jBPM User Guide

Version 6.0.0.CR1

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

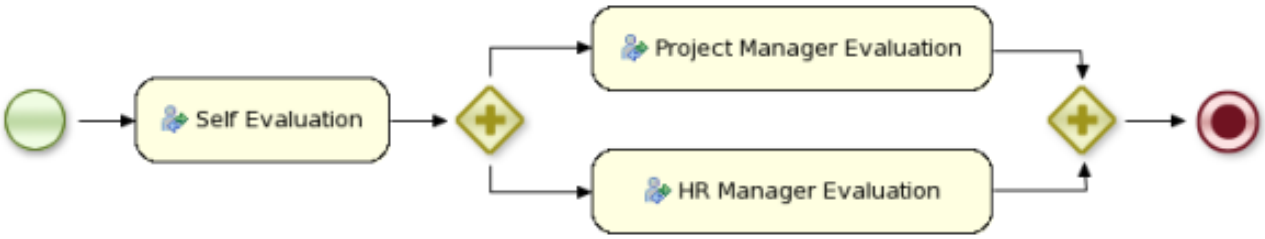

jBPM

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.
- Management console supporting process instance management, task lists and task form management, and reporting
- Task for builder to create, generate and/or edit task forms
- Optional process repository to deploy your process (and other related knowledge)
- History logging (for querying / monitoring / analysis)
- 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

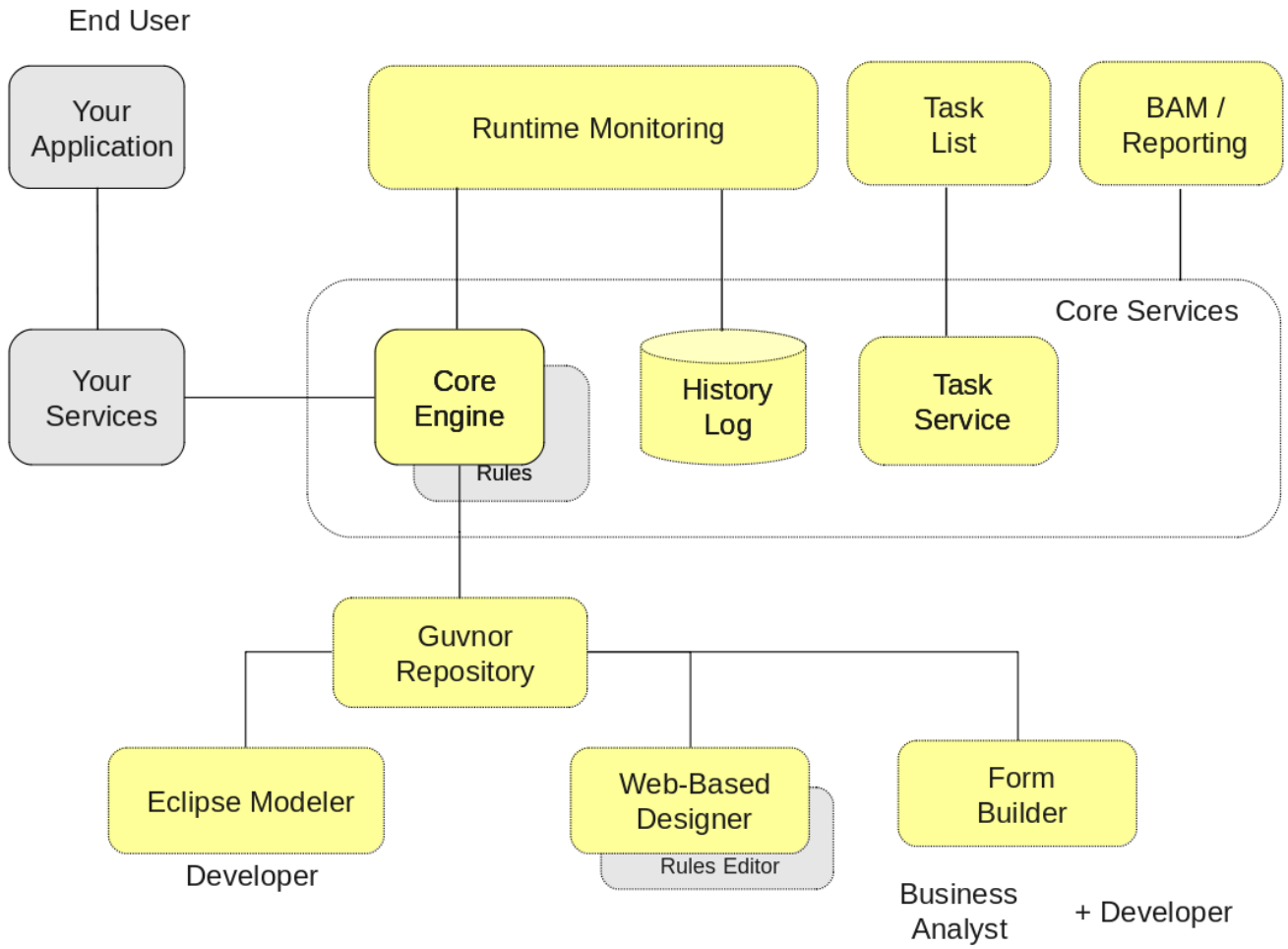


Figure 1.1.

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.

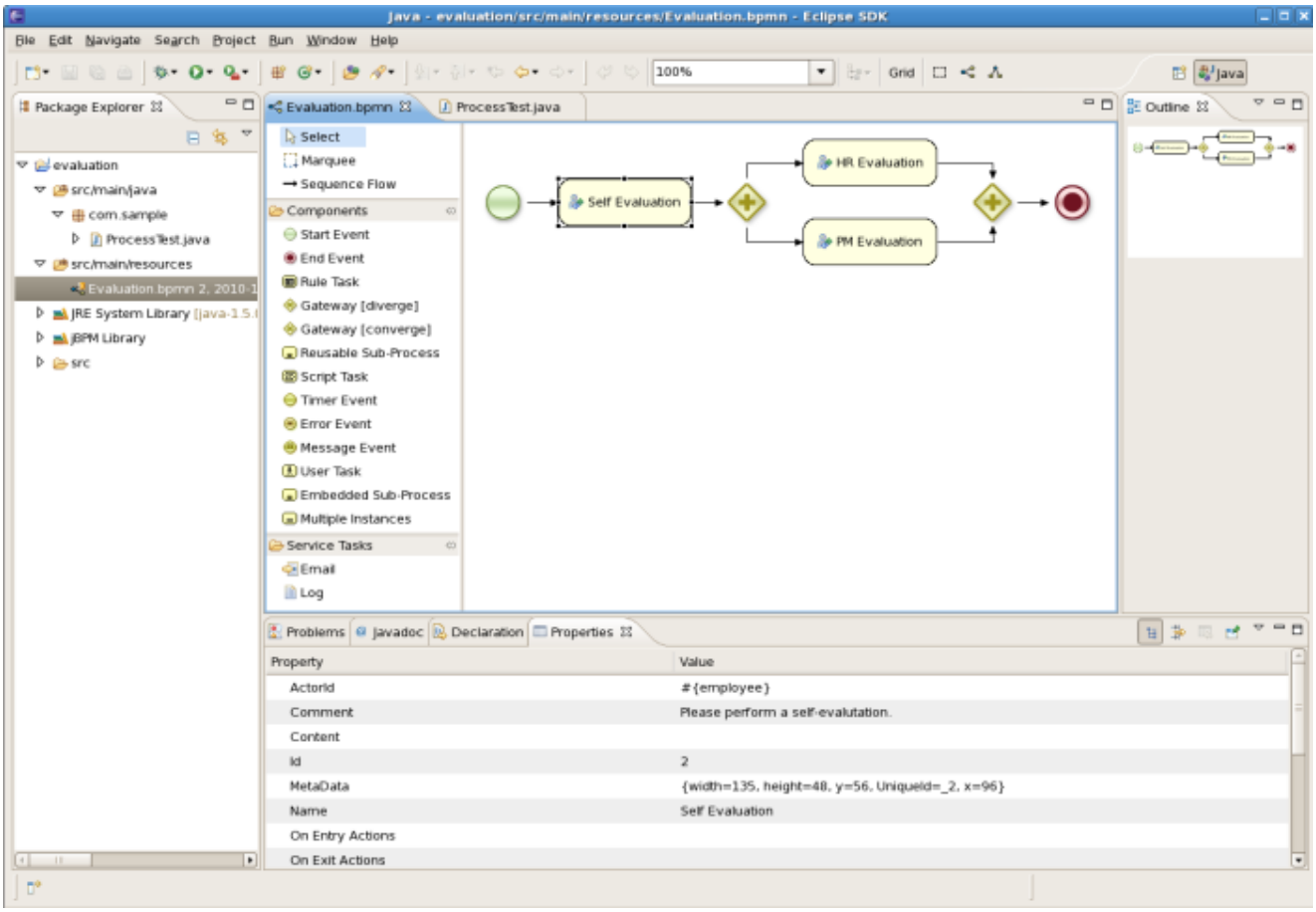


Figure 1.2. Eclipse editor for creating BPMN2 processes

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.
- Audit view to get an overview of what happened at runtime
- The ability to unit test your processes
- Integration with the knowledge repository

1.5. Web-based 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.

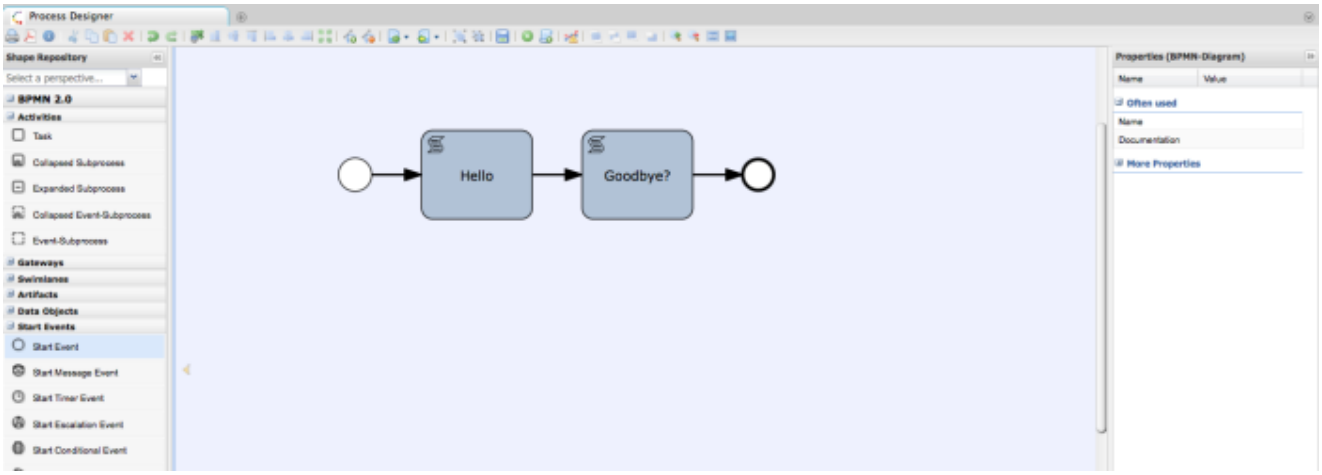


Figure 1.3. Web-based designer for creating BPMN2 processes

1.6. Form Builder

A web-based form builder 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.

1.7. Guvnor Repository

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 JCR 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.

- A knowledge agent to easily create new sessions based on the process definitions in the repository. This also supports (optionally) dynamically updating all sessions if a new process has been deployed.

1.8. Web-based Management Consoles

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).
- Business Activity Monitorong (BAM) and Reporting: 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).

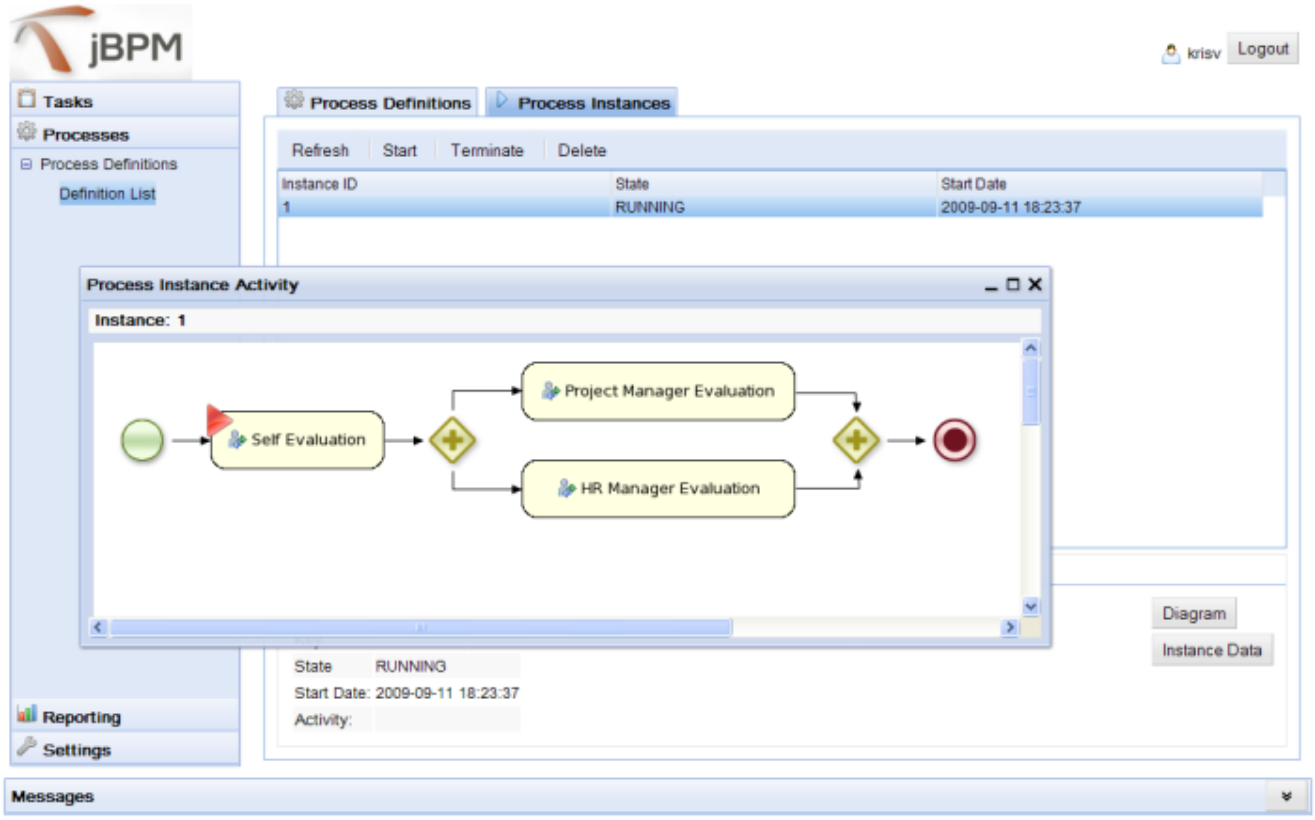


Figure 1.4. Managing your process instances

1.9. 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/]. Select the version you want to download and then select which artefact you want:

- bin: all the jBPM binaries (jars) and their dependencies
- src: the sources of the core components
- gwt-console: the jbpm console, a zip file containing both the server and client war
- 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. 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.

