

# jBPM User Guide

Version 6.1.0-SNAPSHOT

by *The jBPM team* [<http://www.jboss.org/jbpm>]

---

---

---

.....	vii
<b>1. Overview</b> .....	1
1.1. What is jBPM? .....	1
1.2. Overview .....	3
1.3. Core Engine .....	4
1.4. Eclipse Editor .....	5
1.5. Workbench web application .....	6
1.5.1. Process Designer .....	7
1.5.2. Form Modeler .....	7
1.5.3. Process and Task management .....	8
1.5.4. Business Activity Monitoring .....	9
1.6. Documentation .....	10
<b>2. Getting Started</b> .....	13
2.1. Downloads .....	13
2.2. Use with Maven, Gradle, Ivy, Buildr or ANT .....	13
2.3. Getting started .....	14
2.4. Community .....	14
2.5. Sources .....	15
2.5.1. License .....	15
2.5.2. Source code .....	15
2.5.3. Building from source .....	15
<b>3. Installer</b> .....	17
3.1. Prerequisites .....	17
3.2. Download the installer .....	17
3.3. Demo setup .....	17
3.3.1. Control options .....	18
3.4. 10-Minute Tutorial: Using the jBPM Console .....	19
3.5. 10-Minute Tutorial: Integrate Eclipse and Web tooling .....	20
3.6. Using your own database with jBPM .....	21
3.6.1. Introduction .....	21
3.6.2. Database setup .....	22
3.6.3. Configuration .....	22
3.6.4. Using a different database .....	25
3.7. jBPM data base schema scripts (DDL scripts) .....	28
3.8. jBPM installer script .....	28
3.9. What to do if I encounter problems or have questions? .....	30
3.10. Frequently asked questions .....	30
<b>4. Core Engine: API</b> .....	33
4.1. The jBPM API .....	34
4.1.1. Knowledge Base .....	34
4.1.2. Session .....	35
4.1.3. Correlation key and Correlation properties .....	37
4.1.4. Events .....	38
4.2. Knowledge-based API .....	40

4.3. RuntimeManager .....	41
4.4. Control parameters .....	43
<b>5. Core Engine: Basics .....</b>	<b>47</b>
5.1. Creating a process .....	47
5.1.1. Using the graphical BPMN2 Editor .....	47
5.1.2. Defining processes using XML .....	48
5.1.3. Defining Processes Using the Process API .....	50
5.2. Details of different process constructs: Overview .....	51
5.3. Details: Process properties .....	54
5.4. Details: Events .....	55
5.4.1. Start event .....	56
5.4.2. End events .....	57
5.4.3. Intermediate events .....	59
5.5. Details: Activities .....	62
5.5.1. Script task .....	62
5.5.2. Service task .....	63
5.5.3. User task .....	64
5.5.4. Reusable sub-process .....	66
5.5.5. Business rule task .....	67
5.5.6. Embedded sub-process .....	68
5.5.7. Multi-instance sub-process .....	69
5.6. Details: Gateways .....	70
5.6.1. Diverging gateway .....	70
5.6.2. Converging gateway .....	72
5.7. Using a process in your application .....	73
5.8. Other features .....	74
5.8.1. Data .....	74
5.8.2. Constraints .....	76
5.8.3. Action scripts .....	77
5.8.4. Events .....	79
5.8.5. Timers .....	80
5.8.6. Updating processes .....	81
5.8.7. Multi-threading .....	83
<b>6. Core Engine: BPMN 2.0 .....</b>	<b>87</b>
6.1. Business Process Model and Notation (BPMN) 2.0 specification .....	87
6.2. Supported elements / attributes .....	92
6.3. Examples .....	98
<b>7. Core Engine: Persistence and transactions .....</b>	<b>99</b>
7.1. Runtime State .....	99
7.1.1. Binary Persistence .....	100
7.1.2. Safe Points .....	103
7.1.3. Configuring Persistence .....	104
7.1.4. Transactions .....	109
7.1.5. Persistence and concurrency .....	111

---

7.2. Process Definitions .....	112
7.3. History Log .....	112
7.3.1. The jBPM Audit data model .....	112
7.3.2. Storing Process Events in a Database .....	115
7.3.3. Storing Process Events in a JMS queue for further processing .....	117
<b>8. Core Engine: Examples .....</b>	<b>119</b>
8.1. jBPM Examples .....	119
8.2. Examples .....	119
8.3. Unit tests .....	120

---



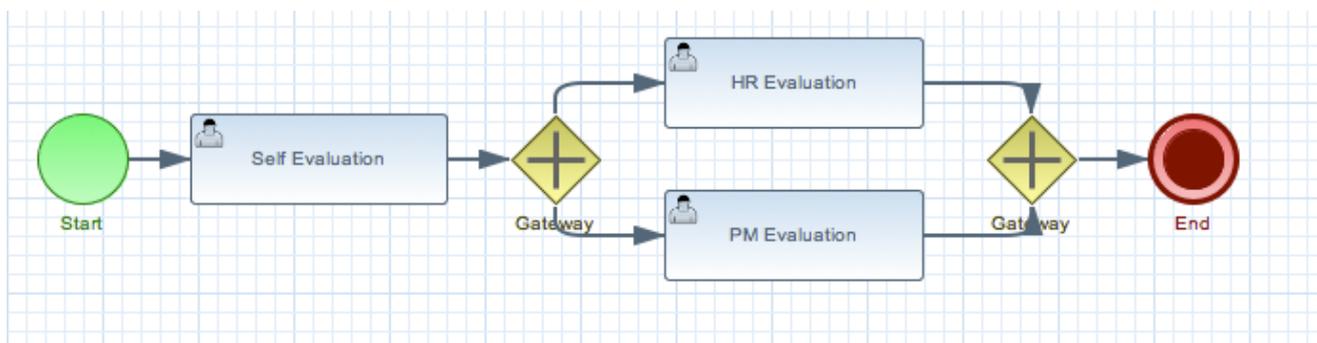
---

# Chapter 1. Overview

## 1.1. What is jBPM?

jBPM is a flexible Business Process Management (BPM) Suite. It is light-weight, fully open-source (distributed under Apache license) and written in Java. It allows you to model, execute, and monitor business processes throughout their life cycle.

A business process allows you to model your business goals by describing the steps that need to be executed to achieve those goals, and the order of those goals are depicted using a flow chart. This process greatly improves the visibility and agility of your business logic. jBPM focuses on executable business processes, which are business processes that contain enough detail so they can actually be executed on a BPM engine. Executable business processes bridge the gap between business users and developers as they are higher-level and use domain-specific concepts that are understood by business users but can also be executed directly.



The core of jBPM is a light-weight, extensible workflow engine written in pure Java that allows you to execute business processes using the latest BPMN 2.0 specification. It can run in any Java environment, embedded in your application or as a service.

On top of the core engine, a lot of features and tools are offered to support business processes throughout their entire life cycle:

- Eclipse-based and web-based editor to support the graphical creation of your business processes (drag and drop).
- Pluggable persistence and transactions based on JPA / JTA.
- Pluggable human task service based on WS-HumanTask for including tasks that need to be performed by human actors.
- Complete BPM life cycle management web console that allows:
  - Model - author your processes, rules, forms and other assets
  - Execute - build and deploy packages to runtime engine

- Work - work on assigned task, manage process instances, etc
- Monitor - keep track of the execution using Business Activity Monitoring capabilities
- Integration with Maven, Spring, OSGi, etc.

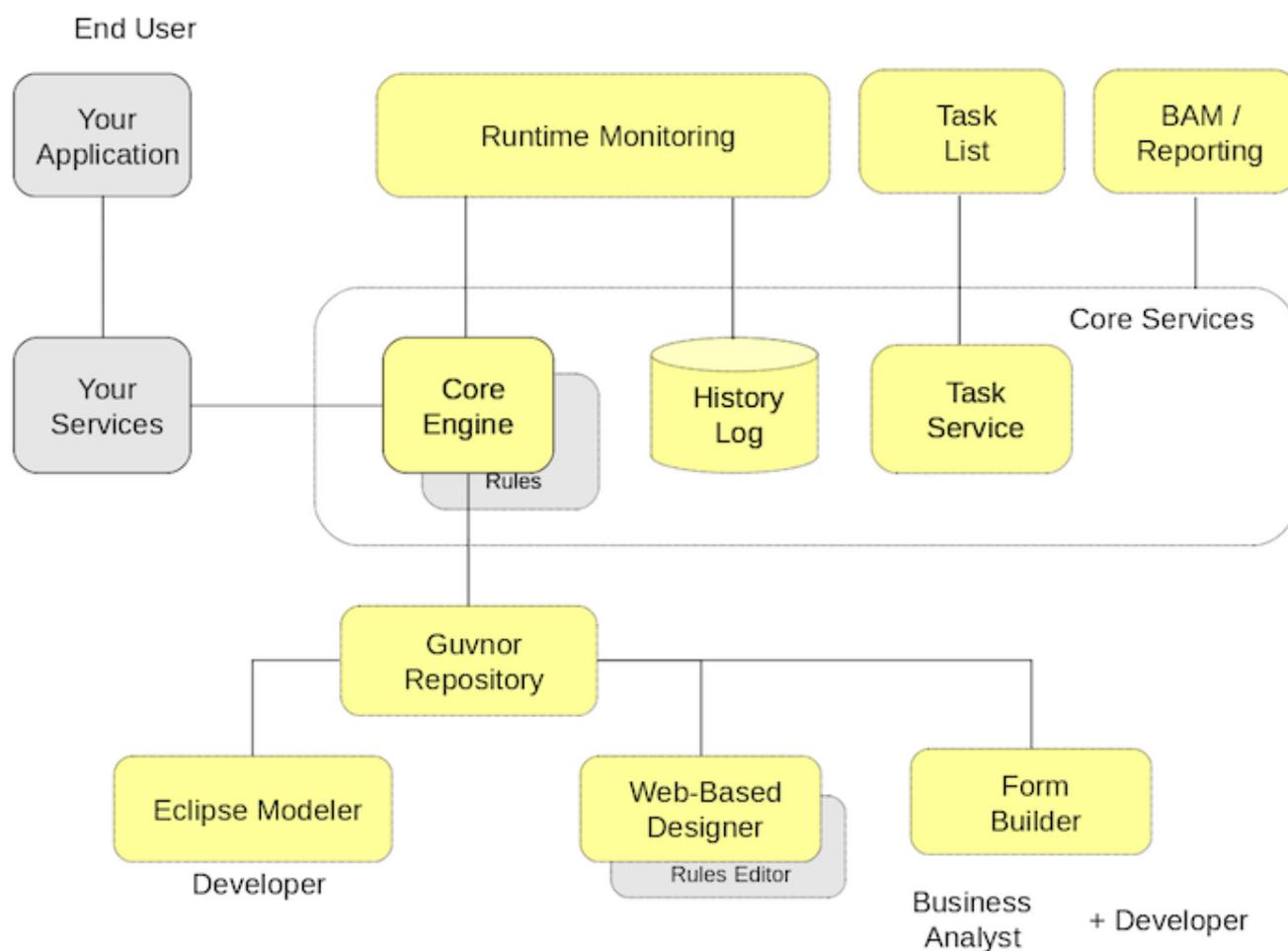
BPM creates the bridge between business analysts, developers and end users by offering process management features and tools in a way that both business users and developers like. Domain-specific nodes can be plugged into the palette, making the processes more easily understood by business users.

jBPM supports adaptive and dynamic processes that require flexibility to model complex, real-life situations that cannot easily be described using a rigid process. We bring control back to the end users by allowing them to control which parts of the process should be executed; this allows dynamic deviation from the process.

jBPM is not just an isolated process engine. Complex business logic can be modeled as a combination of business processes with business rules and complex event processing. jBPM can be combined with the Drools project to support one unified environment that integrates these paradigms where you model your business logic as a combination of processes, rules and events.

Apart from the core engine itself, there are quite a few additional (optional) components that you can use, like an Eclipse-based or web-based designer and a management console.

## 1.2. Overview



**Figure 1.1.**

This figure gives an overview of the different components of the jBPM project. jBPM can integrate with a lot of other services (and we've shown a few using grey boxes on the figure), but here we focus on the components that are part of the jBPM project itself.

- The process engine is the core of the project and is required if you want to execute business processes (all other components are optional, as indicated by the dashed border). Your application services typically invoke the core engine (to start processes or to signal events) whenever necessary.
- An optional core service is the history log; this will log all information about the current and previous state of all your process instances.
- Another optional core service is the human task service that will take care of the human task life cycle if human actors participate in the process.

- Two types of graphical editors are supported for defining your business processes:
  - The Eclipse plugin is an extension to the Eclipse IDE, targeted towards developers, and allows you to create business processes using drag and drop, advanced debugging, etc.
  - The web-based designer allows business users to manage business processes in a web-based environment. A web-based form builder also allows you to create, generate or edit forms related to those processes (to start the process or to complete one of the user tasks).
- The Guvnor repository is an optional component that can be used to store all your business processes. It supports collaboration, versioning, etc. There is integration with both the Eclipse plugin and web-based designer, supporting round-tripping between the different tools.
- The web-based management console allows business users to manage their runtime (manage business processes like start new processes, inspect running instances, etc.), to manage their task list and to perform Business Activity Monitoring (BAM) and see reports.

Each of the components are described in more detail below.

### 1.3. Core Engine

The core jBPM engine is the heart of the project. It's a light-weight workflow engine that executes your business processes. It can be embedded as part of your application or deployed as a service (possibly on the cloud). Its most important features are the following:

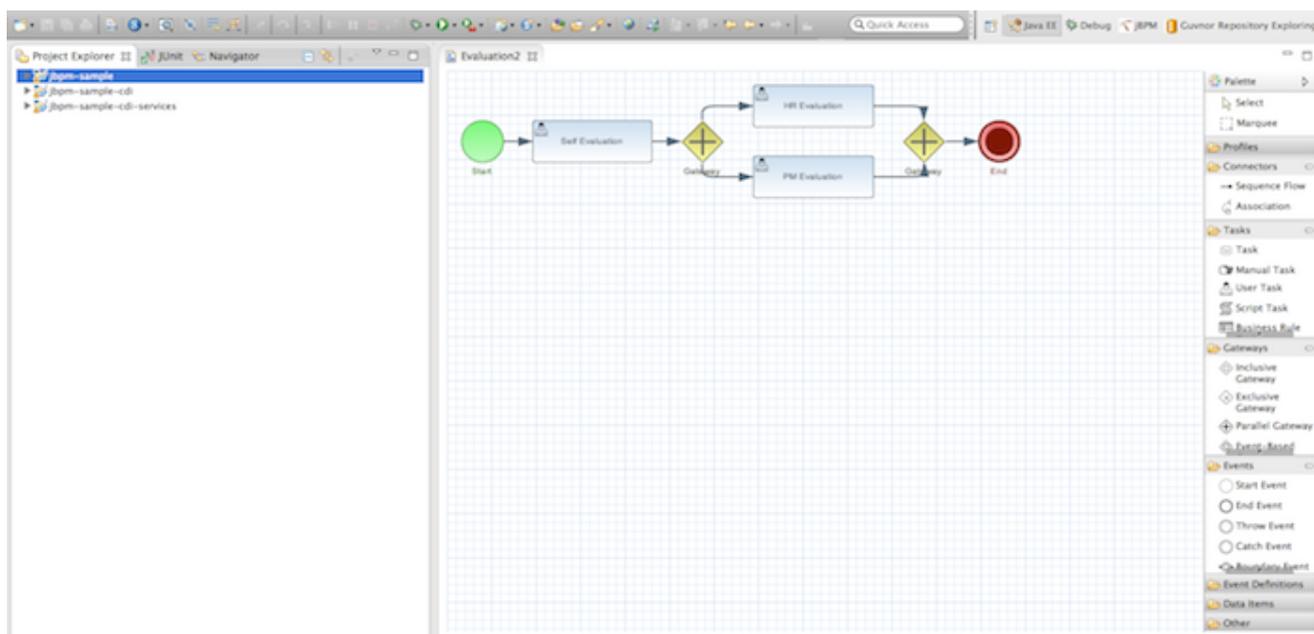
- Solid, stable core engine for executing your process instances.
- Native support for the latest BPMN 2.0 specification for modeling and executing business processes.
- Strong focus on performance and scalability.
- Light-weight (can be deployed on almost any device that supports a simple Java Runtime Environment; does not require any web container at all).
- (Optional) pluggable persistence with a default JPA implementation.
- Pluggable transaction support with a default JTA implementation.
- Implemented as a generic process engine, so it can be extended to support new node types or other process languages.
- Listeners to be notified of various events.
- Ability to migrate running process instances to a new version of their process definition

The core engine can also be integrated with a few other (independent) core services:

- The human task service can be used to manage human tasks when human actors need to participate in the process. It is fully pluggable and the default implementation is based on the WS-HumanTask specification and manages the life cycle of the tasks, task lists, task forms, and some more advanced features like escalation, delegation, rule-based assignments, etc.
- The history log can store all information about the execution of all the processes in the engine. This is necessary if you need access to historic information as runtime persistence only stores the current state of all active process instances. The history log can be used to store all current and historic states of active and completed process instances. It can be used to query for any information related to the execution of process instances, for monitoring, analysis, etc.

## 1.4. Eclipse Editor

The Eclipse editor is a plugin to the Eclipse IDE and allows you to integrate your business processes in your development environment. It is targeted towards developers and has some wizards to get started, a graphical editor for creating your business processes (using drag and drop) and a lot of advanced testing and debugging capabilities.



**Figure 1.2. Eclipse editor for creating BPMN2 processes**

It includes the following features:

- Wizard for creating a new jBPM project
- A graphical editor for BPMN 2.0 processes
- The ability to plug in your own domain-specific nodes

- Validation
- Runtime support (so you can select which version of jBPM you would like to use)
- Graphical debugging to see all running process instances of a selected session, to visualize the current state of one specific process instance, etc.

### 1.5. Workbench web application

Optionally, you can use one or more knowledge repositories to store your business processes (and other related artefacts). The web-based designer is integrated in the Guvnor repository, which is targeted towards business users and allows you to manage your processes separately from your application. It supports the following:

- A repository service to store your business processes and related artefacts, using a GIT repository, which supports versioning, remote accessing (as a file system), and using REST services.
- A web-based user interface to manage your business processes, targeted towards business users; it also supports the visualization (and editing) of your processes (the web-based designer is integrated here), but also categorisation, scenario testing, and deployment.
- Collaboration features to have multiple actors (for example business users and developers) work together on the same process definition.

Workbench application covers complete life cycle of BPM projects starting at authoring phase, going through implementation, execution and monitoring.

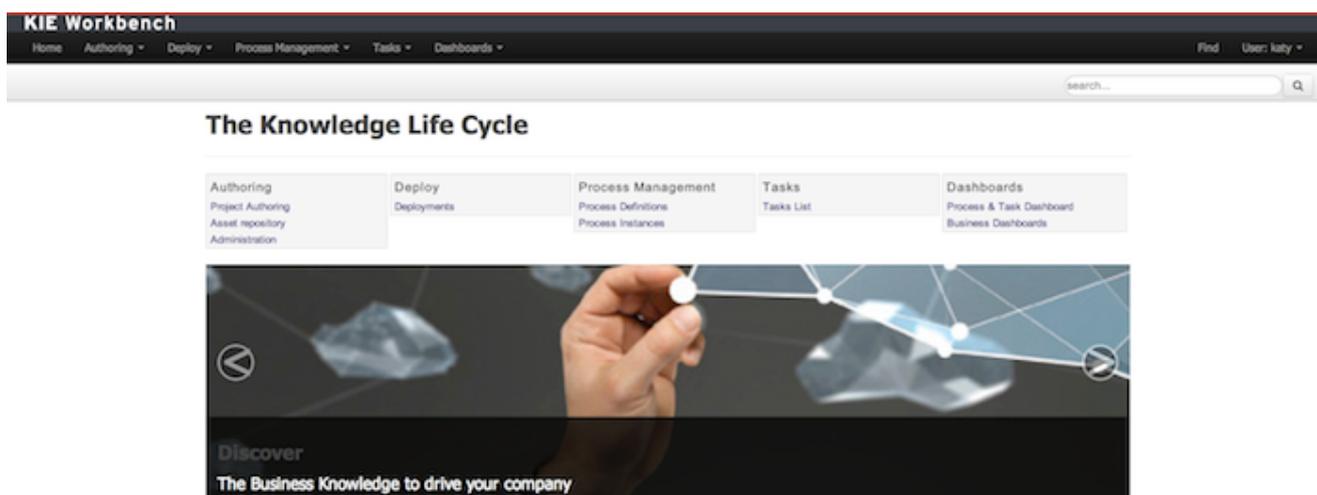
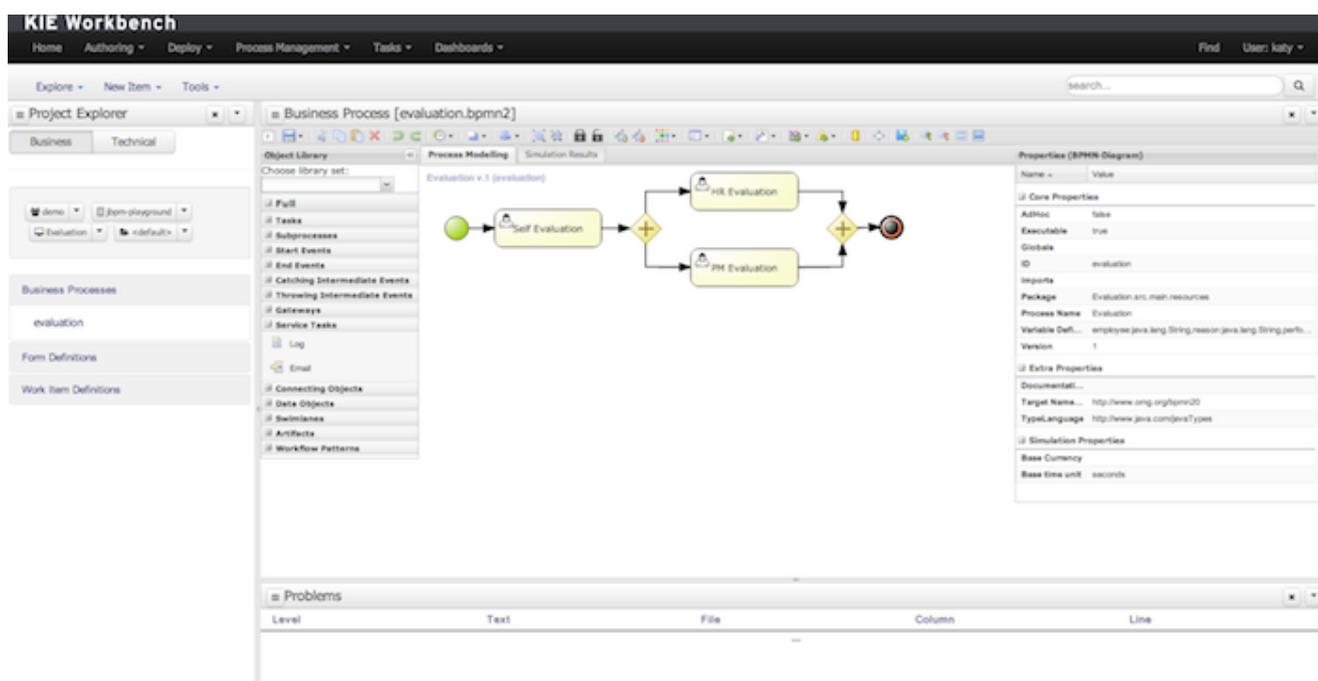


Figure 1.3. KIE workbench application

## 1.5.1. Process Designer

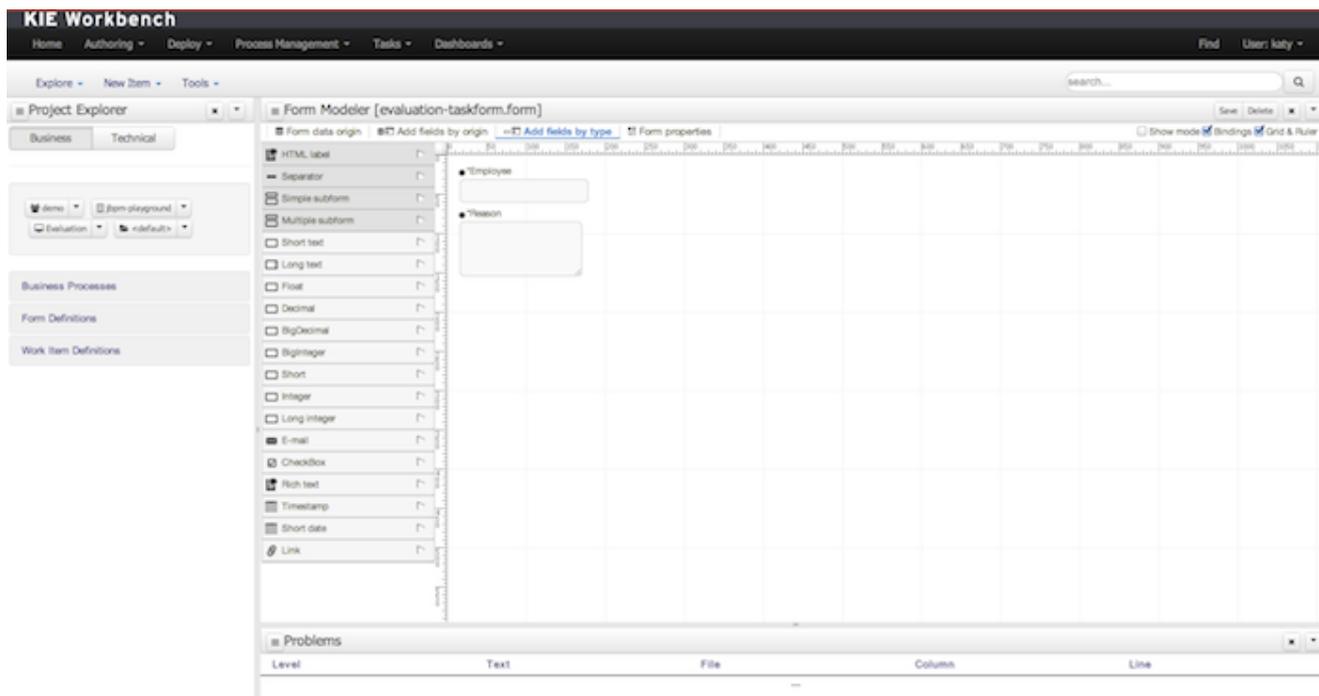
The web-based designer allows you to model your business processes in a web-based environment. It is targeted towards business users and offers a graphical editor for viewing and editing your business processes (using drag and drop), similar to the Eclipse plugin. It supports round-tripping between the Eclipse editor and the web-based designer.



