Table 7.1. SessionInfo

Field	Description	Nullable
<code>id</code>	The primary key	NOT NULL

Field	Description	Nullable
lastmodificationdate	The last time that the entity was saved to the database	
rulesbytearray	The binary dataset containing the state of the session	NOT NULL
startdate	The start time of the session	
optlock	The version field that serves as its optimistic lock value	

The **processinstanceinfo** entity contains the state of the process instance.

Table 7.2. ProcessInstanceInfo

Field	Description	Nullable
instanceid	The primary key	NOT NULL
lastmodificationdate	The last time that the entity was saved to the database	
lastreaddate	The last time that the entity was retrieved (read) from the database	
processid	The name (ID) of the process	
processinstancebytearray	This is the binary dataset containing the state of the process instance	NOT NULL
startdate	The start time of the process	
state	An integer representing the state of the process instance	NOT NULL
optlock	The version field that serves as its optimistic lock value	

The **eventtypes** entity contains information about events that a process instance will undergo or has undergone.

Table 7.3. EventTypes

Field	Description	Nullable
instanceid	This references the processinstanceinfo primary key and there is a foreign key constraint on this column	NOT NULL
element	A text field related to an event that the process has undergone	

The **workiteminfo** entity contains the state of a work item.

Table 7.4. WorkItemInfo

Field	Description	Nullable
workitemid	The primary key	NOT NULL
name	The name of the work item	
processinstanceid	The (primary key) ID of the process: there is no foreign key constraint on this field	NOT NULL
state	An integer representing the state of the work item	NOT NULL
optlock	The version field that serves as its optimistic lock value	
workitembytearray	This is the binary dataset containing the state of the work item	NOT NULL

[Report a bug](#)

7.3. CONFIGURING PERSISTENCE

7.3.1. Configuring Persistence

Persistence is configured by default for JBoss BRMS 5.3 standalone; however, it must be configured for customers deploying JBoss BRMS 5.3 as a deployable web app by adding the jar files to the classpath.

[Report a bug](#)

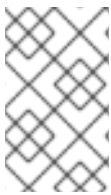
7.3.2. Adding Dependencies

The following packages are required for manual persistence configuration:

- JBoss BRMS 5.3 Deployable Package.
- The database vendor's JDBC driver.
- A transaction manager.

To manually add the necessary dependencies, copy the required jar files from the `jboss-jbpm-engine.zip` archive included with the BRMS 5.3 Deployable archive and make them available on the application classpath.

The following jar files are required when using a combination of Hibernate as the JPA persistence provider with an H2 in-memory database and Bitronix for JTA-based transactions management:



NOTE

This list is for demonstration purposes only. Supported configurations can be found at the following site, <http://www.redhat.com/resourcelibrary/articles/jboss-enterprise-brms-supported-configurations>.

- `jbpm-test`
- `jbpm-persistence-jpa`
- `drools-persistence-jpa`
- `persistence-api`
- `hibernate-entitymanager`
- `hibernate-annotations`
- `hibernate-commons-annotations`
- `hibernate-core`
- `commons-collections`
- `dom4j`
- `jta`
- `btm`
- `javassist`
- `slf4j-api`
- `slf4j-jdk14`
- `h2`

[Report a bug](#)

7.3.3. Configure the Engine to use Persistence

The **JBPMHelper** class of the **jbpm-test** module has a method to create a session and uses a configuration file to configure the session. The information below shows an example **jbpm.properties** file that uses an H2 in-memory database with persistence enabled.

Example 7.1. Example **jbpm.properties** File

```
# for creating a datasource
persistence.datasource.name=jdbc/jbpm-ds
persistence.datasource.user=sa
persistence.datasource.password=
persistence.datasource.url=jdbc:h2:tcp://localhost/~jbpm-db
persistence.datasource.driverClassName=org.h2.Driver

# for configuring persistence of the session
persistence.enabled=true
persistence.persistenceunit.name=org.jbpm.persistence.jpaa
persistence.persistenceunit.dialect=org.hibernate.dialect.H2Dialect

# for configuring the human task service
taskservice.enabled=true
taskservice.datasource.name=org.jbpm.task
taskservice.transport=mina
taskservice.usergroupcallback=org.jbpm.task.service.DefaultUserGroupCall
backImpl
```

The **JBPMHelper** class can be used to register the datasource:

```
JBPMHelper.setupDataSource();
```

The **JBPMHelper** class can be used to create sessions (after the knowledge base has been created):

```
StatefulKnowledgeSession ksession =
JBPMHelper.newStatefulKnowledgeSession(kbase);
```

Methods can now be called on this **ksession** (for instance, **startProcess**) and the engine will persist all runtime state in the created datasource.

The **JBPMHelper** class can be used to recreate sessions by restoring their state from the database by passing in the session ID. The session ID is retrieved using **ksession.getId()**.

```
StatefulKnowledgeSession ksession =
JBPMHelper.loadStatefulKnowledgeSession(kbase, sessionId);
```

[Report a bug](#)

7.3.4. Session ID

7.3.4.1. Session ID

The session ID of the most recently used persistent session is also stored in the `jbpmSessionId.ser` file. If the `jbpmSessionId.ser` file does not exist, it will be created and the session ID stored in the file. If the file exists, the session ID is read from the file and the session is loaded from the database, enabling the session to be reloaded after the server has been restarted.

By default the `jbpmSessionId.ser` file is located in `jboss-as/server/profile/tmp/` directory; however, this can be changed by modifying the `jbpm.console.tmp.dir` property in `jbpm.console.properties` located in the `jboss-as/server/profile/deploy/business-central-server.war/WEB-INF/classes/` directory.

[Report a bug](#)

7.3.5. Transactions

By default, when transaction boundaries are not provided inside an application, the engine will automatically execute each method invocation on the engine in a separate transaction. Transaction boundaries can be specified, allowing, for example, multiple commands to be combined into one transaction.

The following sample code uses the Bitronix transaction manager.

```
// create the entity manager factory and register it in the environment
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory( "org.jbpm.persistence.jpa" );
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );
env.set( EnvironmentName.TRANSACTION_MANAGER,
    TransactionManagerServices.getTransactionManager() );

// create a new knowledge session that uses JPA to store the runtime state
StatefulKnowledgeSession ksession =
    JPAKnowledgeService.newStatefulKnowledgeSession( kbase, null, env );

// start the transaction
UserTransaction ut =
    (UserTransaction) new InitialContext().lookup(
    "java:comp/UserTransaction" );
ut.begin();

// perform multiple commands inside one transaction
ksession.insert( new Person( "John Doe" ) );
ksession.startProcess( "MyProcess" );

// commit the transaction
ut.commit();
```

For Persistence and Concurrency, see [Section 3.12.1, “Multi-threading”](#) for more information.

[Report a bug](#)

7.4. HISTORY LOG

7.4.1. History Log

The history logs store information about the execution of process instances, so that the information can be retrieved and viewed at a later time. History logs contain information, for instance, about the number of processes that have run and are running.

This history log of execution information is created based on events that the process engine generates during execution. This is possible because the jBPM runtime engine provides a generic mechanism to listen to events. The necessary information can easily be extracted from these events and then persisted to a database. Filters can also be used to limit the scope of the logged information.

[Report a bug](#)

7.4.2. The Business Activity Monitoring Data Model

The jbpm-bam module contains an event listener that stores process-related information in a database using JPA or Hibernate directly. The data model itself contains three entities: one for process instance information, one for node instance information, and one for (process) variable instance information.

processinstancelog	nodeinstancelog	variableinstancelog
id: BIGINT [PK]	id: BIGINT [PK]	id: BIGINT [PK]
end_date: TIMESTAMP	log_date: TIMESTAMP	log_date: TIMESTAMP
processid: VARCHAR(255)	nodeid: VARCHAR(255)	processid: VARCHAR(255)
processinstanceid: BIGINT	nodeinstanceid: VARCHAR(255)	processinstanceid: BIGINT
start_date: TIMESTAMP	nodename: VARCHAR(255)	value: VARCHAR(255)
	processid: VARCHAR(255)	variableid: VARCHAR(255)
	processinstanceid: BIGINT	variableinstanceid: VARCHAR(255)
	type: INTEGER	

Figure 7.2. Business Activity Monitoring data model

The **ProcessInstanceLog** table contains the basic log information about a process instance.

Table 7.5. ProcessInstanceLog

Field	Description	Nullable
id	The primary key and ID of the log entity	NOT NULL
end_date	When applicable, the end date of the process instance	
processid	The name (ID) of the process	
processinstanceid	The process instance ID	NOT NULL
start_date	The start date of the process instance	

The **NodeInstanceLog** table contains more information about which nodes were executed inside each process instance. Whenever a node instance is entered from one of its incoming connections or is exited through one of its outgoing connections, that information is stored in this table.

Table 7.6. NodeInstanceLog

Field	Description	Nullable
id	The primary key and ID of the log entity	NOT NULL
log_date	The date of the event	
nodeid	The node ID of the corresponding node in the process definition	
nodeinstanceid	The node instance ID	
nodename	The name of the node	
processid	The ID of the process that the process instance is executing	
processinstanceid	The process instance ID	NOT NULL
type	The type of the event (0 = enter, 1 = exit)	NOT NULL

The **VariableInstanceLog** table contains information about changes in variable instances. The default is to only generate log entries after a variable changes. It is also possible to log entries before the variable's value changes.

Table 7.7. VariableInstanceLog

Field	Description	Nullable
id	The primary key and ID of the log entity	NOT NULL
log_date	The date of the event	
processid	The ID of the process that the process instance is executing	
processinstanceid	The process instance ID	NOT NULL
value	The value of the variable at the time that the log is made	

Field	Description	Nullable
variableid	The variable ID in the process definition	
variableinstanceid	The ID of the variable instance	

[Report a bug](#)

7.4.3. Storing Process Events in a Database

To log process history in a database, register the logger to the session:

```
StatefulKnowledgeSession ksession = ...;
JPAWorkingMemoryDbLogger logger = new JPAWorkingMemoryDbLogger(ksession);

// invoke methods one your session here

logger.dispose();
```

Note that this logger is like any other audit logger, and one or more filters can be added by calling the method **addFilter** to ensure that only relevant information is stored in the database.

The Logger should be disposed of when it is no longer needed.

The persistence database is configured in the **persistence.xml** file, which is located **business-central-server.war/WEB-INF/classes/META-INF/** directory of the JBoss BRMS 5.3 deployable installation.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence
  version="1.0"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
    http://java.sun.com/xml/ns/persistence/orm
    http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/persistence">

  <!--persistence-unit name="ProcessService">
  <jta-data-source>java:/DefaultDS</jta-data-source>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
  </properties>
  </persistence-unit-->

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-
type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
```

```
<mapping-file>META-INF/JBPMorm.xml</mapping-file>

<class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
<class>org.drools.persistence.info.SessionInfo</class>
<class>org.drools.persistence.info.WorkItemInfo</class>
<class>org.jbpm.process.audit.ProcessInstanceLog</class>
<class>org.jbpm.process.audit.NodeInstanceLog</class>
<class>org.jbpm.process.audit.VariableInstanceLog</class>
<properties>
  <property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect"/>
  <property name="hibernate.max_fetch_depth" value="3"/>
  <property name="hibernate.hbm2ddl.auto" value="update" />
  <property name="hibernate.show_sql" value="false" />
  <property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.JBossTransactionManagerLookup" />
</properties>
</persistence-unit>