2.3. 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] related to jBPM

- The *#jbossjbpm twitter account* [http://twitter.com/jbossjbpm].
- A *user forum* [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] for bugs, feature requests and roadmap
- A *continuous build server* [https://hudson.jboss.org/hudson/job/jBPM/] for getting the *latest snapshots* [https://hudson.jboss.org/hudson/job/jBPM/lastSuccessfulBuild/artifact/jbpm-distribution/target/]

Please feel free to join us in our IRC channel at irc.codehaus.org #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://irc.codehaus.org*, input your desired nickname, and specify #jbpm. Then click login to join the fun.

2.4. Sources

2.4.1. 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 BPM console is GNU Lesser General Public License (LGPL) v2.1
- The Drools project is Apache License v2.0.

2.4.2. 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].
- The jBPM Eclipse plugin can be found *here* [http://anonsvn.jboss.org/repos/jbosstools/trunk/bpmn/plugins/org.jboss.tools.jbpm/].

- The new Eclipse BPMN2 plugin can be found *here* [https://github.com/droolsjbpm/bpmn2-eclipse-editor].
- The web-based designer can be found *here* [https://github.com/tsurdilo/process-designer]
- The BPM console can be found *here* [https://github.com/bpmc/bpm-console]

2.4.3. 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].

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 1.5+ (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%205/] the installer. There are two versions, a full installer (which already contains a lot of the dependencies that are necessary during the installation) and a 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 Drools Guvnor into JBoss AS

- Install jBPM Designer into JBoss AS
- Install the jBPM console into JBoss AS
- Install the jBPM Eclipse plugin
- Install the Drools Eclipse plugin

Note

Guvnor (from version 5.4) requires JBoss EAP 5 to run properly, this only applies if installation is ran in AS5 configuration mode.

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.

To limit the amount of data that needs to be downloaded, we have disabled the download of the Eclipse BIRT plugin for reporting by default. If you want to try out reporting as well in the jBPM console, make sure to put the jBPM.birt.download property in the build.properties file to true before running the installer.

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 the H2 database
- Start the JBoss AS
- Start Eclipse
- Start the Human Task Service

Once everything is started, you can start playing with the Eclipse tooling, Guvnor repository and 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.noeclipse
ant start.demo.noeclipse
```

3.4. 10-Minute Tutorial: Using the Eclipse tooling

The *following screencast* [http://people.redhat.com/kverlaen/jbpm-installer-eclipse-5.2.swf] gives an overview of how to run a simple demo process in Eclipse. It shows you:

- How to import an existing example project into your workspace, containing
  - a sample BPMN2 process for requesting a performance evaluation
  - a sample Java class to start the process
- How to start the process

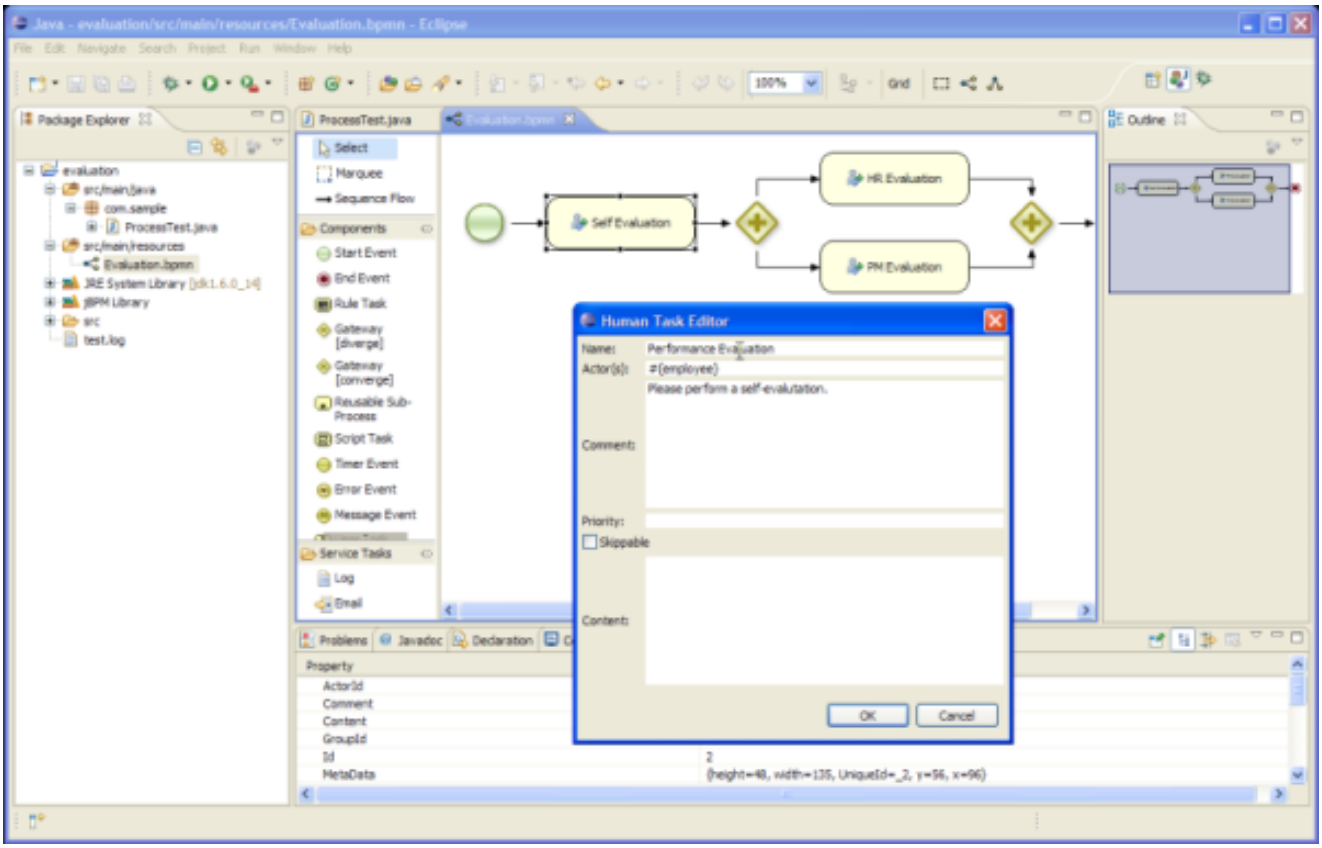


Figure 3.1.

[http://people.redhat.com/kverlaen/jbpm-installer-eclipse-5.2.swf]

Do the following:

- Once Eclipse has opened, simply import (using "File -> Import ..." and then under the General category, select "Existing Projects into Workspace") the existing sample project (in the jbpm-installer/sample/evaluation directory). This should add the sample project, including a simple BPMN2 process and a Java file to start the process.
- You can open the BPMN2 process and the Java class by double-clicking it.
- We will now debug the process, so we can visualize its runtime state using the debug tooling. First put a breakpoint on line "logger.close()" of the ProcessTest class. To start debugging, right-click on ProcessTest.java in the com.sample package (under "src/main/java") and select "Debug As - Java Application", and switch to the debug perspective.
- Open up the various debug views: Under "Window - Show View -> Other ...", select the Process Instances View and Process Instance View (under Drools category) and the Human Task View (under jBPM Task) and click OK.
- The program will hit the breakpoint right after starting the process. In this case, it will simply start the process, which will result in the creation of a new user task for the user "krisv" in the human task service, after which the process will wait for its execution. Go to the Human Task View, fill in "krisv" under UserId and click Refresh. A new Performance Evaluation task should show up.
- To show the state of the process instance you just started graphically, click on the Process Instances View and then select the ksession variable in the Variables View. This will show all active process instances in the selected session. In this case, there is only one process instance. Double-click it to see the state of that process instance annotated on the process flow chart.
- Now go back to the Task View, select the Performance Evaluation task and first start and then complete the selected task. Now go back to the Process Instances view and double click the process instance again to see its new state.

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.

3.5. 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/jbpm-installer-console.5.2.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

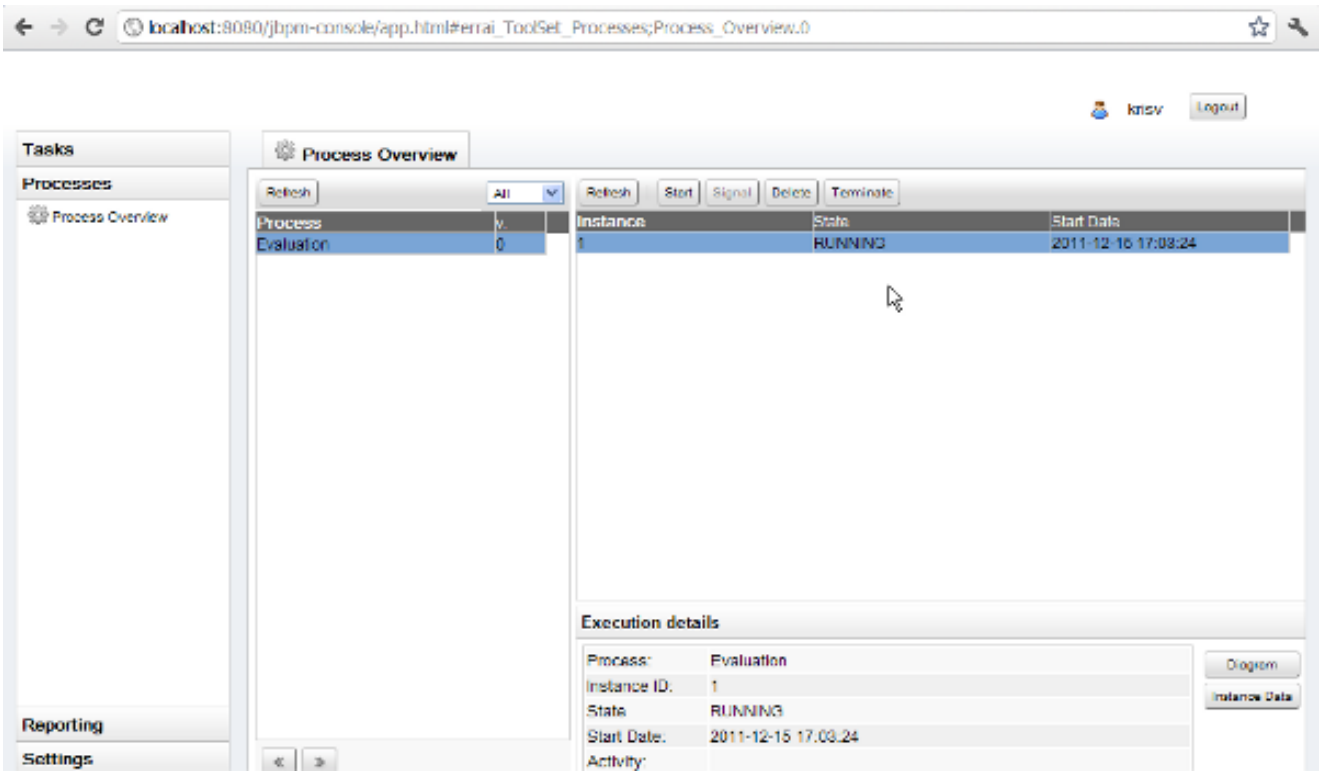


Figure 3.2.

[http://people.redhat.com/kverlaen/jbpm-installer-console.5.2.swf]

- To manage your process instances, click on the "Processes" tab at the left an select "Process Overview". After a slight delay (if you are using the application for the first time, due to session initalization etc.), the "Process" list should show all the known processes. The jbpm-console in the demo setup currently loads all the processes in the "src/main/resources" folder of the evaluation sample in "jbpm-installer/sample/evaluation". If you click the process, it will show you all current running instances. Since there are no running instances at this point, the "Instance" table will remain empty.

- You can start a new process instance by clicking on the "Start" button. After confirming that you want to start a new execution of this process, 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. A new instance should show up in the "Instance" table. If you click the process instance, you can check its details below and the diagram and instance data by clicking on the "Diagram" and "Instance Data" buttons respectively. The 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" tab on the left and select "Personal Tasks" (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 Overview" 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 "Reporting", select "Report Templates". By default, the console has one report template, for generating a generic overview for all processes. Click the "Create Report" button to generate a realtime report of the current status. Notice that the initialization of the reports might take a moment, especially the first time you use the application.

3.6. 10-Minute Tutorial: Using Guvnor repository and Designer

The Guvnor repository can be used as a process repository to store business processes. It also offers a web-based interface to manage your processes. This includes a web-based editor for viewing and editing processes.

Open up Drools Guvnor:

http://localhost:8080/drools-guvnor

Log in (if necessary), using any non-empty username / password (we disabled authentication for demo purposes). The *following screencast* [http://people.redhat.com/kverlaen/jbpm-installer-guvnor.5.2.swf] gives an overview of how to manage your repository. It shows you:

- How to import an existing process (in this case the evaluation process) from eclipse into guvnor

- How to open up the evaluation process in the web editor

- How to build a package so it can be used for creating a session

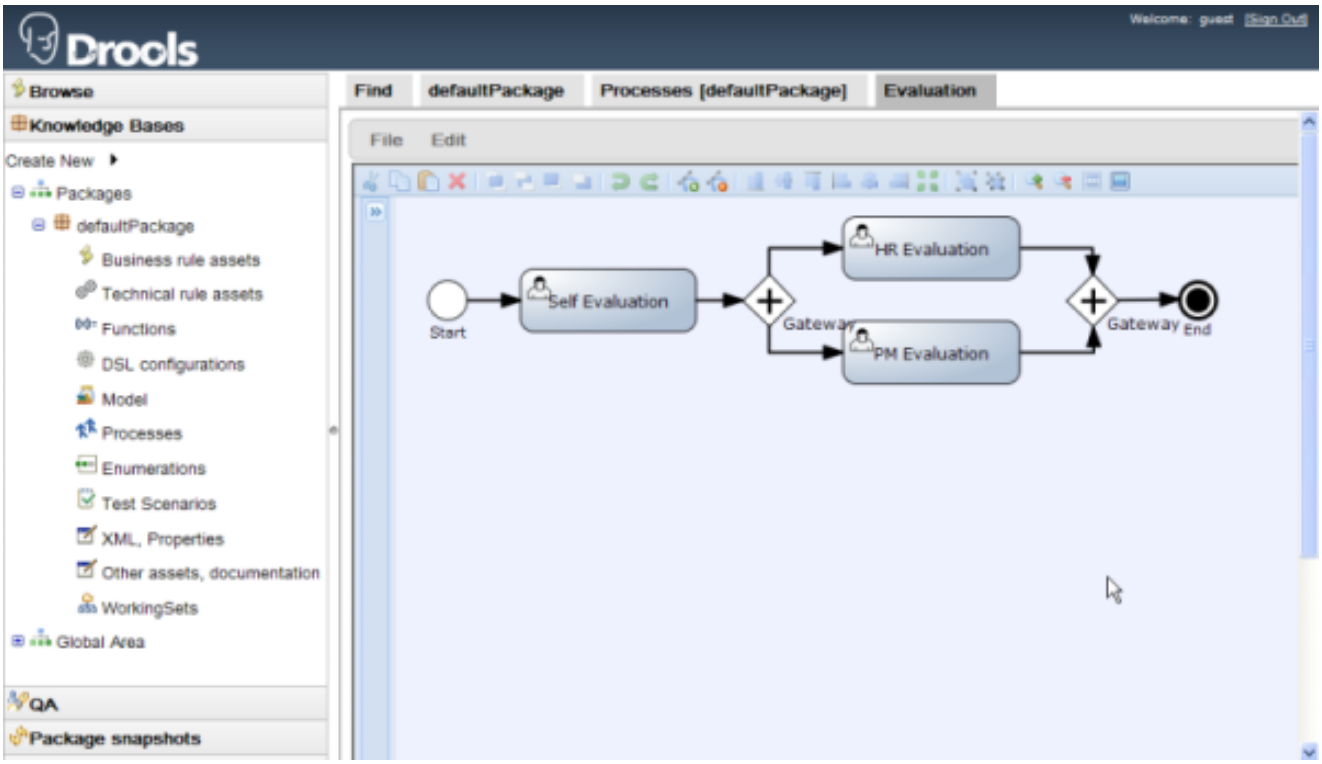


Figure 3.3.

[http://people.redhat.com/kverlaen/jbpm-installer-guvnor.5.2.swf]

If you want to know more, we recommend you take a look at the rest of the Drools Guvnor documentation.

Once you're done playing:

```
ant stop.demo
```

and simply close all the rest.

3.7. 10-Minute Tutorial: Using your own database with jBPM

At the moment, this quickstart does *not* work with JBoss AS 5. However, an update to the quickstart (and installer) is forthcoming which will fix that (and make it work with JBoss AS 5). [01/2012]

3.7.1. Introduction

In this quickstart, we are going to:

1. modify the persistence settings for the process engine
2. modify the persistence settings for the task server

3. 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 three types of persistent entities used by jBPM:

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

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

For reasons that I'll explain later on in this quickstart, the demo uses two different persistent units:

- one for jBPM and the logging/BAM information,
- and one for the task service.

With other jBPM installations, there's no reason not to use only one persistent unit if you want to.

The first persistence unit needs to use JTA, which is why we also need to define a seperate datasource for that persistence unit as well.

3.7.2. Database setup

In the MySQL database that I use in this quickstart, I've created two users:

- user/schema "jbpm5" with password "jbpm5" (for jBPM and the logging/BAM information)
- user/schema "task" with password "task" (for the task service)

If you end up using different names for your user/schemas, please make a note of where we insert "jbpm5" and "task" in the configuation 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.

3.7.3. Quickstart

The following 4 files define the persistence settings for the jbpm-installer demo:

- db/persistence.xml
- task-service/resources/META-INF/persistence.xml
- db/jBPM-ds.xml
  - If you're using the JBoss AS 5 server
- standalone.xml
  - If you're using the JBoss AS 7 server

Do the following:

- `db/persistence.xml`:

  This is the JPA persistence file that defines the persistence settings used by jBPM for both the process engine information and the logging/BAM information. The installer ant script moves this file to the expanded gwt console server war before the server is started.

  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].

- `task-service/resources/META-INF/persistence.xml`:

  The task service (that the installer starts) uses the JPA Persistence settings described in this file.

  The original file contains the following lines:

```
    <properties>
                                    <property      name="hibernate.dialect"
 value="org.hibernate.dialect.H2Dialect"/>
    <property name="hibernate.connection.driver_class" value="org.h2.Driver"/
>
    <property name="hibernate.connection.url" value="jdbc:h2:tcp://localhost/
runtime/task" />
      <property name="hibernate.connection.username" value="sa"/>
      <property name="hibernate.connection.password" value="sasa"/>
```

  Please change these lines so that they look like this:

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

```
                        <property    name="hibernate.connection.driver_class"
 value="com.mysql.jdbc.Driver"/>
            <property  name="hibernate.connection.url"  value="jdbc:mysql://
localhost:3306/task" />
      <property name="hibernate.connection.username" value="task"/>
      <property name="hibernate.connection.password" value="task"/>
```

- `db/jBPM-ds.xml`:

  This step is *only* neccessary if you're using JBoss AS 5.

  This file is the configuration for the (JTA) datasource used by the jBoss AS 5 instance for the process engine persistence. The installer ant script moves this file to the jboss server deploy directory.

  The original file contains the following lines:

```
<datasources>
  <local-tx-datasource>
    <jndi-name>jboss/datasources/jbpmDS</jndi-name>
   <connection-url>jdbc:h2:tcp://localhost/runtime/jbpm-demo</connection-url>
    <driver-class>org.h2.jdbcx.JdbcDataSource</driver-class>
    <user-name>sa</user-name>
    <password></password>
  </local-tx-datasource>
</datasources>
```

  Please change these to the following:

```
<datasources>
  <local-tx-datasource>
    <jndi-name>jboss/datasources/jbpmDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/jbpm5</connection-url>
       <driver-class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</driver-
class>
    <user-name>jbpm5</user-name>
    <password>jbpm5</password>
  </local-tx-datasource>
</datasources>
```

- `standalone.xml`:

  This step is *only* neccessary if you're using AS 7.

This file is the configuration for the standalone JBoss AS 7 server. When the installer starts the demo (using jBoss AS 7), it moves this file to the `standalone/configuration` directory in the jboss server directory

We need to change the datasource configuration in `standalone.xml` so that the (JTA) datasource for the jBPM process engine and logging/BAM points to our MySQL database

The original file contains the following lines:

```
        <subsystem xmlns="urn:jboss:domain:datasources:1.0">
            <datasources>
                        <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>
            </datasources>
        </subsystem>
```

Change the lines to the following:

```
        <subsystem xmlns="urn:jboss:domain:datasources:1.0">
            <datasources>
                 <datasource jndi-name="java:jboss/datasources/jbpmDS" pool-
name="H2DS" enabled="true" use-java-context="true">
                       <connection-url>jdbc:mysql://localhost:3306/jbpm5</
connection-url>
                    <driver>mysql</driver>
                    <pool></pool>
                    <security>
                       <user-name>jbpm5</user-name>
                       <password>jbpm5</password>
                    </security>
                </datasource>
```

```
                <drivers>
                    <driver name="mysql" module="com.mysql">
                                                                <xa-datasource-
class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</xa-datasource-class>
                    </driver>
                </drivers>
            </datasources>
        </subsystem>
```

- `Start the demo`

  We've modified all the neccessary files at this point, all that's left to do is run the demo.

  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:

```
ant install.demo.db
```

  If you have already installed and *run* the demo, it can't hurt to reinstall the demo:

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

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

```
ant start.demo.db
```

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

```
ant stop.demo.db
```

  The `stop.demo` ant task will also work, although it might throw some exceptions.

- `Problems?`

  If you 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.

3.7.4. 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.
- You will need to download the correct driver jar for your database and add it to the `db/drivers` directory. If you're using JBoss AS 5, the installer ant script will make sure that your downloaded driver is installed in the server. If you're using JBoss AS 7, see the next step.
- In order 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].
  - *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.
  - Otherwise, you can *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.

  While the former (deployment) is possibly easier, the latter (module) is slightly more straightforward -- and the installer can help you. If you choose to do the latter, please do the following:
  - Change the `db.driver.jar.name` property in `build.xml` to the name of the downloaded jdbc driver jar you placed in `db/drivers`. For example:

```
  <property name="db.driver.jar.name" value="postgresql-8.4-701.jdbc3.jar" /
>
```

  - 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.xml` 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. For example:

```
  <property name="db.driver.module.prefix" value="org/postgresql" />
```

- Lastly, you'll have to modify the `db/driver_jar_module.xml` file. We need to
  - 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 you 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-8.4-701.jdbc3.jar"/>
   </resources>
```

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

You can always contact the jBPM community for assistance.

Email: jbpm-dev@lists.jboss.org

IRC: #jbpm at irc.codehaus.org

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

3.9. 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 temporarly 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, Guvnor and the Designer. 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. Quickstarts

This chapter contains a number of simple, common task that you can follow to get started.

4.1. Invoking a Java service

It is common that you already have existing Java code that you would like to invoke from your process. How do you do that? There are different ways of doing this, and this quickstart will show you some of these alternatives.

4.1.1. Using a script task

One of the easiest ways to include some Java code into your process is to use a Script Task. This task will execute some script code whenever that node is reached during the execution of the process. This allows you to include some Java code as part of the process. For example, imagine this simple process that contains one Script Task to invoke some existing Java code:

Figure 4.1.

The script task defines a script that needs to be executed when the task is reached. In this case, the script invokes an existing class `org.jbpm.examples.quickstarts.HelloService`:

```
HelloService.getInstance().sayHello(person.getName());
```

where the HelloService class looks like this:

```
package org.jbpm.examples.quickstarts;
public class HelloService {
  private static final HelloService INSTANCE = new HelloService();
  public static HelloService getInstance() {
    return INSTANCE;
  }
  public void sayHello(String name) {
    System.out.println("Hello " + name);
  }
}
```

The script retrieves an instance of the HelloService and passes it the name of the person that started this process. This is possible because `person` is defined as a variable of the process, of type `org.jbpm.examples.quickstarts.Person`, and script tasks can directly reference process variables as if they were local variables (at least for reading, for setting the value of a variable, you should use `kcontext.setVariable(name, value)`). This process also references HelloService without fully qualifying the package as HelloService is defined using an import statement.

The underlying XML might look something like this:

```
<?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"
             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">
<itemDefinition id="_personItem" structureRef="org.jbpm.examples.quickstarts.Person"
  >
    <process processType="Private" isExecutable="true" id="org.jbpm.examples.quickstarts.script"
             name="Sample Process" tns:packageName="defaultPackage" >
      <extensionElements>
        <tns:import name="org.jbpm.examples.quickstarts.HelloService" />
      </extensionElements>
      <!-- process variables -->
      <property id="person" itemSubjectRef="_personItem"/>
      <!-- nodes -->
      <startEvent id="_1" name="StartProcess" />
      <scriptTask id="_2" name="Script" >
        <script>HelloService.getInstance().sayHello(person.getName());</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="org.jbpm.examples.quickstarts.script" >
        <bpmndi:BPMNShape bpmnElement="_1" >
          <dc:Bounds x="45" y="45" width="48" height="48" />
```

```
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_2" >
        <dc:Bounds x="131" y="46" width="80" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_3" >
        <dc:Bounds x="252" y="47" width="48" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge bpmnElement="_1-_2" >
        <di:waypoint x="69" y="69" />
        <di:waypoint x="171" y="70" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge bpmnElement="_2-_3" >
        <di:waypoint x="171" y="70" />
        <di:waypoint x="276" y="71" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>
```

A simple test that executes this process could look something like this: simply create a ksession and start the process by id, passing in a `Person` object that will then be set as the `person` process variable:

```
public class JavaServiceQuickstartTest extends JbpmJUnitTestCase {
  @Test
  public void testProcess() {
    StatefulKnowledgeSession ksession = createKnowledgeSession("test.bpmn");
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("person", new Person("krisv"));
    ksession.startProcess("org.jbpm.examples.quickstarts.script", params);
  }
}
```

This example shows how easy it is to include custom Java code in your process using Script Tasks, to invoke existing code and to pass it process variable values. Note that some node types allow you to specify on-entry and on-exit actions (which will be executed when the node is triggered or left respectively). This allows you to include scripts, just like you would do when using a Script Task, but hiding these more or less from the diagram (as for example business users might not be interested in these details).