**Figure 1.4. Web-based designer for creating BPMN2 processes**

## 1.5.2. Form Modeler

A web-based form modeler allows you to create, generate and/or edit your form (both for starting a process or completing a user task) using a WYSIWYG editor. By dragging and dropping various form elements into a panel and filling in the necessary details, task forms can be created by non-technical experts. It provides advanced capabilities for data mapping including complex type support (e.g. map form data to POJO)

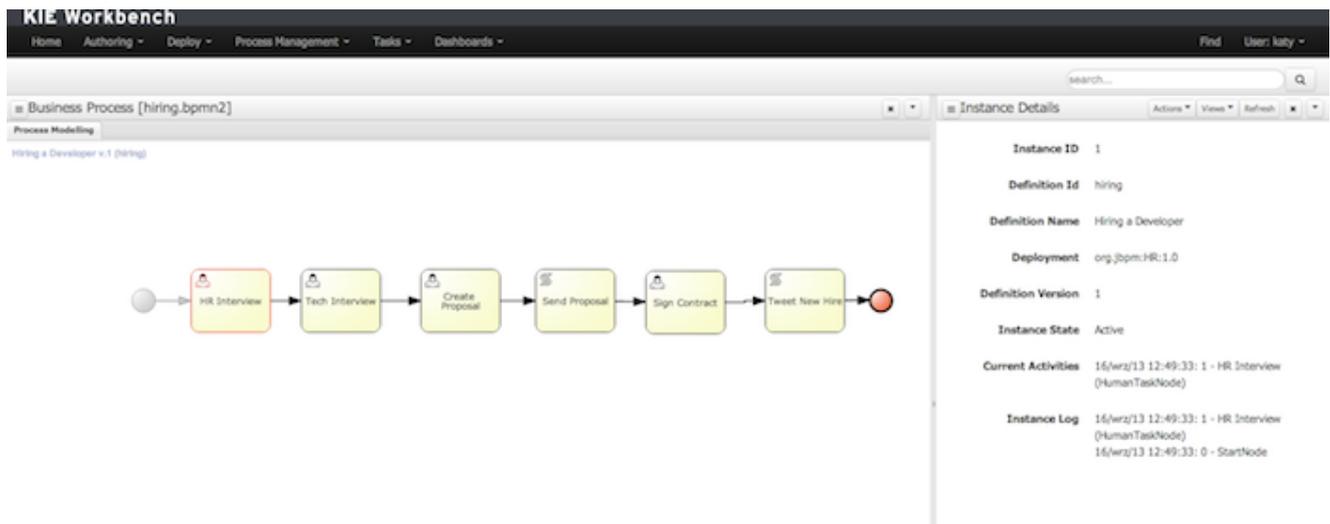


**Figure 1.5. Web-based form modeler**

### 1.5.3. Process and Task management

Business processes can be managed through a web-based management console. It is targeted towards business users and its main features are the following:

- Process instance management: the ability to start new process instances, get a list of running process instances, visually inspect the state of a specific process instances.
- Human task management: being able to get a list of all your current tasks (either assigned to you or that you might be able to claim), and completing tasks on your task list (using customizable task forms).

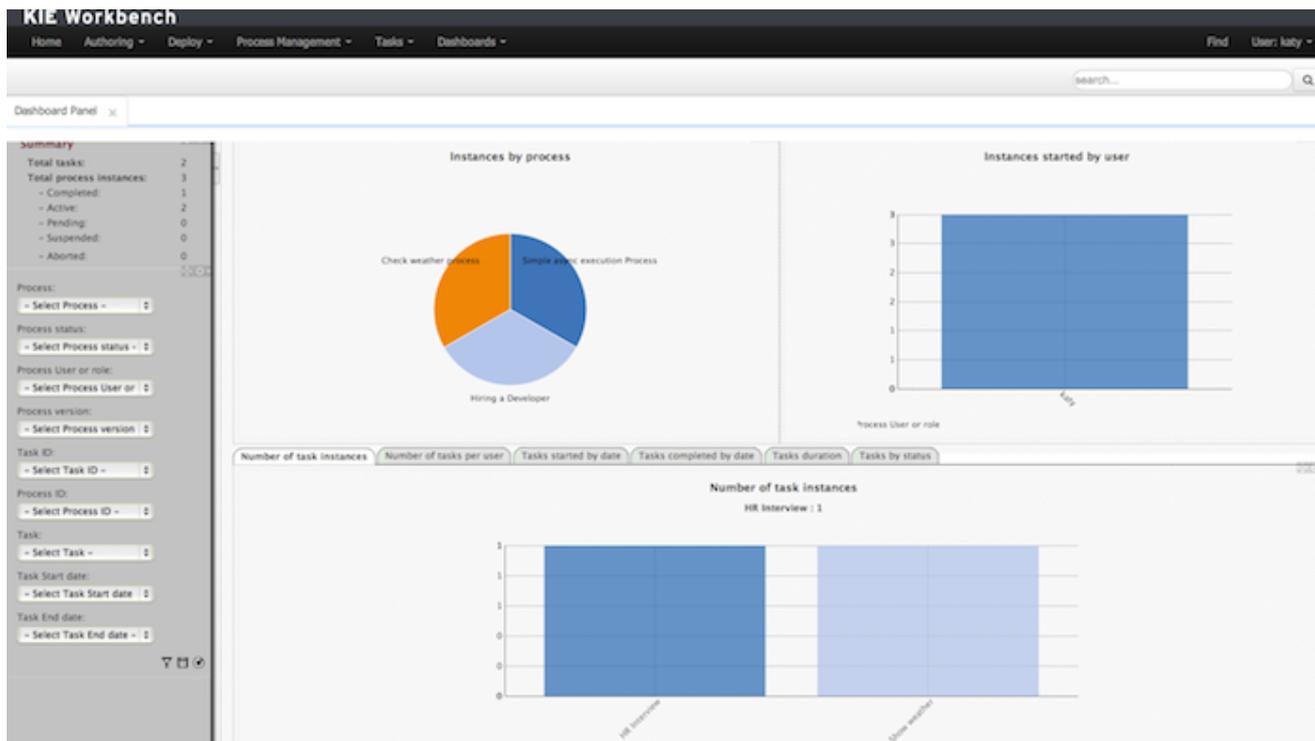


**Figure 1.6. Managing your process instances**

### 1.5.4. Business Activity Monitoring

Business Activity Monitoring (BAM) and Reporting allows to get an overview of the state of your application and/or system using dynamically generated (customizable) reports, that give you an overview of your key performance indicators (KPIs). To name few:

- Process instance charts: illustrates what process instances are active in the system process instances.
- Human task charts: illustrates human interaction within the system (who has been working on what, how long it took to complete particular tasks, etc)
- and much more



**Figure 1.7. Business Activity Monitoring**

## 1.6. Documentation

The documentation is structured as follows:

- Overview: the overview chapter gives an overview of the different components.
- Getting Started: the getting started chapter teaches you where to download the binaries and sources and contains a lot of useful links.
- Installer: the installer helps you setup a running demo, including most of the jBPM components. It runs you through the demos using a simple example and some 10-minute tutorials including screencasts.
- Quickstarts: these are tutorials for common tasks you might want to try out after successfully running the installer.
- Core engine: the next 4 chapters describe the core engine: the process engine API, the process definition language (BPMN 2.0), persistence and transactions, and examples.
- Eclipse editor: the next chapter describes the Eclipse plugin for developers.
- Designer: describes the web-based designer that allows business users to edit business processes in a web-based context.
- Console: the jBPM console can be used for managing process instances, human task lists and reports.

- Important features
  - Human tasks: When using human actors, you need a human task service to manage the life cycle of the tasks, the task lists, etc.
  - Domain-specific processes: plug in your own higher-level, domain-specific nodes in your processes.
  - Testing and debugging: how to test and debug your processes.
  - Process repository: a process repository used to manage your business processes.
- Advanced concepts
  - Business activity monitoring: event processing to monitor the state of your systems.
  - Flexible processes: model much more adaptive, flexible processes using advanced process constructs and integration with business rules and event processing.
  - Integration: how to integrate with other technologies like maven, OSGi, Spring, etc.



# Chapter 2. Getting Started

## 2.1. Downloads

All releases can be downloaded from [SourceForge](https://sourceforge.net/projects/jbpm/files/) [https://sourceforge.net/projects/jbpm/files/]. Select the version you want to download and then select which artifact you want:

- bin: all the jBPM binaries (jars) and their dependencies
- src: the sources of the core components
- docs: the documentation
- examples: some jBPM examples, can be imported into Eclipse
- installer: the jbpm-installer, downloads and installs a demo setup of jBPM
- installer-full: the jbpm-installer, downloads and installs a demo setup of jBPM, already contains a number of dependencies prepackages (so they don't need to be downloaded separately)

## 2.2. Use with Maven, Gradle, Ivy, Buildr or ANT

The jars are also available in [the central maven repository](http://search.maven.org/#search|ga|1|org.jbpm) [http://search.maven.org/#search|ga|1|org.jbpm] (and also in [the JBoss maven repository](https://repository.jboss.org/nexus/index.html#nexus-search;gav~org.jbpm~~~~) [https://repository.jboss.org/nexus/index.html#nexus-search;gav~org.jbpm~~~~]).

If you use Maven, add KIE and jBPM dependencies in your project's `pom.xml` like this:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-bom</artifactId>
      <type>pom</type>
      <version>...</version>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-api</artifactId>
  </dependency>
  <dependency>
    <groupId>org.jbpm</groupId>
```

```
<artifactId>jbpm-flow</artifactId>
<scope>runtime</scope>
</dependency>
...
<dependencies>
```

This is similar for Gradle, Ivy and Buildr. To identify the latest version, check the maven repository.

If you're still using ANT (without Ivy), copy all the jars from the download zip's `binaries` directory and manually verify that your classpath doesn't contain duplicate jars.

### 2.3. Getting started

If you like to take a quick tutorial that will guide you through most of the components using a simple example, take a look at the Installer chapter. This will teach you how to download and use the installer to create a demo setup, including most of the components. It uses a simple example to guide you through the most important features. Screencasts are available to help you out as well.

If you like to read more information first, the following chapters first focus on the core engine (API, BPMN 2.0, etc.). Further chapters will then describe the other components and other more complex topics like domain-specific processes, flexible processes, etc. After reading the core chapters, you should be able to jump to other chapters that you might find interesting.

You can also start playing around with some examples that are offered in a separate download. Check out the examples chapter to see how to start playing with these.

After reading through these chapters, you should be ready to start creating your own processes and integrate the engine with your application. These processes can be started from the installer or be started from scratch.

### 2.4. Community

Here are a lot of useful links part of the jBPM community:

- A feed of [blog entries](http://planet.jboss.org/view/feed.seam?name=jbossjbpm) [http://planet.jboss.org/view/feed.seam?name=jbossjbpm] related to jBPM
- The [#jbossjbpm twitter account](http://twitter.com/jbossjbpm) [http://twitter.com/jbossjbpm].
- A [user forum](http://www.jboss.com/index.html?module=bb&op=viewforum&f=217) [http://www.jboss.com/index.html?module=bb&op=viewforum&f=217] for asking questions and giving answers
- A [JIRA bug tracking system](https://jira.jboss.org/jira/browse/JBPM) [https://jira.jboss.org/jira/browse/JBPM] for bugs, feature requests and roadmap
- A [continuous build server](https://hudson.jboss.org/hudson/job/jBPM/) [https://hudson.jboss.org/hudson/job/jBPM/] for getting the [latest snapshots](https://hudson.jboss.org/hudson/job/jBPM/lastSuccessfulBuild/artifact/jbpm-distribution/target/) [https://hudson.jboss.org/hudson/job/jBPM/lastSuccessfulBuild/artifact/jbpm-distribution/target/]

Please feel free to join us in our IRC channel at chat.freenode.net #jbpm. This is where most of the real-time discussion about the project takes place and where you can find most of the developers most of their time as well. Don't have an IRC client installed? Simply go to <http://webchat.freenode.net/>, input your desired nickname, and specify #jbpm. Then click login to join the fun.

## 2.5. Sources

### 2.5.1. License

The jBPM code itself is using the Apache License v2.0.

Some other components we integrate with have their own license:

- The new Eclipse BPMN2 plugin is Eclipse Public License (EPL) v1.0.
- The web-based designer is based on Oryx/Wapama and is MIT License
- The Drools project is Apache License v2.0.

### 2.5.2. Source code

jBPM now uses git for its source code version control system. The sources of the jBPM project can be found here (including all releases starting from jBPM 5.0-CR1):

<https://github.com/droolsjbpm/jbpm>

The source of some of the other components we integrate with can be found here:

- Other components related to the jBPM and Drools project can be found [here](https://github.com/droolsjbpm) [https://github.com/droolsjbpm].
- The new Eclipse BPMN2 plugin can be found [here](https://git.eclipse.org/c/bpmn2-modeler/org.eclipse.bpmn2-modeler.git) [https://git.eclipse.org/c/bpmn2-modeler/org.eclipse.bpmn2-modeler.git].
- The web-based designer can be found [here](https://github.com/droolsjbpm/jbpm-designer) [https://github.com/droolsjbpm/jbpm-designer]
- The kie workbench can be found [here](https://github.com/droolsjbpm/kie-wb-distribution-wars) [https://github.com/droolsjbpm/kie-wb-distribution-wars] note this is an aggregate of other projects (drools-wb, jbpm-console-ng)

### 2.5.3. Building from source

If you're interested in building the source code, contributing, releasing, etc. make sure to read this [README](https://github.com/droolsjbpm/droolsjbpm-build-bootstrap/blob/master/README.md) [https://github.com/droolsjbpm/droolsjbpm-build-bootstrap/blob/master/README.md].



## Chapter 3. Installer

This guide will assist you in installing and running a demo setup of the various components of the jBPM project. If you have any feedback on how to improve this guide, if you encounter problems, or if you want to help out, do not hesitate to contact the jBPM community as described in the "What to do if I encounter problems or have questions?" section.

### 3.1. Prerequisites

This script assumes you have Java JDK 1.6+ (set as JAVA\_HOME), and Ant 1.7+ installed. If you don't, use the following links to download and install them:

Java: <http://java.sun.com/javase/downloads/index.jsp>

Ant: <http://ant.apache.org/bindownload.cgi>

### 3.2. Download the installer

First of all, you need to [download](https://sourceforge.net/projects/jbpm/files/jBPM%206/) [https://sourceforge.net/projects/jbpm/files/jBPM%206/] the installer. There are two versions

- full installer - which already contains a lot of the dependencies that are necessary during the installation
- minimal installer - which only contains the installer and will download all dependencies

In general, it is probably best to download the full installer: jBPM-{version}-installer-full.zip

You can also find the latest snapshot release here (only minimal installer) here:

<https://hudson.jboss.org/jenkins/job/jBPM/lastSuccessfulBuild/artifact/jbpm-distribution/target/>  
[https://hudson.jboss.org/jenkins/job/jBPM/lastSuccessfulBuild/artifact/jbpm-distribution/target/]

### 3.3. Demo setup

The easiest way to get started is to simply run the installation script to install the demo setup. Simply go into the install folder and run:

```
ant install.demo
```

This will:

- Download JBoss AS
- Download Eclipse
- Install the jBPM console (including repository, designer, BAM) into JBoss AS

- Install the jBPM Eclipse plugin
- Install the Drools Eclipse plugin

This could take a while (REALLY, not kidding, we are downloading an application server and Eclipse installation, even if you downloaded the full installer). The script however always shows which file it is downloading (you could for example check whether it is still downloading by checking the whether the size of the file in question in the `jbpm-installer/lib` folder is still increasing). If you want to avoid downloading specific components (because you will not be using them or you already have them installed somewhere else), check below for running only specific parts of the demo or directing the installer to an already installed component.

Once the demo setup has finished, you can start playing with the various components by starting the demo setup:

```
ant start.demo
```

This will:

- Start H2 server
- Start the JBoss AS
- Start Eclipse

Once everything is started, you can start playing with the Eclipse tooling, jBPM console, as explained in the next three sections.

If you do not wish to use Eclipse in the demo setup, you can use the alternative commands:

```
ant install.demo.noclipse  
ant start.demo.noclipse
```

### 3.3.1. Control options

jbpm console started by installer will by default bring in sample processes from jbpm demo repository that is cloned from origin hosted on github. In some cases where access to Internet is not available or there is a need to start completely clean installation of jbpm console the default behavior can be turned off. To do so following system property needs to be added to startup script (`build.xml` -> `start.jboss` target) or to `standalone.xml`:

```
-Dorg.kie.demo=false
```

with this there will not be any organizational unit nor repository created. To be able to start modeling processes user needs to create:

- Organizational unit
- Repository - brand new or clone existing one

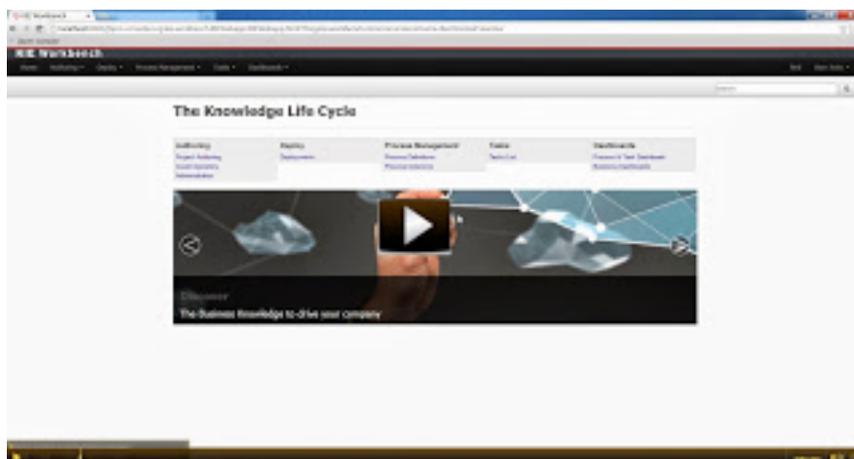
### 3.4. 10-Minute Tutorial: Using the jBPM Console

Open up the process management console:

<http://localhost:8080/jbpm-console>

Log in, using krisv / krisv as username / password. The [following screencast](http://people.redhat.com/kverlaen/jBPM6-console.swf) [http://people.redhat.com/kverlaen/jBPM6-console.swf] gives an overview of how to manage your process instances. It shows you:

- How to start a new process
- How to look up the current status of a running process instance
- How to look up your tasks
- How to complete a task
- How to generate reports to monitor your process execution



**Figure 3.1.**

[<http://people.redhat.com/kverlaen/jBPM6-console.swf>]

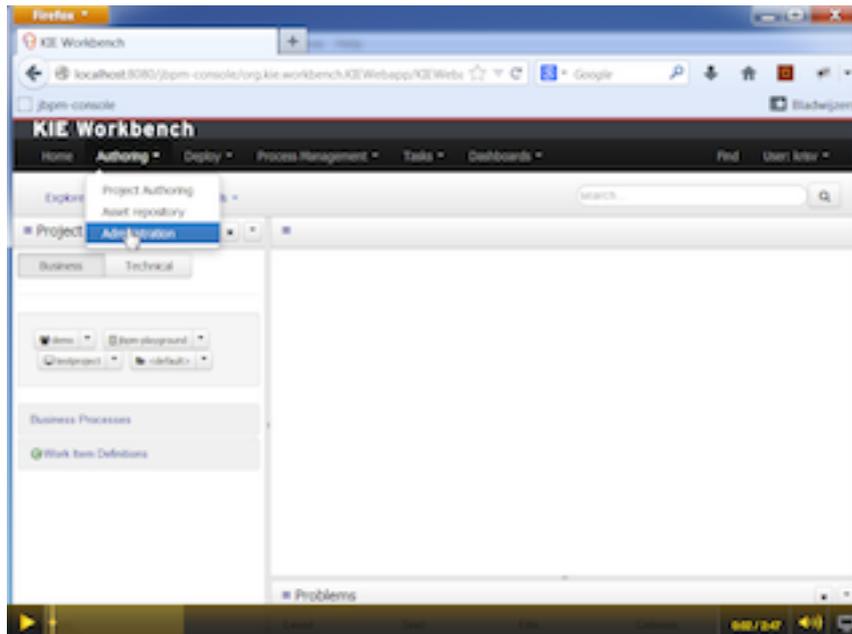
- To manage your process definitions and instances, click on the "Process Management" menu option at the top menu bar and select one of available options depending on your interest:

- Process Definitions - lists all available process definitions
- Process Instances - lists all active process instances (allows to show completed, aborted as well by changing filter criteria)
- Process definitions panel allow you to start a new process instance by clicking on the "Play" button. You will see a process form where you need to fill in the necessary information to start the process. In this case, you need to fill in your username "krisv" and a reason for the request, after which you can complete the form and close the window. After process form window is closed process instance details panel will be shown. From there you can access additional details:
  - Process model - to visualize current state of the process
  - Process variables - to see current values of process variablesThe process instance that you just started is first requiring a self-evaluation of the user and is waiting until the user has completed this task.
- To see the tasks that have been assigned to you, choose the "Tasks" menu option on the top bar and select "Task List" (you may need to click refresh to update your task view). The personal tasks table should show a "Performance Evaluation" task for you. You can complete this task by selecting it and clicking the "View" button. This will open the task form for performance evaluations. You can fill in the necessary data and then complete the form and close the window. After completing the task, you could check the "Process Instances" once more to check the progress of your process instance. You should be able to see that the process is now waiting for your HR manager and project manager to also perform an evaluation. You could log in as "john" / "john" and "mary" / "mary" to complete these tasks.
- After starting and/or completing a few process instances and human tasks, you can generate a report of what has happened so far. Under "Dashboards", select "Process & Task Dashboard". By default, you will see predefined set of charts that allow to directly spot what is going on in the system. Charts can be customized as well which will be described in following chapters.

### 3.5. 10-Minute Tutorial: Integrate Eclipse and Web tooling

The [following screencast](http://people.redhat.com/kverlaen/jBPM6-EclipseGitIntegration.swf) [http://people.redhat.com/kverlaen/jBPM6-EclipseGitIntegration.swf] gives an overview of how to integrate Web Console and Eclipse. It shows you:

- How to create new project inside web console
- How to create new process inside web console
- How to import an existing project into your workspace
- How to apply changes in Eclipse
- How to push back changes done in Eclipse to Web Console



**Figure 3.2.**

[<http://people.redhat.com/kverlaen/jBPM6-EclipseGitIntegration.swf>]

You could also create a new project using the jBPM project wizard. This sample project contains a simple HelloWorld BPMN2 process and an associated Java file to start the process. Simply select "File - New - jBPM Project" (if you cannot see that (because you're not in the jBPM perspective) you can do "File - New ... - Project ..." and under the "jBPM" category, select "jBPM project" and click "Next"). Give the project a name and click "Finish". You should see a new project containing a "sample.bpmn" process and a "com.sample.ProcessMain" Java class and a "com.sample.ProcessTest" JUnit test class. You can open the BPMN2 process by double-clicking it. To execute the process, right-click on ProcessMain.java and select "Run As - Java Application". You should see a "Hello World" statement in the output console. To execute the test, right-click on ProcessTest.java and select "Run As - JUnit Test". You should also see a "Hello World" statement in the output console, and the JUnit test completion in the JUnit view.

## 3.6. Using your own database with jBPM

### 3.6.1. Introduction

In this quickstart, we are going to:

1. modify the persistence settings for the process engine
2. test the startup with our new settings!

You will need a local instance of a database, in this case MySQL in order to complete this quickstart

First though, let's look at the persistence setup that jBPM uses. In the demo, and in general, there are following types of persistent entities used by jBPM:

- entities used for saving the actual session, process and work item information - aka runtime data.
- entities used for logging and generating Business Activity Monitoring (BAM) information - aka history log.
- entities used by the task service.

“persistent entities” in this context, are java classes that represent information in the database.

### 3.6.2. Database setup

In the MySQL database that I use in this quickstart, I've created single user:

- user/schema "jbpm" with password "jbpm" (for all mentioned above entities)

If you end up using different names for your user/schemas, please make a note of where we insert "jbpm" in the configuration files.

If you want to try this quickstart with *another* database, I've included a section at the end of this quickstart that describes what you may need to modify.

### 3.6.3. Configuration

The following files define the persistence settings for the jbpm-installer demo:

- db/jbpm-persistence-JPA2.xml
- Application server configuration
  - standalone-as-7.1.1.Final.xml
  - standalone-full-as-7.1.1.Final.xml

There are two standalone.xml files available as jbpm allows to use JMS component for integration and thus requires standalone-full.xml to be configured. Best practice is to update both to have consistent setup but most important is to have standalone-full-as-7.1.1.Final.xml properly configured.

Do the following:

- Disable H2 default data base and enable mysql data base in build.properties

```
# default is H2
# H2.version=1.3.168
# db.name=h2
# db.driver.jar.name=${db.name}.jar
# db.driver.download.url=http://repo1.maven.org/maven2/com/h2database/h2/
# ${H2.version}/h2-${H2.version}.jar
#mysql
```

```
db.name=mysql
db.driver.module.prefix=com/mysql
db.driver.jar.name=${db.name}-connector-java.jar
db.driver.download.url=https://repository.jboss.org/nexus/service/local/
repositories/central/content/mysql/mysql-connector-java/5.1.18/mysql-
connector-java-5.1.18.jar
```

- db/jbpm-persistence-JPA2.xml:

This is the JPA persistence file that defines the persistence settings used by jBPM for both the process engine information, the logging/BAM information and task service. The installer ant script moves this file to the expanded web console war before the war is installed on the server. So if you have already tried with default settings (H2) best would be to clean and install it again

```
ant clean.demo
```

this time it will be much faster as it does not have to download anything.

In this file, you will have to change the name of the hibernate dialect used for your database.

The original line is:

```
<property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
```

In the case of a MySQL database, you need to change it to:

```
<property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect"/>
```

For those of you who decided to use another database, a list of the available hibernate dialect classes can be found [here](http://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/session-configuration.html#configuration-optional-dialects) [http://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/session-configuration.html#configuration-optional-dialects].

- standalone-as-7.1.1.Final.xml and standalone-full-as-7.1.1.Final.xml:

This file is the configuration for the standalone JBoss AS 7 server. When the installer installs the demo, it moves these files to the `standalone/configuration` directory in the jboss server directory

We need to change datasource configuration in `standalone.xml` so that the jBPM process engine can use our MySQL database

The original file contains the following lines:

```
<datasource jndi-name="java:jboss/datasources/jbpmDS" enabled="true" use-  
java-context="true" pool-name="H2DS">  
  <connection-url>jdbc:h2:tcp://localhost/runtime/jbpm-demo</connection-url>  
  <driver>h2</driver>  
  <pool></pool>  
  <security>  
    <user-name>sa</user-name>  
    <password></password>  
  </security>  
</datasource>  
<drivers>  
  <driver name="h2" module="com.h2database.h2">  
    <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>  
  </driver>  
</drivers>
```

Change the lines to the following:

```
<datasource jndi-name="java:jboss/datasources/jbpmDS" pool-name="MySQLDS"  
enabled="true" use-java-context="true">  
  <connection-url>jdbc:mysql://localhost:3306/jbpm</connection-url>  
  <driver>mysql</driver>  
  <pool></pool>  
  <security>  
    <user-name>jbpm</user-name>  
    <password>jbpm</password>  
  </security>  
</datasource>  
<drivers>  
  <driver name="mysql" module="com.mysql">  
    <xa-datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</  
xa-datasource-class>  
  </driver>  
</drivers>
```

- Start the demo

We've modified all the necessary files at this point. Of course, this would be a good time to start your database up as well!

If you haven't installed the demo yet, do that first: If you have already installed and *run* the demo, it can't hurt to reinstall the demo:

```
ant clean.demo; ant install.demo
```

alternatively

```
ant install.demo.noclipse
```

After you've done that, you can finally start the demo using the following command:

```
ant start.demo
```

alternatively

```
ant start.demo.noclipse
```

If you're done with the demo, you can stop it using this command:

```
ant clean.demo; ant stop.demo
```

- Problems?

If this isn't working for you, please try the following:

- Please double check the files you've modified: I *wrote* this, but still made mistakes when changing files!
- Please make sure that you don't secretly have another instance of jboss AS running.
- If neither of those work (and you're using MySQL), please do then let us know.

### 3.6.4. Using a different database

If you decide to use a different database with this demo, you need to remember the following when going through the steps above:

- Change the JDBC URLs, usernames and passwords, and Hibernate dialect lines to match your database information in the configuration files mentioned above.

- In order to make sure your driver will be correctly installed in the JBoss AS 7 server, you can do one of two things. Both ways are explained [here](https://community.jboss.org/wiki/DataSourceConfigurationinAS7) [https://community.jboss.org/wiki/DataSourceConfigurationinAS7].

- *Install* [https://community.jboss.org/wiki/DataSourceConfigurationinAS7#Installing\_a\_JDBC\_driver\_as\_a\_module] the driver jar as a *module*, which is what the install script does.
- *Otherwise, you can modify and install* [https://community.jboss.org/wiki/DataSourceConfigurationinAS7#Installing\_a\_JDBC\_driver\_as\_a\_deployment] the downloaded jar as a *deployment*. In this case you will have to copy the jar yourself to the `standalone/deployments` directory.

If you choose to install driver as JBoss module, please do the following:

- Disable default H2 driver properties

```
# default is H2
# H2.version=1.3.168
# db.name=h2
# db.driver.jar.name=${db.name}.jar
# db.driver.download.url=http://repo1.maven.org/maven2/com/h2database/h2/
${H2.version}/h2-${H2.version}.jar
```

- Copy one of the example configs (mysql or postgresql)

```
#postgresql
db.name=postgresql
db.driver.module.prefix=org/postgresql
db.driver.jar.name=${db.name}-jdbc.jar
db.driver.download.url=https://repository.jboss.org/nexus/content/
repositories/thirdparty-uploads/postgresql/postgresql/9.1-902.jdbc4/
postgresql-9.1-902.jdbc4.jar
```

- Change the `db.name` property in `build.properties` to the name of the downloaded jdbc driver jar you placed in `db/drivers`.
- Change the `<driver>` information in the `<datasource>` section of `standalone.xml` so that it refers to the name of your driver module (see next step). For example:

```
<driver>postgresql</driver>
```

- Further on in `standalone.xml` is the `<drivers>` section of the `<datasources>` (note the plural: `drivers`, `datasources`). We need to do the following with this file:
  - Change the name of the driver to match the name in the last step,
  - Give an appropriate name to the module,
  - And fill in the correct name of the XA datasource class to use.

For example:

```
<drivers>
  <driver name="postgresql" module="org.postgresql">
    <xa-datasource-class>org.postgresql.xa.PGXADatasource</xa-datasource-
class>
  </driver>
</drivers>
```

- Change the `db.driver.module.prefix` property in `build.properties` to the same "value" you used for the module name in `standalone.xml`. In the example above, I used "org.postgresql" which means that I should then use `org/postgresql` for the `db.driver.module.prefix` property.
- Lastly, you'll have to create the `db/${db.name}_module.xml` file. As an example you can use `db/mysql_module.xml`, so just make a copy of it and:
  - Change the name of the *module* to match the `db.driver.module.prefix` property above
  - Change the name of the module resource to the name of the JDBC driver jar that was downloaded.

The top of the original file looks like this:

```
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java.jar"/>
  </resources>
```

Change those lines to look like this, for example:

```
<module xmlns="urn:jboss:module:1.0" name="org.postgresql">
  <resources>
    <resource-root path="postgresql-9.1-902.jdbc4.jar"/>
```

```
</resources>
```

### 3.7. jBPM data base schema scripts (DDL scripts)

By default demo setup makes use of Hibernate auto ddl generation capabilities to build up complete data base schema including all tables, sequences, etc for given data base. This is not always welcome and thus installer provides DDL scripts for most popular data base

**Table 3.1. DDL scripts**

Data base name	Location
db2	jbpm-installer/db/ddl-scripts/db2
derby	jbpm-installer/db/ddl-scripts/derby
h2	jbpm-installer/db/ddl-scripts/h2
hsqldb	jbpm-installer/db/ddl-scripts/hsqldb
mysql5	jbpm-installer/db/ddl-scripts/mysql5
mysqlinnodb	jbpm-installer/db/ddl-scripts/mysqlinnodb
oracle	jbpm-installer/db/ddl-scripts/oracle
postgresql	jbpm-installer/db/ddl-scripts/postgresql
sqlserver	jbpm-installer/db/ddl-scripts/sqlserver
sqlserver2008	jbpm-installer/db/ddl-scripts/sqlserver2008

DDL scripts are provided for both jBPM and Quartz schemas although Quartz schema DDL script is only required when timer service should be configured with Quartz data base job store. See Timer Service section for additional details.

This can be used to initially create data base schema but it can serve as base for any optimization that needs to be applied - such as indexes, etc.

### 3.8. jBPM installer script

jBPM installer ant script performs most of the work automatically and usually does not require additional attention but in case it does, here is a list of available targets that might be needed to perform some of the steps manually.

**Table 3.2. jBPM installer available targets**

Target	Description
clean.db	cleans up data base used by jBPM demo (applies only to H2 data base)
clean.demo	cleans up entire installation so new installation can be performed

Target	Description
clean.demo.noclipse	same as clean.demo but does not remove eclipse
clean.eclipse	removes eclipse and its workspace
clean.generated.ddl	removes DDL scripts generated if any
clean.jboss	removes application server with all its deployments
clean.jboss.repository	removes repository content for demo setup (guvnor maven repo, niogit, etc)
download.dashboard	downloads jBPM dashboard component (BAM)
download.db.driver	downloads db driver configured in build.properties
download.ddl.dependencies	downloads all dependencies required to run DDL script generation tool
download.droolsjbpm.eclipse	downloads drools and jbpm eclipse plugin
download.eclipse	downloads eclipse distribution
download.jboss	downloads Jboss Application Server
download.jBPM.bin	downloads jBPM binary distribution (jBPM libs and its dependencies)
download.jBPM.console	downloads jBPM console for JBoss AS
install.dashboard.into.jboss	installs jBPM dashboard into JBoss AS
install.db.files	installs db driver as JBoss module
install.demo	installs complete demo environment
install.demo.eclipse	installs Eclipse with all jBPM plugins, no server installation
install.demo.noclipse	similar to install.demo but skips eclipse installation
install.dependencies	installs custom libraries (such as work item handlers, etc) into the jbpm console
install.droolsjbpm-eclipse.into.eclipse	installs droolsjbpm eclipse plugin into eclipse
install.eclipse	install eclipse IDE
install.jboss	installs JBoss AS
install.jBPM-console.into.jboss	installs jBPM console application into JBoss AS

### 3.9. What to do if I encounter problems or have questions?

You can always contact the jBPM community for assistance.

IRC: #jbpm at chat.freenode.net

[jBPM User Forum](http://community.jboss.org/en/jbpm?view=discussions) [http://community.jboss.org/en/jbpm?view=discussions]

Email: [jbpm-user@lists.jboss.org](mailto:jbpm-user@lists.jboss.org)

### 3.10. Frequently asked questions

Some common issues are explained below.

Q: What if the installer complains it cannot download component X?

A: Are you connected to the internet? Do you have a firewall turned on? Do you require a proxy? It might be possible that one of the locations we're downloading the components from is temporarily offline. Try downloading the components manually (possibly from alternate locations) and put them in the `jbpm-installer/lib` folder.

Q: What if the installer complains it cannot extract / unzip a certain jar/war/zip?

A: If your download failed while downloading a component, it is possible that the installer is trying to use an incomplete file. Try deleting the component in question from the `jbpm-installer/lib` folder and reinstall, so it will be downloaded again.

Q: What if I have been changing my installation (and it no longer works) and I want to start over again with a clean installation?

A: You can use `ant clean.demo` to remove all the installed components, so you end up with a fresh installation again.

Q: I sometimes see exceptions when trying to stop or restart certain services, what should I do?

A: If you see errors during shutdown, are you sure the services were still running? If you see exceptions during restart, are you sure the service you started earlier was successfully shutdown? Maybe try killing the services manually if necessary.

Q: Something seems to be going wrong when running Eclipse but I have no idea what. What can I do?

A: Always check the consoles for output like error messages or stack traces. You can also check the Eclipse Error Log for exceptions. Try adding an audit logger to your session to figure out what's happening at runtime, or try debugging your application.

Q: Something seems to be going wrong when running the a web-based application like the `jbpm-console`. What can I do?

A: You can check the server log for possible exceptions: `jbpm-installer/jboss-as-{version}/standalone/log/server.log` (for JBoss AS7) or `jbpm-installer/jboss-as-{version}/server/default/log/server.log` (for earlier versions).

For all other questions, try contacting the jBPM community as described in the Getting Started chapter.

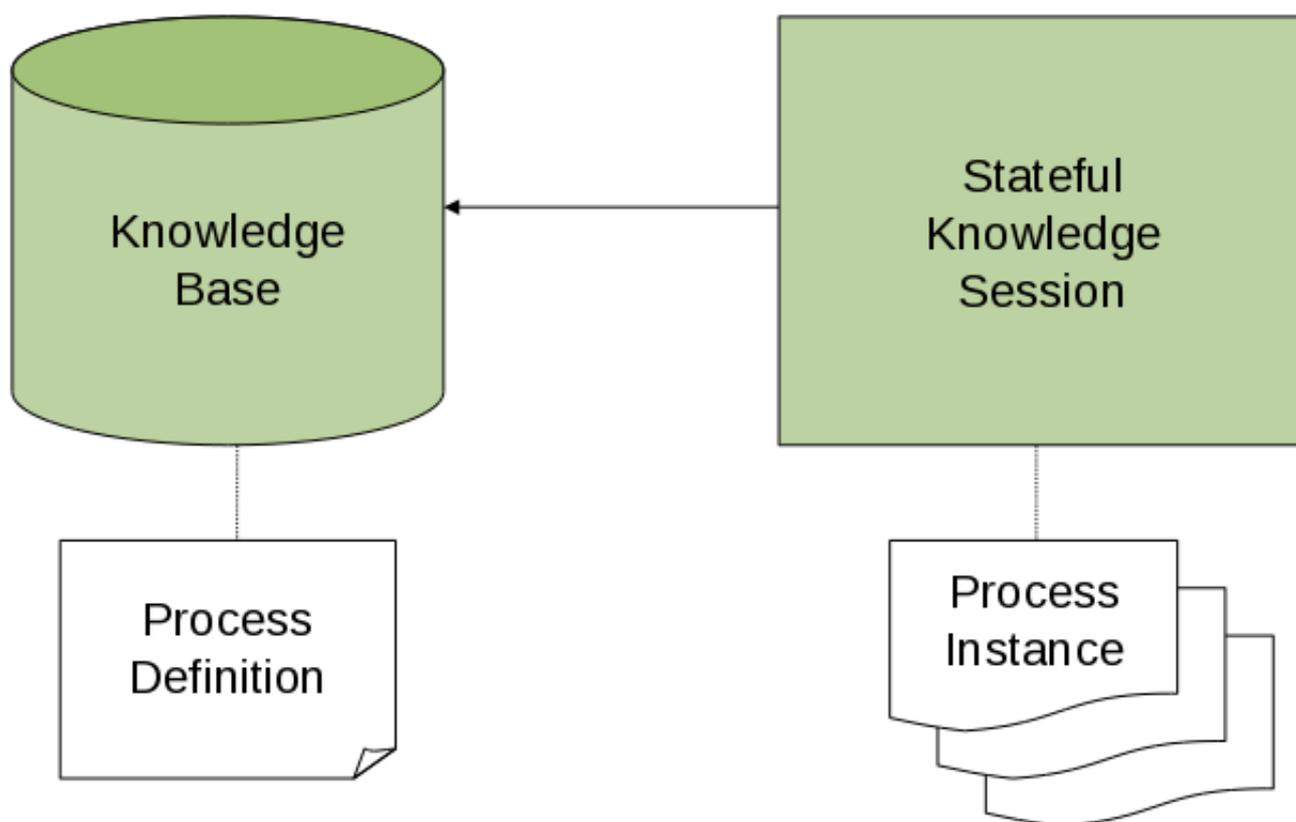


## Chapter 4. Core Engine: API

This chapter introduces the API you need to load processes and execute them. For more detail on how to define the processes themselves, check out the chapter on BPMN 2.0.

To interact with the process engine (for example, to start a process), you need to set up a session. This session will be used to communicate with the process engine. A session needs to have a reference to a knowledge base, which contains a reference to all the relevant process definitions. This knowledge base is used to look up the process definitions whenever necessary. To create a session, you first need to create a knowledge base, load all the necessary process definitions (this can be from various sources, like from classpath, file system or process repository) and then instantiate a session.

Once you have set up a session, you can use it to start executing processes. Whenever a process is started, a new process instance is created (for that process definition) that maintains the state of that specific instance of the process.



For example, imagine you are writing an application to process sales orders. You could then define one or more process definitions that define how the order should be processed. When starting up your application, you first need to create a knowledge base that contains those process definitions. You can then create a session based on this knowledge base so that, whenever a new sales order comes in, a new process instance is started for that sales order. That process instance contains the state of the process for that specific sales request.

A knowledge base can be shared across sessions and usually is only created once, at the start of the application (as creating a knowledge base can be rather heavy-weight as it involves parsing and compiling the process definitions). Knowledge bases can be dynamically changed (so you can add or remove processes at runtime).

Sessions can be created based on a knowledge base and are used to execute processes and interact with the engine. You can create as many independent session as you need and creating a session is considered relatively lightweight. How many sessions you create is up to you. In general, most simple cases start out with creating one session that is then called from various places in your application. You could decide to create multiple sessions if for example you want to have multiple independent processing units (for example, if you want all processes from one customer to be completely independent from processes for another customer, you could create an independent session for each customer) or if you need multiple sessions for scalability reasons. If you don't know what to do, simply start by having one knowledge base that contains all your process definitions and create one session that you then use to execute all your processes.

### 4.1. The jBPM API

The jBPM project has a clear separation between the API the users should be interacting with and the actual implementation classes. The public API exposes most of the features we believe "normal" users can safely use and should remain rather stable across releases. Expert users can still access internal classes but should be aware that they should know what they are doing and that the internal API might still change in the future.

As explained above, the jBPM API should thus be used to (1) create a knowledge base that contains your process definitions, and to (2) create a session to start new process instances, signal existing ones, register listeners, etc.

#### 4.1.1. Knowledge Base

The jBPM API allows you to first create a knowledge base. This knowledge base should include all your process definitions that might need to be executed by that session. To create a knowledge base, use a KieHelper to load processes from various resources (for example from the classpath or from the file system), and then create a new knowledge base from that helper. The following code snippet shows how to create a knowledge base consisting of only one process definition (using in this case a resource from the classpath).

```
KieHelper kieHelper = new KieHelper();
KieBase kieBase = kieHelper
    .addResource(ResourceFactory.newClassPathResource("MyProcess.bpmn"))
    .build();
```

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

This is considered manual creation of knowledge base and while it is simple it is not recommended for real application development but more for try outs. Following you'll find recommended and much more powerful way of building knowledge base, knowledge session and more - `RuntimeManager`.

### 4.1.2. Session

Once you've loaded your knowledge base, you should create a session to interact with the engine. This session can then be used to start new processes, signal events, etc. The following code snippet shows how easy it is to create a session based on the previously created knowledge base, and to start a process (by id).

```
KieSession ksession = kbase.newKieSession();
ProcessInstance processInstance = ksession.startProcess("com.sample.MyProcess");
```

The `ProcessRuntime` interface defines all the session methods for interacting with processes, as shown below.

```
/**
 * 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 in
 * @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 re
* 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();

```

### 4.1.3. Correlation key and Correlation properties

Common requirement when working with processes is ability to assign given process instance sort of business identifier that can be later on referenced without knowing the actual (generated) id of the process instance. To provide such capabilities jBPM allows to use CorrelationKey that is composed of CorrelationProperties. CorrelationKey can have either single property describing it (which is in most cases) but it can be represented as multi valued properties set.

Correlation capabilities are provided as part of interface

```
CorrelationAwareProcessRuntime
```

that exposes following methods:

```

/**
 * 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 correlationKey custom correlation key that can be used to identify process instance
 * @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, CorrelationKey correlationKey, Map<String, Object> parameters);

```

```
/**
 * Creates a new process instance (but does not yet start it). 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. You should only
 * use this method if you need a reference to the process instance before actually
 * starting it. Otherwise, use startProcess.
 *
 * @param processId the id of the process that should be started
 * @param correlationKey custom correlation key that can be used to identify process instance
 * @param parameters the process variables that should be set when creating the process instance
 * @return the ProcessInstance that represents the instance of the process that was created
 */
ProcessInstance createProcessInstance(String processId, CorrelationKey correlationKey, Map<String, String> parameters);

/**
 * Returns the process instance with the given correlationKey. Note that only active process instances
 * will be returned. If a process instance has been completed already, this method will return
 * null.
 *
 * @param correlationKey the custom correlation key assigned when process instance was created
 * @return the process instance with the given id or null if it cannot be found
 */
ProcessInstance getProcessInstance(CorrelationKey correlationKey);
```

Correlation is usually used with long running processes and thus require persistence to be enabled to be able to permanently store correlation information.

### 4.1.4. Events

The session provides methods for registering and removing listeners. A `ProcessEventListener` can be used to listen to process-related events, like starting or completing a process, entering and leaving a node, etc. Below, the different methods of the `ProcessEventListener` class are shown. An event object provides access to related information, like the process instance and node instance linked to the event. You can use this API to register your own event listeners.

```
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);

}
```

A note about before and after events: these 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 node triggered events will occur inbetween 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). Doing that allows us to derive cause relationships between events more easily. Similarly, all node triggered and node left 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, but 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 has ended (for example, when you want to be notified when starting of a particular process instance has ended).

Also note that not all nodes always generate node triggered and/or node left events. Depending on the type of node, some nodes might only generate node left events, others might only generate node triggered 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 are not generating left events (they are only generating triggered events, as they are not really left, as they have no outgoing connection).

jBPM out-of-the-box provides a listener that can be used to create an audit log (either to the console or the a file on the file system). This audit log contains all the different events that occurred at runtime so it's easy to figure out what happened. Note that these loggers should only be used for debugging purposes. The following logger implementations are supported by default:

1. Console logger: This logger writes out all the events to the console.
2. File logger: This logger writes out all the events to a file using an XML representation. This log file might then be used in the IDE to generate a tree-based visualization of the events that occurred during execution.
3. Threaded file logger: Because a file logger writes the events to disk only when closing the logger or when the number of events in the logger reaches a predefined level, it cannot be used when debugging processes at runtime. A threaded file logger writes the events to a file after a specified time interval, making it possible to use the logger to visualize the progress in realtime, while debugging processes.

The `KnowledgeRuntimeLoggerFactory` lets you add a logger to your session, as shown below. When creating a console logger, the knowledge session for which the logger needs to be created must be passed as an argument. The file logger also requires the name of the log file to be created, and the threaded file logger requires the interval (in milliseconds) after which the events should be saved. You should always close the logger at the end of your application.

```
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 Eclipse, using the Audit View in the Drools Eclipse plugin, where the events are visualized as a tree. Events that occur between the before and after event are shown as children of that event. The following screenshot shows a simple example, where a process is started, resulting in the activation of the Start node, an Action node and an End node, after which the process was completed.

- ▼  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]

## 4.2. Knowledge-based API

As you might have noticed, the API as exposed by the jBPM project is a knowledge API. That means that it doesn't just focus on processes, but potentially also allows other types of knowledge to be loaded. The impact for users that are only interested in processes however is very small. It just means that, instead of having a `ProcessBase` or a `ProcessSession`, you are using a `KnowledgeBase` and a `KnowledgeSession`.

However, if you ever plan to use business rules or complex event processing as part of your application, the knowledge-based API allows users to add different types of resources, such as processes and rules, in almost identical ways into the same knowledge base. This enables a user who knows how to use jBPM to start using Drools Expert (for business rules) or Drools Fusion (for event processing) almost instantaneously (and even to integrate these different types of Knowledge) as the API and tooling for these different types of knowledge is unified.

## 4.3. RuntimeManager

RuntimeManager has been introduced to simplify and empower usage of knowledge API especially in context of processes. It provides configurable strategies that control actual runtime execution (how KieSessions are provided) and by default provides following:

- Singleton - runtime manager maintains single KieSession regardless of number of processes available
- Per Request - runtime manager delivers new KieSession for every request
- Per Process Instance - runtime manager maintains mapping between process instance and KieSession and always provides same KieSession whenever working with given process instance

With that the complexity of knowing when to create, dispose, register handlers, etc is taken away from the end user and moved to the runtime manager that knows when/how to perform such operations but still allows to have a fine grained control over this process by providing comprehensive configuration of the RuntimeEnvironment.

```
public interface RuntimeEnvironment {  
  
    KieBase getKieBase();  
  
    Environment getEnvironment();  
  
    KieSessionConfiguration getConfiguration();  
  
    boolean usePersistence();  
  
    RegisterableItemsFactory getRegisterableItemsFactory();  
  
    Mapper getMapper();  
  
    UserGroupCallback getUserGroupCallback();  
  
    ClassLoader getClassLoader();  
  
    void close();  
}
```

While this interface provides mostly access to data kept as part of the runtime environment, the default implementation that is then considered as base for any extensions allows configuring the actual environment

```
public class SimpleRuntimeEnvironment .... {
    ....

    public void addToEnvironment(String name, Object value)

    public void addToConfiguration(String name, String value)

    public void setUserGroupCallback(UserGroupCallback userGroupCallback)

    public void setSessionConfigProperties(Properties sessionConfigProperties)

    public void setUsePersistence(boolean usePersistence)

    public void setKieBase(KieBase kbase)

    public void setMapper(Mapper mapper)

    public void setSchedulerService(GlobalSchedulerService schedulerService)

    public void setRegisterableItemsFactory(RegisterableItemsFactory registerableItemsFactory)

    public void setEmf(EntityManagerFactory emf)

    public void setClassLoader(ClassLoader classLoader)

    ....
}
```

Besides KieSession Runtime Manager provides access to TaskService too as integrated component of a RuntimeEngine that will always be configured and ready for communication between process engine and task service.

More about RuntimeManager and RuntimeEngine can be found in RuntimeManager chapter just to give a short preview of how it is used, here is how you can build RuntimeManager and get RuntimeEngine (that encapsulates KieSession and TaskService) from it:

```
// first configure environment that will be used by RuntimeManager
RuntimeEnvironment environment = RuntimeEnvironmentBuilder.getEmpty()
    .addAsset(ResourceFactory.newClassPathResource("BPMN2-
ScriptTask.bpmn2"), ResourceType.BPMN2)
    .get();

// next create RuntimeManager - in this case singleton strategy is chosen
```

```

RuntimeManager manager = RuntimeManagerFactory.Factory.get().newSingletonRuntimeManager(env);

// then get RuntimeEngine out of manager - using empty context as singleton
// does not keep track
// of runtime engine as there is only one
RuntimeEngine runtime = manager.getRuntimeEngine(EmptyContext.get());

// get KieSession from runtime runtimeEngine - already initialized with all
// handlers, listeners, etc that were configured
// on the environment
KieSession ksession = runtimeEngine.getKieSession();

// add invocations to the process engine here,
// e.g. ksession.startProcess(processId);

// and last dispose the runtime engine
manager.disposeRuntimeEngine(runtimeEngine);

```

## 4.4. Control parameters

There are several control parameters available to alter engine default behavior. This allows to fine tune the execution for the environment needs and actual requirements. All of these parameters are set as JVM system properties, usually with `-D` when starting program e.g. application server.