</persistence>
```

[Report a bug](#)

CHAPTER 8. BUSINESS CENTRAL CONSOLE

8.1. BUSINESS CENTRAL CONSOLE

Business processes can be managed through the web console. The web console includes features for managing your process instances (starting/stopping/inspecting), inspecting and executing your (human) task list, and generating reports.

The Business Central Console is installed with the standalone JBoss BRMS 5.3; please refer to the *BRMS Getting Started Guide* for installation instructions. When using JBoss BRMS 5.3 from the deployable package, it is necessary to install the console. The console consists of two war directories that must be deployed to the application server. These directories contain the necessary libraries and the actual application. One jar contains the server application, the other one the client. Refer to the *BRMS Getting Started Guide* for details.

[Report a bug](#)

8.2. BUSINESS CENTRAL CONSOLE AND BRMS INTEGRATION

8.2.1. Business Central Console and BRMS Integration

The Business Central Console is integrated with the JBoss BRMS repository. The console will retrieve artifacts from all available packages in the repository (please note this does not include packages in the Global Area). Packages must be built in BRMS before they will be visible in the console. When a package has been updated and rebuilt, the updated package can be accessed in the console by pressing the refresh button.

The console will attempt to retrieve all available packages from the BRMS repository, and, as such, all packages must have been successfully built in BRMS. It is possible to limit the packages the console will attempt to retrieve, please see [Section 8.2.2, “Configuring the Business Central Console”](#) for details.

To build all packages in BRMS at once, from the navigation menu select **Knowledge Bases** → **Create New** → **Rebuild all package binaries**.

[Report a bug](#)

8.2.2. Configuring the Business Central Console

The management console is configured with the `jbpm.console.properties` file, which is located in the `jboss-as/server/profile/deploy/business-central-server.war/WEB-INF/classes/` directory. This file configures the following properties:



NOTE

Not all of the properties listed below are included in the `jbpm.console.properties` file by default.

- `jbpm.console.server.host` (default localhost)
- `jbpm.console.server.port` (default 8080)

- `jbpm.console.task.service.host` (default localhost) host where Task Server is deployed applies to all transports
- `jbpm.console.tmp.dir`: An optional property that is used to set the location of the `jbpmSessionId.ser` file, which stores the session ID of the most recently used persisted session.
- `jbpm.console.task.service.port`: The port where the task server is deployed (5446 when using HornetQ)
- `jbpm.console.task.service.strategy` (default HornetQ)
- `JMSTaskClient.connectionFactory` (no default) JNDI name of connection factory only for JMS
- `JMSTaskClient.acknowledgeMode` (no default) acknowledgment mode only for JMS
- `JMSTaskClient.transactedQueue` (no default) transacted queue name only for JMS
- `JMSTaskClient.queueName` (no default) queue name only for JMS
- `JMSTaskClient.responseQueueName` (no default) response queue name only for JMS
- `guvnor.protocol` (default http)
- `guvnor.host` (default localhost:8080)
- `guvnor.subdomain` (default drools-guvnor)
- `guvnor.usr` (default admin)
- `guvnor.pwd` (default admin)
- `guvnor.packages` (comma separated list of packages to load from Guvnor. When no packages have been specified, all packages will be visible. When specifying packages that should be visible, it is important to remember only the specified packages be visible.)
- `guvnor.connect.timeout` (default 10000)
- `guvnor.read.timeout` (default 10000)
- `guvnor.snapshot.name` (default LATEST) This is an optional property that specifies the name of the snapshot to use.



NOTE

The password and username contained in this file must match the credentials set in `jboss-as/server/production/deploy/jboss-brms.war/WEB-INF/components.xml`. If the credentials are changed in one place, they must be changed in both locations.

The Business Central console can be clustered so that multiple instances of the console and the process engine attached to the console can share the same data in a persisted database. Clustering the console makes it possible to spread the available processes across the cluster, and also ensures failover if a node in the cluster fails. See the *BRMS Getting Start Guide* for details.

[Report a bug](#)

8.2.3. User and Group Management

The human task service requires users to be defined as group members so that group members can claim tasks that are assigned to that group. The console uses username / group associations; when using the JBoss BRMS 5.3 standalone installation, the group assignments are specified in the **brms-roles.properties** file in the **server/profile/conf/props/** directory.

Alternative login modules can be configured; for details, please refer to the *BRMS Administrator Guide* about Security Authentication.

[Report a bug](#)

8.3. LOG ON TO THE BUSINESS CENTRAL CONSOLE

The human task service requires users to be defined as group members so that group members can claim tasks that are assigned to that group. The console uses username / group associations; when using the JBoss BRMS 5.3 standalone installation, the group assignments are specified in the **brms-roles.properties** file in the **server/profile/conf/props/** directory.

[Report a bug](#)

8.4. MANAGING PROCESS INSTANCES

The Business Central Console offers the following functionality for managing business processes that are part of the installed knowledge base:

Managing Process Instances

Managing Process Instances

The process instances table, found under the **Processes** menu, shows all running instances for a specific process definition. Select a process instance to show the details of that specific process instance.

Starting new process instances

New process instances are started by selecting the process definition from the process definition list and selecting **Start**. If a form is associated with the process (to ask for additional information before starting the process) the form will be displayed. After completing the form, the process is started with the provided information.

Inspecting Process Instance State

To inspect the state of a specific process instance, click on the **Diagram** button. The process flow chart is displayed with a red triangle overlaid on any currently active nodes.

Inspecting Process Instance Variables

Top level process instance variables can be inspected by clicking the **Instance Data** button.

Terminate Process Instances

To terminate a process instance, click the **Terminate** button.

Delete Process Instances

To delete a process instance, click the **Delete** button.

Signal Process

To signal processes that have catching intermediate signal events defined, click the **signal** button.

[Report a bug](#)

8.5. HUMAN TASK LISTS

The task management section of the console allows users to see their current task list. The group task list shows all the tasks that are not yet assigned to one specific user but that the currently logged in user could claim. The personal task list shows all tasks that are assigned to the currently logged in user. To execute a task, users select it from the personal task list and click **View**. If a form is associated with the selected task (for example, to ask for additional information), the form will be displayed. After completing the form, the task will also be completed.

[Report a bug](#)

8.6. REGISTERING SERVICE HANDLERS

Service handlers must be registered to execute domain-specific services as custom service tasks, this happens because the process only contains a high-level description of the service that needs to be executed; accordingly, a handler is responsible for invoking the service.

Service handlers are registered by adding a configuration file that specifies the implementation of class for each of the handlers to the classpath. You can specify which configuration files are loaded in a **drools.session.conf** file by using the `drools.workItemHandlers` property as a list of space delimited file names:

```
drools.workItemHandlers = CustomerWorkItemHandlers.conf
```

These file names should contain a Map of entries, the name and the corresponding `WorkItemHandler` instance that should be used to execute the service. The configuration file is using the MVEL script language to specify a map of type `Map<String,WorkItemHandler>`:

```
[
  "log" : new
  org.jbpm.process.instance.impl.demo.SystemOutWorkItemHandler(),
]
```

The implementation classes (and dependencies) also need to be added to the classpath of the server war.

[Report a bug](#)

8.7. ADDING NEW PROCESS AND TASK FORMS

Forms can be used to start a new process and complete a human task. To create a form for a specific process definition, create a template with the name {processId}.ftl. The template itself should use HTML to model the form. For example, com.sample.evaluation.ftl file uses HTML to model the form:

```
<html>
<body>
<h2>Start Performance Evaluation</h2>
<hr>
<form action="complete" method="POST" enctype="multipart/form-data">
Please fill in your username: <input type="text" name="employee" /></BR>
<input type="submit" value="Complete">
</form>
</body>
</html>
```

Task forms for a specific type of human task (uniquely identified by its task name) can be linked to that human task by creating a template with the name {taskName}.ftl. The form has access to a task parameter that represents the current human task, so the task form can be dynamically adjusted based on the task input. The task parameter is a task model object as defined in the jbpm-human-task module. This allows the task form to be customized based on the description or input data related to that task. For example, the evaluation form shown earlier uses the task parameter to access the description of the task and show that in the task form:

```
<html>
<body>
<h2>Employee evaluation</h2>
<hr>
${task.descriptions[0].text}<br/>
<br/>
Please fill in the following evaluation form:
<form action="complete" method="POST" enctype="multipart/form-data">
Rate the overall performance: <select name="performance">
<option value="outstanding">Outstanding</option>
<option value="exceeding">Exceeding expectations</option>
<option value="acceptable">Acceptable</option>
<option value="below">Below average</option>
</select><br/>
<br/>
Check any that apply:<br/>
<input type="checkbox" name="initiative" value="initiative">Displaying
initiative<br/>
<input type="checkbox" name="change" value="change">Thriving on
change<br/>
<input type="checkbox" name="communication" value="communication">Good
communication skills<br/>
<br/>
<input type="submit" value="Complete">
</form>
</body>
</html>
```

Task forms also have access to the additional task parameters that might be mapped in the user task node from process variables using parameter mapping. See [Section 10.1, “Human Tasks”](#) for more

details. These task parameters are also directly accessible inside the task form. For example, to make a task form to review customer requests, the user task node copies the `userId` (of the customer that performed the request), the `comment` (the description of the request) and the `date` (the actual date and time of the request) from the process into the task as task parameters, and these parameters will then be accessible directly in the task form:

```
<html>
<body>
<h2>Request Review</h2>
<hr>
UserId: ${userId} <br/>
Description: ${description} <br/>
Date: ${date?date} ${date?time}
<form action="complete" method="POST" enctype="multipart/form-data">
Comment:<BR/>
<textarea cols="50" rows="5" name="comment"></textarea></BR>
<input type="submit" name="outcome" value="Accept">
<input type="submit" name="outcome" value="Reject">
</form>
</body>
</html>
```

Data that is provided by the user when filling in the task form will be added as result parameters when completing the task. The name of the data element will be used as the name of the result parameter. For example, when completing the first task above, the Map of outcome parameters will include result variables called `performance`, `initiative`, `change`, and `communication`. The result parameters can be accessed in the related process by mapping these result parameters to process variables using result mapping.

Forms should either be available on the classpath (for example inside a jar in the **`jbossas/server/profile/lib`** directory or added to the set of sample forms in the `jbpm-gwt-form.jar` in the `jbpm console server war`), or the forms can be stored in the JBoss BRMS process repository.

[Report a bug](#)

8.8. REST INTERFACE

The console offers a REST interface for the functionality it exposes. This allows easy integration with the process engine for features like starting process instances and retrieving task lists.

The list URLs that the REST interface exposes can be inspected if you navigate to the following URL:

<http://localhost:8080/business-central-server/rs/server/resources/jbpm>

For example, tasks can be closed using:

```
/business-central-server/rs/task/{taskId}/close
```

A new processes instance can be started using:

```
/business-central-server/rs/process/definition/{id}/new_instance
```

[Report a bug](#)

CHAPTER 9. DOMAIN-SPECIFIC PROCESSES

9.1. DOMAIN-SPECIFIC SERVICE NODES

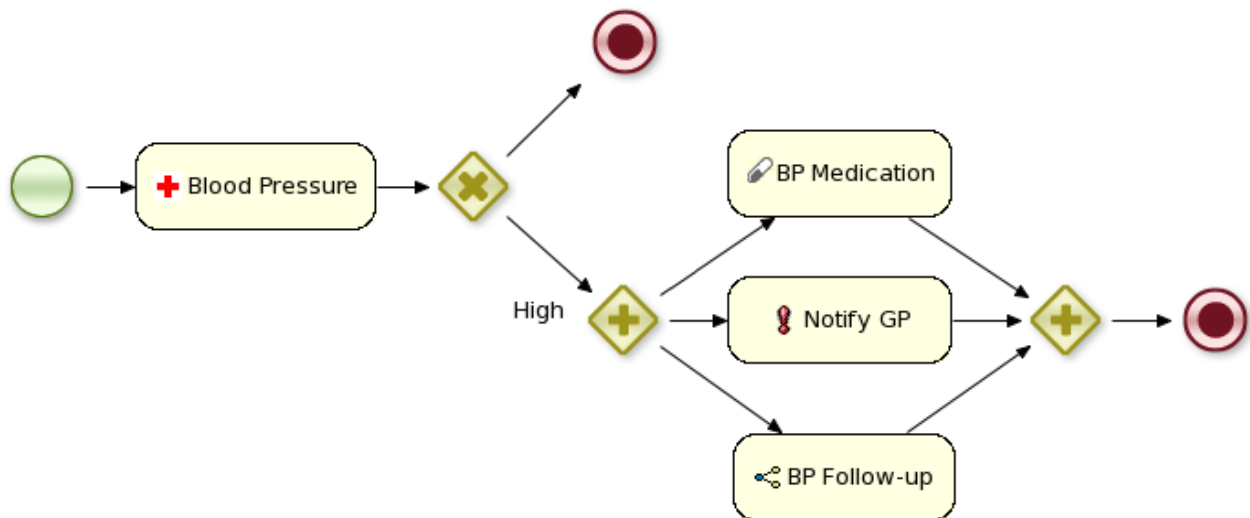
Domain-specific processes extend default process constructs with domain-specific extensions. Because domain-specific languages are targeted to one application domain, they can offer constructs that are closely related to the problem the user is trying to solve, making the process easier to understand and to a degree, self-documenting. Domain-specific work items (or service nodes) represent atomic units of work that need to be executed by specifying what needs to be executed in the context of the process in a declarative manner.

Domain-specific service nodes should be:

1. Declarative (they describe what, not how)
2. high-level (no code)
3. Adaptable to the context

Users can define their own domain-specific service nodes and integrate them into the process language. The image below provides an example of a process in a health care context. The process includes domain-specific service nodes for ordering nursing tasks: measuring blood pressure, prescribing medication, and notifying care providers.

Example 9.1. Ordering Nursing Tasks



[Report a bug](#)

9.2. DEFINE A WORK ITEM

A work item represents an atomic unit of work in a declarative way. The work item includes a unique name and parameters; in addition, each item can include results where results are expected to appear. A notification of a work item with no result parameters or icon could be defined as follows:

```

Name: "Notification"
Parameters
  
```

```

From [String]
To [String]
Message [String]
Priority [String]

```

Work item definitions are specified in one or more configuration files (with a .wid file extension) in the project classpath in the **META-INF** directory. The properties are provided as name-value pairs. Parameters and results are mapped and each parameter name is mapped to the expected data type. The configuration file also includes the display name and icon for the work item.

Example 9.2. Example Notification Work Item using MVEL

```

import org.drools.process.core.datatype.impl.type.StringDataType;
[
  // the Notification work item
  [
    "name" : "Notification",
    "parameters" : [
      "Message" : new StringDataType(),
      "From" : new StringDataType(),
      "To" : new StringDataType(),
      "Priority" : new StringDataType(),
    ],
    "displayName" : "Notification",
    "icon" : "icons/notification.gif"
  ]
]

```

Icons need to be in either .gif or .png format; both formats require a pixel size of 16x16. The icons should be stored in the **resources** directory:

```
project/src/main/resources/icons/notification.gif
```

In addition to the properties defined in the work item, all work items also have these three properties:

1. Parameter Mapping:

Maps the value of a variable in the process to a parameter of the work item. The work item can be customized based on the current state of the actual process instance (for example, the priority of the notification could be dependent on process-specific information).

2. Result Mapping:

Maps a result to a process variable, which is returned after the work item has been executed, and it makes the variable available to the rest of the process.

3. Wait for completion:

By default, the process waits until the requested work item has been completed before continuing with the process. It is also possible to continue immediately after the work item has been requested (and not wait for the results) by setting wait for completion to false.

The process below is an example that creates a domain-specific node to execute Java, asking for the class and method parameters. It includes a custom java.gif icon and consists of the following files and resulting screenshot:

Example 9.3.

```
import org.drools.process.core.datatype.impl.type.StringDataType;
[
  // the Java Node work item located in:
  // project/src/main/resources/META-INF/JavaNodeDefinition.conf
  [
    "name" : "JavaNode",
    "parameters" : [
      "class" : new StringDataType(),
      "method" : new StringDataType(),
    ],
    "displayName" : "Java Node",
    "icon" : "icons/java.gif"
  ]
]
```

[Report a bug](#)

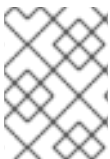
9.3. REGISTER THE WORK DEFINITION

When using JBoss Developer Studio to edit processes, you are able to register custom work item definition files for your project using the `drools.workDefinitions` property. This property represents a list of files containing work definitions, which are separated using spaces. For example, include a `drools.rulebase.conf` file in the META-INF directory of your project and add the following line:

```
drools.workDefinitions = MyWorkDefinitions.wid
```

This will replace the default domain specific node types EMAIL and LOG with the newly defined NOTIFICATION node in the JBoss Developer Studio process editor. Should you wish to just add a newly created node definition to the existing palette nodes, adjust the `drools.workDefinitions` property as follows and include the default set configuration file:

```
drools.workDefinitions = MyWorkDefinitions.conf WorkDefinitions.conf
```



NOTE

It is recommended to use extension `.wid` for your own definitions of domain specific nodes.

To update work definitions in the JBoss Enterprise BRMS user interface, please refer to [Section 5.6](#), “Defining Domain-Specific Service Nodes”.

[Report a bug](#)

9.4. EXECUTING SERVICE NODES

The process engine contains a `WorkItemManager` that delegates the work items to the `WorkItemHandlers` to execute the work item. The `WorkItemHandlers` notify the `WorkItemManager` when the work item has been completed. For executing notification work items, a `NotificationWorkItemHandler` should be created (implementing the `WorkItemHandler` interface):

```
package com.sample;

import org.drools.runtime.process.WorkItem;
import org.drools.runtime.process.WorkItemHandler;
import org.drools.runtime.process.WorkItemManager;

public class NotificationWorkItemHandler implements WorkItemHandler {

    public void executeWorkItem(WorkItem workItem, WorkItemManager manager)
    {
        // extract parameters
        String from = (String) workItem.getParameter("From");
        String to = (String) workItem.getParameter("To");
        String message = (String) workItem.getParameter("Message");
        String priority = (String) workItem.getParameter("Priority");
        // send email
        EmailService service =
ServiceRegistry.getInstance().getEmailService();
        service.sendEmail(from, to, "Notification", message);
        // notify manager that work item has been completed
        manager.completeWorkItem(workItem.getId(), null);
    }

    public void abortWorkItem(WorkItem workItem, WorkItemManager manager) {
        // Do nothing, notifications cannot be aborted
    }
}
```

This `WorkItemHandler` sends a notification as an email and then immediately notifies the `WorkItemManager` that the work item has been completed. Note that not all work items can be completed directly. In cases where executing a work item takes some time, execution can continue asynchronously and the work item manager can be notified later. If a work item is aborted before it has completed, the `abort` method should be used to specify how the item should be aborted.

`WorkItemHandlers` should be registered at the `WorkItemManager` using the following API:

```
ksession.getWorkItemManager().registerWorkItemHandler(
    "Notification", new NotificationWorkItemHandler());
```

Decoupling the execution of work items from the process itself has the following advantages:

1. The process is more declarative, specifying what should be executed, not how.
2. Changes to the environment can be implemented by adapting the work item handler. The process itself should not be changed.

3. The same processes can be used in different environments, where the work item handler is responsible for integration with the right services.
4. It is easy to share work item handlers across processes and projects (which would be more difficult if the code would be embedded in the process itself).
5. Different work item handlers could be used depending on the context. For example, during testing or simulation, it might not be necessary to execute the work items. In this case, specialized dummy work item handlers could be used during testing.

[Report a bug](#)

9.5. SERVICE REPOSITORY

Domain-specific services can be added to a repository to make them available to other users.

Services are added to a service repository by creating a configuration file that contains all the necessary information and links to the required files. Note, the configuration file is an extended version of a normal work definition configuration file as shown in [Section 9.2, “Define a Work Item”](#).

Example 9.4. Service Repository Configuration File