4.1.2. Using a Java handler

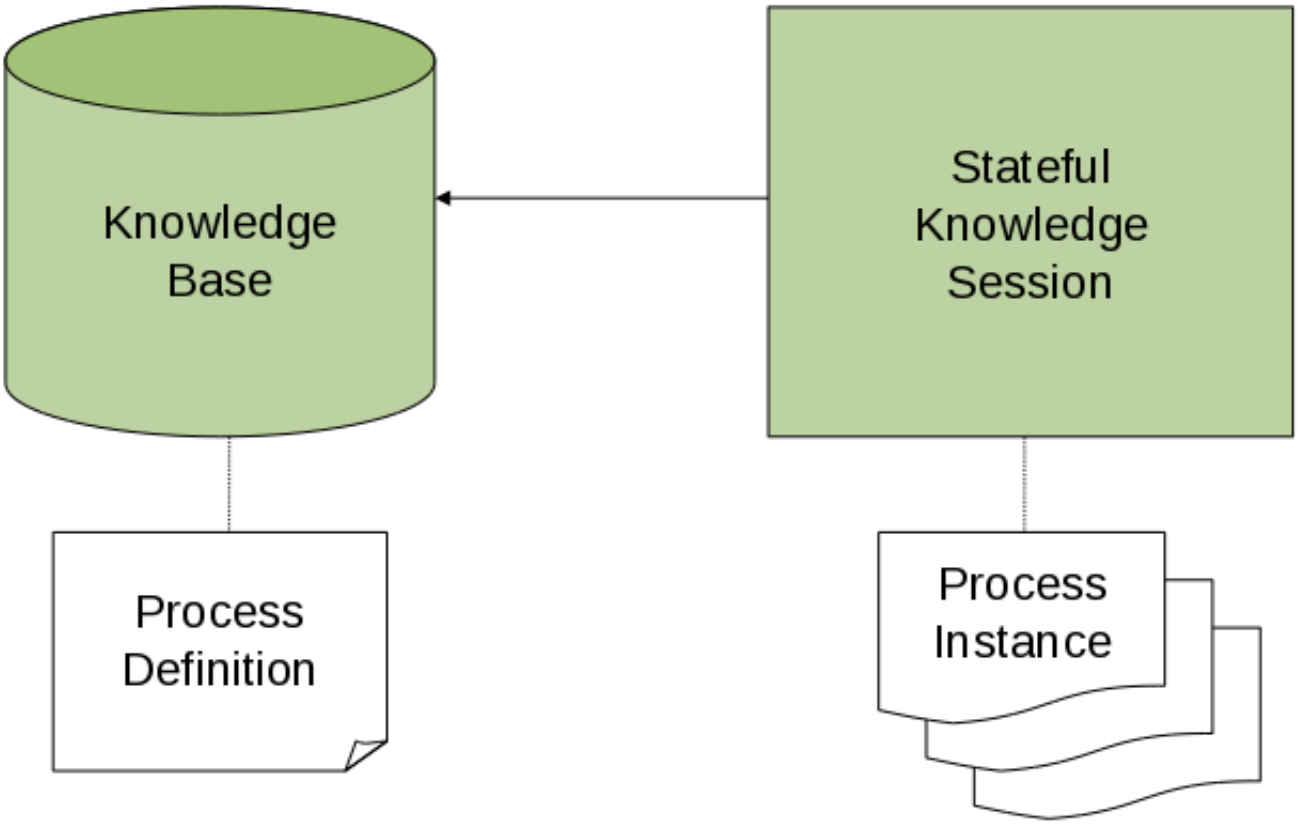
4.1.3. Writing your own domain-specific task

Chapter 5. 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.

5.1. 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.

5.1.1. 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 knowledge builder 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 builder. 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).

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

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

If you don't want to list all resources in your Java code, you could use a configuration file, called a changeset, to define these. These are simple XML configuration files that then list the resources. For example:

```
<change-set  xmlns='http://drools.org/drools-5.0/change-set'  xmlns:xs='http://
www.w3.org/2001/XMLSchema-instance'       xs:schemaLocation='http://drools.org/
drools-5.0/change-set      http://anonsvn.jboss.org/repos/labs/labs/jbossrules/
trunk/drools-api/src/main/resources/change-set-1.0.0.xsd'>
   <add>
      <resource source='file:/path_to_process/MyProcess.bpmn' type='BPMN2'/>
   </add>
</change-set>
```

You can also use a change set to load all processes from one or multiple folder for example:

```
<change-set  xmlns='http://drools.org/drools-5.0/change-set'  xmlns:xs='http://
www.w3.org/2001/XMLSchema-instance'       xs:schemaLocation='http://drools.org/
drools-5.0/change-set      http://anonsvn.jboss.org/repos/labs/labs/jbossrules/
trunk/drools-api/src/main/resources/change-set-1.0.0.xsd'>
   <add>
      <resource source='file:/path_to_process/folder1/' type='BPMN2'/>
      <resource source='file:/path_to_process/folder2/' type='BPMN2'/>
   </add>
</change-set>
```

You can create a process from a changeset file by using the ResourceType CHANGE_SET:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(ResourceFactory.newClassPathResource("changeset.xml"), ResourceType.CHANGE_SET);
KnowledgeBase kbase = kbuilder.newKnowledgeBase();
```

You can also use a knowledge agent to create a knowledge base. The main advantage of a knowledge agent is that you can configure it to automatically update the knowledge base if the resource(s) it is based on are updated. When initializing the knowledge agent, you need to use a changeset to define which resources it should monitor. This could either be files, or folders, in which case it will automatically update itself for all files added, updated or removing in that folder. For example, you could use the following snippet to create a kbase from a folder on the file system, and it will check every ten seconds (this is configurable of course) for updates, and will add, update or remove processes based on updates.

```
ResourceChangeScannerConfiguration sconf = ResourceFactory.getResourceChangeScannerService().ne
sconf.setProperty( "drools.resource.scanner.interval", "10" ); // every 10s
```

```
ResourceFactory.getResourceChangeScannerService().configure( sconf );
ResourceFactory.getResourceChangeScannerService().start();
ResourceFactory.getResourceChangeNotifierService().start();
KnowledgeAgentConfiguration aconf = KnowledgeAgentFactory.newKnowledgeAgentConfiguration();
aconf.setProperty("drools.agent.newInstance", "false");
KnowledgeAgent    kagent    =    KnowledgeAgentFactory.newKnowledgeAgent("Folder
 changeset", aconf);
kagent.applyChangeSet(ResourceFactory.newClassPathResource("changesetFolder.xml"));
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

```
<change-set  xmlns='http://drools.org/drools-5.0/change-set'  xmlns:xs='http://
www.w3.org/2001/XMLSchema-instance'       xs:schemaLocation='http://drools.org/
drools-5.0/change-set      http://anonsvn.jboss.org/repos/labs/labs/jbossrules/
trunk/drools-api/src/main/resources/change-set-1.0.0.xsd'>
   <add>
      <resource source='file:/path_to_process/folder1/' type='BPMN2'/>
   </add>
</change-set>
```

A knowledge agent can also load a kbase from the Guvnor repository. An example is provided in the process repository chapter.

5.1.2. 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).