**Table 4.1. Control parameters**

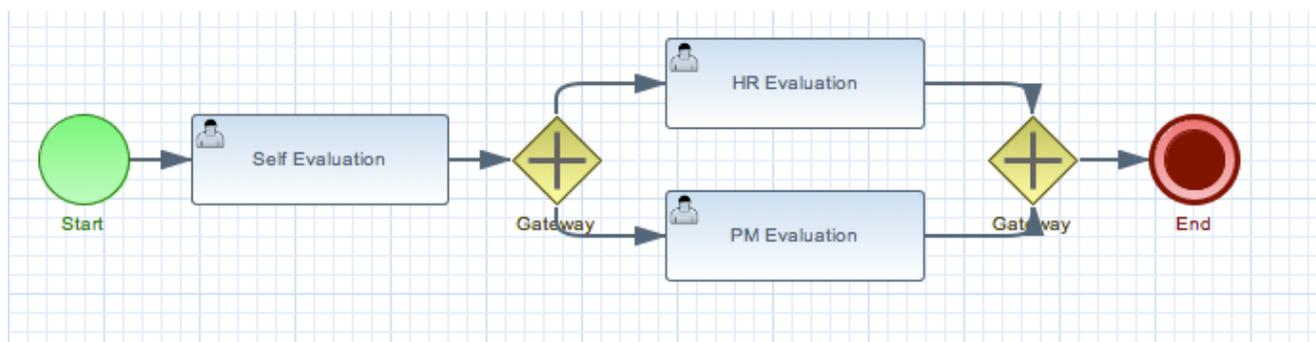
Name	Possible values	Default value	Description
<code>jbpm.ut.jndi.lookup</code>	String		Alternative JNDI name to be used when there is no access to the default one (java:comp/UserTransaction)
<code>jbpm.enable.multiple</code>	true false	false	Enables multiple incoming/outgoing sequence flows support for activities
<code>jbpm.business.calendar</code>	String properties	/ <code>jbpm.business.calendar.properties</code>	Allows to provide alternative properties classpath location of

Name	Possible values	Default value	Description
jbpm.overdue.timer.delay	Long	2000	business calendar configuration file Specifies delay for overdue timers to allow proper initialization, in milliseconds
jbpm.process.name.comparator	String		Allows to provide alternative comparator class to empower start process by name feature, if not set NumberVersionComparator is used
jbpm.loop.level.disabled	Boolean	true	Allows to enable or disable loop iteration tracking, to allow advanced loop support when using XOR gateways
org.kie.mail.session	String	mail/jbpmMailSession	Allows to provide alternative JNDI name for mail session used by Task Deadlines
jbpm.usergroup.callback.properties	String	/jbpm.usergroup.callback.properties	Allows to provide alternative properties classpath location for user group callback implementation (LDAP, DB)
jbpm.user.group.roles	String	\$(jboss.server.config.dir)/roles.properties	Allows to provide alternative location of roles.properties

Name	Possible values	Default value	Description
jbpm.user.info.properties	String	/jbpm.user.info.properties	for JBossUserGroupCallbackImpl Allows to provide alternative classpath location of user info configuration (used by LDAPUserInfolmp)
org.jbpm.ht.user.separator	String	,	Allows to provide alternative separator of actors and groups for user tasks, default is comma (,)
org.quartz.properties	String		Allows to provide location of the quartz config file to activate quartz based timer service
jbpm.data.dir	String	\${jboss.server.data.dir} if available otherwise \${java.io.tmpdir}	Allows to provide location where data files produced by jbpm should be stored
org.kie.executor.poolsize	Integer	1	Allows to provide thread pool size for jbpm executor
org.kie.executor.retry	Integer	3	Allows to provide number of retries attempted in case of error by jbpm executor
org.kie.executor.interval	Integer	3	Allows to provide frequency used to check for pending jobs by

Name	Possible values	Default value	Description
org.kie.executor.disable	true/false	true	jbpm executor, in seconds Enables or disables jbpm executor

## Chapter 5. Core Engine: Basics



**Figure 5.1.**

A business process is a graph that describes the order in which a series of steps need to be executed, using a flow chart. 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 while the connections specify how to transition from one node to the other. A large selection of predefined node types have been defined. This chapter describes how to define such processes and use them in your application.

### 5.1. Creating a process

Processes can be created by using one of the following three methods:

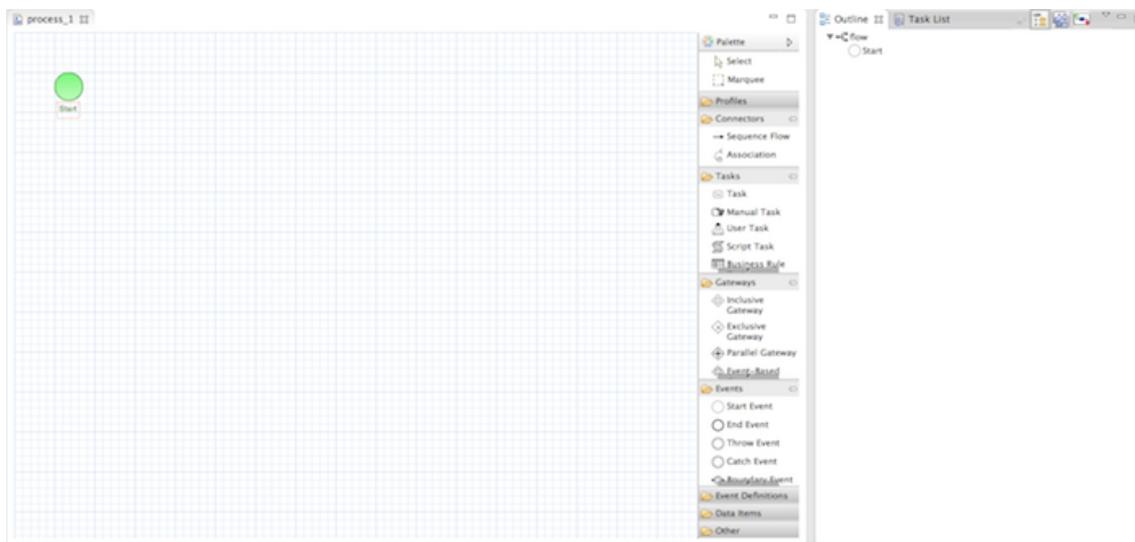
1. Using the graphical process editor such as jBPM web designer or Eclipse BPMN2 modeler
2. As an XML file, according to the XML process format as defined in the XML Schema Definition in the BPMN 2.0 specification.
3. By directly creating a process using the Process API.

#### 5.1.1. Using the graphical BPMN2 Editor

The graphical BPMN2 editor is an editor that allows you to create a process by dragging and dropping different nodes on a canvas and editing the properties of these nodes. The graphical BPMN2 modeler is an Eclipse plugin hosted on [eclipse.org](http://www.eclipse.org/bpmn2-modeler/) [http://www.eclipse.org/bpmn2-modeler/] that provides number of contributors where one of them is jBPM project. Once you have set up a jBPM project (see the installer for creating a working Eclipse environment where you can start), you can start adding processes. When in a project, launch the "New" wizard (use Ctrl+N) or right-click the directory you would like to put your process in and select "New", then "File". Give the file a name and the extension bpmn (e.g. MyProcess.bpmn). This will open up the process editor (you can safely ignore the warning that the file could not be read, this is just because the file is still empty).

First, ensure that you can see the Properties View down the bottom of the Eclipse window, as it will be necessary to fill in the different properties of the elements in your process. If you cannot

see the properties view, open it using the menu "Window", then "Show View" and "Other...", and under the "General" folder select the Properties View.



**Figure 5.2. New process**

The process editor consists of a palette, a canvas and an outline view. To add new elements to the canvas, select the element you would like to create in the palette and then add them to the canvas by clicking on the preferred location. For example, click on the "End Event" icon in the palette of the GUI. Clicking on an element in your process allows you to set the properties of that element. You can connect the nodes (as long as it is permitted by the different types of nodes) by using "Sequence Flow" from the palette.

You can keep adding nodes and connections to your process until it represents the business logic that you want to specify.

### 5.1.2. Defining processes using XML

It is also possible to specify processes using the underlying BPMN 2.0 XML directly. The syntax of these XML processes is defined using the BPMN 2.0 XML Schema Definition. For example, the following XML fragment shows a simple process that contains a sequence of a Start Event, a Script Task that prints "Hello World" to the console, and an End Event.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"
  targetNamespace="http://www.jboss.org/drools"
  typeLanguage="http://www.java.com/javaTypes"
  expressionLanguage="http://www.mvel.org/2.0"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" Rule Task
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
  BPMN20.xsd"
  xmlns:g="http://www.jboss.org/drools/flow/gpd"
```

```

        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.hello" name="Hello
Process" >

    <!-- nodes -->
    <startEvent id="_1" name="Start" />
    <scriptTask id="_2" name="Hello" >
        <script>System.out.println("Hello World");</script>
    </scriptTask>
    <endEvent id="_3" name="End" >
        <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="com.sample.hello" >
        <bpmndi:BPMNShape bpmnElement="_1" >
            <dc:Bounds x="16" y="16" width="48" height="48" />
        </bpmndi:BPMNShape>
        <bpmndi:BPMNShape bpmnElement="_2" >
            <dc:Bounds x="96" y="16" width="80" height="48" />
        </bpmndi:BPMNShape>
        <bpmndi:BPMNShape bpmnElement="_3" >
            <dc:Bounds x="208" y="16" width="48" height="48" />
        </bpmndi:BPMNShape>
        <bpmndi:BPMNEdge bpmnElement="_1_2" >
            <di:waypoint x="40" y="40" />
            <di:waypoint x="136" y="40" />
        </bpmndi:BPMNEdge>
        <bpmndi:BPMNEdge bpmnElement="_2_3" >
            <di:waypoint x="136" y="40" />
            <di:waypoint x="232" y="40" />
        </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
</bpmndi:BPMNDiagram>

</definitions>

```

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 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, defining the various parameters and, possibly, sub-elements for that node type.

### 5.1.3. Defining Processes Using the Process API

While it is recommended to define processes using the graphical editor or the underlying XML (to shield yourself from internal APIs), it is also possible to define a process using the Process API directly. The most important process model elements are defined in the packages `org.jbpm.workflow.core` and `org.jbpm.workflow.core.node`. A "fluent API" is provided that allows you to easily construct processes in a readable manner using factories. At the end, you can validate the process that you were constructing manually.

#### 5.1.3.1. Example

This is a simple example of a basic process with a script task only:

```
RuleFlowProcessFactory factory =
    RuleFlowProcessFactory.createProcess("org.jbpm.HelloWorld");
factory
    // Header
    .name("HelloWorldProcess")
    .version("1.0")
    .packageName("org.jbpm")
    // Nodes
    .startNode(1).name("Start").done()
    .actionNode(2).name("Action")
        .action("java", "System.out.println(\"Hello World\");").done()
    .endNode(3).name("End").done()
    // Connections
    .connection(1, 2)
    .connection(2, 3);

RuleFlowProcess process = factory.validate().getProcess();
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(ResourceFactory.newByteArrayResource(
    XmlBPMNProcessDumper.INSTANCE.dump(process).getBytes()), ResourceType.BPMN2);
KnowledgeBase kbase = kbuilder.newKnowledgeBase();
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
ksession.startProcess("org.jbpm.HelloWorld");
```

You can see that we start by calling the static `createProcess()` method from the `RuleFlowProcessFactory` class. This method creates a new process with the given id and returns the `RuleFlowProcessFactory` that can be used to create the process. A typical process consists of three parts. The header part comprises global elements like the name of the process, imports, variables, etc. The nodes section contains all the different nodes that are part of the process. The connections section finally links these nodes to each other to create a flow chart.

In this example, the header contains the name and the version of the process and the package name. After that, you can start adding nodes to the current process. If you have auto-completion you can see that you have different methods to create each of the supported node types at your disposal.

When you start adding nodes to the process, in this example by calling the `startNode()`, `actionNode()` and `endNode()` methods, you can see that these methods return a specific `NodeFactory`, that allows you to set the properties of that node. Once you have finished configuring that specific node, the `done()` method returns you to the current `RuleFlowProcessFactory` so you can add more nodes, if necessary.

When you are finished adding nodes, you must connect them by creating connections between them. This can be done by calling the method `connection`, which will link previously created nodes.

Finally, you can validate the generated process by calling the `validate()` method and retrieve the created `RuleFlowProcess` object.

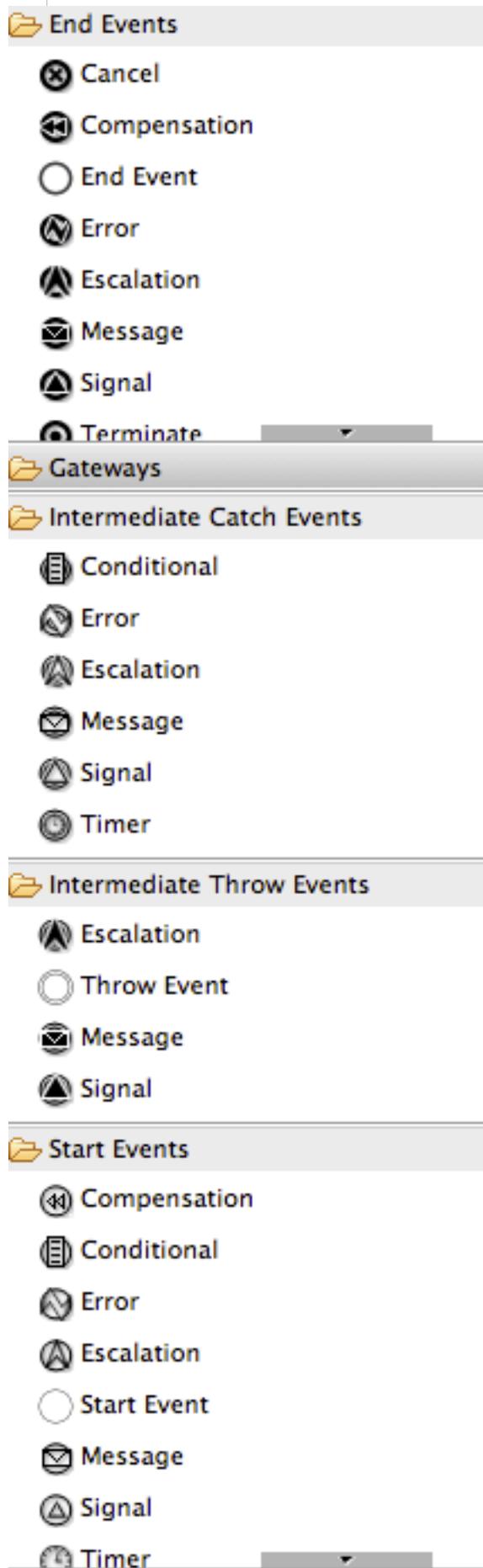
## 5.2. Details of different process constructs: Overview

The following chapters will describe the different constructs that you can use to model your processes (and their properties) in detail. Executable processes in BPMN consist of different types of nodes being connected to each other using sequence flows. The BPMN 2.0 specification defines three main types of nodes:

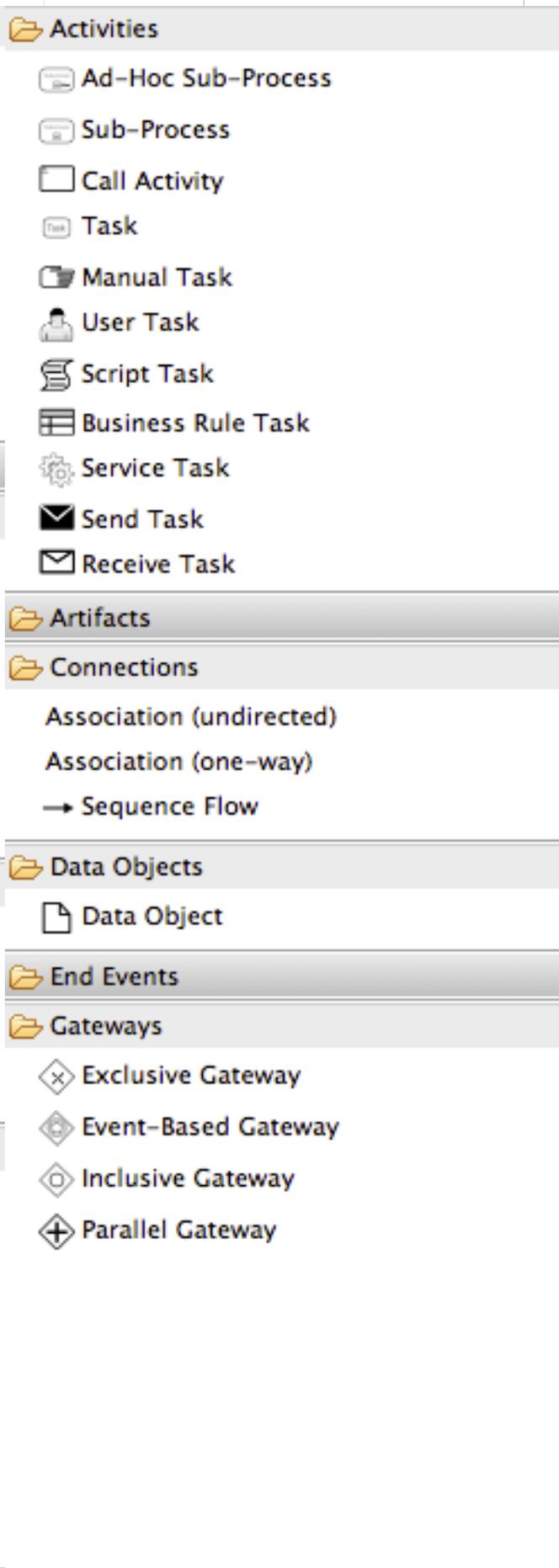
- **Events:** They are used to model the occurrence of a particular event. This could be a start event (that is used to indicate the start of the process), end events (that define the end of the process, or of that subflow) and intermediate events (that indicate events that might occur during the execution of the process).
- **Activities:** These define the different actions that need to be performed during the execution of the process. Different types of tasks exist, depending on the type of activity you are trying to model (e.g. human task, service task, etc.) and activities could also be nested (using different types of sub-processes).
- **Gateways:** Can be used to define multiple paths in the process. Depending on the type of gateway, these might indicate parallel execution, choice, etc.

The following sections will describe the properties of the process itself and of each of these different node types in detail, as supported by the Eclipse plugin and shown in the following figure of the palette. Note that the Eclipse property editor might show more properties for some of the

supported node types, but only the properties as defined in this section are supported when using the BPMN 2.0 XML format.



**Figure 5.3. The different types of BPMN2 events**

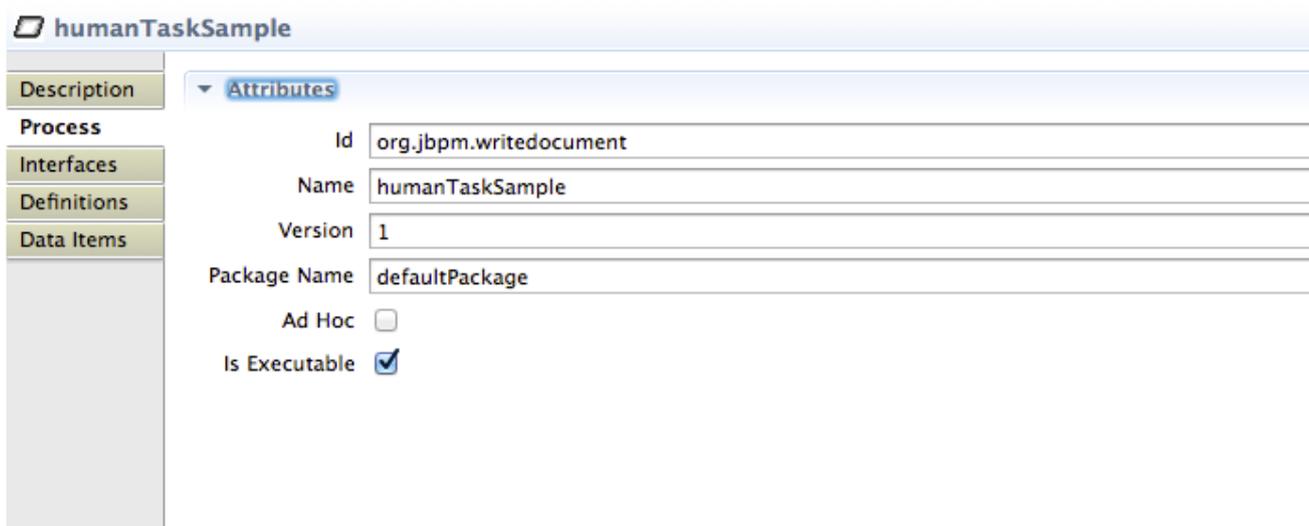


**Figure 5.4. The different types of BPMN2 activities and gateways**

### 5.3. Details: Process properties

A BPMN2 process is a flow chart where different types of nodes are linked using connections. The process itself exposes 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.



**Figure 5.5. BPMN2 process properties**

In addition to that following can be defined as well:

- *Variables*: Variables can be defined to store data during the execution of your process. See section “[Data](#)” for details.
- *Swimlanes*: Specify the swimlanes used in this process for assigning human tasks. See chapter “[???](#)” for details.

The screenshot shows a software interface for editing a BPMN2 process named "humanTaskSample". On the left is a vertical navigation menu with the following items: "Description", "Process", "Interfaces", "Definitions", and "Data Items". The "Data Items" item is currently selected. The main area of the interface displays the "Variable List for Process 'humanTaskSample'". This list is presented as a table with two columns: "Name" and "Data Type".

Name	Data Type
approval_document	String
approval_translatedDocument	String
approval_reviewComment	String

Figure 5.6. BPMN2 process variables

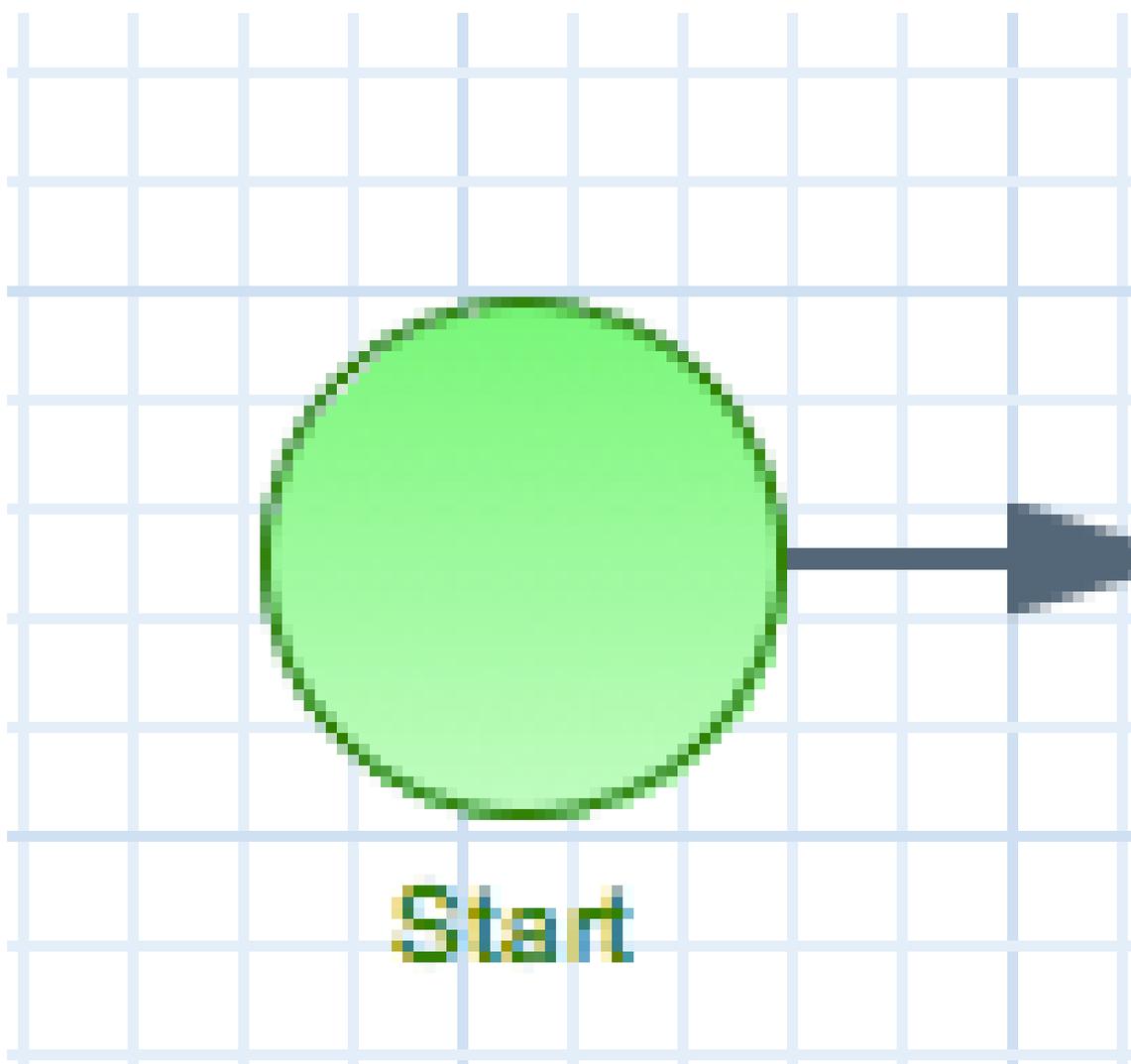
## 5.4. Details: Events



### Note

Following sections provide only introduction to BPMN2 constructs and their properties and complete reference can be found in BPMN2 chapter.

### 5.4.1. Start event



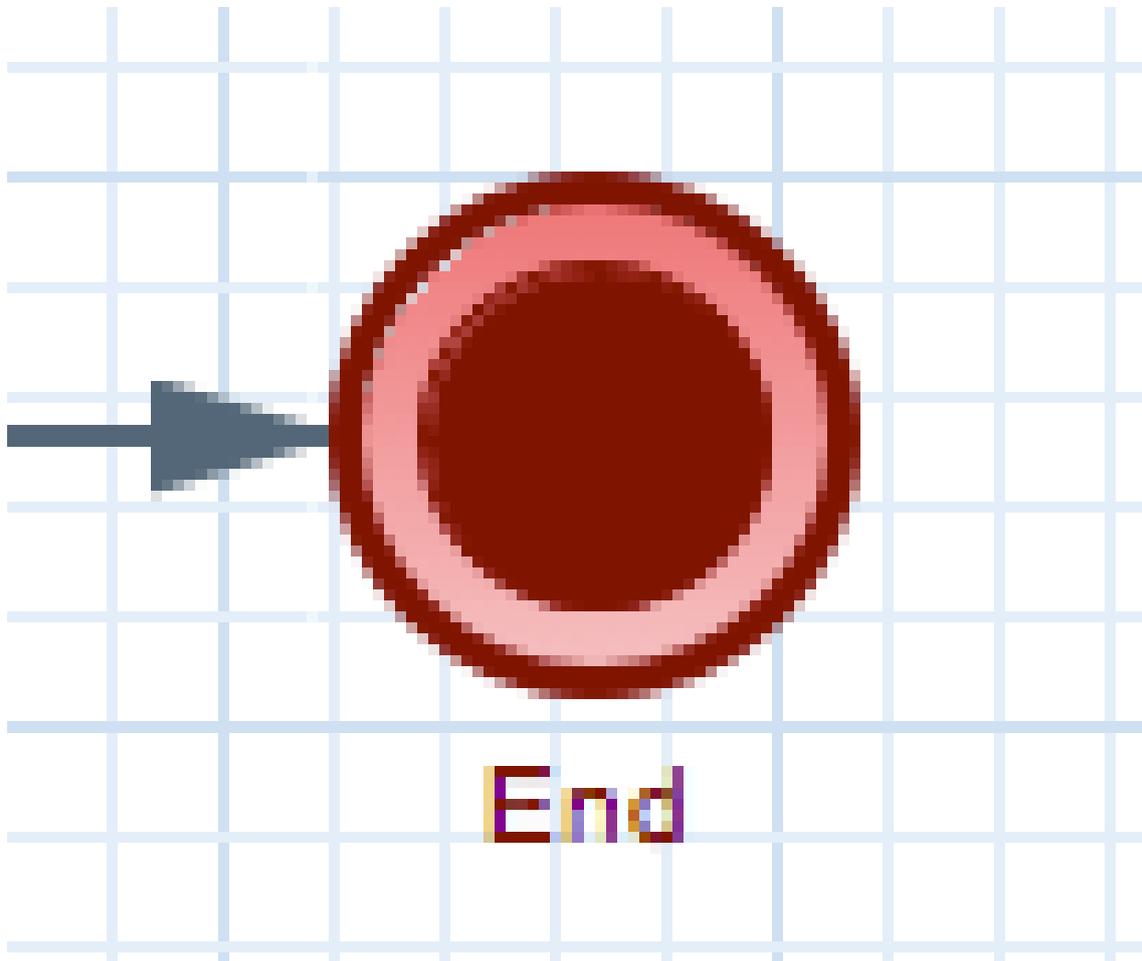
**Figure 5.7. Start event**

The start of the process. A process should have exactly one start node (none start node which does not have event definitions), which cannot have incoming connections and should have one outgoing connection. Whenever a process is started, execution will start at this node and automatically continue to the first node linked to this start event, and so on. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.

## 5.4.2. End events

### 5.4.2.1. End event



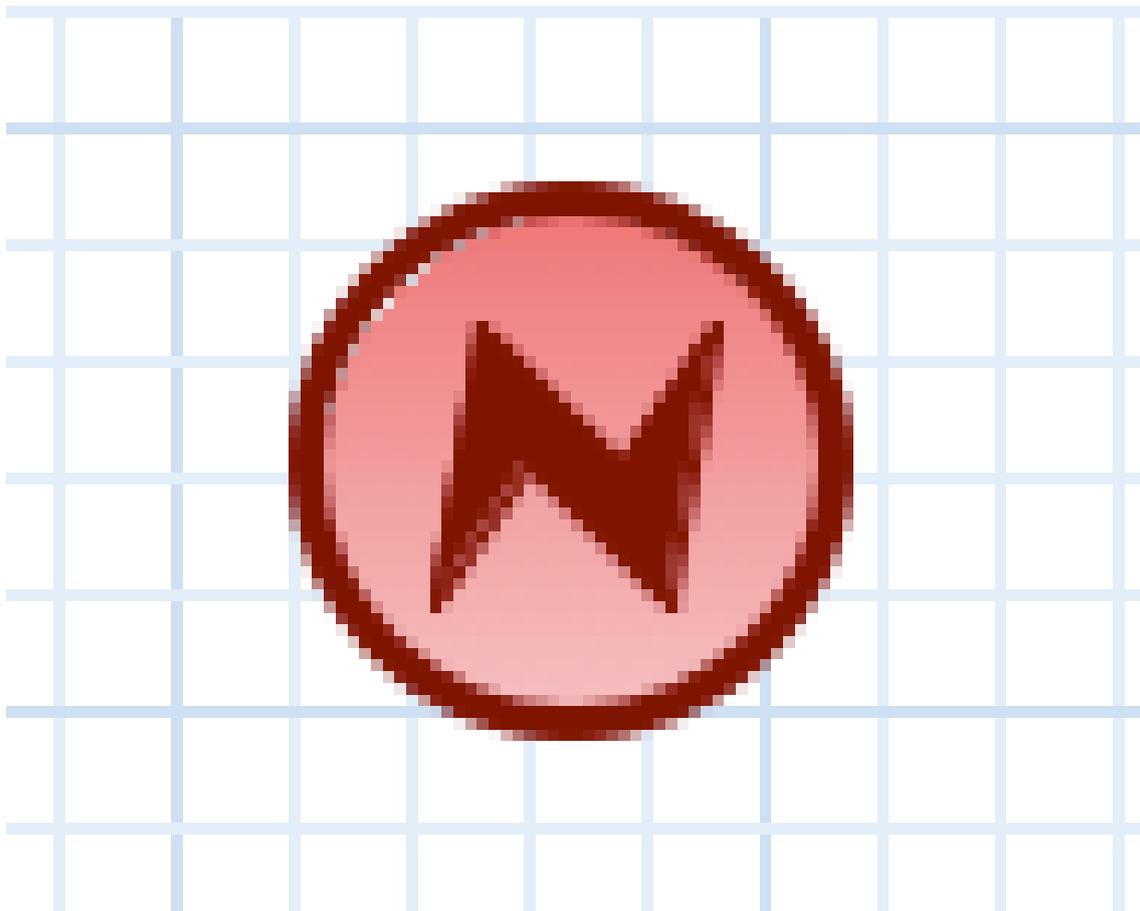
**Figure 5.8. End event**

The end of the process. A process should have one or more end events. The End Event should have one incoming connection and cannot have any outgoing connections. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Terminate*: An End Event can terminate the entire process or just the path. When a process instance is terminated, it means its state is set to completed and all other nodes that might still be active (on parallel paths) in this process instance are cancelled. Non-terminating end events are simply end for this path (execution of this branch will end here), but other parallel paths can still continue. A process instance will automatically complete if there are no more active paths inside that process instance (for example, if a process instance reaches a non-terminating end node but there are no more active branches inside the process instance, the process instance

will be completed anyway). Terminating end events are visualized using 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 just that sub-process and top level continues.

### 5.4.2.2. Throwing error event



**Figure 5.9. Throwing error event**

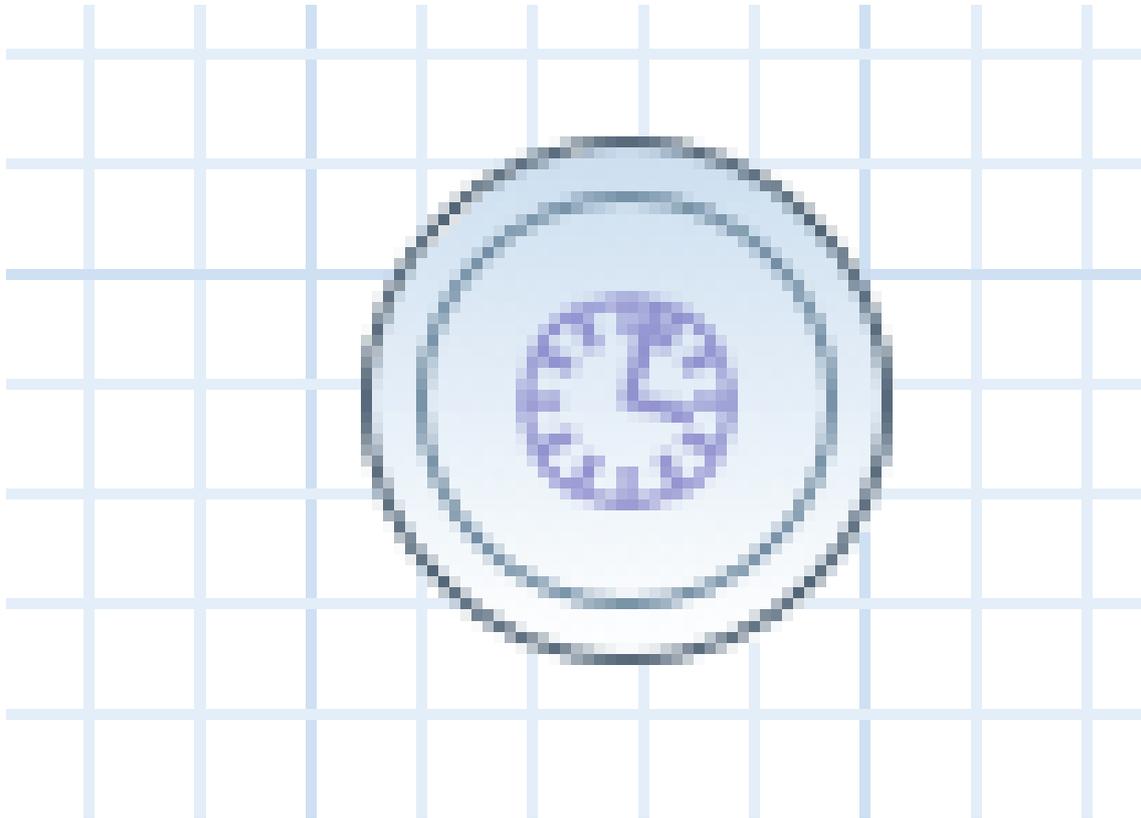
An Error Event can be used to signal an exceptional condition in the process. It should have one incoming connection and no outgoing connections. When an Error Event is reached in the process, it will throw an error with the given name. The process will search for an appropriate error handler that is capable of handling this kind of fault. If no error handler is found, the process instance will be aborted. An Error Event contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *FaultName*: The name of the fault. This name is used to search for appropriate exception handlers that are capable of handling this kind of fault.
- *FaultVariable*: The name of the variable that contains the data associated with this fault. This data is also passed on to the exception handler (if one is found).

Error handlers can be specified using boundary events.

### 5.4.3. Intermediate events

#### 5.4.3.1. Catching timer event



**Figure 5.10. Catching timer event**

Represents 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. The timer delay specifies how long the timer should wait before triggering the first time. When a Timer Event is reached in the process, it will start the associated timer. The timer is cancelled if the timer node is cancelled (e.g., by completing or aborting the enclosing process instance). Consult the section “[Timers](#)” for more information. The Timer Event contains the following properties:

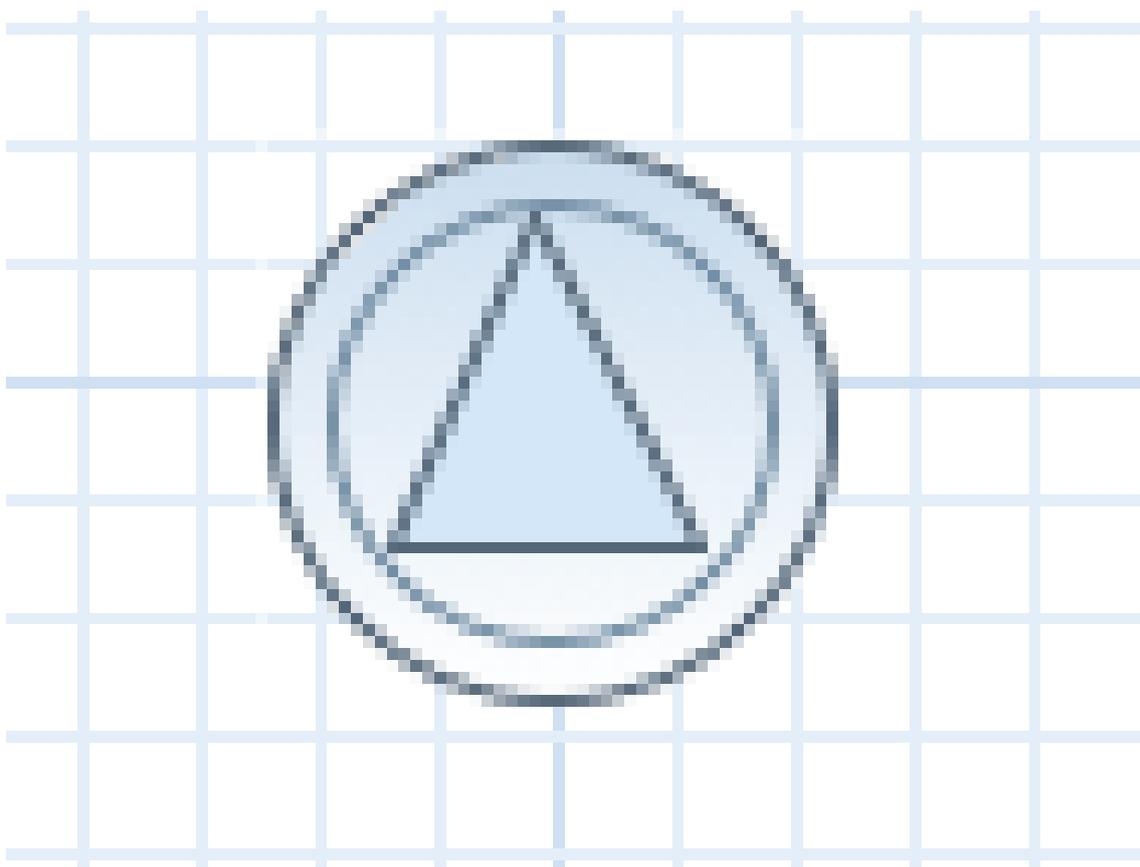
- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Timer delay*: The delay that the node should wait before triggering the first time. The expression should be of the form `[#d][#h][#m][#s][#[ms]]`. This allows you to specify the number of days, hours, minutes, seconds and milliseconds (which is the default if you don't specify anything). For example, the expression "1h" will wait one hour before triggering the timer. The expression could also use `#{expr}` to dynamically derive the delay based on some process

variable. Expr in this case could be a process variable, or a more complex expression based on a process variable (e.g. myVariable.getValue()).

- *Timer period*: The period between two subsequent triggers. If the period is 0, the timer should only be triggered once. The expression should be of the form [#d][#h][#m][#s][#ms]. You can specify the number of days, hours, minutes, seconds and milliseconds (which is the default if you don't specify anything). For example, the expression "1h" will wait one hour before triggering the timer again. The expression could also use #{expr} to dynamically derive the period based on some process variable. Expr in this case could be a process variable, or a more complex expression based on a process variable (e.g. myVariable.getValue()).

Timer events could also be specified as boundary events on sub-processes and tasks that are not automatic tasks like script task that have no wait state as timer will not have a change to fire before task completion.

### 5.4.3.2. Catching signal event



**Figure 5.11. Catching signal event**

A Signal Event can be used to respond to internal or external events during the execution of the process. A Signal Event should have one 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. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *EventType*: The type of event that is expected.
- *VariableName*: The name of the variable that will contain the data associated with this event (if any) when this event occurs.

A process instance can be signaled that a specific event occurred using

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

This will trigger all (active) signal event nodes in the given 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.

It is also possible to use event nodes inside sub-processes. These event nodes will however only be active when the sub-process is active.

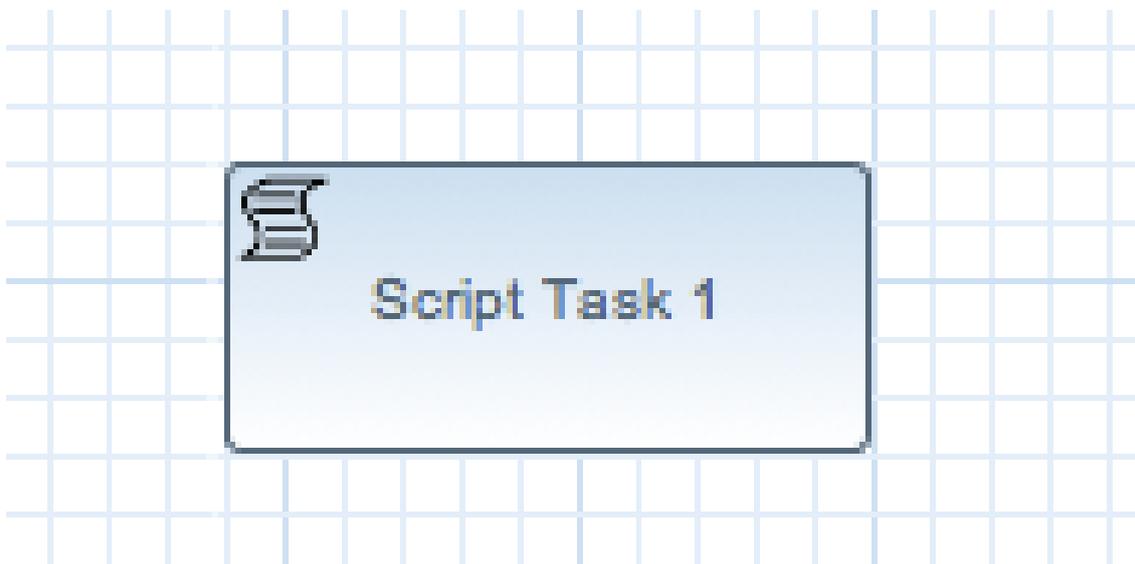
You can also generate a signal from inside a process instance. A script (in a script task or using on entry or on exit actions) can use