```
import org.drools.process.core.datatype.impl.type.StringDataType;
[
  [
    "name" : "Twitter",
    "description" : "Send a twitter message",
    "parameters" : [
      "Message" : new StringDataType()
    ],
    "displayName" : "Twitter",
    "eclipse:customEditor" :
"org.drools.eclipse.flow.common.editor.editpart.work.SampleCustomEditor"
  ,
    "icon" : "twitter.gif",
    "category" : "Communication",
    "defaultHandler" :
"org.jbpm.process.workitem.twitter.TwitterHandler",
    "documentation" : "index.html",
    "dependencies" : [
      "file:./lib/jbpm-twitter.jar",
      "file:./lib/twitter4j-core-2.2.2.jar"
    ]
  ]
]
```

- The icon property should refer to a file with the given file name in the same folder as the extended configuration file (so it can be downloaded by the import wizard and used in the process diagrams). Icons should be 16x16 GIF files.
- The category property defines the category this service should be placed under when browsing the repository.

- The `defaultHandler` property defines the default handler implementation (i.e. the Java class that implements the **WorkItemHandler** interface and can be used to execute the service). This can automatically be registered as the handler for that service when importing the service from the repository.
- The `documentation` property defines a documentation file that describes what the service does and how it works. This property should refer to a HTML file with the given name in the same folder as the extended configuration file (so it can be shown by the import wizard when browsing the repository).
- The `dependencies` property defines additional dependencies that are necessary to execute this service. This usually includes the handler implementation jar but could also include additional external dependencies. These dependencies should also be located on the repository on the given location (relative to the folder where the extended configuration file is located), so they can be downloaded by the import wizard when importing the service.

The root of the repository should also contain an **index.conf** file that references all the folders that should be processed when searching for services in the repository. Each of those folders should then contain:

- An extended configuration file with the same name as the folder (e.g. **Twitter.conf**) that defines the service task
- The icon as references in the configuration file
- The documentation as references in the configuration file
- The dependencies as references in the configuration file (for example in a lib folder)

Include an additional **index.conf** in each sub-directory of the repository that can be used to scan additional sub-folders. Note that the hierarchical structure of the repository is not shown when browsing the repository using the import wizard, as the category property in the configuration file is used for that.

The example below explains how to import resources from the service repository when working with JBoss Developer Studio.

Procedure 9.1. Import Resources From the Service Repository

1. Right click the project and select **Import . . .**
2. Select the source of the resources to import. i.e., **Guvnor** → **Resources from Guvnor** and click **Next**.
3. Navigate to the resources. i.e., select the package where the resource is located in Guvnor and highlight the resources. Click **Menu**.
4. Select the destination location for the resources and click **Finish**.

[Report a bug](#)

CHAPTER 10. HUMAN TASKS

10.1. HUMAN TASKS

Human Tasks are tasks within a process that must be carried out by human actors. BRMS Business Process Management supports a human task node inside processes for modeling the interaction with human actors. The human task node allows process designers to define the properties related to the task that the human actor needs to execute; for example, the type of task, the actor, and the data associated with the task can be defined by the human task node. A back-end human task service manages the lifecycle of the tasks at runtime. The implementation of the human task service is based on the WS-HumanTask specification, and the implementation is fully pluggable; this means users can integrate their own human task solution if necessary.

To include human tasks in processes, the following steps are required:

- Human tasks nodes must be included inside the process model.
- A task management component must integrate with the BRMS BPM (BRMS 5.3 standalone comes with the WS-HumanTask implementation included).
- End users must interact with a human task client to request their tasks, claim and complete tasks.

[Report a bug](#)

10.2. ADDING HUMAN TASKS TO PROCESSES

10.2.1. User Task Node



A user task node represents an atomic task that needs to be executed by a human actor.



NOTE

Human task nodes are considered the same as any other external service and must be invoked as a domain-specific service, for further details see [Section 9.1, “Domain-Specific Service Nodes”](#)

A user task node contains the following properties:

- *ID*: The ID of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *TaskName*: The name of the human task.
- *Priority*: An integer indicating the priority of the human task.

- *Comment*: A comment associated with the human task.
- *ActorId*: The actor ID that is responsible for executing the human task. A list of actor ID's can be specified using a comma (',') as separator.
- *GroupId*: The group ID that is responsible for executing the human task. A list of group ID's can be specified using a comma (',') as separator.
- *Skippable*: Specifies whether the human task can be skipped, i.e., whether the actor may decide not to execute the task.
- *Content*: The data associated with this task.
- *Swimlane*: The swimlane this human task node is part of. Swimlanes make it easy to assign multiple human tasks to the same actor.
- *On entry and on exit actions*: Action scripts that are executed upon entry and exit of this node.
- *Parameter mapping*: Allows copying the value of process variables to parameters of the human task. Upon creation of the human tasks, the values will be copied.
- *Result mapping*: Allows copying the value of result parameters of the human task to a process variable. Upon completion of the human task, the values will be copied. A human task has a result variable "Result" that contains the data returned by the human actor. The variable "ActorId" contains the ID of the actor that actually executed the task.

User task nodes can be edited in the properties view when editing in JBoss Developer Studio or in the Properties window when editing in the process designer embedded in the BRMS user interface.

[Report a bug](#)

10.2.2. Dynamic Human Task Properties

Properties such as name, actorID, and priority can be added to the user task node when the process is created; however, some properties in human tasks can be dependent on data that comes from another part of the process instance that the node belongs to, making it necessary to add human task properties dynamically. There are two ways to make human task properties dynamic.

`#{expression}`

Task parameters of type String can use `#{expression}` to embed the value of the given expression in the String. For example, the comment related to a task might be "Please review this request from user `#{user}`", where `user` is a variable in the process. At runtime, `#{user}` will be replaced by the actual user name for that specific process instance. The value of `#{expression}` will be resolved when creating human task and the `#{...}` will be replaced by the `toString()` value of the value it resolves to. The expression could simply be the name of a variable (in which case it will be resolved to the value of the variable), but more advanced MVEL expressions are possible as well; for example, `#{person.name.firstname}` could be used as the expression. Note that this approach can only be used for String parameters. Other parameters should use parameter mapping to map a value to that parameter.

Parameter mapping

The value of a process variable (or a value derived from a variable) can be mapped to a task parameter. For example, if a task needs to be assigned to a user whose ID is a variable in the process, the task could be completed by mapping the ID variable to the parameter `ActorId`.

[Report a bug](#)

10.2.3. User and Group Assignment

Tasks can be assigned to one specific user, and the task will show up on the task list of the specified user only. If a task is assigned to more than one user, any of those users can claim and execute the task.

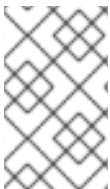
Tasks can also be assigned to one or more groups, the task will show up on the task list for the specified group, and any member of that group can claim and execute the task.

[Report a bug](#)

10.2.4. Standard Human Roles

Tasks and notifications within Business Process Management can be assigned to generic human roles. The following list contains some of these generic roles:

- *Task Initiator* - creates the task instance.
- *Task Stakeholders* - are responsible for the outcome of the task instance.
- *Potential Owners* - receive the task so they can complete it.
- *Actual Owner* - actually performs the task.
- *Excluded Owners* - may not reserve or start the task.
- *Business Administrators* - perform the Task Stakeholders role but at a task type level.
- *Notification recipients* - receive the notifications pertaining to the task.



NOTE

Task Stakeholder has no special authority within BRMS BPM 5.3.1; however, Business Administrator has the authority to claim, stop, release, suspend, resume, skip, delegate, forward, activate, and exit tasks even if they are not the task owner.

[Report a bug](#)

10.2.5. Task Escalation and Notification

In certain situations, the escalation of a task is necessary. For example, a user assigned to a task may be unable to complete that task within a certain period of time. In such cases, tasks should be automatically reassigned to another actor or group. Escalations can be defined for tasks that are in the following statuses:

- not started (READY or RESERVED)
- not completed (IN_PROGRESS)

Whenever an escalation occurs, users and groups defined in it will be assigned to the task as potential owners, replacing those who were previously assigned. If an actual owner is assigned to the task, the escalation will reset and the task will be put in READY state.

Users	Groups	Expires At	Type
john	sales	4d	not-started

Buttons: Add, Remove, Cancel, OK

The following is a list of attributes that can be specified during the escalation process:

- *Users*. This attribute is a comma separated list of user ids that should be assigned to the task on escalation. String values and expressions `#{user-id}` are acceptable for this attribute.
- *Groups*. This comma separated list of group ids should be assigned to the task on escalation. String values and expressions `#{group-id}` are acceptable for this attribute.
- *Expires At*. This attribute defines the time an escalation should take place. It should be defined as time definition (2m, 4h, 6d, etc.) in same way as for timers. String values and expressions `#{expiresAt}` are acceptable for this attribute.
- *Type*. This attribute identifies the type of task state the escalation should take place in (not-started | not-completed).

Email notifications can be sent out through the 'Notifications' tab. It is very similar to escalation in terms of definition. Email notifications can be sent for tasks that are in following statuses:

- not started (READY or RESERVED)
- not completed (IN_PROGRESS)

Notifications

Please take care of this task!

The image shows a configuration dialog box for an email notification. It has several sections with text input fields and a list of recipients.

- Type:** A dropdown menu with "not-started" selected.
- ExpiresAt:** A text field containing "4d".
- From:** An empty text field.
- To Users:** A text field containing "john".
- To Groups:** A text field containing "sales".
- Reply To:** An empty text field.
- Subject:** A text field containing "Please take care of this task!".
- Body:** A large text area containing "Hello," and "Please take care of this task instead of John." with a mouse cursor over the second line.
- Regards:** A text field that is currently empty.

Below the "Regards" field are four buttons: "Add", "Remove", "Update", and "Clear". At the bottom of the dialog are "Cancel" and "OK" buttons.

Email notifications have the following properties:

- *Type*. This attribute identifies the type of task state on which escalation should take place (not-started | not-completed).
- *Expires At*. This time definition property determines when escalation should take place. It should be defined as time definition (2m, 4h, 6d, etc.) in the same way as for timers. String values and expressions `#{expiresAt}` are acceptable properties.

- *From*. An optional user or group id that will be used in the 'From' field for email messages; it accepts Strings and expressions.
- *To Users*. A comma separated list of user ids that will become recipients of the notification.
- *To Groups*. A comma separated list of group ids that will become recipients of the notification.
- *Reply To*. An optional user or group id that should receive replies to the notification.
- *Subject*. The subject of the notification; it accepts Strings and expressions.
- *Body*. The body of the notification; it accepts Strings and expressions.

Notifications can reference process variables by `#{processVariable}` and task variables by `#{taskVariable}`. Accordingly, process variables resolve at task creation time, and task variables resolve at notification time. The following list contains several task variables that can be used while working with notifications:

- *taskId*: An internal id of a task instance.
- *processInstanceId*: An internal id of a process instance that the task belongs to.
- *workItemId*: An internal id of a work item that created this task.
- *processSessionId*: A session internal id of a runtime engine.
- *owners*: A list of users/groups that are potential owners of the task.
- *doc*: A map that contains regular task variables.

Below is an example notification message that illustrates how different variables can be accessed.

```
<html>
  <body>
    <b>#{owners[0].id} you have been assigned to a task (task-id #{taskId})</b><br>
    You can access it in your task
    <a href="http://localhost:8080/jbpm-console/app.html#errai_ToolSet_Tasks;Group_Tasks.3">inbox</a><br>
    Important technical information that can be of use when working on it<br>
    - process instance id - #{processInstanceId}<br>
    - work item id - #{workItemId}<br>

  <hr/>

  Here are some task variables available:
  <ul>
    <li>ActorId = #{doc['ActorId']}</li>
    <li>GroupId = #{doc['GroupId']}</li>
    <li>Comment = #{doc['Comment']}</li>
  </ul>
  <hr/>
  Here are all potential owners for this task:
  <ul>
    $foreach{orgEntity : owners}
      <li>Potential owner = #{orgEntity.id}</li>
```

```
$end{}  
</ul>  
  
<i>Regards</i>  
</body>  
</html>
```

[Report a bug](#)

10.2.6. Data Mapping

10.2.6.1. Data Mapping

Human tasks typically present some data related to the task that needs to be performed to the actor that is executing the task. Human tasks usually also request the actor to provide some result data related to the execution of the task. Task forms are typically used to present this data to the actor and request results.

[Report a bug](#)

10.2.6.2. Task Parameters

Data that needs to be displayed in a task form should be passed to the task using parameter mapping. Parameter mapping allows you to copy the value of a process variable to a task parameter (as described in [Section 10.2.2, “Dynamic Human Task Properties”](#)). This could be the customer name that needs to be displayed in the task form, the actual request, etc. To copy data to the task, map the variable to a task parameter. This parameter will then be accessible in the task form.

[Report a bug](#)

10.2.6.3. Task Results

Data that needs to be returned to the process should be mapped from the task back into process variables, using result mapping. Result mapping allows you to copy the value of a task result to a process variable. This could be some data that the actor filled in. To copy a task result to a process variable, map the task result parameter to the variable in the result mapping. The value of the task result will then be copied after completion of the task so it can be used in the remainder of the process.

[Report a bug](#)

10.2.7. Swimlanes

User tasks can be used in combination with swimlanes to assign multiple human tasks to the same actor. Whenever the first task in a swimlane is created, and that task has an actorId specified, that actorId will be assigned to all the tasks in that swimlane. Note that this would override the actorId of subsequent tasks in that swimlane (if specified), so only the actorId of the first human task in a swimlane will be taken into account, all others will then take the actorId as assigned in the first one.

To add a human task to a swimlane, first ensure the swimlane has been defined in the process. If it has not been defined, define the swimlane by editing the swimlane property. Next, specify the name of the swimlane as the value of the *Swimlane* parameter of the user task node.

[Report a bug](#)

10.2.8. Removing Tasks from the Database

Human tasks information can be removed from the database with the `org.jbpm.task.admin.TaskCleanUpProcessEventListener`, which is a `DefaultProcessEventListener` that archives and removes completed tasks with the associated process ID.

The `TaskCleanUpProcessEventListener` uses an instance of `org.jbpm.task.admin.TasksAdmin`, which can be obtained from `org.jbpm.task.service.TaskService#createTaskAdmin()`.

Example 10.1. Attaching the Event Listener

```
TasksAdmin admin = new TaskService(...).createTaskAdmin();
StatefulKnowledgeSession ksession = ...
ksession.addEventListener(new TaskCleanUpProcessEventListener(admin));
```

[Report a bug](#)

10.3. HUMAN TASK SERVICE

10.3.1. Human Task Service

Human tasks are similar to any other external service and are implemented as a domain-specific service. Refer to [Section 9.1, “Domain-Specific Service Nodes”](#) for details about including domain-specific services in a process. Because a human task is a domain-specific service, the process itself contains a high-level, abstract description of the human tasks that need to be executed, and a work item handler is responsible for binding this abstract tasks to a specific implementation. With this pluggable work item handler approach, users can plug in the human task service that is provided, as described below, or they can register their own implementation.

The default implementation of a human task service is based on the WS-HumanTask specification. It manages the life cycle of the tasks (creation, claiming, completion, etc.) and stores the state of all the tasks, task lists, etc. It also supports features like internationalization, calendar integration, different types of assignments, delegation, deadlines, etc. It is implemented as part of the `jbpm-human-task` module. The WS-HumanTask (WS-HT) specification can be downloaded from the following location http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf.

[Report a bug](#)

10.3.2. Task Life Cycle

Whenever a user task node is triggered during the execution of a process instance, a human task is created, and the process only leaves that node when the human task has been completed or aborted.

The human task life cycle is as follows:

1. The task is created and starts at the 'created' stage.
2. The task is usually transferred to the 'Ready' stage automatically, and it is displayed on the task lists of users who can claim the task.
3. A user claims the task and the status is set to 'Reserved'.
4. The user starts the task (executes the task) and the status is set to 'InProgress'
5. The user completes the task and specifies the result data of the task, and the status is set to 'Completed'. If the user was unable to complete the task, they indicate this with a fault response (including the associated fault data) and the status is set to 'Failed'.

The human task life cycle can also include the following steps:

- Delegating or forwarding a task to be assigned to another user.
- Revoking a task. After claiming a task, a user can revoke the task and it will become available again to all the users who can claim it.
- Temporarily suspending and resuming a task.
- Stopping a task in progress.
- Skipping a task (if the task has been marked as skippable), in which case the task will not be executed.

[Report a bug](#)

10.3.3. Integrate a Human Task Service

To integrate an alternative human task service, a custom work item handler must be registered. The custom work item handler can be registered as follows:

```
StatefulKnowledgeSession ksession = ...;
ksession.getWorkItemManager().registerWorkItemHandler("Human Task", new
CommandBasedHornetQWSHumanTaskHandler());
```

By default, this handler will connect to the human task service on the local machine on port 5446. To change the address and port of the human task service, invoke the `setConnection(ipAddress, port)` method on the `CommandBasedHornetQWSHumanTaskHandler`.

The communication between the human task service and the process engine, or any task client, is done by sending messages between the client and the server. HornetQ is the default transport mechanism for client server communication.

[Report a bug](#)

10.3.4. Interacting with the Human Task Service

The Business Central Console offers a graphical interface for users to interact with the human task service, see [Section 8.1, “Business Central Console”](#) for more details. The human task service exposes various methods to manage the life cycle of the tasks through a Java API.

A task client (class `org.jbpm.task.service.TaskClient`) offers the following methods for managing the life cycle of human tasks:

```
public void start( long taskId, String userId,
TaskOperationResponseHandler responseHandler )
public void stop( long taskId, String userId, TaskOperationResponseHandler
responseHandler )
public void release( long taskId, String userId,
TaskOperationResponseHandler responseHandler )
public void suspend( long taskId, String userId,
TaskOperationResponseHandler responseHandler )
public void resume( long taskId, String userId,
TaskOperationResponseHandler responseHandler )
public void skip( long taskId, String userId, TaskOperationResponseHandler
responseHandler )
public void delegate( long taskId, String userId, String targetUserId,
TaskOperationResponseHandler responseHandler )
public void complete( long taskId, String userId, ContentData outputData,
TaskOperationResponseHandler responseHandler )
...

```

All of the above methods take the following arguments:

- **taskId**: The ID of the task. This is usually extracted from the currently selected task in the user task list in the user interface.
- **userId**: The ID of the user that is executing the action. This is usually the ID of the user that is logged in to the application.
- **responseHandler**: Communication with the task service is asynchronous, so you should use a response handler that will be notified when the results are available.

When a message is invoked on the `TaskClient`, a message is created that will be sent to the server, and the server will execute the logic that implements the correct action.

The following code sample shows how to create a task client and interact with the task service to create, start and complete a task:

```
//Use UUID.randomUUID() to ensure the HornetQ Connector has a unique name
TaskClient client = new TaskClient(new
HornetQTaskClientConnector("HornetQConnector" + UUID.randomUUID(), new
HornetQTaskClientHandler(SystemEventListenerFactory.getSystemEventListener
())));
client.connect("127.0.0.1", 5446);
CommandBasedHornetQSHumanTaskHandler handler = new
CommandBasedHornetQSHumanTaskHandler(ksession);
handler.setClient(client);
handler.connect();
ksession.getWorkItemManager().registerWorkItemHandler("Human Task",
handler);

// adding a task

```

```

BlockingAddTaskResponseHandler addTaskResponseHandler = new
BlockingAddTaskResponseHandler();
Task task = ...;
client.addTask( task, null, addTaskResponseHandler );
long taskId = addTaskResponseHandler.getTaskId();

// getting tasks for user "bobba"
BlockingTaskSummaryResponseHandler taskSummaryResponseHandler =
    new BlockingTaskSummaryResponseHandler();
client.getTasksAssignedAsPotentialOwner("bobba", "en-UK",
taskSummaryResponseHandler);
List<TaskSummary> tasks = taskSummaryResponseHandler.getResults();

// starting a task
BlockingTaskOperationResponseHandler responseHandler =
    new BlockingTaskOperationResponseHandler();
client.start( taskId, "bobba", responseHandler );
responseHandler.waitTillDone(1000);

// completing a task
responseHandler = new BlockingTaskOperationResponseHandler();
client.complete( taskId, "bobba".getId(), null, responseHandler );
responseHandler.waitTillDone(1000);

```

[Report a bug](#)

10.3.5. User and Group Assignment

Tasks can be assigned to one or more users. If a task is assigned to one user, it will show up in that user's task list. If the task is assigned to more than one user, any one of those users can claim and execute the task. Tasks can also be assigned to groups, and any user who is a member of one of the groups the task is assigned to can claim the task.

Users and groups need to be registered before tasks can be assigned to them. This can be done dynamically.

```

EntityManagerFactory emf =
Persistence.createEntityManagerFactory("org.jbpm.task");
TaskService taskService = new TaskService(emf,
SystemEventListenerFactory.getSystemEventListener());
TaskServiceSession taskSession = taskService.createSession();
// now register new users and groups
taskSession.addUser(new User("userA"));
taskSession.addGroup(new Group("groupA"));

```

The human tasks service does not maintain the relationships between users and groups. A user group callback class must be created and listed in the `jbpm-human-task.war/WEB-INF/web.xml` file. The default implementation, `org.jbpm.task.service.DefaultUserGroupCallbackImpl`, assigns all users to all groups and is provided for testing purposes only.

To add the user group callback class open `jbpm-human-task.war/WEB-INF/web.xml` and add the class as in the following example:

```

<!-- use org.jbpm.task.service.DefaultUserGroupCallbackImpl to configure

```

```

sample user group callback for demo purpose-->
<init-param>
  <param-name>user.group.callback.class</param-name>
  <param-value>org.jbpm.task.service.DefaultUserGroupCallbackImpl</param-
value>
</init-param>

```

The `jbpm-human-task` module contains a `org.jbpm.task.RunTaskService` class in the `src/test/java` source folder that can be used to start a task server. It automatically adds users and groups as defined in `LoadUsers.mvel` and `LoadGroups.mvel` configuration files, which are located in the **server/profile/deploy/jbpm-human-task.war/WEB-INF/classes/org/jbpm/task/servlet/** directory.

[Report a bug](#)

10.3.6. Starting the Human Task Service

When using an independent human task service that the process engine communicates with, it is necessary to start the service:

```

org.jbpm.task.service.TaskService taskService = new
org.jbpm.task.service.TaskService(
  emf, SystemEventListenerFactory.getSystemEventListener());

TaskServiceSession taskServiceSession = taskService.createSession();

//adding users to TaskServiceSession
taskServiceSession.addUser(new User("Administrator"));
taskServiceSession.addUser(new User("jsmith"));

LocalTaskService localTaskService = new LocalTaskService( taskService );
humanTaskHandler = new SyncWSHumanTaskHandler( localTaskService, ksession
);
humanTaskHandler.setLocal( true );
humanTaskHandler.connect();
ksession.getWorkItemManager().registerWorkItemHandler( "Human Task",
humanTaskHandler );

//using HT API ...
List<TaskSummary> tasks =
localTaskService.getTasksAssignedAsPotentialOwner("jsmith", "en-US");

```

The task management component uses the Java Persistence API (JPA) to store all task information in a persistent manner. To configure persistence, edit the `persistence.xml` configuration file.

The following example shows how to use the task management component with hibernate and an in-memory H2 database. Please note that H2 databases are not supported in a production environment. The following is provided as an example only, and providing **create** as the value for the **hibernate.hbm2ddl.auto** property will result in the jBPM schemas being recreated every time the server is restarted. The indexes should only be created once and then this functionality should be disabled. This can be achieved by placing comment tags around the `hibernate.hbm2ddl.auto` property after the schema has been created:

```

<!-- <property name="hibernate.hbm2ddl.auto" value="create" /> -->

```



```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence
  version="1.0"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/persistence
     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
     http://java.sun.com/xml/ns/persistence/orm
     http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/persistence">

  <persistence-unit name="org.jbpm.task">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>org.jbpm.task.Attachment</class>
    <class>org.jbpm.task.Content</class>
    <class>org.jbpm.task.BooleanExpression</class>
    <class>org.jbpm.task.Comment</class>
    <class>org.jbpm.task.Deadline</class>
    <class>org.jbpm.task.Comment</class>
    <class>org.jbpm.task.Deadline</class>
    <class>org.jbpm.task.Delegation</class>
    <class>org.jbpm.task.Escalation</class>
    <class>org.jbpm.task.Group</class>
    <class>org.jbpm.task.I18NText</class>
    <class>org.jbpm.task.Notification</class>
    <class>org.jbpm.task.EmailNotification</class>
    <class>org.jbpm.task.EmailNotificationHeader</class>
    <class>org.jbpm.task.PeopleAssignments</class>
    <class>org.jbpm.task.Reassignment</class>
    <class>org.jbpm.task.Status</class>
    <class>org.jbpm.task.Task</class>
    <class>org.jbpm.task.TaskData</class>
    <class>org.jbpm.task.SubTasksStrategy</class>
    <class>org.jbpm.task.OnParentAbortAllSubTasksEndStrategy</class>
    <class>org.jbpm.task.OnAllSubTasksEndParentEndStrategy</class>
    <class>org.jbpm.task.User</class>