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

The `ProcessRuntime` interface defines all the session methods for interacting with processes, as shown 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 complet
     * (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();
```

5.1.3. 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:

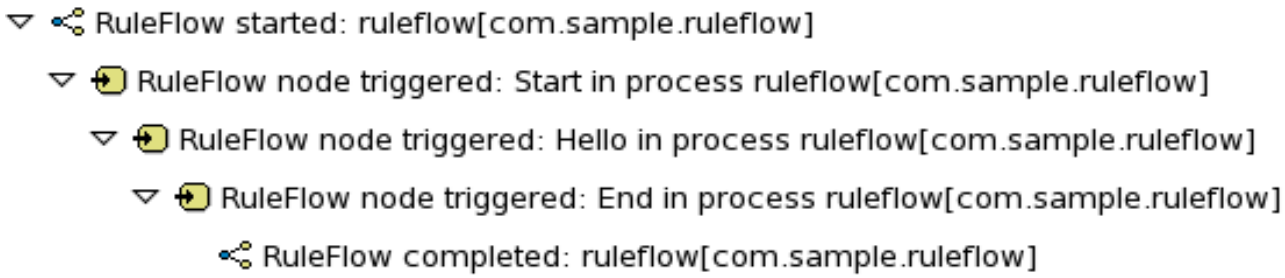
1. Console logger: This logger writes out all the events to the console.
2. File logger: This logger writes out all the events to a file using an XML representation. This log file might then be used in the IDE to generate a tree-based visualization of the events that occurred during execution.

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.



5.2. 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.

Chapter 6. Core Engine: Basics

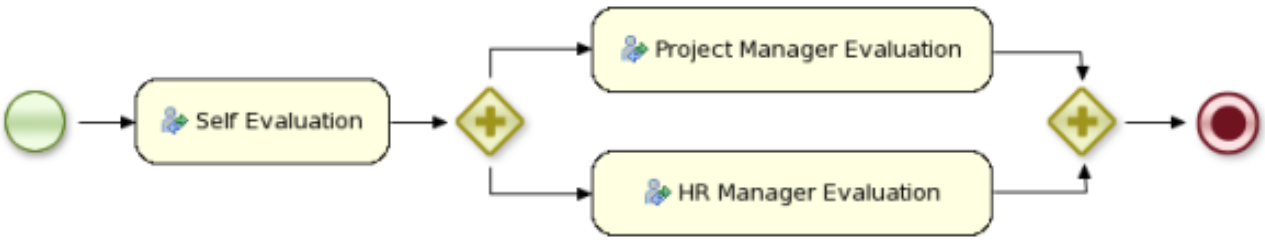


Figure 6.1.

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.

6.1. Creating a process

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

1. Using the graphical process editor in the Eclipse plugin
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.

6.1.1. 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 editor is part of the jBPM / Drools Eclipse plugin. 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.

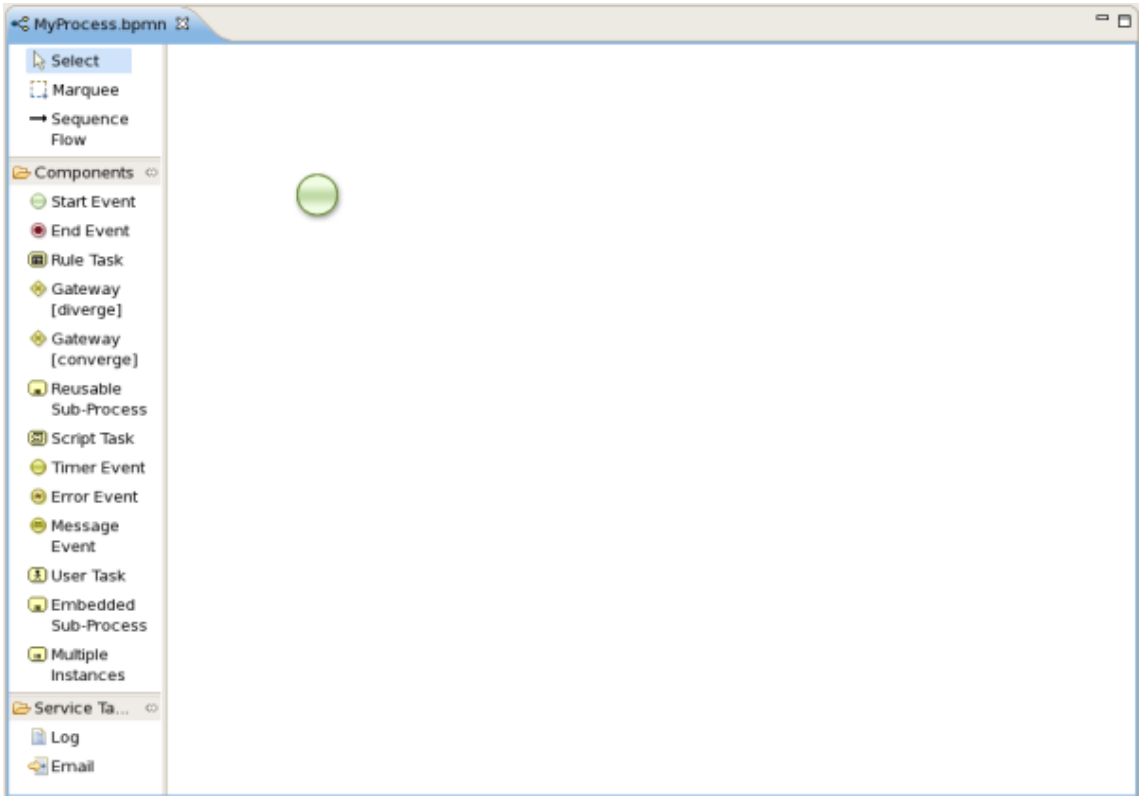


Figure 6.2. New process

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 "Components" 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 "Components" palette.

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

6.1.2. 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">

<processprocessType="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.

6.1.3. 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.

6.1.3.1. 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.

6.2. 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.

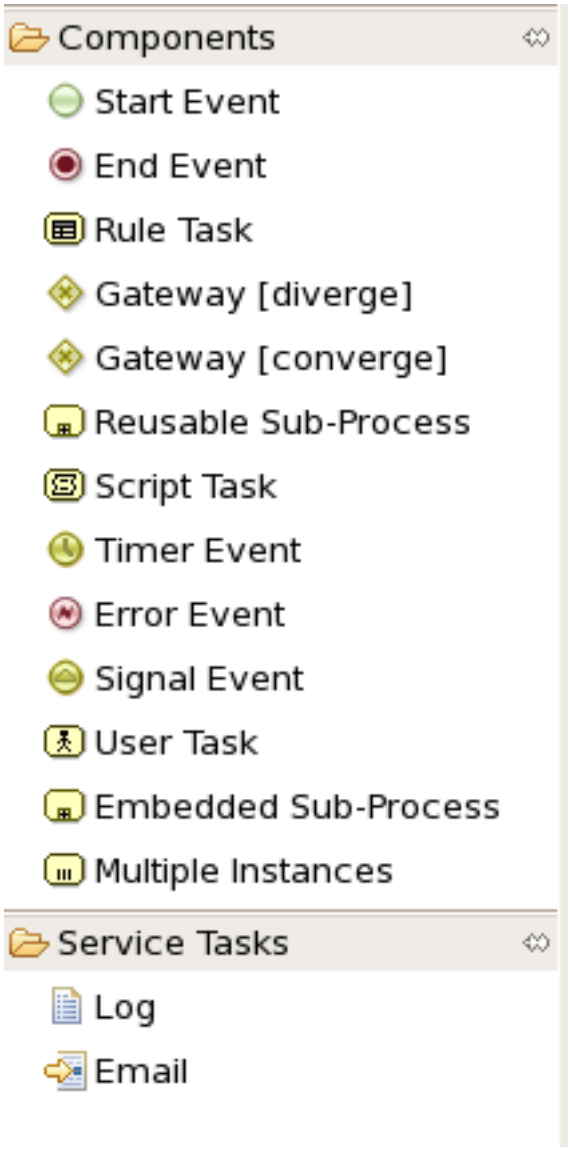


Figure 6.3. The different types of BPMN2 nodes

6.3. 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.
- *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 “*Human Tasks*” for details.

6.4. Details: Events

6.4.1. Start event

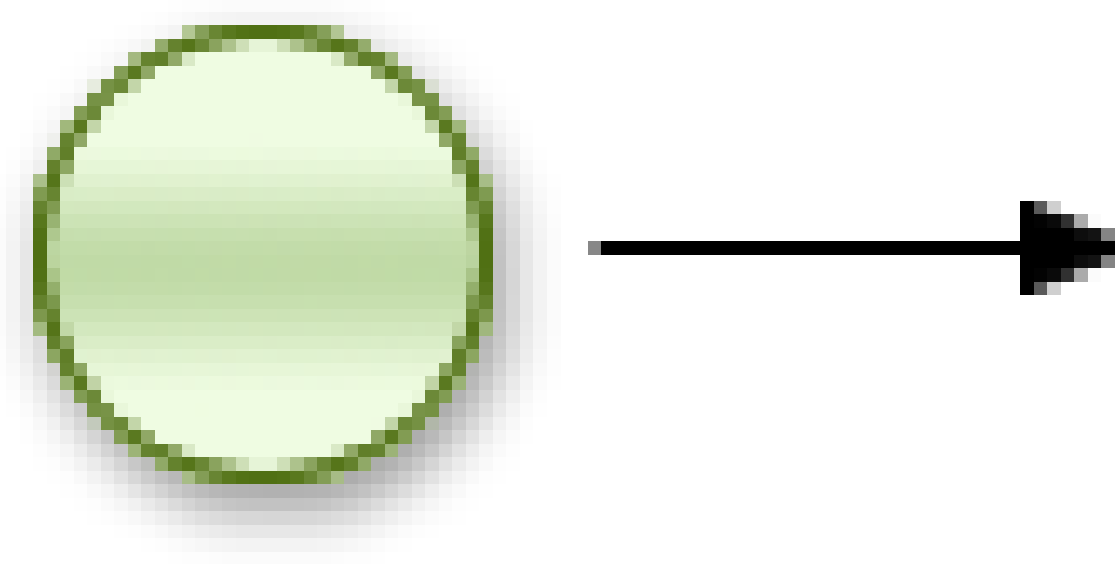


Figure 6.4. Start event

The start of the process. A process should have exactly one start node, 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.

6.4.2. End events

6.4.2.1. End event

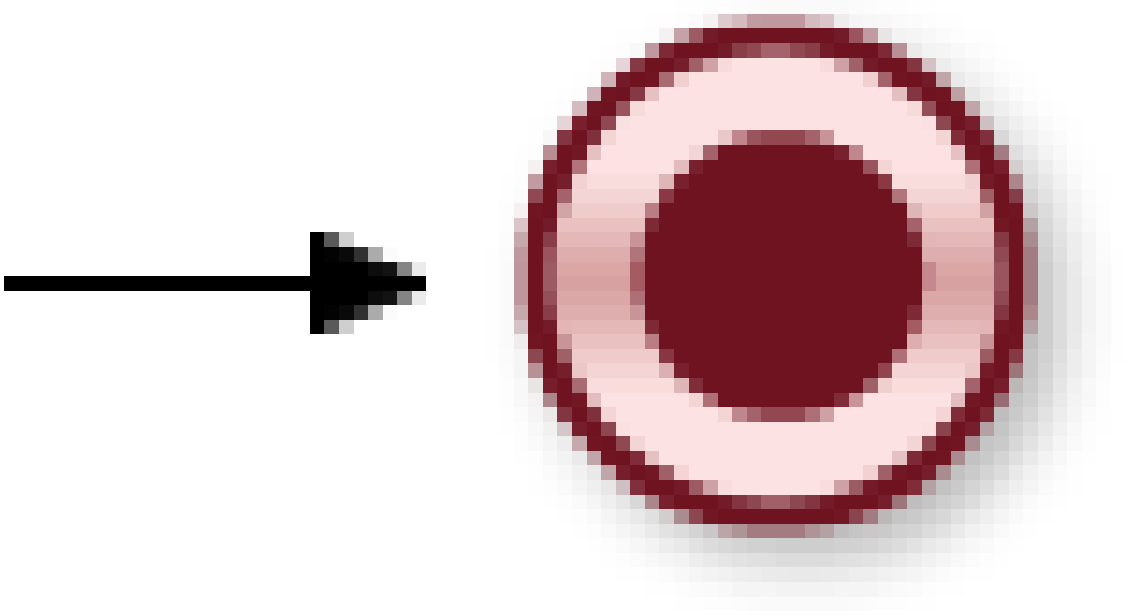


Figure 6.5. End event

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 ends 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 the top-level process instance, not just that sub-process.

6.4.2.2. Throwing error event

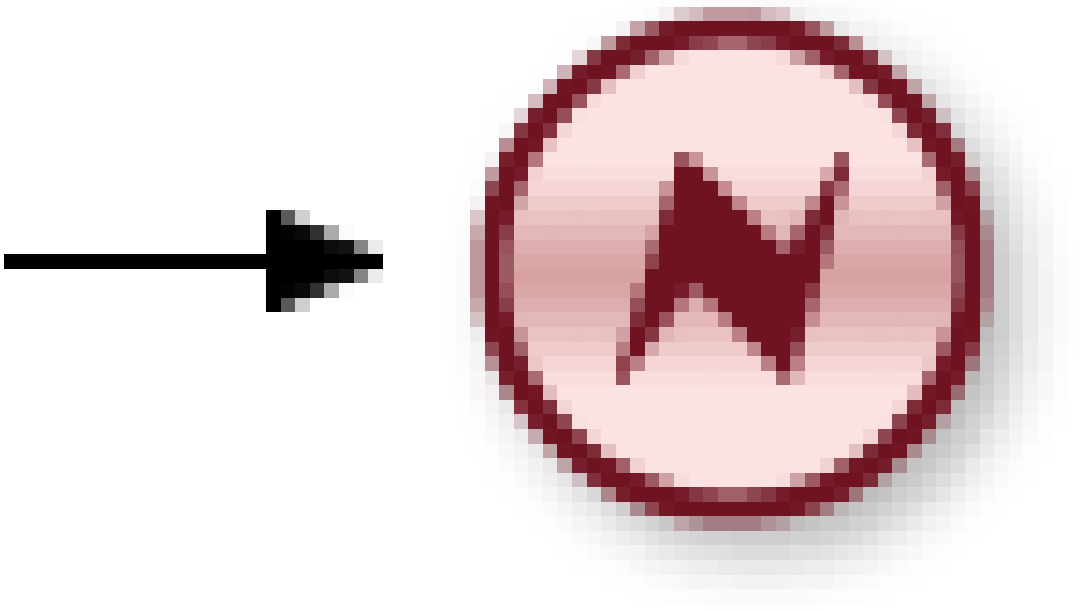


Figure 6.6. Throwing error event

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. This is however currently only possible when working with XML directly. We will be adding support for graphically specifying this in the new BPMN2 editor.

6.4.3. Intermediate events

6.4.3.1. Catching timer event

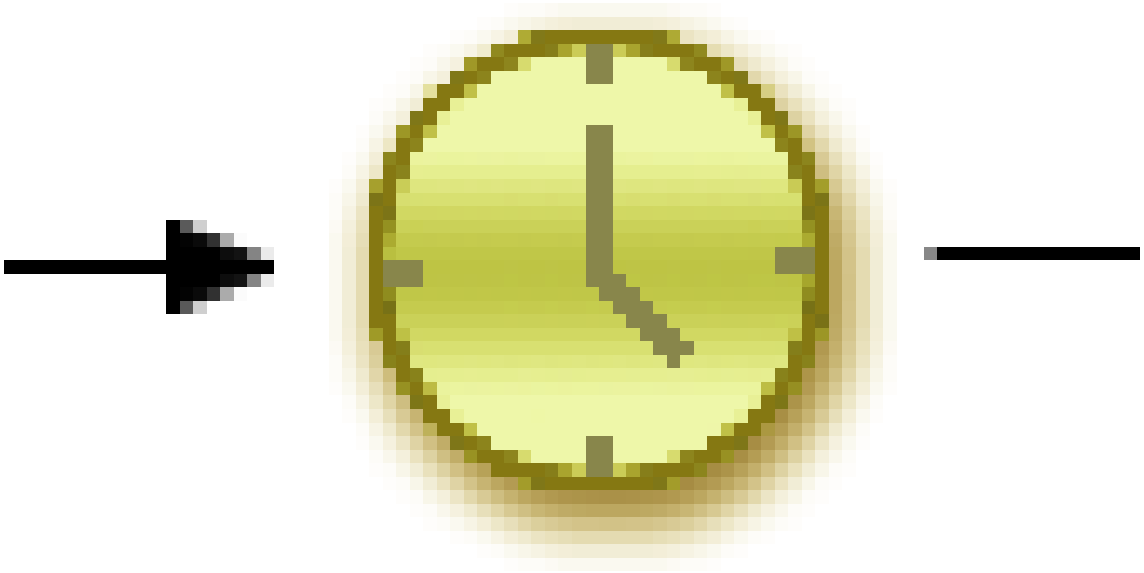


Figure 6.7. Catching timer event

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. This is however currently only possible when working with XML directly. We will be adding support for graphically specifying this in the new BPMN2 editor.

6.4.3.2. Catching signal event

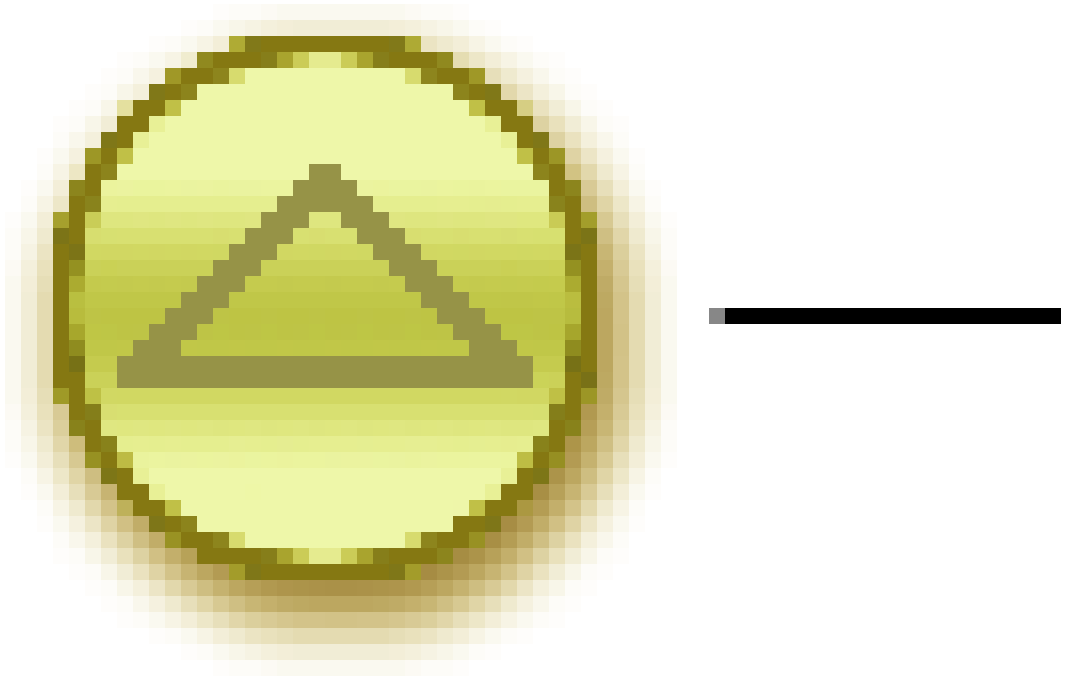


Figure 6.8. Catching signal event

A Signal Event can be used to respond to internal or external events during the execution of the process. A Signal Event should have no incoming connections and one outgoing connection. It specifies the type of event that is expected. Whenever that type of event is detected, the node connected to this event node will be triggered. 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. This is however currently only possible when working with XML directly. We will be adding support for graphically specifying this in the new BPMN2 editor.

6.5. Details: Activities

6.5.1. Script task

Figure 6.9. 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.

6.5.2. Service task

Figure 6.10. 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}`.

6.5.3. User task

Figure 6.11. 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.

6.5.4. Reusable sub-process

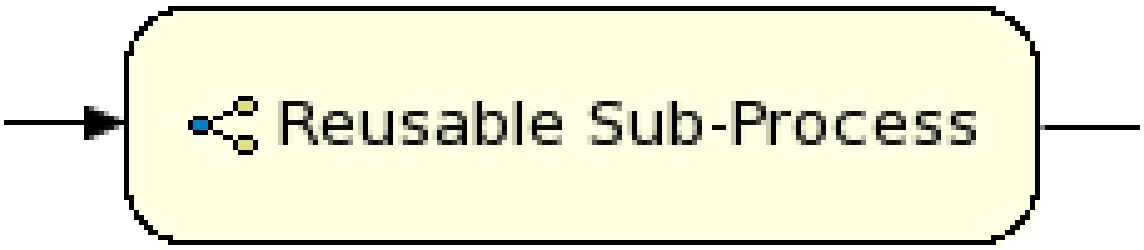


Figure 6.12. Reusable sub-process

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.

6.5.5. Business rule task

Figure 6.13. 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.

6.5.6. Embedded sub-process

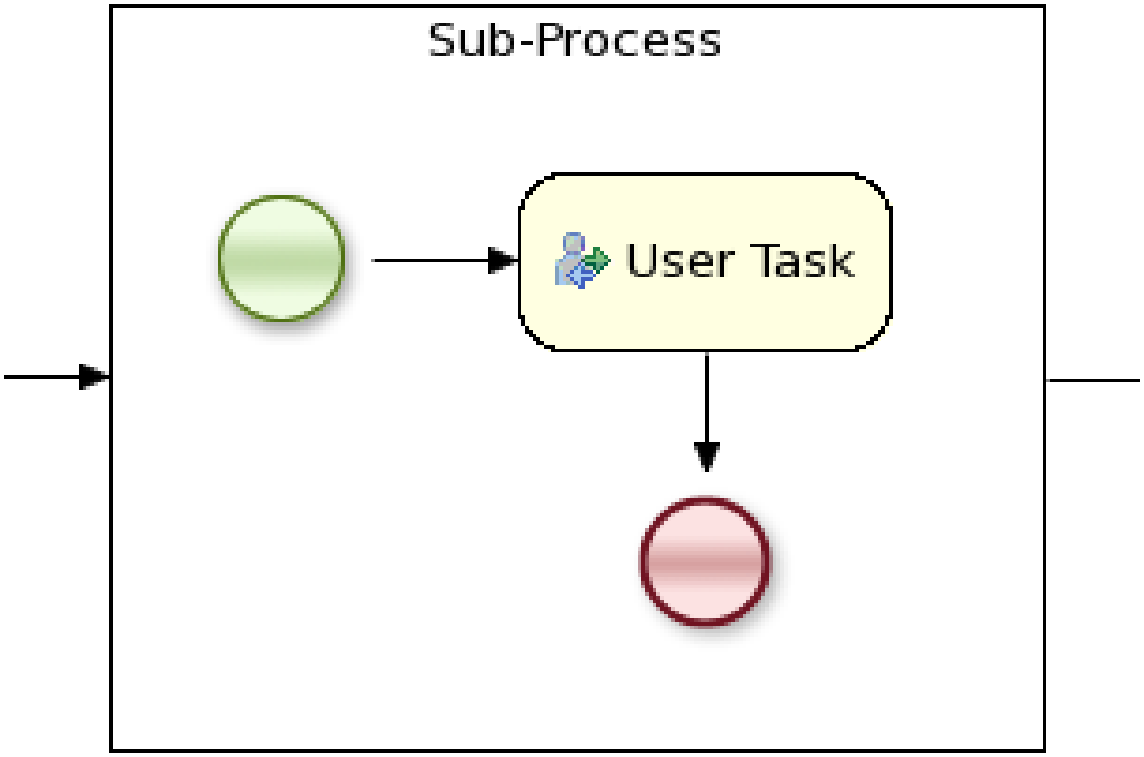


Figure 6.14. Embedded sub-process

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 the top-level process instance, not just that sub-process, so in general you should use non-terminating end nodes inside a 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.

6.5.7. Multi-instance sub-process

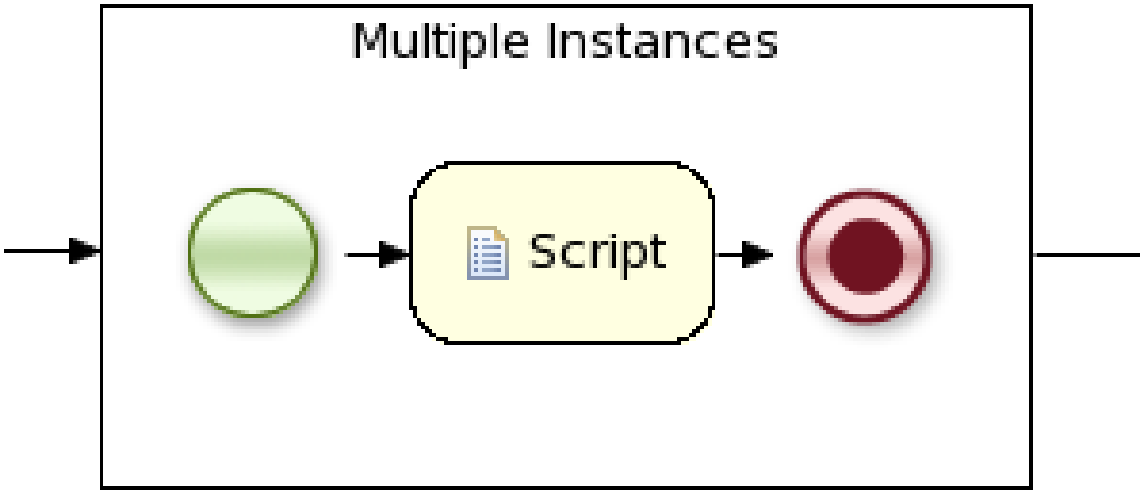


Figure 6.15. Multi-instance sub-process

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.

6.6. Details: Gateways

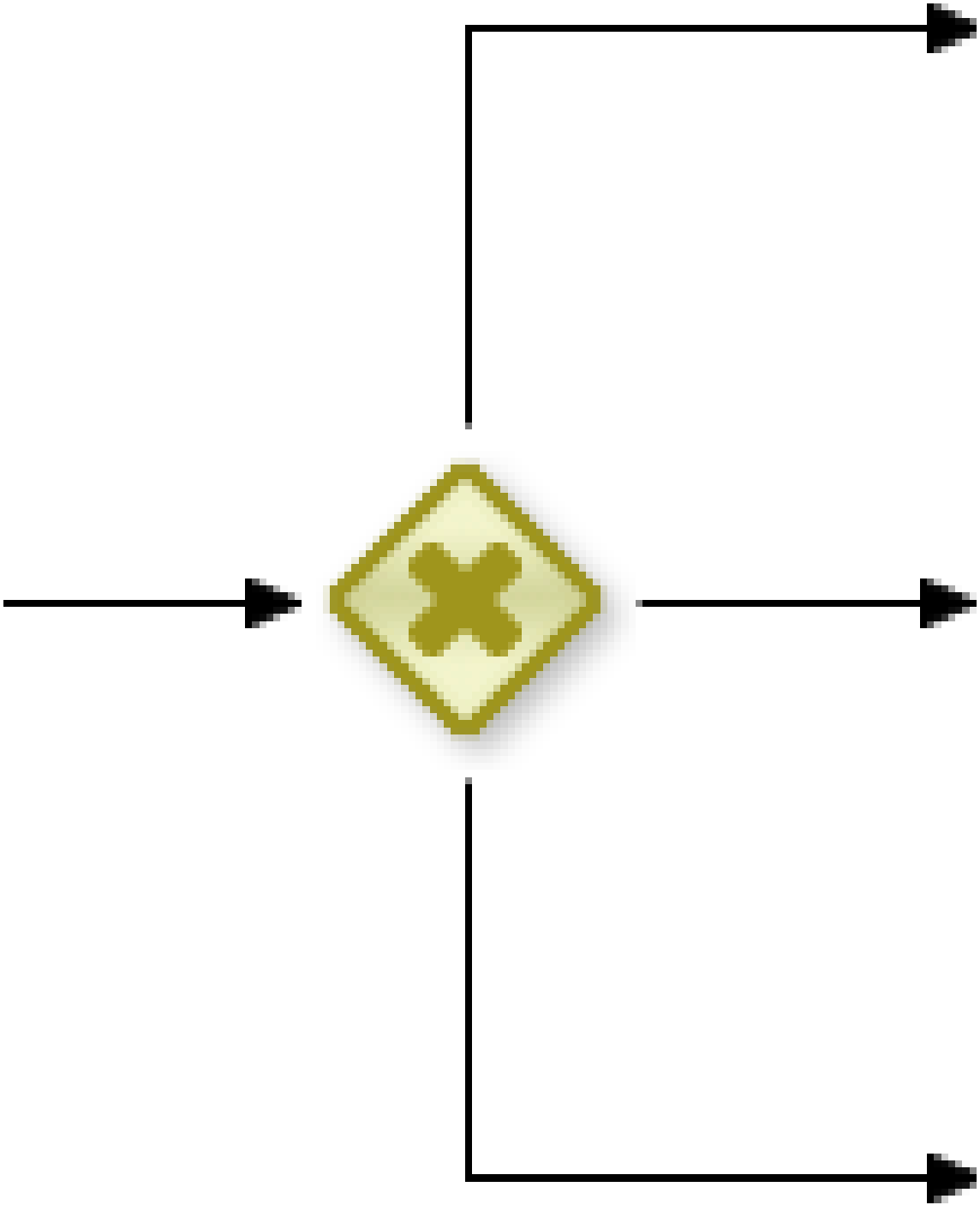


Figure 6.16. Diverging gateway

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 ruleflow 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 process 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).

6.6.2. Converging gateway

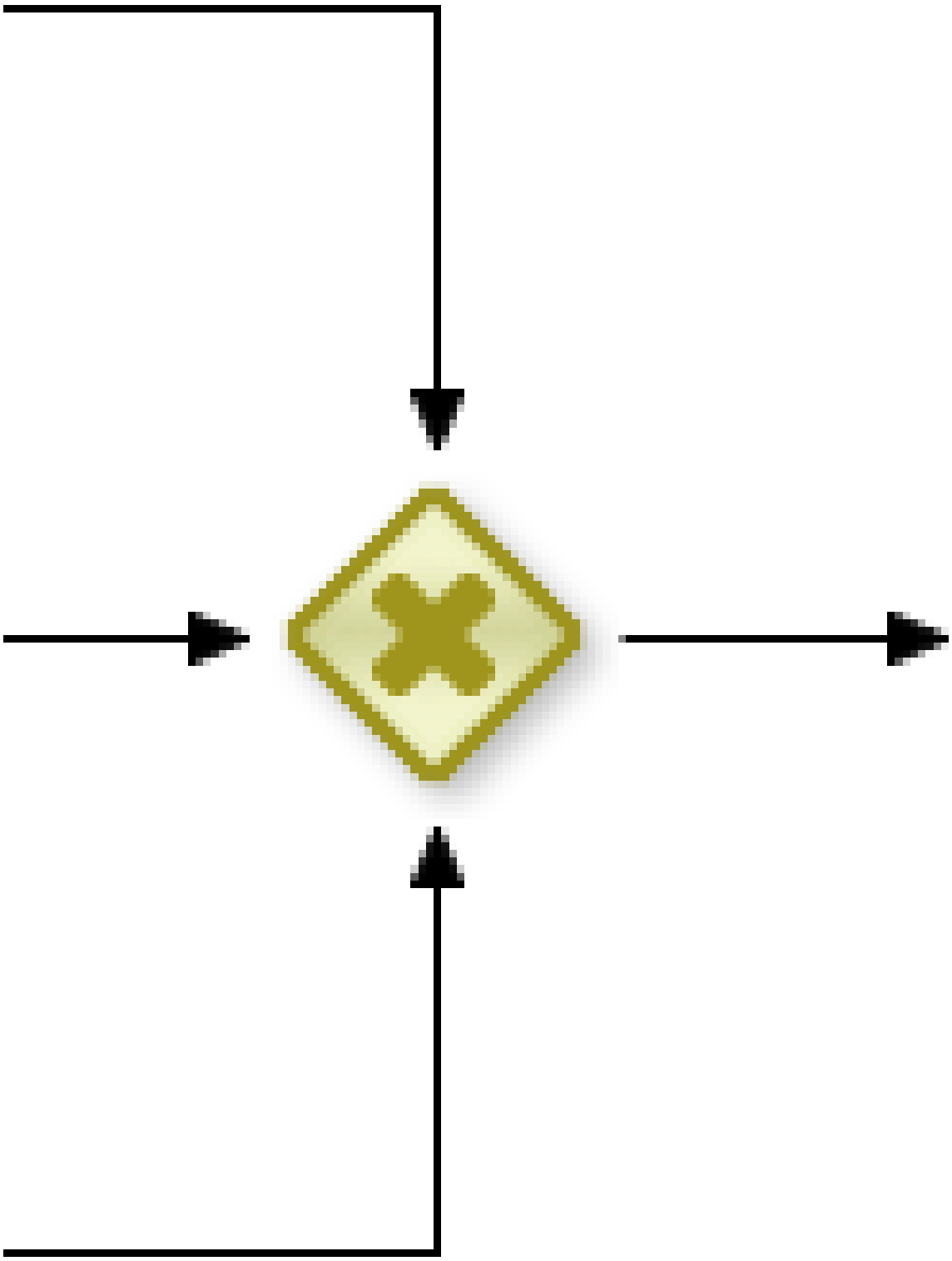


Figure 6.17. Converging gateway

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

- AND or parallel means that is 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.

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 XOR.

6.7. 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:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource("MyProcess.bpmn2"),
              ResourceType.BPMN2 );
```