```
kcontext.getKnowledgeRuntime().signalEvent(eventType, data,  
kcontext.getProcessInstance().getId());
```

A throwing signal event could also be used to model the signaling of an event.

## 5.5. Details: Activities

### 5.5.1. Script task



**Figure 5.12. Script task**

Represents a script that should be executed in this process. A Script Task should have one incoming connection and one outgoing connection. 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, for example, be used to access the current `ProcessInstance` or `NodeInstance`, and to get and set variables, or get access to the `ksession` using `kcontext.getKnowledgeRuntime()`). When a Script Task is reached in the process, it will execute the action and then continue with the next node. It contains the following properties:

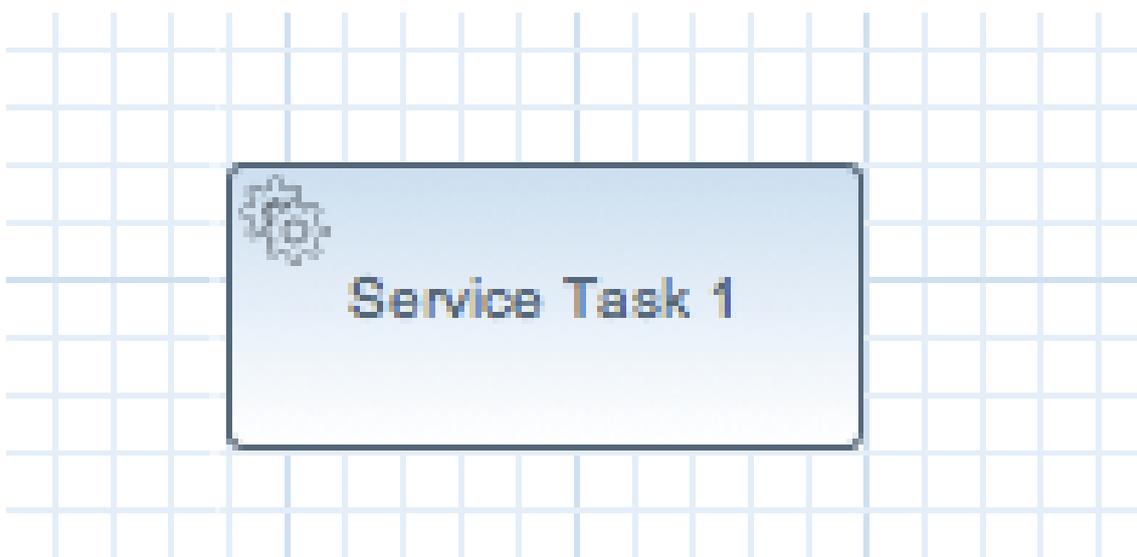
- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Action*: The action script associated with this action node.

Note that you can write any valid Java code inside a script node. This basically allows you to do anything inside such a script node. There are some caveats however:

- When trying to create a higher-level business process, that should also be understood by business users, it is probably wise to avoid low-level implementation details inside the process, including inside these script tasks. A Script Task could still be used to quickly manipulate variables etc. but other concepts like a Service Task could be used to model more complex behaviour in a higher-level manner.

- Scripts should be immediate. They are using the engine thread to execute the script. Scripts that could take some time to execute should probably be modeled as an asynchronous Service Task.
- You should try to avoid contacting external services through a script node. Not only does this usually violate the first two caveats, it is also interacting with external services without the knowledge of the engine, which can be problematic, especially when using persistence and transactions. In general, it is probably wiser to model communication with an external service using a service task.
- Scripts should not throw exceptions. Runtime exceptions should be caught and for example managed inside the script or transformed into signals or errors that can then be handled inside the process.

### 5.5.2. Service task



**Figure 5.13. Service task**

Represents 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. Users can define domain-specific services or work items, using a unique name and by defining the parameters (input) and results (output) that are associated with this type of work. Check the chapter on domain-specific processes for a detailed explanation and illustrative examples of how to define and use work items in your processes. When a Service Task is reached in the process, the associated work is executed. A Service Task should have one incoming connection and one outgoing connection.

- *Id*: The id of the node (which is unique within one node container).

- *Name*: The display name of the node.
- *Parameter mapping*: Allows copying the value of process variables to parameters of the work item. Upon creation of the work item, the values will be copied.
- *Result mapping*: Allows copying the value of result parameters of the work item to a process variable. Each type of work can define result parameters that will (potentially) be returned after the work item has been completed. A result mapping can be used to copy the value of the given result parameter to the given variable in this process. For example, the "FileFinder" work item returns a list of files that match the given search criteria within the result parameter `Files`. This list of files can then be bound to a process variable for use within the process. Upon completion of the work item, the values will be copied.
- *On-entry and on-exit actions*: Actions that are executed upon entry or exit of this node, respectively.
- *Additional parameters*: Each type of work item can define additional parameters that are relevant for that type of work. For example, the "Email" work item defines additional parameters such as `From`, `To`, `Subject` and `Body`. The user can either provide values for these parameters directly, or define a parameter mapping that will copy the value of the given variable in this process to the given parameter; if both are specified, the mapping will have precedence. Parameters of type `String` can use `#{expression}` to embed a value in the string. 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 simply be the name of a variable (in which case it resolves to the value of the variable), but more advanced MVEL expressions are possible as well, e.g., `#{person.name.firstname}`.

### 5.5.3. User task

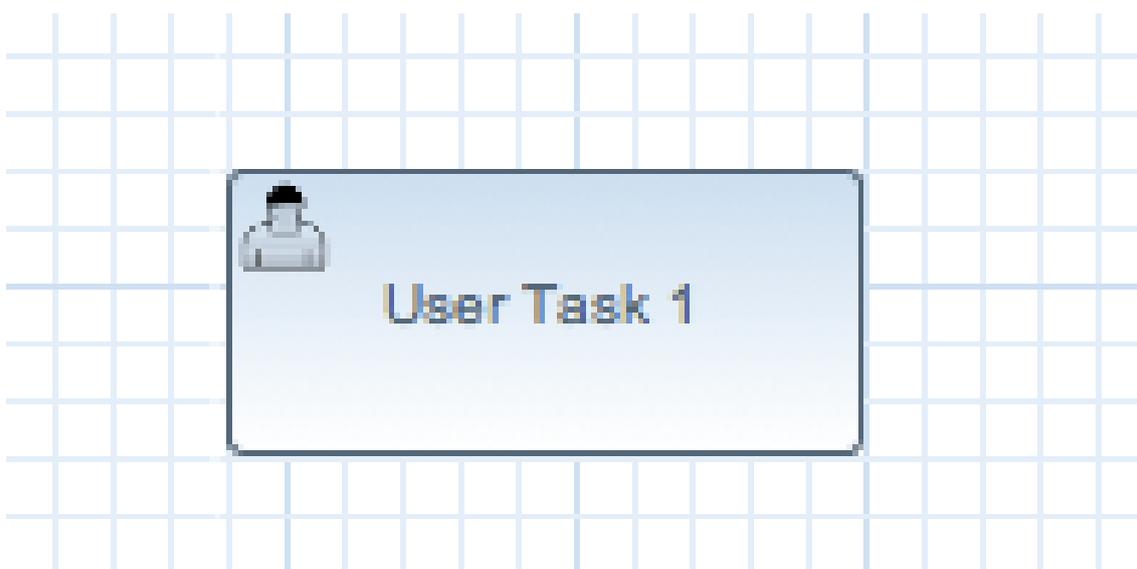


Figure 5.14. User task

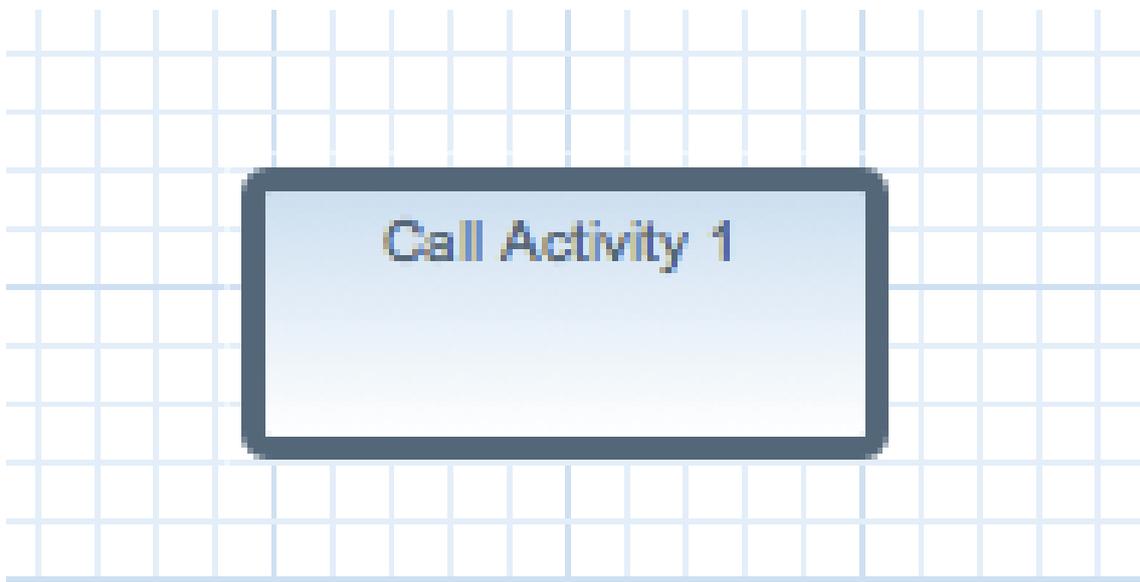
Processes can also involve tasks that need to be executed by human actors. A User Task represents an atomic task to be executed by a human actor. It should have one incoming connection and one outgoing connection. User Tasks can be used in combination with Swimlanes to assign multiple human tasks to similar actors. Refer to the chapter on human tasks for more details. A User Task is actually nothing more than a specific type of service node (of type "Human Task"). A User Task 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. See the human tasks chapter for more detail on how to use swimlanes.
- *On entry and on exit actions*: Action scripts that are executed upon entry and exit of this node, respectively.
- *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.

A user task should define the type of task that needs to be executed (using properties like TaskName, Comment, etc.) and who needs to perform it (using either actorId or groupId). Note that if there is data related to this specific process instance that the end user needs when performing the task, this data should be passed as the content of the task. The task for example does not

have access to process variables. Check out the chapter on human tasks to get more detail on how to pass data between human tasks and the process instance.

### 5.5.4. Reusable sub-process



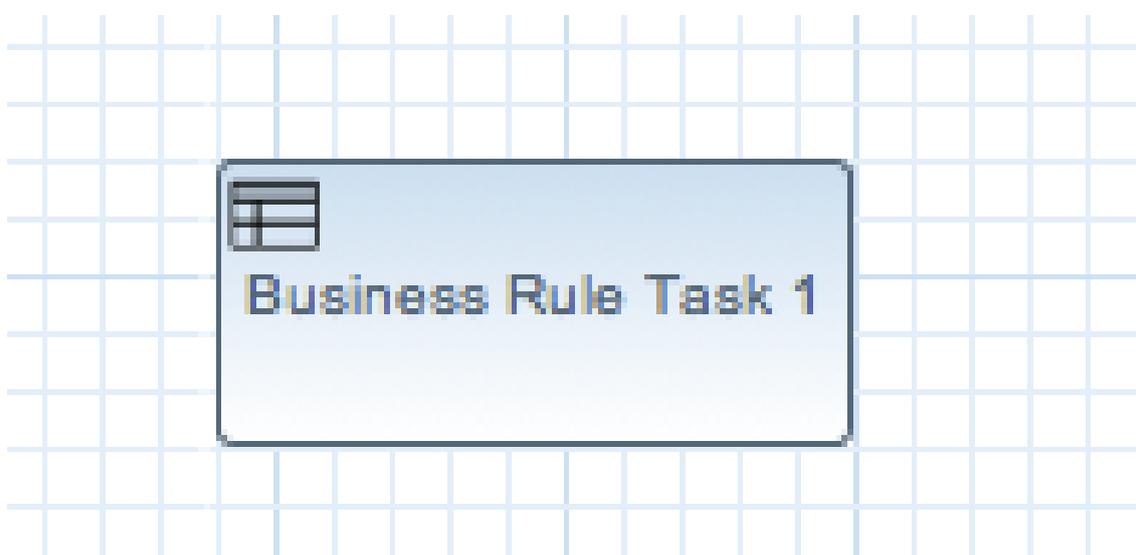
**Figure 5.15. Reusable sub-process - Call activity**

Represents the invocation of another process from within this process. A sub-process node should have one incoming connection and one outgoing connection. When a Reusable Sub-Process node is reached in the process, the engine will start the process with the given id. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *ProcessId*: The id of the process that should be executed.
- *Wait for completion* (by default true): If this property is true, this sub-process node will only continue if the child process that was started has terminated its execution (completed or aborted); otherwise it will continue immediately after starting the subprocess (so it will not wait for its completion).
- *Independent* (by default true): If this property is true, the child process is started as an independent process, which means that the child process will not be terminated if this parent process is completed (or this sub-process node is cancelled for some other reason); otherwise the active sub-process will be cancelled on termination of the parent process (or cancellation of the sub-process node). Note that you can only set independent to "false" only when "Wait for completion" is set to true.

- *On-entry and on-exit actions:* Actions that are executed upon entry or exit of this node, respectively.
- *Parameter in/out mapping:* A sub-process node can also 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 you can use "out" mappings only when "Wait for completion" is set to true.

### 5.5.5. Business rule task



**Figure 5.16. Business rule task**

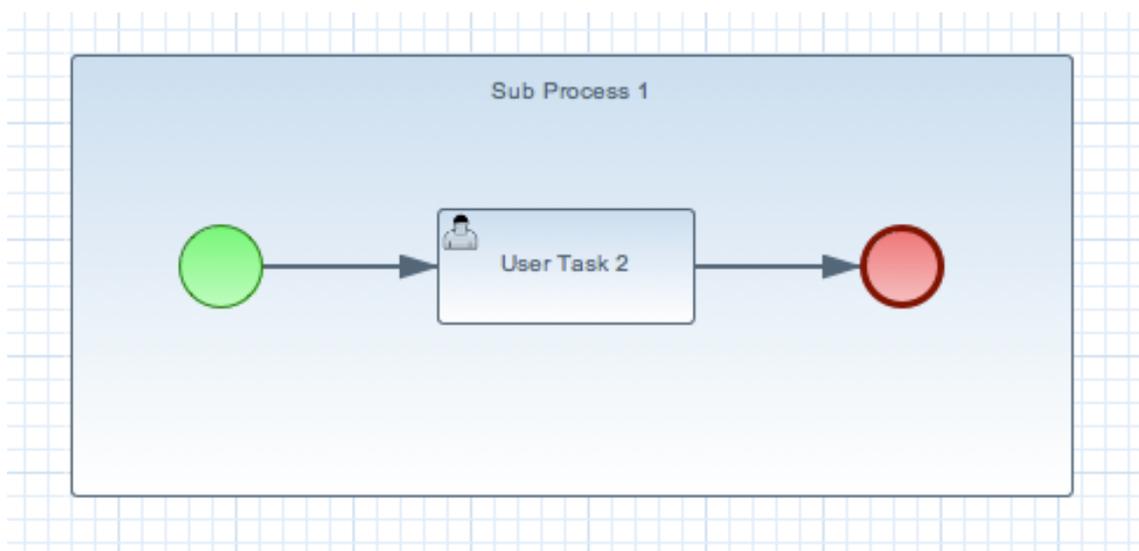
A Business Rule Task Represents a set of rules that need to be evaluated. The rules are evaluated when the node is reached. A Rule Task should have one incoming connection and one outgoing connection. Rules are defined in separate files using the Drools rule format. Rules can become part of a specific 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. As a result, during the execution of a ruleflow group, new activations belonging to the currently active ruleflow group can be added to the Agenda due to changes made to the facts by the other rules. Note that the process will immediately continue with the next node if it encounters a ruleflow group where there are no active rules at that time.

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 has been completed. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *RuleFlowGroup*: The name of the ruleflow group that represents the set of rules of this RuleFlowGroup node.

### 5.5.6. Embedded sub-process

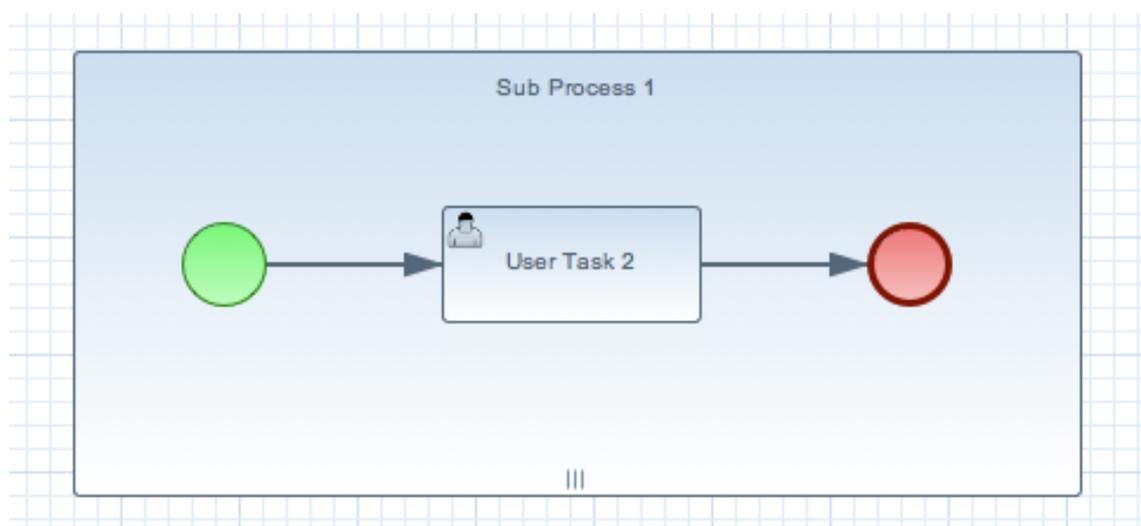


**Figure 5.17. Embedded sub-process**

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 should have one incoming connection and one outgoing connection. It should also contain one start node that defines where to start (inside the Sub-Process) when you reach the sub-process. It should also contain one or more end events. Note that, if you use a terminating event node inside a sub-process, you are terminating just that sub-process. A sub-process ends when there are no more active nodes inside the sub-process. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *Variables*: Additional variables can be defined to store data during the execution of this node. See section "[Data](#)" for details.

### 5.5.7. Multi-instance sub-process



**Figure 5.18. Multi-instance sub-process**

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. A multiple instance sub-process should have one incoming connection and one outgoing connection. It waits until the embedded process fragment is completed for each of the elements in the given collection before continuing. It contains the following properties:

- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *CollectionExpression*: The name of a variable that represents the collection of elements that should be iterated over. The collection variable should be an array or of type `java.util.Collection`. 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.
- *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.

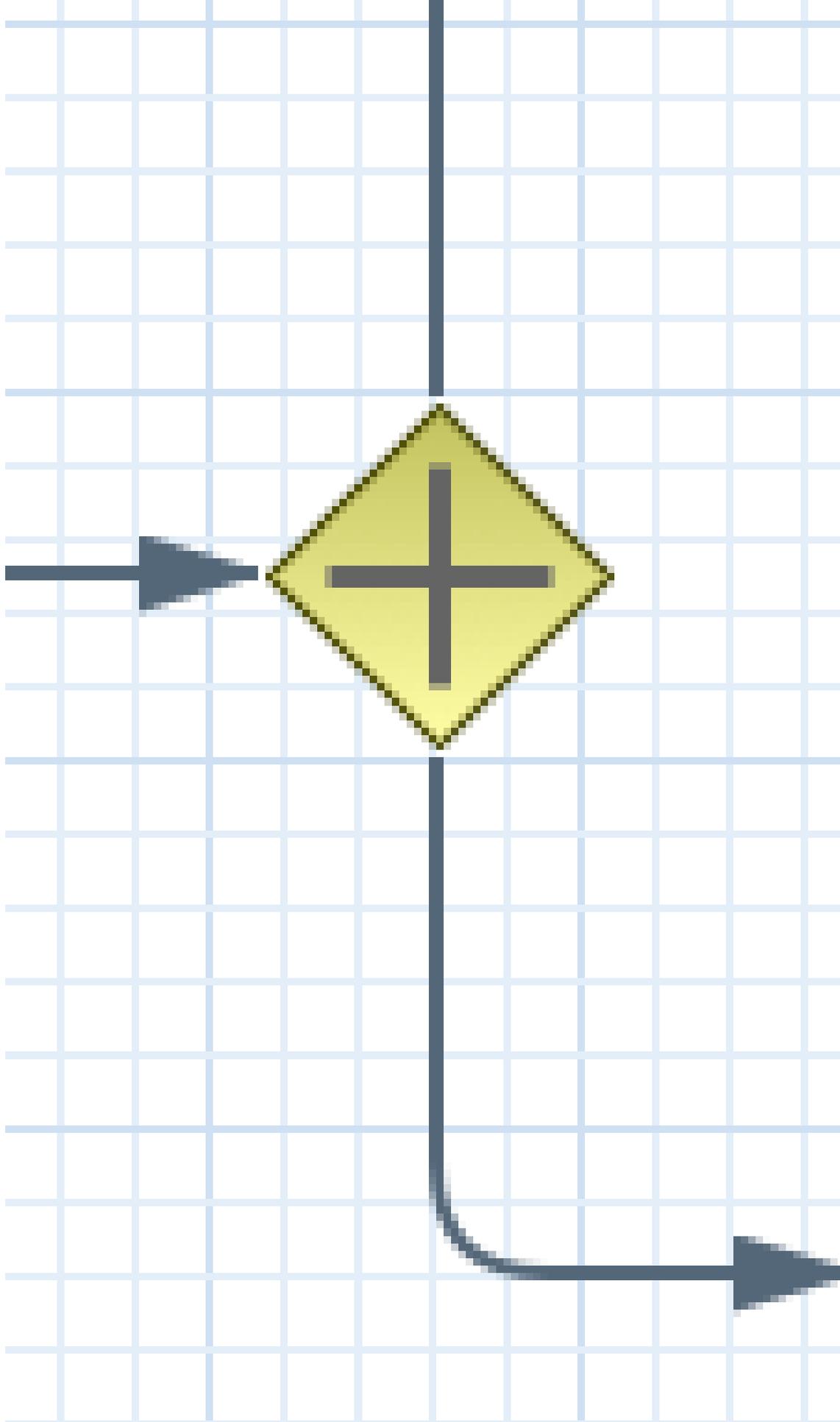


Figure 5.19. Diverging gateway

Allows you to create branches in your process. A Diverging Gateway should have one incoming connection and two or more outgoing connections. There are three types of gateway nodes currently supported:

- **AND** or parallel means that the control flow will continue in all outgoing connections simultaneously.
- **XOR** or exclusive means that exactly one of the outgoing connections will be chosen. The decision is made by evaluating the constraints that are linked to each of the outgoing connections. The constraint with the *lowest* priority number that evaluates to true is selected. Constraints can be specified using different dialects. Note that you should always make sure that at least one of the outgoing connections will evaluate to true at runtime (the engine will throw an exception at runtime if it cannot find at least one outgoing connection).
- **OR** or inclusive means that all outgoing connections whose condition evaluates to true are selected. Conditions are similar to the exclusive gateway, except that no priorities are taken into account. Note that you should make sure that at least one of the outgoing connections will evaluate to true at runtime because the engine will throw an exception at runtime if it cannot determine an outgoing connection.

It 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*: The type of the split node, i.e., AND, XOR or OR (see above).
- *Constraints*: The constraints linked to each of the outgoing connections (in case of an exclusive or inclusive gateway).

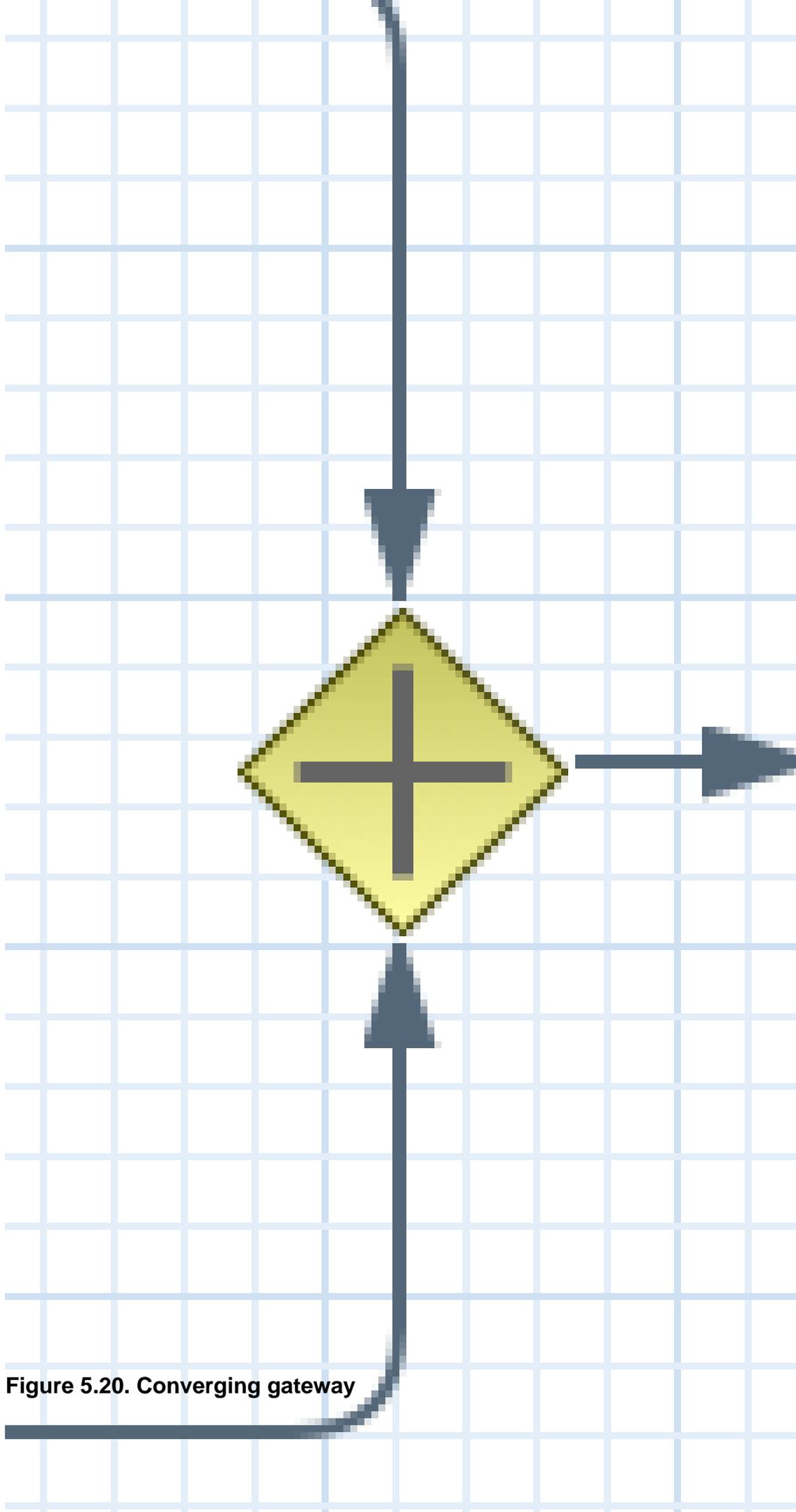


Figure 5.20. Converging gateway

Allows you to synchronize multiple branches. A Converging Gateway should have two or more incoming connections and one outgoing connection. There are three types of splits currently supported:

- AND or parallel means that it will wait until *all* incoming branches are completed before continuing.
- XOR or exclusive means that it continues as soon as *one* of its incoming branches has been completed. If it is triggered from more than one incoming connection, it will trigger the next node for each of those triggers.
- OR or inclusive means that it continues as soon as *all direct active paths* of its incoming branches has been completed. This is complex merge behaviour that is described in BPMN2 specification but in most cases it means that OR join will wait for all active flows that started in OR split. Some advanced cases (including other gateways in between or repeatable timers) will be causing different "direct active path" calculation.

It 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*: The type of the Join node, i.e. AND, OR or XOR.

## 5.7. Using a process in your application

As explained in more detail in the API chapter, there are two things you need to do to be able to execute processes from within your application: (1) you need to create a Knowledge Base that contains the definition of the process, and (2) you need to start the process by creating a session to communicate with the process engine and start the process.

1. *Creating a Knowledge Base*: Once you have a valid process, you can add the process to the Knowledge Base:

```
KieHelper kieHelper = new KieHelper();
KieBase kieBase = kieHelper
    .addResource(ResourceFactory.newClassPathResource("MyProcess.bpmn2"));
```

After adding all your process (you can add more than one process), you can create a new knowledge base like this:

```
KieBase kbase = kieHelper.build();
```

Note that this will throw an exception if the knowledge base contains errors (because it could not parse your processes correctly).

2. *Starting a process*: To start a particular process, you will need to call the `startProcess` method on your session and pass the id of the process you want to start. For example:

```
KieSession ksession = kbase.newKieSession();  
ksession.startProcess("com.sample.hello");
```

The parameter of the `startProcess` method is the id of the process that needs to be started. When defining a process, this process id needs to be specified as a property of the process (as for example shown in the Properties View in Eclipse when you click the background canvas of your process).

When you start the process, you may specify additional parameters that are used to pass additional input data to the process, using the `startProcess(String processId, Map parameters)` method. The additional set of parameters is a set of name-value pairs. These parameters are copied to the newly created process instance as top-level variables of the process, so they can be accessed in the remainder of your process directly.

## 5.8. Other features

### 5.8.1. Data

While the flow chart focuses on specifying the control flow of the process, it is usually also necessary to look at the process from a data perspective. Throughout the execution of a process, data can be retrieved, stored, passed on and used.

For storing runtime data, during the execution of the process, process variables can be used. A variable is defined by a name and a data type. This could be a basic data type, such as boolean, int, or String, or any kind of Object subclass (it must implement Serializable interface). Variables can be defined inside a variable scope. The top-level scope is the variable scope of the process itself. Subscopes can be defined using a Sub-Process. Variables that are defined in a subscope 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 of variable scopes is allowed. A node will always search for a variable in its parent container. If the variable cannot be found, it will look in that one'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, with the process continuing its execution.

Variables can be used in various 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` right before the service is being invoked. You can also inject the value of process variable into a hard-coded parameter String using `#{expression}`. For example, the description of a human task could be defined as `You need to contact person #{person.getName()}` (where `person` is a process variable), which will replace this expression by the actual name of the person when the service needs to be invoked. Similarly results of a service (or reusable sub-process) can also be copied back to a variable using a result mapping.
- Various other nodes can also access data. Event nodes for example can store the data associated to the event in a variable, etc. Check the properties of the different node types for more information.
- Process variables can be accessed also from the Java code of your application. It is done by casting of `ProcessInstance` to `WorkflowProcessInstance`. See the following example:

```
variable = ((WorkflowProcessInstance) processInstance).getVariable("variableName");
```

To list all the process variables see the following code snippet:

```
org.jbpm.process.instance.ProcessInstance processInstance = ...;
VariableScopeInstance variableScope = (VariableScopeInstance) processInstance.getContextInstance();
Map<String, Object> variables = variableScope.getVariables();
```

Note that when you use persistence then you have to use a command based approach to get all process variables:

```
Map<String, Object> variables = ksession.execute(new GenericCommand<Map<String, Object>>() {
    public Map<String, Object> execute(Context context) {
        KieSession ksession = ((KnowledgeCommandContext) context).getStatefulKnowledgeSession();
        org.jbpm.process.instance.ProcessInstance processInstance = (org.jbpm.process.instance.ProcessInstance) ksession.getContextInstance();
        VariableScopeInstance variableScope = (VariableScopeInstance) processInstance.getContextInstance();
        Map<String, Object> variables = variableScope.getVariables();
        return variables;
    }
});
```

Finally, processes (and rules) all have access to globals, i.e. globally defined variables and data in the Knowledge Session. Globals are directly accessible in actions just like variables. Globals need to be defined as part of the process before they can be used. You can for example define globals by clicking the globals button when specifying an action script in the Eclipse action property editor. You can also set the value of a global from the outside using `ksession.setGlobal(name, value)` or from inside process scripts using `kcontext.getKnowledgeRuntime().setGlobal(name, value);`

### 5.8.2. Constraints

Constraints can be used in various locations in your processes, for example in a diverging gateway. jBPM supports two types of constraints:

- *Code constraints* are boolean expressions, evaluated directly whenever they are reached. We currently support two dialects for expressing these code constraints: Java and 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;
```

A similar example of a valid MVEL code constraint is:

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

- *Rule constraints* are equals to normal Drools rule conditions. They use the Drools Rule Language syntax to express possibly 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 being in the Working Memory.

Rule constraints do not have direct access to variables defined inside the process. It is however 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. We have added special logic to make sure that a variable `processInstance` of type `WorkflowProcessInstance` will only match to the current process instance and not to other process instances in the Working Memory. Note that you are however responsible yourself to insert the process instance into the session and, possibly, to update it, for example, using Java code or an on-entry or on-exit or explicit action in your 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 ...
```

### 5.8.3. Action scripts

Action scripts can be used in different ways:

- Within a Script Task,
- As entry or exit actions, with a number of nodes.

Actions have access to globals and the variables that are defined for the process and the predefined variable `kcontext`. This variable is of type `org.kie.api.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.

```
ProcessInstance proc = kcontext.getProcessInstance();
proc.signalEvent( type, eventObject );
```

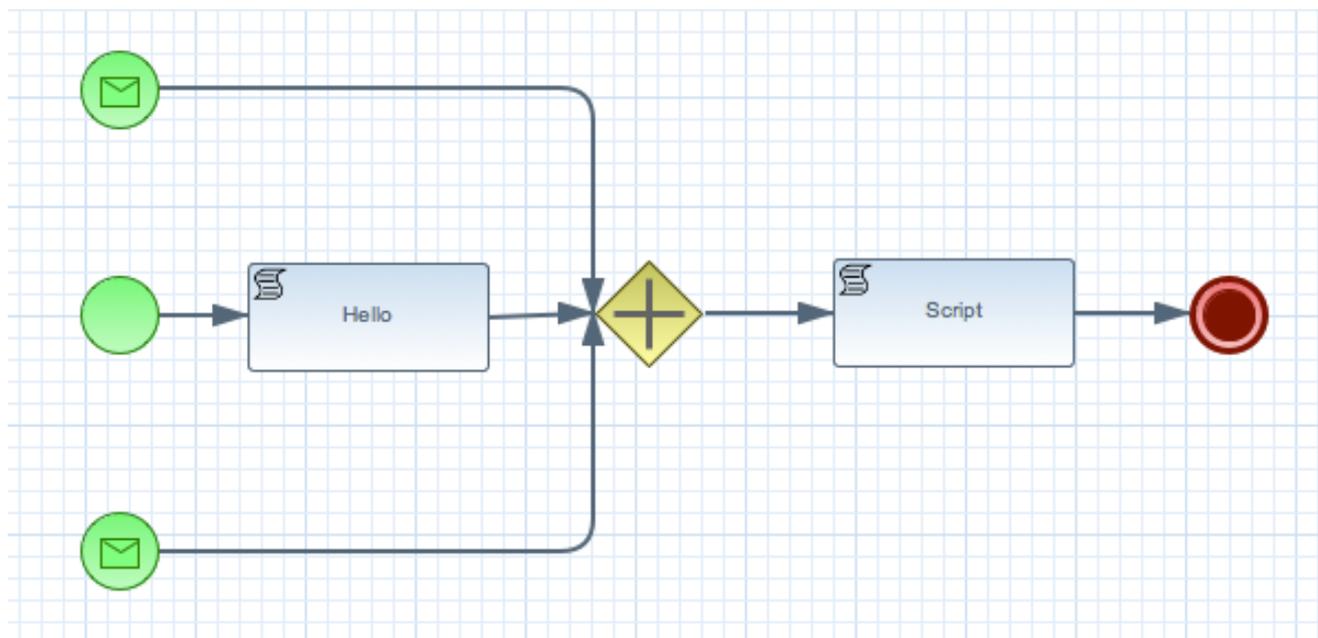
- Getting or setting the value of variables.
- Accessing the Knowledge Runtime allows you do things like starting a process, signaling (external) events, inserting data, etc.

jBPM currently supports two dialects, Java and MVEL. 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 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 );
```

### 5.8.4. Events



**Figure 5.21. A sample process using events**

During the execution of a process, the process engine makes sure that all the relevant tasks are executed according to the process plan, by requesting the execution of work items and waiting for the results. However, it is also possible that the process should respond to events that were not directly requested by the process engine. Explicitly representing these events in a process allows the process author to specify how the process should react to such events.

Events have a type and possibly data associated with them. Users are free to define their own event types and their 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, using code like the following:

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

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

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

- External event using event correlation: Instead of notifying a process instance directly, it is also 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, write code such as:

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

Events could also be used to start a process. Whenever a Message Start Event defines an event trigger of a specific type, a new process instance will be started every time that type of event is signalled to the process engine.

### 5.8.5. Timers

Timers wait for a predefined amount of time, before triggering, once or repeatedly. They can be used to trigger certain logic after a certain period, or to repeat some action at regular intervals.

#### 5.8.5.1. Configure timer with delay and period

A Timer node is set up with 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 one-shot timer.

The (period and delay) expression should be of the form `[#d][#h][#m][#s][#ms]`. You can specify the amount of days, hours, minutes, seconds and milliseconds (which is the default if you don't specify anything). For example, the expression "1h" will wait one hour before triggering the timer (again).