    <properties>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.connection.driver_class"
value="org.h2.Driver"/>
      <property name="hibernate.connection.url" value="jdbc:h2:mem:mydb"
/>
      <property name="hibernate.connection.username" value="sa"/>
      <property name="hibernate.connection.password" value="sasa"/>
      <property name="hibernate.connection.autocommit" value="false" />
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="create" />
      <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>

```

[Report a bug](#)

10.3.7. Starting the Human Task Service as a Web Application

The human task service can be started as a web application to simplify deployment. As part of the application configuration, users can select a number of settings to be applied on startup. Configuration is done via `web.xml` which is located in the `jbpm-human-task-war/WEB-INF/` directory by setting init parameters of the `HumanTaskServiceServlet`.

The following is a list of the supported parameters and their meaning:

General Settings:

- `task.persistence.unit`: The name of persistence unit that will be used to build `EntityManagerFactory` (default `org.jbpm.task`).
- `user.group.callback.class`: The implementation of `UserGroupCallback` interface to be used to resolve users and groups (default `DefaultUserGroupCallbackImpl` which is provided for testing purposes).
- `escalated.deadline.handler.class`: The implementation of `EscalatedDeadlineHandler` interface to be used to handle escalations and notifications (default `DefaultEscalatedDeadlineHandler`).
- `user.info.class`: The implementation of `UserInfo` interface to be used to resolve user/group information such as email address and preferred language.
- `load.users`: This specifies the location of a file that will be used to initially populate task server db with users. It accepts two types of files: MVEL and properties; It must be suffixed with `.mvel` or `.properties`. Location of the file can be either on classpath (with prefix `classpath:`) or valid URL. NOTE: That with custom user files, Administrator user must always be present.
- `load.groups`: This specifies the location of a file that will be used to initially populate task server db with groups. It accepts two types of files: MVEL and properties; the file must be suffixed with `.mvel` or `.properties`. Location of the file can be either on classpath (with prefix `classpath:`) or valid URL.

Transport Settings:

- `active.config`: The main parameter that controls what transport is configured for Task Server. By default this is set to `HornetQ`, but it also accepts `Mina` and `JMS`.

Apache Mina:

- `mina.host`: The host/ip address used to bind Apache Mina server (localhost).
- `mina.port`: The port used to bind Apache Mina server (default 9123).

HornetQ:

- `hornetq.port`: The port used to bind HornetQ server (default 5446).

JMS:

- `JMSTaskServer.connectionFactory`: JNDI name of `QueueConnectionFactory` to look up (no default).

- `JMSTaskServer.transacted` : A boolean flag that indicates if JMS session will be transacted or not (no default).
- `JMSTaskServer.acknowledgeMode`: Acknowledgment mode (default `DUPS_OK_ACKNOWLEDGE`).
- `JMSTaskServer.queueName`: The name of JMS queue (no default).
- `JMSTaskServer.responseQueueName`: The name of JMS response queue (no default).

[Report a bug](#)

10.4. HUMAN TASK PERSISTENCE

10.4.1. Human Task Persistence

The model below is an entity relationship diagram (ERD) that shows the persistent entities used by the Human Task service.

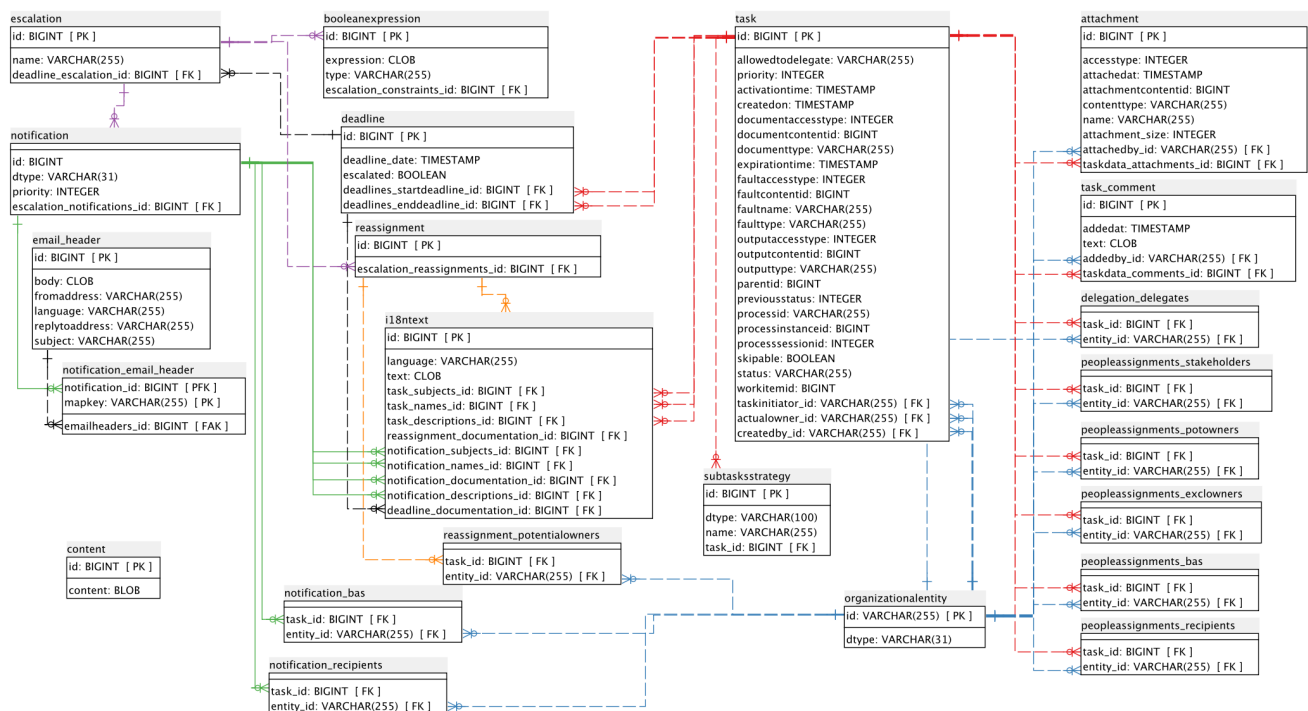


Figure 10.1. Human Task Service Data Model

The data model above is organized around 2 groups of entities:

- The ***task*** entity which represents the main information for a task. (See the right-hand side of the ERD above.)
- The ***deadline***, ***escalation***, and ***notification*** entities represent deadlines and escalations for tasks and notifications associated with those deadlines. (See the left-hand side of the ERD above.)

Two other important entities in the data model are the ***i18ntext*** and ***organizationalentity***.

- The *i18ntext* entity is used to store text which may be language related, such as names or descriptions entered by users.
- The **organizationalentity** entity represents a user.

The foreign key column in the tables in [Section 10.4.2, “Task Related Entities”](#) indicate whether or not a column in a database table has a foreign key constraint on it. The null column describes whether or not the database table column being describe can be null.

Please note, that if an entry is described as not allowing a null value and there is no associated entry, the column will contain the value **-1** or **0**.

[Report a bug](#)

10.4.2. Task Related Entities

The **task** entity contains the information for describing a task.

Table 10.1. Task

Field	Description	Null	Foreign Key
id	The primary key of the task identity	✗	✗
priority	The priority of the task	✗	✗
allowedtodelegate	The group this task may be delegated to	✓	✗
status	The status of the task	✓	✗
previousstatus	The previous status of the task	✓	✗
actualowner_id	The ID of the organizational entity who owns the task	✗	✓
createdby_id	The ID of the organizational entity who created the task	✗	✓
createdon	The timestamp describing when this task was created	✓	✗

Field	Description	Null	Foreign Key
activationtime	The timestamp describing when this task was activated	✓	✗
expirationtime	The timestamp describing when this task will expire	✓	✗
skipable	Whether or not this task may be skipped	✗	✗
workitemid	The ID of the work item associated with this task (see jBPM core schema)	✗	✗
processinstanceid	The ID of the process instance associated with this task (see jBPM core schema)	✗	✗
documentaccessstype	How a document associated with the task can be accessed	✓	✗
documenttype	The type of data in the document	✓	✗
documentcontentid	The ID of the content entity containing the document data	✗	✗
outputaccessstype	How the output document associated with the task can be accessed	✓	✗
outputtype	The type of data in the output document	✓	✗
outputcontentid	The ID of the content entity containing the output document data	✗	✗
faultname	The name of the fault generated, if a fault occurs	✓	✗

Field	Description	Null	Foreign Key
faultaccesstype	How the document associated with the fault can be accessed	✓	✗
faulttype	The type of data in the fault document	✓	✗
faultcontentid	The ID of the content entity containing the fault document data	✗	✗
parentid	This is the ID of the parent task	✗	✗
processid	The name (ID) of the associated process	✓	✗
processsessionid	The ID of the associated (knowledge) session	✗	✗
taskinitiator_id	The ID of the organizational entity who created the task	✗	✓

The **subtasksstrategy** entity is used to save the strategy that describes how parent and sub-tasks should react when either parent or sub-tasks are ended.

Table 10.2. SubTasksStrategy

Field	Description	Null	Foreign Key
id	The primary key	✗	✗
dtype	A discriminator column	✗	✗
name	The name of the strategy	✓	✗
task_id	The primary key of the associated task	✗	✓

The **organizationalentity** entity is extended to represent the different people assignments that are part of the task.

Table 10.3. OrganizationalEntity

Field	Description	Null
id	The primary key	✗
dtype	The discriminator column	✗

The **attachment** entity describes attachments that have been added to the task.

Table 10.4. Attachment

Field	Description	Null	Foreign Key
id	The primary key	✗	✗
name	The (file) name of the attachment	✓	✗
accesstype	How the attachment can be accessed	✓	✗
attachedat	When the attachment was attached to the task	✓	✗
attachment_size	The size (in bytes) of the attachment	✓	✗
attachmentcontentid	The ID of the content entity storing the raw data of the attachment	✗	✗
contenttype	The MIME type of the attachment data	✓	✗
attachedby_id	The ID of the organizationalentity entity that attached the attachment	✗	✓
taskdata_attachments_id	The ID of the task entity to which this attachment belongs	✗	✓

The **task_comment** entity describes comments added to tasks.

Table 10.5. task_comment

Field	Description	Null	Foreign Key
id	The primary key	✗	✗
addedat	The timestamp of when the comment was added to the task	✓	✗
text	The text of the comment	✓	✗
addedby_id	The primary key of the associated organizationalentity entity	✗	✓
taskdata_comments_id	The primary key of the associated task entity	✗	✓

The **delegation_delegates** table is a join table for relationships between the **task** entity and the **organizationalentity**.

Table 10.6. delegation_delegates

Field	Description	Null	Foreign Key
task_id	The primary key of the associated task	✗	✓
entity_id	The primary key of the associated organizationalentity	✗	✓

The **peopleassignments_stakeholders** table is a join table that describes which **organizationalentity** entities are *task stakeholders* of a particular task.

Table 10.7. peopleassignments_stakeholders

Field	Description	Null	Foreign Key
task_id	The primary key of the associated task entity	✗	✓

Field	Description	Null	Foreign Key
entity_id	The primary key of the associated organizationalentity entity	✗	✓

The **peopleassignments_potowners** table is a join table that describes which **organizationalentity** entities are *potential* owners of a particular task.

Table 10.8. peopleassignments_potowners

Field	Description	Null	Foreign Key
task_id	The primary key of the associated task entity	✗	✓
entity_id	The primary key of the associated organizationalentity entity	✗	✓

The **peopleassignments_exclowners** table is a join table that describes which **organizationalentity** entities are the *excluded* owners of a particular task.

Table 10.9. peopleassignments_exclowners

Field	Description	Null	Foreign Key
task_id	The primary key of the associated task entity	✗	✓
entity_id	The primary key of the associated organizationalentity entity	✗	✓

The **peopleassignments_bas** table is a join table that describes which **organizationalentity** entities are *business administrators* of a particular task.





Table 10.10. peopleassignments_bas

Field	Description	Null	Foreign Key
task_id	The primary key of the associated task entity	✗	✓

Field	Description	Null	Foreign Key
entity_id	The primary key of the associated organizationalentity entity		

The **peopleassignments_recipients** table is a join table that describes which **organizationalentity** entities are *notification recipients* for a particular task.

Table 10.11. peopleassignments_recipients

Field	Description	Null	Foreign Key
task_id	The primary key of the associated task entity		
entity_id	The primary key of the associated organizationalentity entity		







[Report a bug](#)

10.4.3. Deadline, Escalation, and Notification Related Entities

The following paragraphs and tables describe the group of entities having to do with deadline, escalation, and notification information.

The **deadline** entity represents a deadline for a task.

Table 10.12. deadline

Field	Description	Null	Foreign Key
id	The primary key		
deadline_date	The deadline date		
escalated	Whether or not the deadline has been escalated		

Field	Description	Null	Foreign Key
deadlines_startdeadline_id	The ID of the associated task entity which uses this deadline as its start deadline.	✗	✓
deadlines_enddeadline_id	The ID of the associated task entity which uses this deadline as its end deadline.	✗	✓

The **escalation** entity describes an escalation action that should be taken for a particular deadline.

Table 10.13. escalation

Field	Description	Null	Foreign Key
id	The primary key	✗	✗
name	The name of the escalation event	✓	✗
deadline_escalation_id	The ID of the associated deadline entity	✗	✓

The **booleanexpression** entity represents an expression that evaluates to a boolean. These expressions are used to determine if a constraint should be applied.

Table 10.14. booleanexpression

Field	Description	Null	Foreign Key
id	The primary key	✗	✗
expression	The expression text	✓	✗
type	The type of expression	✓	✗
escalation_constraints_id	The ID of the escalation constraint used on the expression	✗	✓

The **notification** entity describes a notification generated by an escalation action.

Table 10.15. notification

Field	Description	Null	Foreign Key
id	The primary key	✗	✗
dtype	The discriminator column	✗	✗
priority	The priority of the notification	✗	✗
escalation_notifications_id	The ID of the associated escalation entity	✗	✓







The **email_header** entity describes an email that will be sent as part of a notification.

Table 10.16. email_header

Field	Description	Null
id	The primary key	✗
fromaddress	The email address the e-mail is sent from	✓
replytoaddress	The reply-to address used in the e-mail	✓
language	The language the email is written in	✓
subject	The subject of the email	✓
body	The body of the email	✓





The **notification_email_header** table is a join table that describes and qualifies which **email_header** entities are part of a notification.

Table 10.17. notification_email_header

Field	Description	Null	Foreign Key
notification_id	Together with the mapkey , this field is part of the primary key. This field refers to the notification entity that the email_header is associated with		
mapkey	Together with the mapkey , this field is part of the primary key. This field describes what the type is of the associated email_header		
emailheaders_id	The ID of the associated email_header entity		

The **reassignment** entity describes reassignments associated with escalations.



Table 10.18. reassignment

Field	Description	Null	Foreign Key
id	The primary key		
escalation_reassignments_id	The ID of the associated escalation entity		

The **reassignments_potentialowners** table is a join table that describes which **organizationalentity** entities are potential owners if a reassignment happens as part of an escalation.





Table 10.19. reassignment_potentialowners

Field	Description	Null	Foreign Key
task_id	The primary key of the associated reassignment entity		

Field	Description	Null	Foreign Key
entity_id	The primary key of the associated organizationalentity entity		





The **notification_bas** table is a join table that describes which *business administrators* will be notified by a **notification**.

Table 10.20. notification_bas

Field	Description	Null	Foreign Key
task_id	The primary key of the associated notification entity		
entity_id	The primary key of the associated organizationalentity entity		


The **notification_recipients** table is a join table that describes which **recipients** entities will be received a **notification**.

Table 10.21. notification_recipients

Field	Description	Null	Foreign Key
task_id	The primary key of the associated notification entity		
entity_id	The primary key of the associated organizationalentity entity		

The **content** entity represents the content of a document, output document, fault or other object.

Table 10.22. content

Field	Description	Null
id	The primary key	







Field	Description	Null
content	The content data	✗

The **i18ntext** entity is used by a number of other entities to store text fields. The **deadline**, **notification**, **reassignment**, and **task** entities use this entity to store descriptions, subjects, names, and other documentation.

Foreign keys can not be set to null, and any foreign key that is not being used will be set to 0.

Table 10.23. i18ntext

Field	Description	Null	Foreign Key
id	The primary key	✗	✗
language	The language of the text	✓	✗
text	The text	✓	✗
task_subjects_id	The ID of the task entity that this subject refers to	✗	✓
task_names_id	The ID of the task entity this name refers to	✗	✓
task_descriptions_id	The ID of the task entity this description refers to	✗	✓
reassignment_documentation_id	The ID of the reassignment entity this documentation refers to	✗	✓
notification_subjects_id	The ID of the notification entity this subject refers to	✗	✓
notification_names_id	The ID of the notification entity this name refers to	✗	✓

Field	Description	Null	Foreign Key
notification_documentation_id	The ID of the notification entity this documentation refers to		
notification_descriptions_id	The ID of the notification entity this description refers to		
deadline_documentation_id	The ID of the deadline entity this documentation refers to		

[Report a bug](#)

CHAPTER 11. TESTING AND DEBUGGING

11.1. UNIT TESTING

11.1.1. Unit Testing

Business Processes should be designed at a high level with no implementation details; however, just like other development artifacts, they still have a lifecycle; since business processes can be updated dynamically, it is important that they are tested.

Unit tests are conducted to ensure processes behave as expected in specific use cases, for example, to test the output based on the specific input. The helper class `JbpmJUnitTestCase` (in the `jbpm-test` module) has been included to simplify unit testing. `JbpmJUnitTestCase` provides the following:

- Helper methods to create a new knowledge base and session for a given set of processes.
- Assert statements to check:
 - The state of a process instance (active, completed, aborted).
 - Which node instances are currently active.
 - Which nodes have been triggered (to check the path that has been followed).
 - The value of variables.

The image below contains a start event, a script task, and an end event. Within the example junit Test, a new session is created, the process is started, and the process instance is verified based on successful completion. It also checks whether these three nodes have been executed.

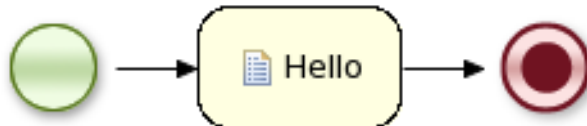


Figure 11.1. Example Hello World Process

Example 11.1. example junit Test