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

```
KnowledgeBase kbase = kbuilder.newKnowledgeBase();
```

   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:

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

6.8. Other features

6.8.1. 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 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. 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.getContextInsta
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) {
        StatefulKnowledgeSession ksession = ((KnowledgeCommandContext) context).getStatefulKn
        org.jbpm.process.instance.ProcessInstance processInstance = (org.jbpm.process.instanc
```

```
        VariableScopeInstance variableScope = (VariableScopeInstance) processInstance.getCont
        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);`.

6.8.2. 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 ...
```

6.8.3. 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 );
```

6.8.4. Events

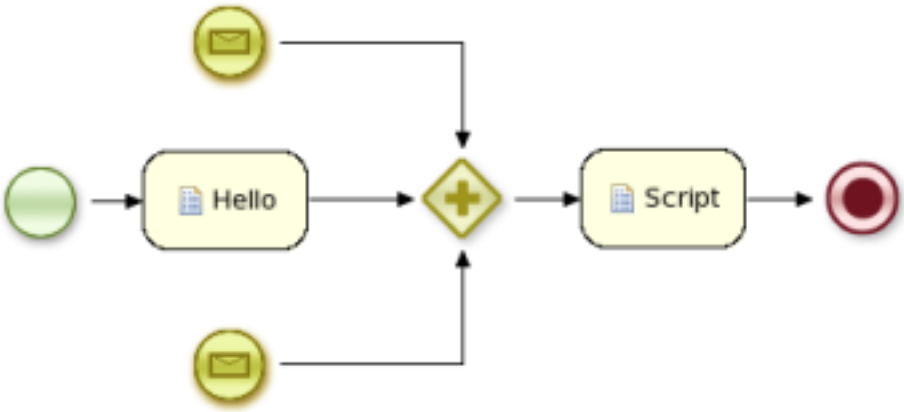


Figure 6.18. A sample process using events

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.

6.8.5. 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.

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).

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 as a boundary event. This is however currently only possible when working with XML directly. We will be adding support for graphically specifying this in the new BPMN2 editor.

6.8.6. 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.

6.8.6.1. 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"
KnowledgeBuilder kbuilder = ...
StatefulKnowledgeSession ksession = ...
ProcessInstance processInstance = ksession.startProcess("com.sample.process");

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

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

If this kind of mapping is 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.