#### 5.8.5.2. Configure timer ISO-8601 date format

since version 6 timers can be configured with valid [ISO8601](http://en.wikipedia.org/wiki/ISO_8601) [http://en.wikipedia.org/wiki/ISO\_8601] date format that supports both one shot timers and repeatable timers. Timers can be defined as data dn time representation, time duration or repeating intervals

- Date - 2013-12-24T20:00:00.000+02:00 - fires exactly at Christmas Eve at 8PM
- Duration - PT1S - fires once after 1 second
- Repeatable intervals - R/PT1S - fires every second, no limit, alternatively R5/PT1S will fire 5 times every second

### 5.8.5.3. Configure timer with process variables

In addition to two configuration options above timers can be specified using process variable that consists of string representation of either delay and period or ISO8601 date format. By specifying `#{variable}` engine will dynamically extract process variable and use it as timer expression.

The timer service is responsible for making sure that timers get triggered at the appropriate times. Timers can also be cancelled, meaning that the timer will no longer be triggered.

Timers can be used in two ways inside a process:

- 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. Subsequently, the outgoing connection of a timer with a positive period is triggered multiple times. Cancelling a Timer node also cancels the associated timer, after which no more triggers will occur.
- Timers can be associated with a Sub-Process or tasks as a boundary event.

### 5.8.6. Updating processes

Over time, processes may evolve, for example because the process itself needs to be improved, or due to changing requirements. Actually, you cannot really update a process, you can only deploy a new version of the process, the old process will still exist. That is because existing process instances might still need that process definition. So the new process should have a different id, though the name could be the same, and you can use the version parameter to show when a process is updated (the version parameter is just a String and is not validated by the process framework itself, so you can select your own format for specifying minor/major updates, etc.).

Whenever a process is updated, it is important to determine what should happen to the already running process instances. There are various strategies one could consider for each running instance:

- *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, meaning that - once it has been migrated successfully - it will continue executing based on the updated process logic.

By default, jBPM uses the proceed approach, meaning that multiple versions of the same process can be deployed, but existing process instances will simply continue executing based on the process definition that was used when starting the process instance. Running process instances could always be aborted as well of course, using the process management API. Process instance migration is more difficult and is explained in the following paragraphs.

### 5.8.6.1. 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 (as variables), but also the current state in the process diagram. For each node that is currently active, a node instance is used to represent this. This node instance can also contain additional state linked to the execution of that specific node only. 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 (this clear separation of definition and runtime state allows reuse of the definition across all process instances based on this process and 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 to the new id.

However, this does not take into account that the state of the process instance (the variable instances and the node instances) might need to be migrated as well. In cases where the process is only extended and all existing wait states are kept, this is pretty straightforward, 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 that is waiting in that state cannot simply be updated. Or when a new process variable is introduced, that 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. Of course, you need to provide the process instance and the new process id. By default, jBPM will automatically map old node instances to 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 inbetween), 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 simply the new node id in the node container (so no unique node id here, simply the new node id). The following code snippet shows a simple example.

```
// create the session and start the process "com.sample.process"
KieSession 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 still insufficient, you can still describe your own custom mappers for specific situations. Be sure to first disconnect the process instance, change the state accordingly and then reconnect the process instance, similar to how the `WorkflowProcessInstanceUpgrader` does it.

## 5.8.7. 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, for example. From a functional standpoint, the original process will then split into two processes that are executed in a parallel fashion.

Of course, the jBPM engine supports logical multi-threading: for example, processes that include a parallel gateway. We've chosen to implement logical multi-threading using one thread: a jBPM 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.

### 5.8.7.1. Engine execution

In general, the jBPM 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.

### 5.8.7.2. Asynchronous handlers

How can we implement an asynchronous service handler? To start with, this depends on the technology you're using. If you're only using Java, you could 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) {
    }

}
```

It's advisable to have your handler contact a service that executes the business operation, instead of having it perform the actual work. If anything goes wrong with a business operation, it doesn't affect your process. The loose coupling that this provides also gives you greater flexibility in reusing services and developing them.

For example, you can have your human task handler simply invoke the human task service to add a task there. To implement an asynchronous handler, you usually have to simply 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, or sending a JMS message to the external service.

---

jbpm-executor component provides advanced capabilities for asynchronous execution to ease the development efforts to empower solution when using jBPM. It provides out of the box

- persistent storage of the requested operations
- retry mechanism
- error handling
- history log of executions

See chapter jbpm executor for more details.

### 5.8.7.3. Multiple knowledge sessions and persistence

The simplest way to run multiple processes is to run them all using one knowledge session. However, there are cases in which it's necessary to run multiple processes in different knowledge sessions, even in different (technical) threads. Both are supported by jBPM.

When we add persistence (using a database, for example) to a situation in which we have multiple knowledge sessions (and processes), there is a guideline that users should be aware of. The following paragraphs explain why this guideline is important to follow.

- Please make sure to use a database that allows row-level locks as well as table-level locks.

For example, a user could have a situation in which there are 2 (or more) threads running, each with its own knowledge session instance. On each thread, jBPM processes are being started using the local knowledge session instance.

In this use case, a race condition exists in which both thread A and thread B will have coincidentally simultaneously finished a process. At this point, because persistence is being used, both thread A and B will be committing changes to the database. If row-level locks are *not* possible, then the following situation can occur:

- Thread A has a lock on the `ProcessInstanceInfo` table, having just committed a change to that table.
- Thread A *wants* a lock on the `SessionInfo` table in order to commit a change there.
- Thread B has the opposite situation: it has a lock on the `SessionInfo` table, having just committed a change there.
- Thread B *wants* a lock on the `ProcessInstanceInfo` table, even though Thread A already has a lock on it

This is a deadlock situation which the database and application will not be able to solve.

However, if row-level locks are possible (and enabled!!) in the database (and tables used), then this situation will not occur.



# Chapter 6. Core Engine: BPMN 2.0

## 6.1. Business Process Model and Notation (BPMN) 2.0 specification



### Note

"The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes."

The Business Process Model and Notation (BPMN) 2.0 specification is an OMG specification that not only defines a standard on how to graphically represent a business process (like BPMN 1.x), but now also includes execution semantics for the elements defined, and an XML format on how to store (and share) process definitions.

jBPM6 allows you to execute processes defined using the BPMN 2.0 XML format. That means that you can use all the different jBPM6 tooling to model, execute, manage and monitor your business processes using the BPMN 2.0 format for specifying your executable business processes. Actually, the full BPMN 2.0 specification also includes details on how to represent things like choreographies and collaboration. The jBPM project however focuses on that part of the specification that can be used to specify executable processes.

Executable processes in BPMN consist of a different types of nodes being connected to each other using sequence flows. The BPMN 2.0 specification defines three main types of nodes:

- **Events:** They are used to model the occurrence of a particular event. This could be a start event (that is used to indicate the start of the process), end events (that define the end of the process, or of that subflow) and intermediate events (that indicate events that might occur during the execution of the process).
- **Activities:** These define the different actions that need to be performed during the execution of the process. Different types of tasks exist, depending on the type of activity you are trying to model (e.g. human task, service task, etc.) and activities could also be nested (using different types of sub-processes).
- **Gateways:** Can be used to define multiple paths in the process. Depending on the type of gateway, these might indicate parallel execution, choice, etc.

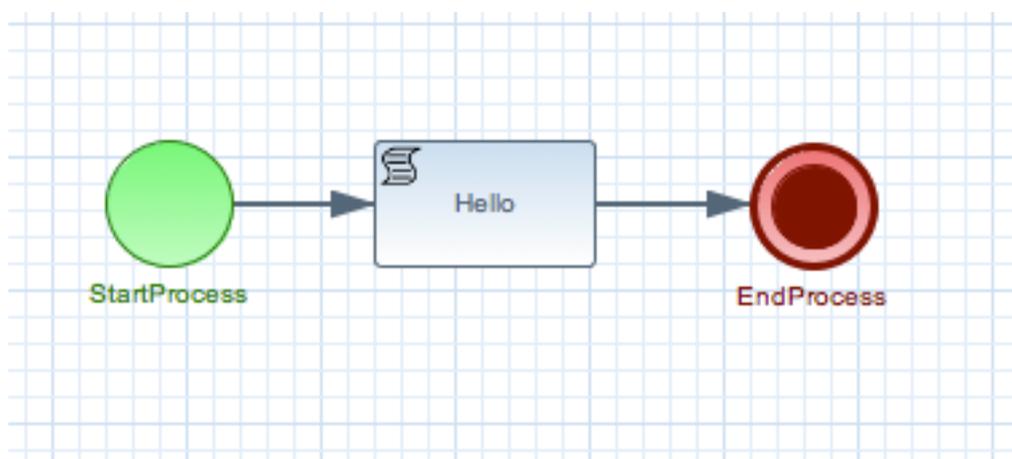
jBPM6 does not implement all elements and attributes as defined in the BPMN 2.0 specification. We do however support a significant subset, including the most common node types that can be used inside executable processes. This includes (almost) all elements and attributes as defined in

the "Common Executable" subclass of the BPMN 2.0 specification, extended with some additional elements and attributes we believe are valuable in that context as well. The full set of elements and attributes that are supported can be found below, but it includes elements like:

- *Flow objects*
  - Events
    - Start Event (None, Conditional, Signal, Message, Timer)
    - End Event (None, Terminate, Error, Escalation, Signal, Message, Compensation)
    - Intermediate Catch Event (Signal, Timer, Conditional, Message)
    - Intermediate Throw Event (None, Signal, Escalation, Message, Compensation)
    - Non-interrupting Boundary Event (Escalation, Signal, Timer, Conditional, Message)
    - Interrupting Boundary Event (Escalation, Error, Signal, Timer, Conditional, Message, Compensation)
  - Activities
    - Script Task
    - Task
    - Service Task
    - User Task
    - Business Rule Task
    - Manual Task
    - Send Task
    - Receive Task
    - Reusable Sub-Process (Call Activity)
    - Embedded Sub-Process
    - Event Sub-Process
    - Ad-Hoc Sub-Process
    - Data-Object
  - Gateways
    - Diverging

- Exclusive
- Inclusive
- Parallel
- Event-Based
- Converging
  - Exclusive
  - Inclusive
  - Parallel
- Lanes
- *Data*
  - Java type language
  - Process properties
  - Embedded Sub-Process properties
  - Activity properties
- *Connecting objects*
  - Sequence flow

For example, consider the following "Hello World" BPMN 2.0 process, which does nothing more than writing out a "Hello World" statement when the process is started.



An executable version of this process expressed using BPMN 2.0 XML would look something like this:

```

<?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" Executable="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>

```

To create your own process using BPMN 2.0 format, you can

- The jBPM Designer is an open-source web-based editor that supports the BPMN 2.0 format. We have embedded it into jbpm console for BPMN 2.0 process visualization and editing. You could use the Designer (either standalone or integrated) to create / edit BPMN 2.0 processes and then export them to BPMN 2.0 format or save them into repository and import them so they can be executed.
- A new BPMN2 Eclipse plugin is being created to support the full BPMN2 specification.
- You can always manually create your BPMN 2.0 process files by writing the XML directly. You can validate the syntax of your processes against the BPMN 2.0 XSD, or use the validator in the Eclipse plugin to check both syntax and completeness of your model.



### Note

Drools Eclipse Process editor has been deprecated in favor of BPMN2 Modeler for process modeling. It can still be used for limited number of supported elements but should be faced out as it is not being developed any more.

Create a new Process file using the Drools Eclipse plugin wizard and in the last page of the wizard, make sure you select Drools 5.1 code compatibility. This will create a new process using the BPMN 2.0 XML format. Note however that this is not exactly a BPMN 2.0 editor, as it still uses different attributes names etc. It does however save the process using valid BPMN 2.0 syntax. Also note that the editor does not support all node types and attributes that are already supported in the execution engine.

The following code fragment shows you how to load a BPMN2 process into your knowledge base ...

```

private static KnowledgeBase createKnowledgeBase() throws Exception {
    KieHelper kieHelper = new KieHelper();
    KieBase kieBase = kieHelper
        .addResource(ResourceFactory.newClassPathResource("sample.bpmn2"))

```

```
.build();

return kieBase;
}
```

... and how to execute this process ...

```
KieBase kbase = createKnowledgeBase();
KieSession ksession = kbase.newKieSession();
ksession.startProcess("com.sample.HelloWorld");
```

For more detail, check out the chapter on the API and the basics.

## 6.2. Supported elements / attributes

Table 6.1. Keywords

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
definitions		<ul style="list-style-type: none"> <li>rootElement</li> <li>BPMNDiagram</li> </ul>		
process	<ul style="list-style-type: none"> <li>processType</li> <li>isExecutable</li> <li>name</li> <li>id</li> </ul>	<ul style="list-style-type: none"> <li>property</li> <li>laneSet</li> <li>flowElement</li> </ul>	<ul style="list-style-type: none"> <li>packageName</li> <li>adHoc</li> <li>version</li> </ul>	<ul style="list-style-type: none"> <li>import</li> <li>global</li> </ul>
sequenceFlow	<ul style="list-style-type: none"> <li>sourceRef</li> <li>targetRef</li> <li>isImmediate</li> <li>name</li> <li>id</li> </ul>	<ul style="list-style-type: none"> <li>conditionExpression</li> </ul>	<ul style="list-style-type: none"> <li>priority</li> </ul>	
interface	<ul style="list-style-type: none"> <li>name</li> <li>id</li> <li>implementationRef</li> </ul>	<ul style="list-style-type: none"> <li>operation</li> </ul>		
operation	<ul style="list-style-type: none"> <li>name</li> <li>id</li> </ul>	<ul style="list-style-type: none"> <li>inMessageRef</li> </ul>		

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
laneSet	<ul style="list-style-type: none"> <li>implementationRef</li> </ul>	<ul style="list-style-type: none"> <li>lane</li> </ul>		
lane	<ul style="list-style-type: none"> <li>name</li> <li>id</li> </ul>	<ul style="list-style-type: none"> <li>flowNodeRef</li> </ul>		
import*		<ul style="list-style-type: none"> <li>name</li> </ul>		
global*		<ul style="list-style-type: none"> <li>identifier</li> <li>type</li> </ul>		
<b>Events</b>				
startEvent	<ul style="list-style-type: none"> <li>name</li> <li>id</li> <li>isInterrupting</li> </ul>	<ul style="list-style-type: none"> <li>dataOutput</li> <li>dataOutputAssociation</li> <li>outputSet</li> <li>eventDefinition</li> </ul>		
endEvent	<ul style="list-style-type: none"> <li>name</li> <li>id</li> </ul>	<ul style="list-style-type: none"> <li>dataInput</li> <li>dataInputAssociation</li> <li>inputSet</li> <li>eventDefinition</li> </ul>		
intermediateCatchEvent	<ul style="list-style-type: none"> <li>name</li> <li>id</li> </ul>	<ul style="list-style-type: none"> <li>dataOutput</li> <li>dataOutputAssociation</li> <li>outputSet</li> <li>eventDefinition</li> </ul>		
intermediateThrowEvent	<ul style="list-style-type: none"> <li>name</li> <li>id</li> </ul>	<ul style="list-style-type: none"> <li>dataInput</li> <li>dataInputAssociation</li> <li>inputSet</li> <li>eventDefinition</li> </ul>		
boundaryEvent	<ul style="list-style-type: none"> <li>cancelActivity</li> <li>attachedToRef</li> <li>name</li> </ul>	<ul style="list-style-type: none"> <li>eventDefinition</li> </ul>		

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
terminateEventDefinition	<ul style="list-style-type: none"> <li>• id</li> </ul>			
compensateEventDefinitionRef		<ul style="list-style-type: none"> <li>• documentation</li> <li>• extensionElements</li> </ul>		
conditionalEventDefinition		<ul style="list-style-type: none"> <li>• condition</li> </ul>		
errorEventDefinition	<ul style="list-style-type: none"> <li>• errorRef</li> </ul>			
error	<ul style="list-style-type: none"> <li>• errorCode</li> </ul>			
escalationEventDefinitionRef	<ul style="list-style-type: none"> <li>• id</li> </ul>			
escalation	<ul style="list-style-type: none"> <li>• escalationCode</li> </ul>			
messageEventDefinitionRef	<ul style="list-style-type: none"> <li>• id</li> </ul>			
message	<ul style="list-style-type: none"> <li>• itemRef</li> </ul>			
signalEventDefinitionRef	<ul style="list-style-type: none"> <li>• id</li> </ul>			
signalEventDefinition	<ul style="list-style-type: none"> <li>• signalRef</li> </ul>			
timerEventDefinition		<ul style="list-style-type: none"> <li>• timeCycle</li> <li>• timeDuration</li> <li>• timerDate</li> </ul>		
<b>Activities</b>				
task	<ul style="list-style-type: none"> <li>• name</li> <li>• id</li> </ul>	<ul style="list-style-type: none"> <li>• ioSpecification</li> <li>• dataInputAssociation</li> <li>• dataOutputAssociation</li> </ul>	<ul style="list-style-type: none"> <li>• taskName</li> </ul>	
scriptTask	<ul style="list-style-type: none"> <li>• scriptFormat</li> <li>• name</li> </ul>	<ul style="list-style-type: none"> <li>• script</li> </ul>		
script	<ul style="list-style-type: none"> <li>• id</li> </ul>	<ul style="list-style-type: none"> <li>• text[mixed content]</li> </ul>		
userTask	<ul style="list-style-type: none"> <li>• name</li> <li>• id</li> </ul>	<ul style="list-style-type: none"> <li>• ioSpecification</li> <li>• dataInputAssociation</li> </ul>		<ul style="list-style-type: none"> <li>• onEntry-script</li> <li>• onExit-script</li> </ul>

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
potentialOwner		<ul style="list-style-type: none"> <li>• dataOutputAssociation</li> <li>• resourceRole</li> <li>• loopCharacteristics</li> <li>• resourceAssignmentExpression</li> </ul>		
resourceAssignmentExpression		<ul style="list-style-type: none"> <li>• expression</li> </ul>		
businessRuleTask	<ul style="list-style-type: none"> <li>• name</li> <li>• id</li> </ul>	<ul style="list-style-type: none"> <li>• ioSpecification</li> <li>• dataInputAssociation</li> <li>• dataOutputAssociation</li> </ul>	<ul style="list-style-type: none"> <li>• ruleFlowGroup</li> </ul>	<ul style="list-style-type: none"> <li>• onEntry-script</li> <li>• onExit-script</li> </ul>
manualTask	<ul style="list-style-type: none"> <li>• name</li> <li>• id</li> </ul>			<ul style="list-style-type: none"> <li>• onEntry-script</li> <li>• onExit-script</li> </ul>
sendTask	<ul style="list-style-type: none"> <li>• messageRef</li> <li>• name</li> <li>• id</li> </ul>	<ul style="list-style-type: none"> <li>• ioSpecification</li> <li>• dataInputAssociation</li> <li>• loopCharacteristics</li> <li>• ioSpecification</li> </ul>		<ul style="list-style-type: none"> <li>• onEntry-script</li> <li>• onExit-script</li> </ul>
receiveTask	<ul style="list-style-type: none"> <li>• messageRef</li> <li>• name</li> <li>• id</li> </ul>	<ul style="list-style-type: none"> <li>• dataOutputAssociation</li> <li>• loopCharacteristics</li> <li>• ioSpecification</li> </ul>		<ul style="list-style-type: none"> <li>• onEntry-script</li> <li>• onExit-script</li> </ul>
serviceTask	<ul style="list-style-type: none"> <li>• operationRef</li> <li>• name</li> <li>• id</li> </ul>	<ul style="list-style-type: none"> <li>• ioSpecification</li> <li>• dataInputAssociation</li> <li>• dataOutputAssociation</li> </ul>		<ul style="list-style-type: none"> <li>• onEntry-script</li> <li>• onExit-script</li> </ul>
subProcess	<ul style="list-style-type: none"> <li>• implementation</li> <li>• name</li> <li>• id</li> </ul>	<ul style="list-style-type: none"> <li>• loopCharacteristics</li> <li>• flowElement</li> <li>• property</li> </ul>		
adHocSubProcess	<ul style="list-style-type: none"> <li>• triggeredByEvent</li> <li>• cancelRemainingInstances</li> <li>• name</li> <li>• id</li> </ul>	<ul style="list-style-type: none"> <li>• loopCharacteristics</li> <li>• extensionCondition</li> <li>• flowElement</li> <li>• property</li> </ul>		
callActivity	<ul style="list-style-type: none"> <li>• calledElement</li> <li>• name</li> </ul>	<ul style="list-style-type: none"> <li>• ioSpecification</li> <li>• dataInputAssociation</li> </ul>	<ul style="list-style-type: none"> <li>• waitForCompletion</li> <li>• independent</li> </ul>	<ul style="list-style-type: none"> <li>• onEntry-script</li> <li>• onExit-script</li> </ul>

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
multiInstanceLoop	<ul style="list-style-type: none"> <li>id</li> </ul>	<ul style="list-style-type: none"> <li>dataOutputAssociation</li> <li>loopDataInputRef</li> <li>inputDataItem</li> <li>loopDataOutputRef</li> <li>outputDataItem</li> </ul>		
onEntry-script*	<ul style="list-style-type: none"> <li>scriptFormat</li> </ul>		<ul style="list-style-type: none"> <li>script</li> </ul>	
onExit-script*	<ul style="list-style-type: none"> <li>scriptFormat</li> </ul>		<ul style="list-style-type: none"> <li>script</li> </ul>	
<b>Gateways</b>				
parallelGateway	<ul style="list-style-type: none"> <li>gatewayDirection</li> <li>name</li> <li>id</li> </ul>			
eventBasedGateway	<ul style="list-style-type: none"> <li>gatewayDirection</li> <li>name</li> <li>id</li> </ul>			
exclusiveGateway	<ul style="list-style-type: none"> <li>default</li> <li>gatewayDirection</li> <li>name</li> <li>id</li> </ul>			
inclusiveGateway	<ul style="list-style-type: none"> <li>default</li> <li>gatewayDirection</li> <li>name</li> <li>id</li> </ul>			
<b>Data</b>				
property	<ul style="list-style-type: none"> <li>itemSubjectRef</li> <li>id</li> <li>name</li> </ul>			
dataObject	<ul style="list-style-type: none"> <li>itemSubjectRef</li> </ul>			

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
itemDefinition	<ul style="list-style-type: none"> <li>id</li> <li>structureRef</li> </ul>			
ioSpecification	<ul style="list-style-type: none"> <li>id</li> </ul>	<ul style="list-style-type: none"> <li>dataInput</li> <li>dataOutput</li> <li>inputSet</li> <li>outputSet</li> </ul>		
dataInput	<ul style="list-style-type: none"> <li>name</li> <li>id</li> </ul>			
dataInputAssociation		<ul style="list-style-type: none"> <li>sourceRef</li> <li>targetRef</li> <li>assignment</li> </ul>		
dataOutput	<ul style="list-style-type: none"> <li>name</li> <li>id</li> </ul>			
dataOutputAssociation		<ul style="list-style-type: none"> <li>sourceRef</li> <li>targetRef</li> <li>assignment</li> </ul>		
inputSet		<ul style="list-style-type: none"> <li>dataInputRefs</li> </ul>		
outputSet		<ul style="list-style-type: none"> <li>dataOutputRefs</li> </ul>		
assignment		<ul style="list-style-type: none"> <li>from</li> <li>to</li> </ul>		
formalExpression	<ul style="list-style-type: none"> <li>language</li> </ul>	<ul style="list-style-type: none"> <li>text[mixed content]</li> </ul>		
<b>BPMNDI</b>				
BPMNDiagram		<ul style="list-style-type: none"> <li>BPMNPlane</li> </ul>		
BPMNPlane	<ul style="list-style-type: none"> <li>bpmnElement</li> </ul>	<ul style="list-style-type: none"> <li>BPMNEdge</li> <li>BPMNShape</li> </ul>		
BPMNShape	<ul style="list-style-type: none"> <li>bpmnElement</li> </ul>	<ul style="list-style-type: none"> <li>Bounds</li> </ul>		
BPMNEdge	<ul style="list-style-type: none"> <li>bpmnElement</li> </ul>	<ul style="list-style-type: none"> <li>waypoint</li> </ul>		

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
Bounds	<ul style="list-style-type: none"> <li>• x</li> <li>• y</li> <li>• width</li> <li>• height</li> </ul>			
waypoint	<ul style="list-style-type: none"> <li>• x</li> <li>• y</li> </ul>			

### 6.3. Examples

The BPMN 2.0 specification defines the attributes and semantics of each of the node types (and other elements).

TODO: provide more examples with more advanced constructs

The jbpm-bpmn2 module contains a lot of junit tests for each of the different node types. These test processes can also serve as simple examples: they don't really represent an entire real life business processes but can definitely be used to show how specific features can be used. For example, the following figures shows the flow chart of a few of those examples. The entire list can be found in the src/test/resources folder for the jbpm-bpmn2 module like [here](http://github.com/droolsjbpm/jbpm/tree/master/jbpm-bpmn2/src/test/resources/) [http://github.com/droolsjbpm/jbpm/tree/master/jbpm-bpmn2/src/test/resources/].

# Chapter 7. Core Engine:

## Persistence and transactions

jBPM allows the persistent storage of certain information. This chapter describes these different types of persistence, and how to configure them. An example of the information stored is the process runtime state. Storing the process runtime state is necessary in order to be able to continue execution of a process instance at any point, if something goes wrong. Also, the process definitions themselves, and the history information (logs of current and previous process states already) can also be persisted.

### 7.1. Runtime State

Whenever a process is started, a process instance is created, which represents the execution of the process in that specific context. For example, when executing a process that specifies how to process a sales order, one process instance is created for each sales request. The process instance represents the current execution state in that specific context, and contains all the information related to that process instance. Note that it only contains the (minimal) runtime state that is needed to continue the execution of that process instance at some later time, but it does not include information about the history of that process instance if that information is no longer needed in the process instance.

The runtime state of an executing process can be made persistent, for example, in a database. This allows to restore the state of execution of all running processes in case of unexpected failure, or to temporarily remove running instances from memory and restore them at some later time. jBPM allows you to plug in different persistence strategies. By default, if you do not configure the process engine otherwise, process instances are not made persistent.

If you configure the engine to use persistence, it will automatically store the runtime state into the database. You do not have to trigger persistence yourself, the engine will take care of this when persistence is enabled. Whenever you invoke the engine, it will make sure that any changes are stored at the end of that invocation, at so-called safe points. Whenever something goes wrong and you restore the engine from the database, you also should not reload the process instances and trigger them manually to resume execution, as process instances will automatically resume execution if they are triggered, like 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.

The runtime persistence data should in general be considered internal, meaning that you probably should not try to access these database tables directly and especially not try to modify these directly (as changing the runtime state of process instances without the engine knowing might have unexpected side-effects). In most cases where information about the current execution state of process instances is required, the use of a history log is mostly recommended (see below). In some cases, it might still be useful to for example query the internal database tables directly, but you should only do this if you know what you are doing.

### 7.1.1. Binary Persistence

jBPM uses a binary persistence mechanism, otherwise known as marshalling, which converts the state of the process instance into a binary dataset. When you use persistence with jBPM, this mechanism is used to save or retrieve the process instance state from the database. 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:

- First, the process instance information is transformed into a binary blob. For performance reasons, a custom serialization mechanism is used and not normal Java serialization.
- This blob is then stored, alongside other metadata about this process instance. This metadata includes, among other things, the process instance id, process id, and the process start date.