```

public class MyProcessTest extends JbpmJUnitTestCase {

    public void testProcess() {
        // create your session and load the given process(es)
        StatefulKnowledgeSession ksession =
createKnowledgeSession("sample.bpmn");
        // start the process
        ProcessInstance processInstance =
ksession.startProcess("com.sample.bpmn.hello");
        // check whether the process instance has completed successfully
        assertProcessInstanceCompleted(processInstance.getId(),
ksession);
        // check whether the given nodes were executed during the
process execution
        assertNodeTriggered(processInstance.getId(), "StartProcess",
  
```

```
"Hello", "EndProcess");  
    }  
}
```

[Report a bug](#)

11.1.2. Helper Methods to Create Sessions

Several methods are provided to simplify the creation of a knowledge base and a session to interact with the engine:

createKnowledgeBase(String... process):

Returns a new knowledge base containing all the processes in the given filenames (loaded from the classpath).

createKnowledgeBase(Map<String, ResourceType> resources):

Returns a new knowledge base containing all the resources from the given filenames (loaded from the classpath).

createKnowledgeBaseGuvnor(String... packages):

Returns a new knowledge base containing all the processes loaded from Guvnor (the process repository) from the given packages.

createKnowledgeSession(KnowledgeBase kbase):

Creates a new stateful knowledge session from the given knowledge base.

restoreSession(StatefulKnowledgeSession ksession, boolean noCache):

Completely restores this session from the database; it can be used to recreate a session to simulate a critical failure and test recovery. If noCache is true, the existing persistence cache will not be used to restore the data.

[Report a bug](#)

11.1.3. Assertions

The following assertions are provided to simplify testing the current state of a process instance:

assertProcessInstanceActive(long processInstanceId, StatefulKnowledgeSession ksession):

Checks whether the process instance with the given ID is still active.

assertProcessInstanceCompleted(long processInstanceId, StatefulKnowledgeSession ksession):

Checks whether the process instance with the given ID has completed successfully.

assertProcessInstanceAborted(long processInstanceId, StatefulKnowledgeSession ksession):

Checks whether the process instance with the given ID was aborted.

assertNodeActive(long processInstanceId, StatefulKnowledgeSession ksession, String... name):

Checks whether the process instance with the given ID contains at least one active node with the given node name (for each of the given names).

assertNodeTriggered(long processInstanceId, String... nodeNames):

Checks for each given node name whether a node instance was triggered (but not necessarily active anymore) during the execution of the process instance with the given node name (for each of the given names).

getVariableValue(String name, long processInstanceId, StatefulKnowledgeSession ksession):

Retrieves the value of the variable with the given name from the given process instance; it can then be used to check the value of process variables:

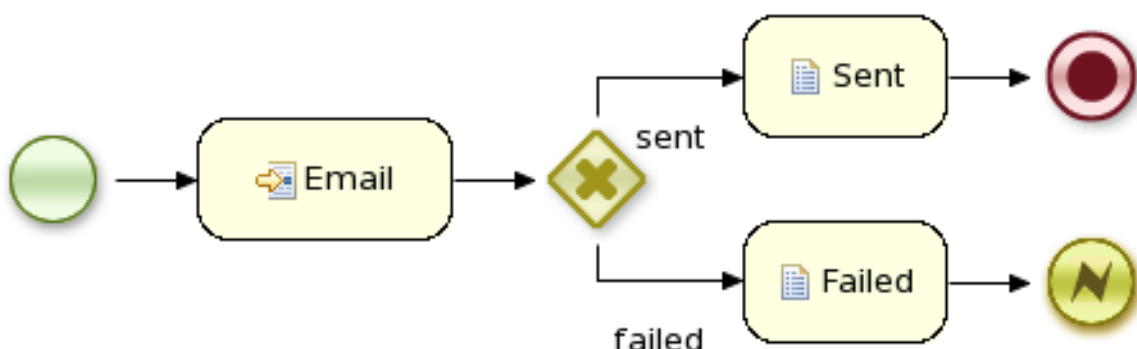
[Report a bug](#)

11.1.4. Testing Integration with External Services

Using domain-specific processes makes it possible to use testing handlers to verify whether or not specific services are requested correctly.

A `TestWorkItemHandler` is provided by default that can be registered to collect all work items (each work item represents one unit of work, for example, sending a specific email or invoking a specific service, and it contains all the data related to that task) for a given type. The test handler can be queried during unit testing to check whether specific work was actually requested during the execution of the process and that the data associated with the work was correct.

The following example describes how a process that sends an email could be tested. The test case tests whether an exception is raised when the email could not be sent (which is simulated by notifying the engine that sending the email could not be completed). The test case uses a test handler that simply registers when an email was requested and the data associated with the request. When the engine is notified the email could not be sent (using `abortWorkItem(..)`), the unit test verifies that the process handles this case successfully by logging this and generating an error, which aborts the process instance in this case.



```
public void testProcess2() {
    // create your session and load the given process(es)
    StatefulKnowledgeSession ksession =
    createKnowledgeSession("sample2.bpmn");
    // register a test handler for "Email"
    TestWorkItemHandler testHandler = new TestWorkItemHandler();
```

```

    ksession.getWorkItemManager().registerWorkItemHandler("Email",
testHandler);
    // start the process
    ProcessInstance processInstance =
ksession.startProcess("com.sample.bpmn.hello2");
    assertProcessInstanceActive(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "StartProcess", "Email");
    // check whether the email has been requested
    WorkItem workItem = testHandler.getWorkItem();
    assertNotNull(workItem);
    assertEquals("Email", workItem.getName());
    assertEquals("me@mail.com", workItem.getParameter("From"));
    assertEquals("you@mail.com", workItem.getParameter("To"));
    // notify the engine the email has been sent
    ksession.getWorkItemManager().abortWorkItem(workItem.getId());
    assertProcessInstanceAborted(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "Gateway", "Failed",
"Error");
}

```

[Report a bug](#)

11.1.5. Configuring Persistence

By default, persistence is turned off and processes exist in memory.

To turn persistence on, pass a boolean to the super constructor when creating the test case and `setPersistence` to true as shown below:

```

public class MyProcessTest extends JbpmJUnitTestCase {

    public MyProcessTest() {
        // configure this test to use persistence
        super(true);
        setPersistence(true);
    }
    ...
}

```

[Report a bug](#)

11.2. DEBUGGING

11.2.1. Debugging

JBoss Developer Studio can be used to debug processes and examine the state of the running processes.

[Report a bug](#)

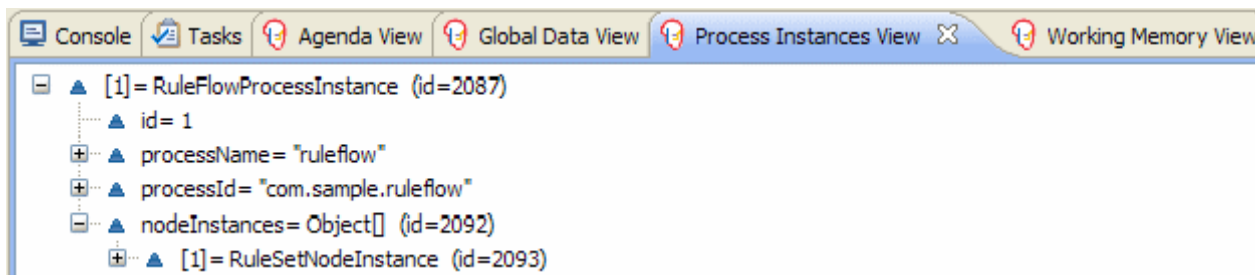
11.2.2. The Process Instances View

The process instances view shows the currently running process instances.

To open the process instances viewer, select **Window** → **Show View** → **Other**, then select **Drools** → **Process Instances**.

The Sample Process Instances View below shows that there is currently one running process (instance), currently executing one node instance, i.e. business rule task. When double-clicking a process instance, the process instance viewer will graphically display the progress of the process instance.

Example 11.2. Sample Process Instances View



[Report a bug](#)

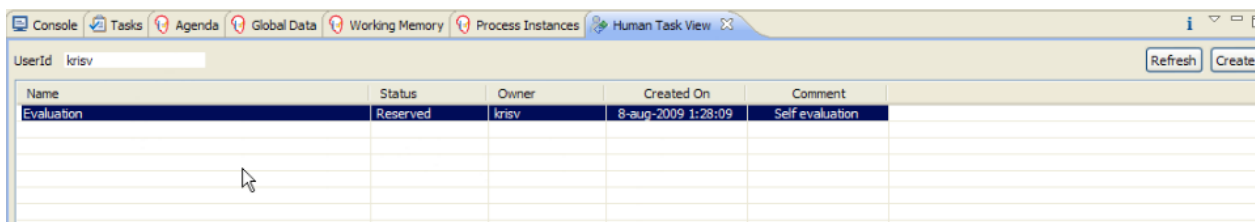
11.2.3. The Human Task View

The Human Task View can connect to a running human task service and request the relevant tasks for a particular user (i.e. the tasks where the user is either a potential owner or the tasks that the user already claimed and is executing). The life cycle of these tasks can then be executed, i.e. claiming or releasing a task, starting or stopping the execution of a task, completing a task, etc.

To open the human task viewer, select **Window** → **Show View** → **Other**, then select **jBPM Task** → **Human Task View**.

To configure the task service to connect to, select **Window** → **Preferences** → **Drools Tasks** and enter the IP address, port, and language.

Example 11.3. Sample Human Task View



[Report a bug](#)

11.2.4. The Audit View

The audit view shows the audit log, which is a log of all events that were logged from the session. To create a logger, use the KnowledgeRuntimeLoggerFactory to create a new logger and attach it to a session. Note that using a threaded file logger will save the audit log to the file system at regular






intervals, and the audit viewer will then be able to show the latest state. The Threaded File Logger below shows an example with the audit log file and the interval (in milliseconds) specified.

Example 11.4. Threaded File Logger

```
KnowledgeRuntimeLogger logger = KnowledgeRuntimeLoggerFactory
    .newThreadedFileLogger(ksession, "logdir/mylogfile", 1000);
// do something with the session here
logger.close();
```

To open the audit view, select **Window** → **Show View** → **Audit**.

To open up an audit tree in the audit view, open the selected log file in the audit view or simply drag the file into the audit view. A tree-based view is generated based on the audit log. An event is shown as a sub node of another event if the child event is caused by (a direct consequence of) the parent event:

- ▼  RuleFlow started: ruleflow[com.sample.ruleflow]
- ▼  RuleFlow node triggered: Start in process ruleflow[com.sample.ruleflow]
- ▼  RuleFlow node triggered: Hello in process ruleflow[com.sample.ruleflow]
- ▼  RuleFlow node triggered: End in process ruleflow[com.sample.ruleflow]
-  RuleFlow completed: ruleflow[com.sample.ruleflow]

[Report a bug](#)

CHAPTER 12. BUSINESS ACTIVITY MONITORING

12.1. BUSINESS ACTIVITY MONITORING

Processes should be monitored so that unexpected events or behaviors can be detected and responded to as soon as possible. Business Activity Monitoring is real-time process monitoring that provides the option to intervene directly or through automation, based on the analysis of the events.



IMPORTANT

Red Hat does not include a monitoring solution with JBoss BRMS 5.3.

[Report a bug](#)

CHAPTER 13. INTEGRATION

13.1. INTEGRATION

The Business Process Management engine can be integrated with other technologies.

[Report a bug](#)

13.2. OSGI

OSGi is a dynamic module system for declarative services. Each jar in OSGi is called a bundle and has its own classloader. Each bundle specifies the packages it exports (makes publicly available) and which packages it imports (external dependencies). OSGi will use this information to wire the classloaders of different bundles together; the key distinction is you don't specify what bundle you depend on or have a single monolithic classpath; instead, you specify your package import and version, and OSGi attempts to satisfy this from the available bundles.

All core jBPM jars (and core dependencies) are OSGi-enabled. That means that they contain MANIFEST.MF files (in the META-INF directory) that describe their dependencies. These manifest files are automatically generated by the build, and you can plug these jars directly into an OSGi environment.

The following jBPM jars are OSGi-enabled:

- jbpm-flow
- jbpm-flow-builder
- jbpm-bpmn2

The example below looks up the necessary services in an OSGi environment using the service registry and creates a session that can then be used to start processes, signal events, etc.

Example 13.1. OSGi Example

```
ServiceReference serviceRef = bundleContext.getServiceReference(
ServiceRegistry.class.getName() );
ServiceRegistry registry = (ServiceRegistry) bundleContext.getService(
serviceRef );

KnowledgeBuilderFactoryService knowledgeBuilderFactoryService =
registry.get( KnowledgeBuilderFactoryService.class );
KnowledgeBaseFactoryService knowledgeBaseFactoryService = registry.get(
KnowledgeBaseFactoryService.class );
ResourceFactoryService resourceFactoryService = registry.get(
ResourceFactoryService.class );

KnowledgeBuilderConfiguration kbConf =
knowledgeBuilderFactoryService.newKnowledgeBuilderConfiguration( null,
getClass().getClassLoader() );
KnowledgeBuilder kbuilder =
knowledgeBuilderFactoryService.newKnowledgeBuilder( kbConf );
kbuilder.add( resourceFactoryService.newClassPathResource(
"MyProcess.bpmn", Dummy.class ), ResourceType.BPMN2 );
```



```

KnowledgeBaseConfiguration kbaseConf =
knowledgeBaseFactoryService.newKnowledgeBaseConfiguration( null,
getClass().getClassLoader() );
KnowledgeBase kbase = knowledgeBaseFactoryService.newKnowledgeBase(
kbaseConf );
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );

StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();

```

[Report a bug](#)

13.3. SPRING

A Spring XML configuration file can be used to easily define and configure knowledge bases and sessions in a Spring environment, making it possible to access a session and invoke processes from within a Spring application.

The Example Spring Configuration File below sets up a new session based on a knowledge base with one process definition loaded from the classpath.

Example 13.2. Example Spring Configuration File

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jbpm="http://drools.org/schema/drools-spring"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://drools.org/schema/drools-spring
org/drools/container/spring/drools-spring-1.2.0.xsd"

<jbpm:kbase id="kbase">
<jbpm:resources>
<jbpm:resource type="BPMN2" source="classpath:HelloWorld.bpmn2"/>
</jbpm:resources>
</jbpm:kbase>

<jbpm:ksession id="ksession" type="stateful" kbase="kbase" />

</beans>

```

The following code loads the above Spring configuration, retrieves the session, and starts the process:

```

ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("spring-conf.xml");
StatefulKnowledgeSession ksession = (StatefulKnowledgeSession)
context.getBean("ksession");
ksession.startProcess("com.sample.HelloWorld");

```

Note that you can also inject the session in one of your domain objects; for example, add the following fragment in the configuration file:

```
<bean id="myObject" class="org.jbpm.sample.MyObject">
  <property name="session" ref="ksession" />
</bean>
```

As a result, the session will be injected into the domain object and can then be accessed directly. For example:

```
public class MyObject {
  private StatefulKnowledgeSession ksession;
  public void setSession(StatefulKnowledgeSession ksession) {
    this.ksession = ksession;
  }
  public void doSomething() {
    ksession.startProcess("com.sample.HelloWorld");
  }
}
```

[Report a bug](#)

13.4. MAVEN

Maven integration is not supported by Red Hat in JBoss Enterprise BRMS 5.3.1 and the following is provided as an example only.

Procedure 13.1. Importing the Drools and jBPM Jar Files to the Local Maven Repository

1. Download the deployable package zip file from the Red Hat Customer Support Portal at <https://access.redhat.com>. Select **Downloads** → **Download your software** → **BRMS Platform**, then select the version and JBoss BRMS 5.3.1.
2. Extract the zip archive.
3. Extract the **jboss-brms-engine.zip** and **jboss-jbpm-engine.zip** archives.
4. Run the following maven commands to import the Drools and jBPM jar files to the local maven repository:

```
mvn install:install-file -Dfile=knowledge-api-5.3.1.BRMS.jar -
DgroupId=org.drools -DartifactId=knowledge-api -Dversion=5.3.1.BRMS
-Dpackaging=jar
mvn install:install-file -Dfile=drools-core-5.3.1.BRMS.jar -
DgroupId=org.drools -DartifactId=drools-core -Dversion=5.3.1.BRMS -
Dpackaging=jar
mvn install:install-file -Dfile=drools-compiler-5.3.1.BRMS.jar -
DgroupId=org.drools -DartifactId=drools-compiler -
Dversion=5.3.1.BRMS -Dpackaging=jar
mvn install:install-file -Dfile=drools-decisiontables-5.3.1.BRMS.jar -
DgroupId=org.drools -DartifactId=drools-decisiontables -
Dversion=5.3.1.BRMS -Dpackaging=jar
mvn install:install-file -Dfile=drools-templates-5.3.1.BRMS.jar -
```

```

DgroupId=org.drools -DartifactId=drools-templates -
Dversion=5.3.1.BRMS -Dpackaging=jar
mvn install:install-file -Dfile=drools-persistence-jpa-
5.3.1.BRMS.jar -DgroupId=org.drools -DartifactId=drools-persistence-
jpa -Dversion=5.3.1.BRMS -Dpackaging=jar
mvn install:install-file -Dfile=jbpm-flow-5.3.1.BRMS.jar -
DgroupId=org.jbpm -DartifactId=jbpm-flow -Dversion=5.3.1.BRMS -
Dpackaging=jar
mvn install:install-file -Dfile=jbpm-flow-builder-5.3.1.BRMS.jar -
DgroupId=org.jbpm -DartifactId=jbpm-flow-builder -
Dversion=5.3.1.BRMS -Dpackaging=jar
mvn install:install-file -Dfile=jbpm-bam-5.3.1.BRMS.jar -
DgroupId=org.jbpm -DartifactId=jbpm-bam -Dversion=5.3.1.BRMS -
Dpackaging=jar
mvn install:install-file -Dfile=jbpm-bpmn2-5.3.1.BRMS.jar -
DgroupId=org.jbpm -DartifactId=jbpm-bpmn2 -Dversion=5.3.1.BRMS -
Dpackaging=jar
mvn install:install-file -Dfile=jbpm-human-task-5.3.1.BRMS.jar -
DgroupId=org.jbpm -DartifactId=jbpm-human-task -Dversion=5.3.1.BRMS
-Dpackaging=jar
mvn install:install-file -Dfile=jbpm-persistence-jpa-5.3.1.BRMS.jar
-DgroupId=org.jbpm -DartifactId=jbpm-persistence-jpa -
Dversion=5.3.1.BRMS -Dpackaging=jar
mvn install:install-file -Dfile=jbpm-workitems-5.3.1.BRMS.jar -
DgroupId=org.jbpm -DartifactId=jbpm-workitems -Dversion=5.3.1.BRMS -
Dpackaging=jar

```



NOTE

The list of imported jar files above is not complete. Make sure you import all **drools**, **knowledge**, and **jbpm** jar files.

A Maven pom.xml file defines project dependencies, which are automatically retrieved when building the project with Maven. The following pom.xml file is an example that shows how to declare dependencies for different use cases:

```

<?xml version="1.0" encoding="utf-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>org.jbpm</groupId>
    <artifactId>jbpm-maven-example</artifactId>
    <name>jBPM Maven Project</name>
    <version>1.0-SNAPSHOT</version>
    <brms-version>5.3.1.BRMS<brms-version>
    ...
  <dependencies>
    <!-- core dependencies -->
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>knowledge-api</artifactId>
      <version>${brms-version}</version>

```

```
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-core</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.mvel</groupId>
  <artifactId>mvel2</artifactId>
  <version>2.1.Beta6</version>
</dependency>
<dependency>
  <groupId>com.thoughtworks.xstream</groupId>
  <artifactId>xstream</artifactId>
  <version>1.3.1</version>
</dependency>

<!-- required to compile DRL -->
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-compiler</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.jdt.core.compiler</groupId>
  <artifactId>ecj</artifactId>
  <version>3.5.1</version>
</dependency>
<dependency>
  <groupId>org.antlr</groupId>
  <artifactId>antlr-runtime</artifactId>
  <version>3.3</version>
  <exclusions>
    <exclusion>
      <groupId>org.antlr</groupId>
      <artifactId>stringtemplate</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<!-- required to compile decision tables -->
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-decisiontables</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-templates</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>net.sourceforge.jexcelapi</groupId>
  <artifactId>jxl</artifactId>
  <version>2.6.10</version>
</dependency>
```

```

<!-- required for jBPM5 processes -->
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-flow</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-flow-builder</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-bam</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-bpmn2</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-human-task</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-persistence-jpa</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-workitems</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-persistence-jpa</artifactId>
  <version>${brms-version}</version>
</dependency>
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>2.4.1</version>
</dependency>
</dependencies>
</project>

```

To use this as the basis for a project in Eclipse, either use M2Eclipse or use "mvn eclipse:eclipse" to generate eclipse .project and .classpath files based on this pom.

[Report a bug](#)

APPENDIX A. REVISION HISTORY

Revision 5.3.1-86.400

2013-10-31

Rüdiger Landmann

Rebuild with publican 4.0.0

Revision 5.3.1-86

Tue Dec 11 2012

L Carlon

Updated documentation for the JBoss Enterprise BRMS Platform 5.3.1 release.