Apart from the process instance state, the session itself can also store some state, such as the state of timer jobs, or the session data that any business rules would be evaluated over. This session state is stored separately as a binary blob, along with the id of the session and some metadata. You can always restore session state 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, this usually contains one or a few node instances, i.e., any node that is currently executing, and any existing variable values.

As a result of jBPM using marshalling, the data model is both simple and small:

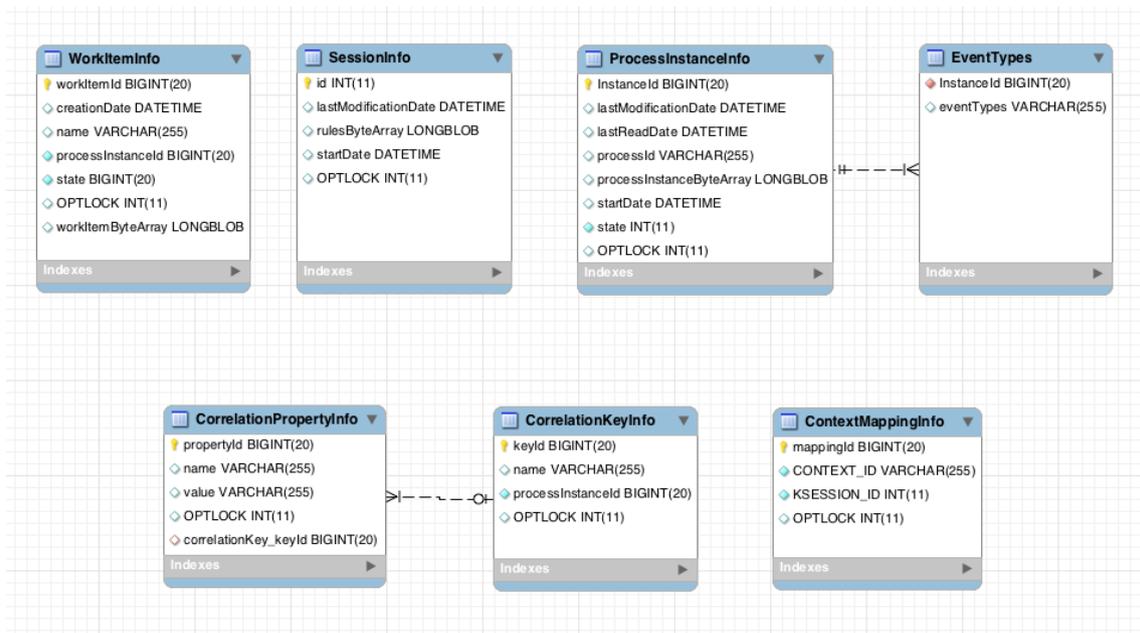


Figure 7.1. jBPM data model

[images/Chapter-Persistence/jbpm\_schema.png]

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

**Table 7.1. SessionInfo**

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

The `processinstanceinfo` entity contains the state of the jBPM process instance.

**Table 7.2. ProcessInstanceInfo**

Field	Description	Nullable
<code>instanceid</code>	The primary key	NOT NULL
<code>lastmodificationdate</code>	The last time that the entity was saved to the database	
<code>lastreaddate</code>	The last time that the entity was retrieved (read) from the database	
<code>processid</code>	The name (id) of the process	
<code>processinstancebytearray</code>	This is the binary dataset containing the state of the process instance	NOT NULL
<code>startdate</code>	The start time of the process	
<code>state</code>	An integer representing the state of the process instance	NOT NULL
<code>optlock</code>	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. EventTypees**

Field	Description	Nullable
instanceid	This references the <code>processinstanceinfo</code> primary key and there is a foreign key constraint on this column.	NOT NULL
eventTypes	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
creationDate	The name of the work item	
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

The `CorrelationKeyInfo` entity contains information about correlation keys assigned to given process instance - loose relationship as this table is considered optional used only when correlation capabilities are required.

**Table 7.5. CorrelationKeyInfo**

Field	Description	Nullable
keyid	The primary key	NOT NULL
name	assigned name of the correlation key	

Field	Description	Nullable
processinstanceid	The id of the process instance which is assigned to this correlation key	NOT NULL
optlock	The version field that serves as its optimistic lock value	

The `CorrelationPropertyInfo` entity contains information about correlation properties for given correlation key that is assigned to given process instance.

**Table 7.6. CorrelationPropertyInfo**

Field	Description	Nullable
propertyid	The primary key	NOT NULL
name	The name of the property	
value	The value of the property	NOT NULL
optlock	The version field that serves as its optimistic lock value	
correlationKey-keyid	Foreign key to map to correlation key	NOT NULL

The `ContextMappingInfo` entity contains information about contextual information mapped to ksession. This is an internal part of `RuntimeManager` and can be considered optional when `RuntimeManager` is not used.

**Table 7.7. ContextMappingInfo**

Field	Description	Nullable
mappingid	The primary key	NOT NULL
context_id	Identifier of the context	NOT NULL
ksession?id	Identifier of the ksession mapped to this context	NOT NULL
optlock	The version field that serves as its optimistic lock value	

## 7.1.2. Safe Points

The state of a process instance is stored at so-called "safe points" during the execution of the process engine. Whenever a process instance is executing (for example when it started or continuing from a previous wait state, the engine executes the process instance until no more actions can be performed (meaning that the process instance either has completed (or was aborted), or that it has reached a wait state in all of its parallel paths). At that point, the engine has reached the next safe state, and the state of the process instance (and all other process instances that might have been affected) is stored persistently.

### 7.1.3. Configuring Persistence

By default, the engine does not save runtime data persistently. This means you can use the engine completely without persistence (so not even requiring an in memory database) if necessary, for example for performance reasons, or when you would like to manage persistence yourself. It is, however, possible to configure the engine to do use persistence by configuring it to do so. This usually requires adding the necessary dependencies, configuring a datasource and creating the engine with persistence configured.

#### 7.1.3.1. Adding dependencies

You need to make sure the necessary dependencies are available in the classpath of your application if you want to user persistence. By default, persistence is based on the Java Persistence API (JPA) and can thus work with several persistence mechanisms. We are using Hibernate by default.

If you're using the Eclipse IDE and the jBPM Eclipse plugin, you should make sure the necessary jars are added to your jBPM runtime directory. You don't really need to do anything (as the necessary dependencies should already be there) if you are using the jBPM runtime that is configured by default when using the jBPM installer, or if you downloaded and unzipped the jBPM runtime artifact (from the downloads) and pointed the jBPM plugin to that directory.

If you would like to manually add the necessary dependencies to your project, first of all, you need the jar file `jbpm-persistence-jpa.jar`, as that contains code for saving the runtime state whenever necessary. Next, you also need various other dependencies, depending on the persistence solution and database you are using. For the default combination with Hibernate as the JPA persistence provider and using an H2 in-memory database and Bitronix for JTA-based transaction management, the following list of additional dependencies is needed:

- `jbpm-persistence-jpa` (org.jbpm)
- `drools-persistence-jpa` (org.drools)
- `persistence-api` (javax.persistence)
- `hibernate-entitymanager` (org.hibernate)
- `hibernate-annotations` (org.hibernate)
- `hibernate-commons-annotations` (org.hibernate)
- `hibernate-core` (org.hibernate)
- `commons-collections` (commons-collections)
- `dom4j` (dom4j)
- `jta` (javax.transaction)
- `btm` (org.codehaus.btm)

- javassist (javassist)
- slf4j-api (org.slf4j)
- slf4j-jdk14 (org.slf4j)
- h2 (com.h2database)
- jbpm-test (org.jbpm) for testing only, do not include it in the actual application

### 7.1.3.2. Manually configuring the engine to use persistence

You can use the `JPAKnowledgeService` to create your knowledge session. This is slightly more complex, but gives you full access to the underlying configurations. You can create a new knowledge session using `JPAKnowledgeService` based on a knowledge base, a knowledge session configuration (if necessary) and an environment. The environment needs to contain a reference to your Entity Manager Factory. For example:

```
// 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 );

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

// invoke methods on your method here
ksession.startProcess( "MyProcess" );
ksession.dispose();
```

You can also use the `JPAKnowledgeService` to recreate a session based on a specific session id:

```
// recreate the session from database using the sessionId
ksession = JPAKnowledgeService.loadStatefulKnowledgeSession(sessionId, kbase, null, env );
```

You need to add a persistence configuration to your classpath to configure JPA to use Hibernate and the H2 database (or your own preference), called `persistence.xml` in the META-INF directory, as shown below. For more details on how to change this for your own configuration, we refer to the JPA and Hibernate documentation for more information.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```

<persistence
  version="2.0"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
java.sun.com/xml/ns/persistence/persistence_2_0.xsd
  http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/
persistence/orm_2_0.xsd"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</class>

    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/
    >
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.transaction.jta.platform"
        value="org.hibernate.service.jta.platform.internal.BitronixJtaPlatform"/
    >
    </properties>
  </persistence-unit>
</persistence>

```

This configuration file refers to a data source called "jdbc/jbpm-ds". If you run your application in an application server (like for example JBoss AS), these containers typically allow you to easily set up data sources using some configuration (like for example dropping a datasource configuration file in the deploy directory). Please refer to your application server documentation to know how to do this.

For example, if you're deploying to JBoss Application Server v5.x, you can create a datasource by dropping a configuration file in the deploy directory, for example:

```

<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>

```

```

<jndi-name>jdbc/jbpm-ds</jndi-name>
<connection-url>jdbc:h2:tcp://localhost/~/test</connection-url>
<driver-class>org.h2.jdbcx.JdbcDataSource</driver-class>
<user-name>sa</user-name>
<password></password>
</local-tx-datasource>
</datasources>

```

If you are however executing in a simple Java environment, you can use the `JBPMHelper` class to do this for you (see below for tests only) or the following code fragment could be used to set up a data source (where we are using the H2 in-memory database in combination with Bitronix in this case).

```

PoolingDataSource ds = new PoolingDataSource();
ds.setUniqueName("jdbc/jbpm-ds");
ds.setClassName("bitronix.tm.resource.jdbc.lrc.LrcXADataSource");
ds.setMaxPoolSize(3);
ds.setAllowLocalTransactions(true);
ds.getDriverProperties().put("user", "sa");
ds.getDriverProperties().put("password", "sasa");
ds.getDriverProperties().put("URL", "jdbc:h2:mem:jbpm-db");
ds.getDriverProperties().put("driverClassName", "org.h2.Driver");
ds.init();

```

### 7.1.3.3. Configuring the engine to use persistence using `JBPMHelper` - for tests only

You need to configure the jBPM engine to use persistence, usually simply by using the appropriate constructor when creating your session. There are various ways to create a session (as we have tried to make this as easy as possible for you and have several utility classes for you, depending for example if you are trying to write a process junit test).

The easiest way to do this is to use the `jbpm-test` module that allows you to easily create and test your processes. The `JBPMHelper` class has a method to create a session, and uses a configuration file to configure this session, like whether you want to use persistence, the datasource to use, etc. The helper class will then do all the setup and configuration for you.

To configure persistence, create a `jbpm.properties` file and configure the following properties (note that the example below are the default properties, using an H2 in-memory database with persistence enabled, if you are fine with all of these properties, you don't need to add new properties file, as it will then use these properties by default):

```
# 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.usergroupcallback=org.jbpm.services.task.identity.JBossUserGroupCallbackImpl
taskservice.usergroupmapping=classpath:/usergroups.properties
```

If you want to use persistence, you must make sure that the datasource (that you specified in the `jbpm.properties` file) is initialized correctly. This means that the database itself must be up and running, and the datasource should be registered using the correct name. If you would like to use an H2 in-memory database (which is usually very easy to do some testing), you can use the `JBPMHelper` class to start up this database, using:

```
JBPMHelper.startH2Server();
```

To register the datasource (this is something you always need to do, even if you're not using H2 as your database, check below for more options on how to configure your datasource), use:

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

Next, you can use the `JBPMHelper` class to create your session (after creating your knowledge base, which is identical to the case when you are not using persistence):

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

Once you have done that, you can just call methods on this `ksession` (like `startProcess`) and the engine will persist all runtime state in the created datasource.

You can also use the `JBPMHelper` class to recreate your session (by restoring its state from the database, by passing in the session id (that you can retrieve using `ksession.getId()`):

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

## 7.1.4. Transactions

The jBPM engine supports JTA transactions. It also supports local transactions *only* when using Spring. It does not support pure local transactions at the moment. For more information about using Spring to set up persistence, please see the Spring chapter in the Drools integration guide.

Whenever you do not provide transaction boundaries inside your application, the engine will automatically execute each method invocation on the engine in a separate transaction. If this behavior is acceptable, you don't need to do anything else. You can, however, also specify the transaction boundaries yourself. This allows you, for example, to combine multiple commands into one transaction.

You need to register a transaction manager at the environment before using user-defined transactions. The following sample code uses the Bitronix transaction manager. Next, we use the Java Transaction API (JTA) to specify transaction boundaries, as shown below:

```
// 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 );

// 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();
```

Note that, if you use Bitronix as the transaction manager, you should also add a simple `jndi.properties` file in your root classpath to register the Bitronix transaction manager in JNDI. If you are using the `jbpm-test` module, this is already included by default. If not, create a file named `jndi.properties` with the following content:

```
java.naming.factory.initial=bitronix.tm.jndi.BitronixInitialContextFactory
```

If you would like to use a different JTA transaction manager, you can change the `persistence.xml` file to use your own transaction manager. For example, when running inside JBoss Application Server v5.x or v7.x, you can use the JBoss transaction manager. You need to change the transaction manager property in `persistence.xml` to:

```
<property name="hibernate.transaction.jta.platform"
value="org.hibernate.transaction.JBossTransactionManagerLookup" />
```

### 7.1.4.1. Container managed transaction

Special consideration need to be taken when embedding jBPM inside an application that executes in Container Managed Transaction (CMT) mode, for instance EJB beans. This especially applies to application servers that does not allow accessing `UserTransaction` instance from JNDI when being part of container managed transaction, e.g. WebSphere Application Server. Since default implementation of transaction manager in jBPM is based on `UserTransaction` to get transaction status which is used to decide if transaction should be started or not, in environments that prevent accessing `UserTransaction` it won't do its job. To secure proper execution in CMT environments a dedicated transaction manager implementation is provided:

```
org.jbpm.persistence.jta.ContainerManagedTransactionManager
```

This transaction manager expects that transaction is active and thus will always return `ACTIVE` when invoking `getStatus` method. Operations like `begin`, `commit`, `rollback` are no-op methods as transaction manager runs under managed transaction and can't affect it.



### Note

To make sure that container is aware of any exceptions that happened during process instance execution, user needs to ensure that exceptions thrown by the engine are propagated up to the container to properly rollback transaction.

To configure this transaction manager following must be done:

- Insert transaction manager and persistence context manager into environment prior to creating/loading session

```
Environment env = EnvironmentFactory.newEnvironment();
env.set(EnvironmentName.ENTITY_MANAGER_FACTORY, emf);
env.set(EnvironmentName.TRANSACTION_MANAGER, new
    ContainerManagedTransactionManager());
env.set(EnvironmentName.PERSISTENCE_CONTEXT_MANAGER, new
    JpaProcessPersistenceContextManager(env));
```

- configure JPA provider (example hibernate and WebSphere)

```
<property name="hibernate.transaction.factory_class"
    value="org.hibernate.transaction.CMTTransactionFactory" />
<property name="hibernate.transaction.manager_lookup_class"
    value="org.hibernate.transaction.WebSphereExtendedJTATransactionLookup" />
```

With following configuration jBPM should run properly in CMT environment.

#### 7.1.4.1.1. CMT dispose ksession command

Usually when running within container managed transaction disposing ksession directly will cause exceptions on transaction completion as there are some transaction synchronization registered by jBPM to clean up the state after invocation is finished. To overcome this problem specialized command has been provided `org.jbpm.persistence.jta.ContainerManagedTransactionDisposeCommand` which allows to simply execute this command instead of regular `ksession.dispose` which will ensure that ksession will be disposed at the transaction completion.

### 7.1.5. Persistence and concurrency

Please see the [Multi-threading](#) section for more information.

## 7.2. Process Definitions

Process definition files are usually written in an XML format. These files can easily be stored on a file system during development. However, whenever you want to make your knowledge accessible to one or more engines in production, we recommend using a knowledge repository that (logically) centralizes your knowledge in one or more knowledge repositories.

jBPM comes with web tooling that allows to author various business assets such as processes, rules, decision tables, etc and then store them in GIT repository. Next you can directly build and deploy your packages (kjars - knowledge archives) to the runtime engine or get access to the GIT repository with your favourite GIT tool e.g. eGit plugin for eclipse.

Since kjars are essentially maven artifacts they can be simply build with maven and deployed to remote repository for deployment to the process engine. More on this in deployment chapter

## 7.3. History Log

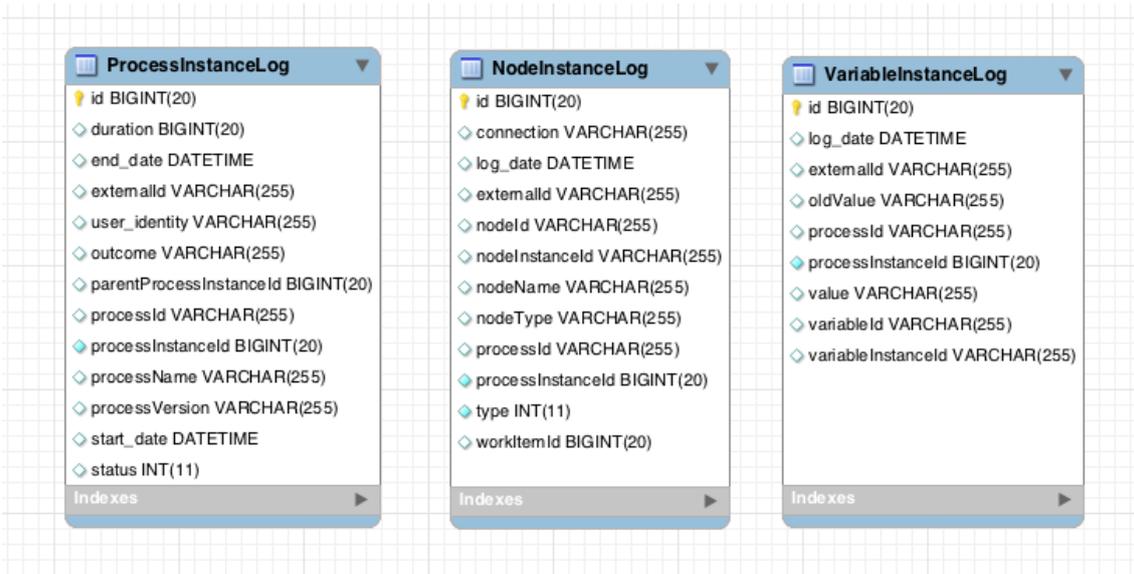
In many cases it will be useful (if not necessary) to store information *about* the execution of process instances, so that this information can be used afterwards. For example, sometimes we want to verify which actions have been executed for a particular process instance, or in general, we want to be able to monitor and analyze the efficiency of a particular process.

However, storing history information in the runtime database can result in the database rapidly increasing in size, not to mention the fact that monitoring and analysis queries might influence the performance of your runtime engine. This is why process execution history information can be stored separately.

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.

### 7.3.1. The jBPM Audit data model

The jbpm-audit 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.



**Figure 7.2. jBPM Audit data model**

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

**Table 7.8. ProcessInstanceLog**

Field	Description	Nullable
<code>id</code>	The primary key and id of the log entity	NOT NULL
<code>duration</code>	Actual duration of this process instance since its start date	
<code>end_date</code>	When applicable, the end date of the process instance	
<code>externalId</code>	Optional external identifier used to correlate to some elements - e.g. deployment id	
<code>user_identity</code>	Optional identifier of the user who started the process instance	
<code>outcome</code>	The outcome of the process instance, for instance error code in case of process instance was finished with error event	
<code>parentProcessInstanceId</code>	The process instance id of the parent process instance if any	

Field	Description	Nullable
processid	The id of the process	
processinstanceid	The process instance id	NOT NULL
processname	The name of the process	
processversion	The version of the process	
start_date	The start date of the process instance	
status	The status of process instance that maps to process instance state	

The `NodeInstanceLog` table contains more information about which nodes were actually 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 7.9. NodeInstanceLog**

Field	Description	Nullable
id	The primary key and id of the log entity	NOT NULL
connection	Actual identifier of the sequence flow that led to this node instance	
log_date	The date of the event	
externalId	Optional external identifier used to correlate to some elements - e.g. deployment id	
nodeid	The node id of the corresponding node in the process definition	
nodeinstanceid	The node instance id	
nodename	The name of the node	
nodetype	The type 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

Field	Description	Nullable
workItemId	Optional - only for certain node types - The identifier of work item	

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

**Table 7.10. VariableInstanceLog**

Field	Description	Nullable
id	The primary key and id of the log entity	NOT NULL
externalId	Optional external identifier used to correlate to some elements - e.g. deployment id	
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
oldvalue	The previous value of the variable at the time that the log is made	
value	The value of the variable at the time that the log is made	
variableid	The variable id in the process definition	
variableinstanceid	The id of the variable instance	

### 7.3.2. Storing Process Events in a Database

To log process history information in a database like this, you need to register the logger on your session (or working memory) like this:

```
EntityManagerFactory emf = ...;
StatefulKnowledgeSession ksession = ...;
AbstractAuditLogger auditLogger = AuditLoggerFactory.newJPAInstance(emf);
ksession.addProcessEventListener(auditLogger);

// invoke methods on your session here
```

Note that this logger is like any other audit logger, which means that you can add one or more filters by calling the method `addFilter` to ensure that only relevant information is stored in the database. Only information accepted by all your filters will appear in the database.

To specify the database where the information should be stored, modify the file `persistence.xml` file to include the audit log classes as well (`ProcessInstanceLog`, `NodeInstanceLog` and `VariableInstanceLog`), as shown below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<persistence
  version="2.0"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
java.sun.com/xml/ns/persistence/persistence_2_0.xsd
  http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/
persistence/orm_2_0.xsd"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</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.H2Dialect"/
    >
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.transaction.jta.platform"
        value="org.hibernate.service.jta.platform.internal.BitronixJtaPlatform"/>
    </properties>
  </persistence-unit>
</persistence>
```

```
</persistence-unit>  
</persistence>
```

All this information can easily be queried and used in a lot of different use cases, ranging from creating a history log for one specific process instance to analyzing the performance of all instances of a specific process.

This audit log should only be considered a default implementation. We don't know what information you need to store for analysis afterwards, and for performance reasons it is recommended to only store the relevant data. Depending on your use cases, you might define your own data model for storing the information you need, and use the process event listeners to extract that information.

### 7.3.3. Storing Process Events in a JMS queue for further processing

Process events are stored in data base synchronously and within the same transaction as actual process instance execution. That obviously takes some time especially in highly loaded systems and might have some impact on data base when both history log and runtime data are kept in the same data base. To provide alternative option for storing process events a JMS based logger has been provided. It allows to be configured to submit messages to JMS queue instead of directly persisting them in data base. It can be configured to be transactional as well to avoid issues with inconsistent data in case of process engine transaction is rolled back.

```
ConnectionFactory factory = ...;  
Queue queue = ...;  
StatefulKnowledgeSession ksession = ...;  
Map<String, Object> jmsProps = new HashMap<String, Object>();  
jmsProps.put("jbpm.audit.jms.transacted", true);  
jmsProps.put("jbpm.audit.jms.connection.factory", factory);  
jmsProps.put("jbpm.audit.jms.queue", queue);  
AbstractAuditLogger auditLogger = AuditLoggerFactory.newInstance(Type.JMS, session, jmsProps);  
ksession.addProcessEventListener(auditLogger);  
  
// invoke methods one your session here
```

This is just one of possible ways to configure JMS audit logger, see javadocs for `AuditLoggerFactory` for more details.



# Chapter 8. Core Engine: Examples

## 8.1. jBPM Examples

There is a separate jBPM examples module that contains a set of example processes that show how to use the jBPM engine and the behavior or the different process constructs as defined by the BPMN 2.0 specification.

To start using these, simply unzip the file somewhere and open up your Eclipse development environment with all required plugins installed. If you don't know how to do this yet, take a look at the installer chapter, where you can learn how to create a demo environment, including a fully configured Eclipse IDE, using the jBPM installer. You can also take a look at the Eclipse plugin chapter if you want to learn how to manually install and configure this.

To take a look at the examples, simply import the downloaded examples project into Eclipse (File -> Import ... -> Under General: Existing Projects into Workspace), browse to the folder where you unzipped the jBPM examples artifact and click finish. This should import the examples project in your workspace, so you can start looking at the processes and executing the classes.

## 8.2. Examples

The examples module contains a number of examples, from basic to advanced:

- **Looping:** An example that shows how you can use exclusive gateways to loop a part your process until the loop condition is no longer valid. The process takes the 'count' (the number of times the loop needs to be repeated) as input and simply prints out a statement during every loop until the process is completed.
- **MultInstance:** This example shows how to execute a sub-process for each element in a collection. The process takes a collection of names as input and creates a review task for a sales representative for each person in that list. The process completes if the task has been executed for every person on that list.
- **Evaluation:** A performance evaluation process that shows how to integrate human actors in the process. While the basic example simply shows tasks assigned to predefined users, the more advanced version shows data passing from the process to the task and back, group assignment, task delegation, etc.
- **HumanTask:** An advanced example when using human tasks. It shows how to do data passing between tasks, task forms, swimlanes, etc. This example can also be deployed to the Guvnor repository (including all the forms etc.) and executed on the jBPM console out-of-the-box.
- **Request:** An advanced example that shows various ways in which processes and rules can work together, like a rule task for invoking validation rules, rules as expression language for

constraints inside the process, rules for exception handling, event processing for monitoring, ad hoc rules for more flexible processes, etc.

### 8.3. Unit tests

The examples project contains a large number of simple BPMN2 processes for each of the different node types that are supported by jBPM5. In the junit folder under src/main/resources you can for example find process examples for constructs like a conditional start event, exclusive diverging gateways using default connections, etc. So if you're looking for a simple working example that shows the behavior of one specific element, you can probably find one here. The folder already contains well over 50 sample processes. Simply double-click them to open them in the graphical editor.

Each of those processes is also accompanied by a small junit test that tests the implementation of that construct. The `org.jbpm.examples.junit.BPMN2JUnitTests` class contains one test for each of the processes in the junit resources folder. You can execute these tests yourself by selecting the method you want to execute (or the entire class) and right-click and then Run as -> JUnit test.

Check out the chapter on testing and debugging if you want to learn more how to debug these example processes.