6.8.7. 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 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.

6.8.7.1. 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.

6.8.7.2. 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.

6.8.7.3. 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 commiting changes to the databse. 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 posible (and enabled!!) in the database (and tables used), then this situation will not occur.

Chapter 7. Core Engine: BPMN 2.0

7.1. Business Process Model and Notation (BPMN) 2.0 specification

```
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.

jBPM5 allows you to execute processes defined using the BPMN 2.0 XML format. That means that you can use all the different jBPM5 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.

jBPM5 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, Timer)
    - Interrupting Boundary Event (Escalation, Error, Timer, 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
    - Ad-Hoc Sub-Process
    - Data-Object
  - Gateways
    - Diverging
      - Exclusive

        - Inclusive
        - Parallel
        - Event-Based
      - Converging
        - Exclusive
        - 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 that 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" isExecutable="true" id="com.sample.HelloWorld" name="Hello
  World" >

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

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

  </process>

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

</definitions>
```

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

- Create a new Flow 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 Designer is an open-source web-based editor that supports the BPMN 2.0 format. We have embedded it into Guvnor 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 Guvnor and import them so they can be executed.
- A new BPMN2 Eclipse plugin is being created to support the full BPMN2 specification. It is currently still under development and only supports a limited number of constructs and attributes, but can already be used to create simple BPMN2 processes. To create a new BPMN2 file for this editor, use the wizard (under Examples) to create a new BPMN2 file, which will generate a .bpmn2 file and a .prd file containing the graphical information. Double-click the .prd file to edit the file using the graphical editor. For more detail, check out the chapter on the new BPMN2 Eclipse plugin.
- 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.

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

```
private static KnowledgeBase createKnowledgeBase() throws Exception {
  KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
  kbuilder.add(ResourceFactory.newClassPathResource("sample.bpmn2"), ResourceType.BPMN2);
  return kbuilder.newKnowledgeBase();
}
```

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

```
KnowledgeBase kbase = createKnowledgeBase();
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
ksession.startProcess("com.sample.HelloWorld");
```

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

7.2. Examples

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

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/].

7.3. Supported elements / attributes

Table 7.1. Keywords

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
definitions		• rootElement • BPMNDiagram		
process	• processType • isExecutable • name • id	• property • laneSet • flowElement	• packageName • adHoc • version	• import • global
sequenceFlow	• sourceRef • targetRef • isImmediate • name • id	• conditionExpression	• priority	
interface	• name • id • implementationRef	• operation		
operation	• name • id • implementationRef	• inMessageRef		
laneSet		• lane		
lane	• name • id	• flowNodeRef		
import*		• name		

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
global*		• identifier • type		
Events				
startEvent	• name • id • isInterrupting	• dataOutput • dataOutputAssociation • outputSet • eventDefinition		
endEvent	• name • id	• dataInput • dataInputAssociation • inputSet • eventDefinition		
intermediateCatchEvent	• name • id	• dataOutput • dataOutputAssociation • outputSet • eventDefinition		
intermediateThrowEvent	• name • id	• dataInput • dataInputAssociation • inputSet • eventDefinition		
boundaryEvent	• cancelActivity • attachedToRef • name • id	• eventDefinition		
terminateEventDefinition				
compensateEventDefinition	• activityRef	• documentation • extensionElements		
conditionalEventDefinition		• condition		

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
errorEventDefinition	• errorRef			
error	• errorCode • id			
escalationEventDefinition	• escalationRef			
escalation	• escalationCode • id			
messageEventDefinition	• messageRef			
message	• itemRef • id			
signalEventDefinition	• signalRef			
timerEventDefinition		• timeCycle • timeDuration • timerDate		
Activities				
task	• name • id	• ioSpecification • dataInputAssociation • dataOutputAssociation	• taskName	
scriptTask	• scriptFormat • name • id	• script		
script		• text[mixed content]		
userTask	• name • id	• ioSpecification • dataInputAssociation • dataOutputAssociation • resourceRole • loopCharacteristics		• onEntry-script • onExit-script
potentialOwner		• resourceAssignmentExpression		

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
resourceAssignmentExpression		• expression		
businessRuleTask	• name • id	• ioSpecification • dataInputAssociation • dataOutputAssociation	• ruleFlowGroup	• onEntry-script • onExit-script
manualTask	• name • id			• onEntry-script • onExit-script
sendTask	• messageRef • name • id	• ioSpecification • dataInputAssociation • loopCharacteristics		• onEntry-script • onExit-script
receiveTask	• messageRef • name • id	• ioSpecification • dataOutputAssociation • loopCharacteristics		• onEntry-script • onExit-script
serviceTask	• operationRef • name • id • implementation	• ioSpecification • dataInputAssociation • dataOutputAssociation • loopCharacteristics		• onEntry-script • onExit-script
subProcess	• name • id • triggeredByEvent	• flowElement • property • loopCharacteristics		
adHocSubProcess	• cancelRemainingInstances • name • id	• completionCondition • flowElement • property		
callActivity	• calledElement • name • id	• ioSpecification • dataInputAssociation • dataOutputAssociation	• waitForCompletion • independent	• onEntry-script • onExit-script
multiInstanceLoopCharacteristics		• loopDataInputRef • inputDataItem		

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
		• loopDataOutputRef • outputDataItem		
onEntry-script*	• scriptFormat		• script	
onExit-script*	• scriptFormat		• script	
Gateways				
parallelGateway	• gatewayDirection • name • id			
eventBasedGateway	• gatewayDirection • name • id			
exclusiveGateway	• default • gatewayDirection • name • id			
inclusiveGateway	• default • gatewayDirection • name • id			
Data				
property	• itemSubjectRef • id • name			
dataObject	• itemSubjectRef • id			
itemDefinition	• structureRef			

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

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
waypoint	• width • height • x • y			

Chapter 8. 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.

8.1. 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.

8.1.1. 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 the 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:

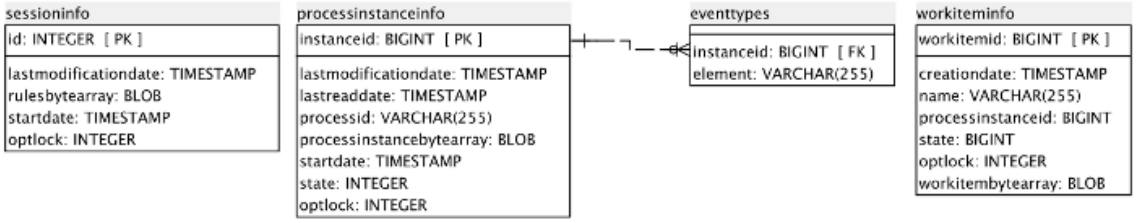


Figure 8.1. jBPM data model

[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 8.1. SessionInfo

Field	Description	Nullable
`id`	The primary key.	NOT NULL
`lastmodificationdate`	The last time that the entity was saved to the database	

Field	Description	Nullable
`rulesbytearray`	The binary dataset containing the state of the session	NOT NULL
`startdate`	The start time of the session	
`optlock`	The version field that serves as its optimistic lock value	

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

Table 8.2. ProcessInstanceInfo

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

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

Table 8.3. EventTypes

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

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

Table 8.4. WorkItemInfo

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

8.1.2. 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.

8.1.3. 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.

8.1.3.1. 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 artefact (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-test (org.jbpm)
- 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)

8.1.3.2. Configuring the engine to use persistence using `JBPMHelper`

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 enables, 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.jpa
persistence.persistenceunit.dialect=org.hibernate.dialect.H2Dialect

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

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);
```

8.1.3.3. Manually configuring the engine to use persistence

You can also 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, env );
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="1.0"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/persistence
     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
     http://java.sun.com/xml/ns/persistence/orm
     http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/persistence">

  <persistence-unit name="org.jbpm.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>

    <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.manager_lookup_class"
                value="org.hibernate.transaction.BTMTransactionManagerLookup"/>
    </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 above) 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:tcp://localhost/~/jbpm-db");
ds.getDriverProperties().put("driverClassName", "org.h2.Driver");
ds.init();
```

8.1.4. 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, env );

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

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

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

Note that, if you use Bitronix as the transaction manager, you should also add a simple `jndi.properties` file in you 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, you can use the JBoss transaction manager. You need to change the transaction manager property in `persistence.xml` to:

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

8.1.4.1. Container managed transaction

Special consideration need to be taken when embedding jBPM inside an application that exucutes in Container Managed Tansaction (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 transation, 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 UserTrancation 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 peristence 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.

8.1.5. Persistence and concurrency

Please see the *Multi-threading* section for more information.

8.2. 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.

Guvnor is a Drools sub-project that does exactly that. It consists of a repository for storing different kinds of knowledge, as well a web application that allows users to view and update the information in the repository. It not only stores process definitions but also can hold rule definitions, object models, and much more.

Easy programmatic retrieval of knowledge packages is possible either using WebDAV or by using a knowledge agent. The knowledge agent will automatically download the information from Guvnor, for example, during the creation of a knowledge base.

Check out the Drools Guvnor documentation for more information on how to do this.

8.3. 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 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.

8.3.1. 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.

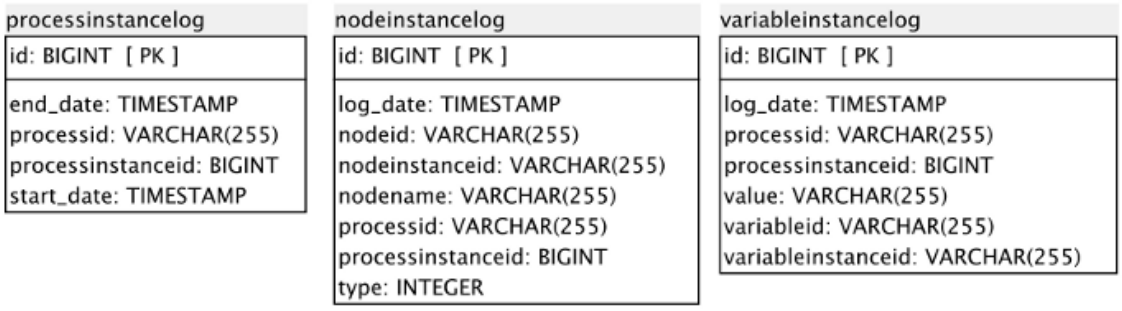


Figure 8.2. jBPM Audit data model

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

Table 8.5. ProcessInstanceLog

Field	Description	Nullable
`id`	The primary key and id of the log entity	NOT NULL
`end_date`	When applicable, the end date of the process instance	
`processid`	The name (id) of the process	
`processinstanceid`	The process instance id	NOT NULL
`start_date`	The start date of the process instance	
`status`	The status of process instance that maps to process instance state	
`parentProcessInstanceId`	The process instance id of the parent process instance if any	
`outcome`	The outcome of the process instance, for instance error code in case of process instance was finished with error event	

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 incomming

connections or is exited through one of its outgoing connections, that information is stored in this table.

Table 8.6. NodeInstanceLog

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

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

Table 8.7. VariableInstanceLog

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

8.3.2. 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:

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

// invoke methods one your session here

logger.dispose();
```

Note that this logger is like any other audit logger, 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. You should dispose the logger when it is no longer needed.

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="1.0"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/persistence
     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
     http://java.sun.com/xml/ns/persistence/orm
     http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/persistence">

  <persistence-unit name="org.jbpm.persistence.jpa">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/processInstanceDS</jta-data-source>
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.process.audit.ProcessInstanceLog</class>
    <class>org.jbpm.process.audit.NodeInstanceLog</class>
    <class>org.jbpm.process.audit.VariableInstanceLog</class>

    <properties>
```

```
    <property name="hibernate.dialect" value="org.hibernate.dialect.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.manager_lookup_class"
                value="org.hibernate.transaction.BTMTransactionManagerLookup"/>
    </properties>
  </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.

Chapter 9. Eclipse BPMN 2.0 Plugin

We are working on a new BPMN 2.0 Eclipse editor that allows you to specify business processes, choreographies, etc. using the BPMN 2.0 XML syntax (including BPMNDI for the graphical information). The editor itself is based on the Eclipse Graphiti framework and the Eclipse BPMN 2.0 EMF meta-model.

Features:

- It supports almost all BPMN 2.0 process constructs and attributes (including lanes and pools, annotations and all the BPMN2 node types).
- Support for the few custom attributes that jBPM5 introduces.
- Allows you to configure which elements and attributes you want use when modeling processes (so we can limit the constructs for example to the subset currently supported by jBPM5, which is a profile we will support by default, or even more if you like).

Many thanks go out to the people at Codehoop that did a great job in creating a first version of this editor.

9.1. Installation

Requirements

- Eclipse 3.6 (Helios) or newer

To install, startup Eclipse and install the Eclipse BPMN2 Modeler from the following update site (from menu Help -> Install new software and then add the update site in question by clicking the Add button, filling in a name and the correct URL as shown below). It will automatically download all other dependencies as well (e.g. Graphiti etc.)

Eclipse 3.6 (Helios): *http://download.eclipse.org/bpmn2-modeler/site-helios/*

Eclipse 3.7 (Indigo): *http://download.eclipse.org/bpmn2-modeler/site/*

The project is hosted at eclipse.org and open for anyone to contribute. The project home page can he found *here* [http://eclipse.org/projects/project.php?id=soa.bpmn2-modeler]. Sources are available *here* [http://git.eclipse.org/c/bpmn2-modeler/org.eclipse.bpmn2-modeler.git]

9.2. Creating your BPMN 2.0 processes

You can use a simple wizard to create a new BPMN 2.0 process (under File -> New - Other ... select BPMN - BPMN2 Diagram).

A video that shows some sample BPMN 2.0 processes from the examples that are part of the BPMN 2.0 specification:

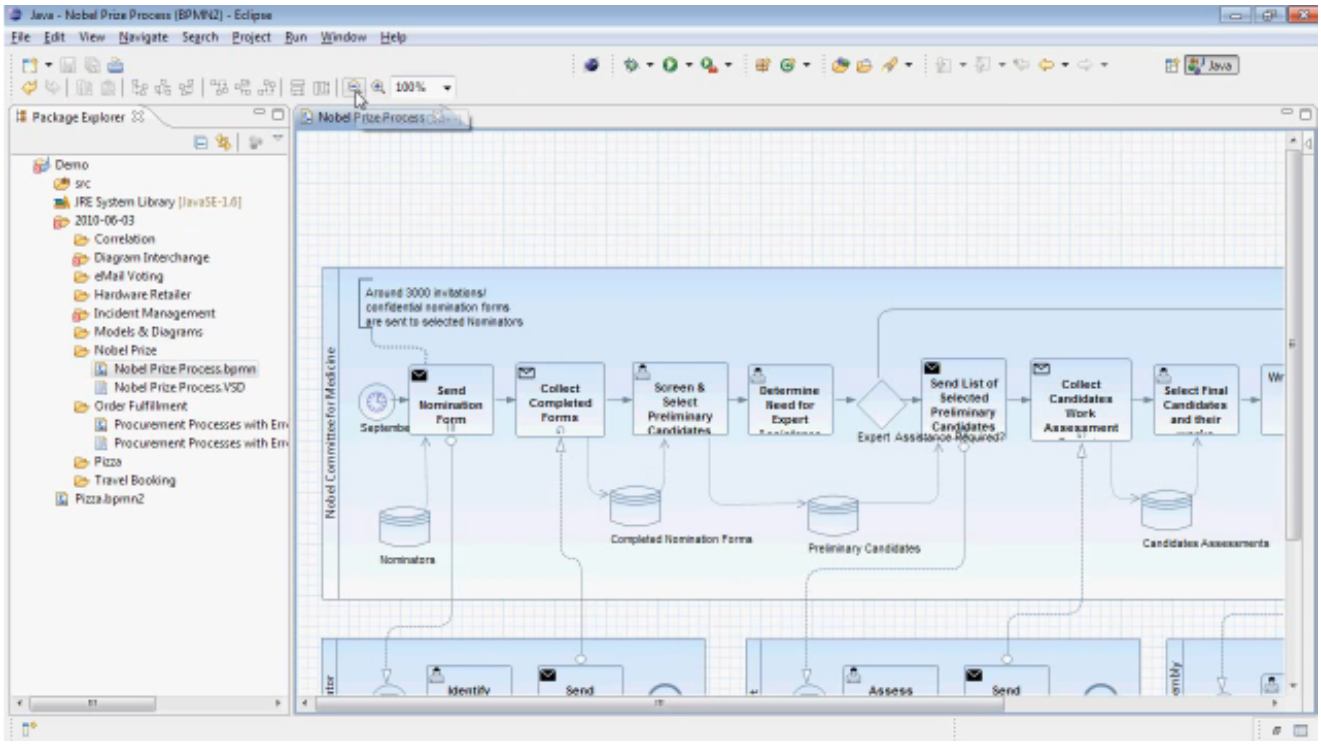


Figure 9.1.

[http://vimeo.com/22021856]

Here are some screenshots of the editor in action.

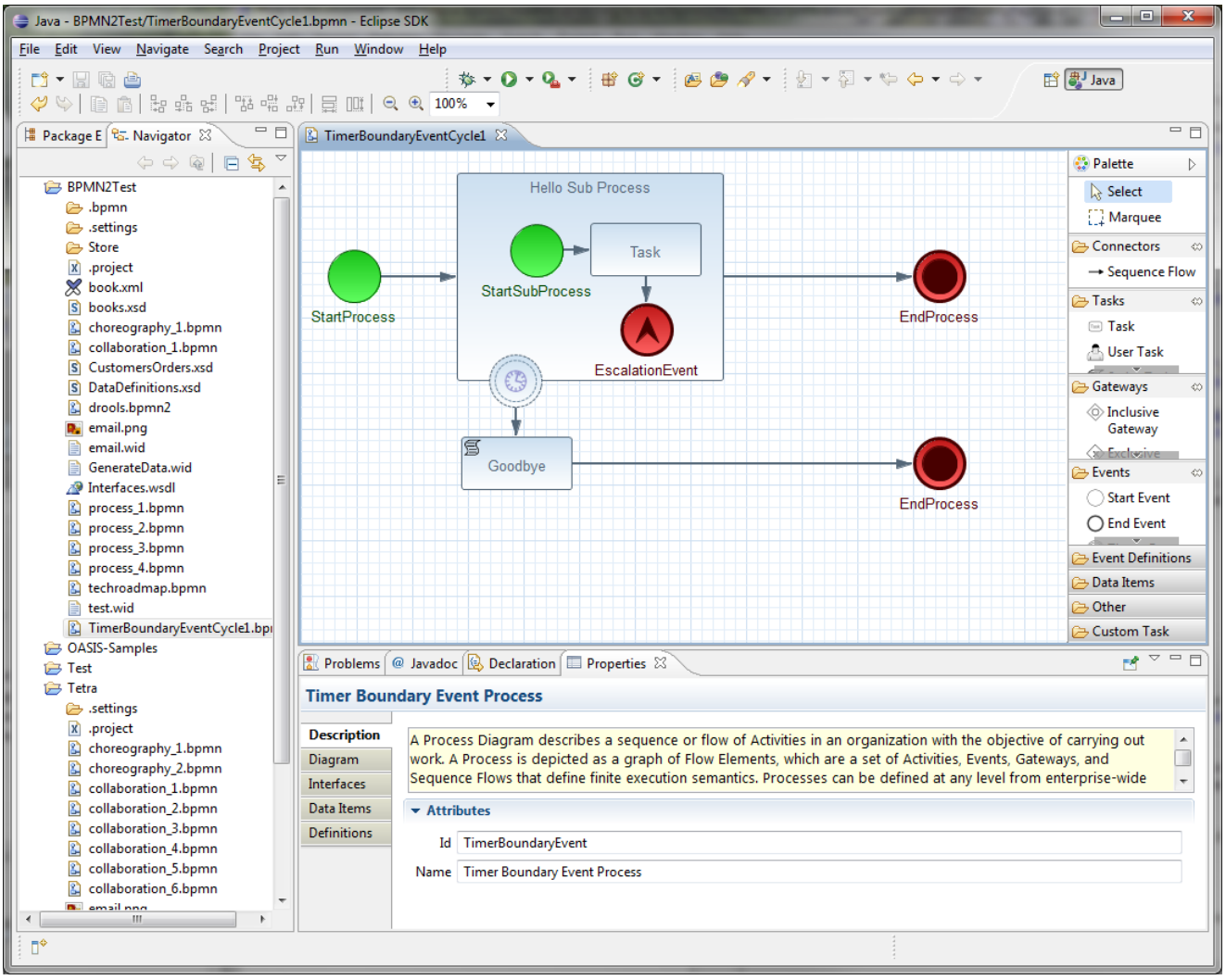


Figure 9.2.

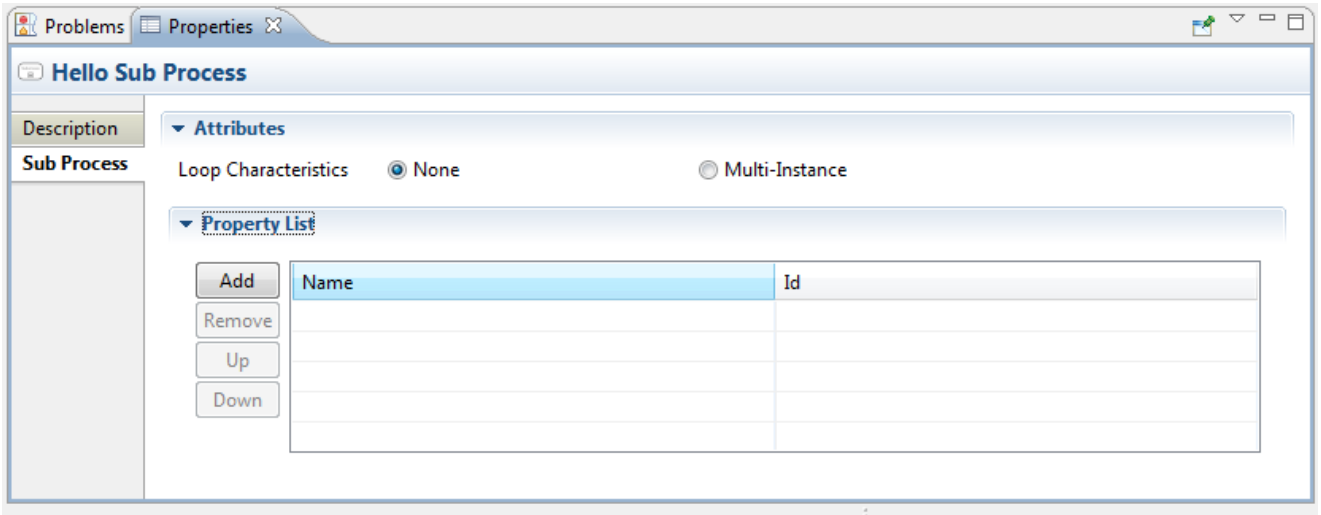


Figure 9.3.

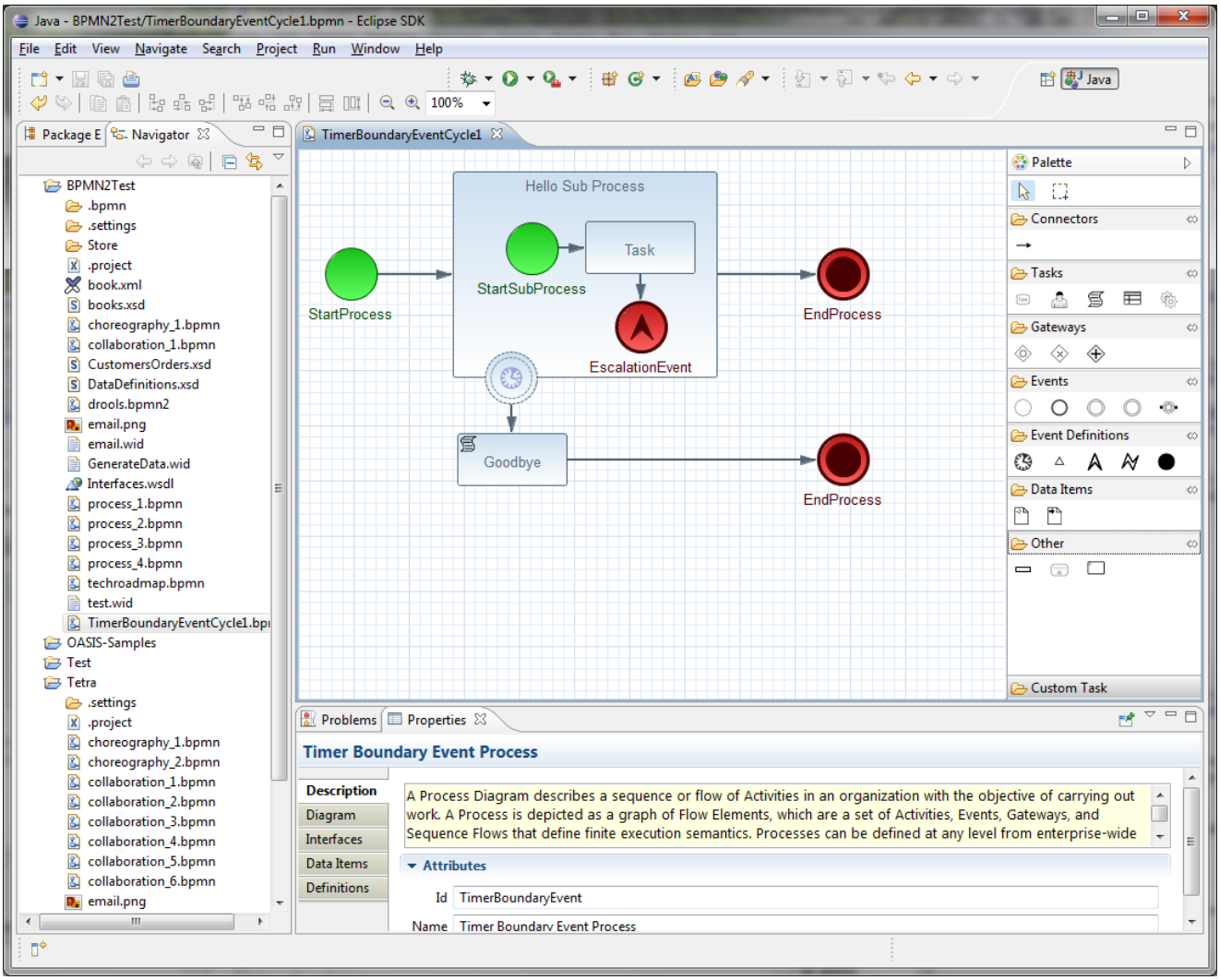

Java - BPMN2Test/TimerBoundaryEventCycle1.bpmn - Eclipse SDK
File Edit View Navigate Search Project Run Window Help
Package E
Navigator
BPMN2Test
.bpmn
.settings
Store
.project
book.xml
books.xsd
choreography_1.bpmn
collaboration_1.bpmn
CustomersOrders.xsd
DataDefinitions.xsd
drools.bpmn2
email.png
email.wid
GenerateData.wid
Interfaces.wsdl
process_1.bpmn
process_2.bpmn
process_3.bpmn
process_4.bpmn
techroadmap.bpmn
test.wid
TimerBoundaryEventCycle1.bpı
OASIS-Samples
Test
Tetra
.settings
.project
choreography_1.bpmn
choreography_2.bpmn
collaboration_1.bpmn
collaboration_2.bpmn
collaboration_3.bpmn
collaboration_4.bpmn
collaboration_5.bpmn
collaboration_6.bpmn
email.png
TimerBoundaryEventCycle1
Hello Sub Process
StartProcess
StartSubProcess
Task
EscalationEvent
EndProcess
Goodbye
EndProcess
Palette
Connectors
Tasks
Gateways
Events
Event Definitions
Data Items
Other
Custom Task
Problems
Properties
Timer Boundary Event Process
Description
Diagram
Interfaces
Data Items
Definitions
A Process Diagram describes a sequence or flow of Activities in an organization with the objective of carrying out work. A Process is depicted as a graph of Flow Elements, which are a set of Activities, Events, Gateways, and Sequence Flows that define finite execution semantics. Processes can be defined at any level from enterprise-wide
Attributes
Id
TimerBoundaryEvent
Name
Timer Boundary Event Process

Figure 9.4.

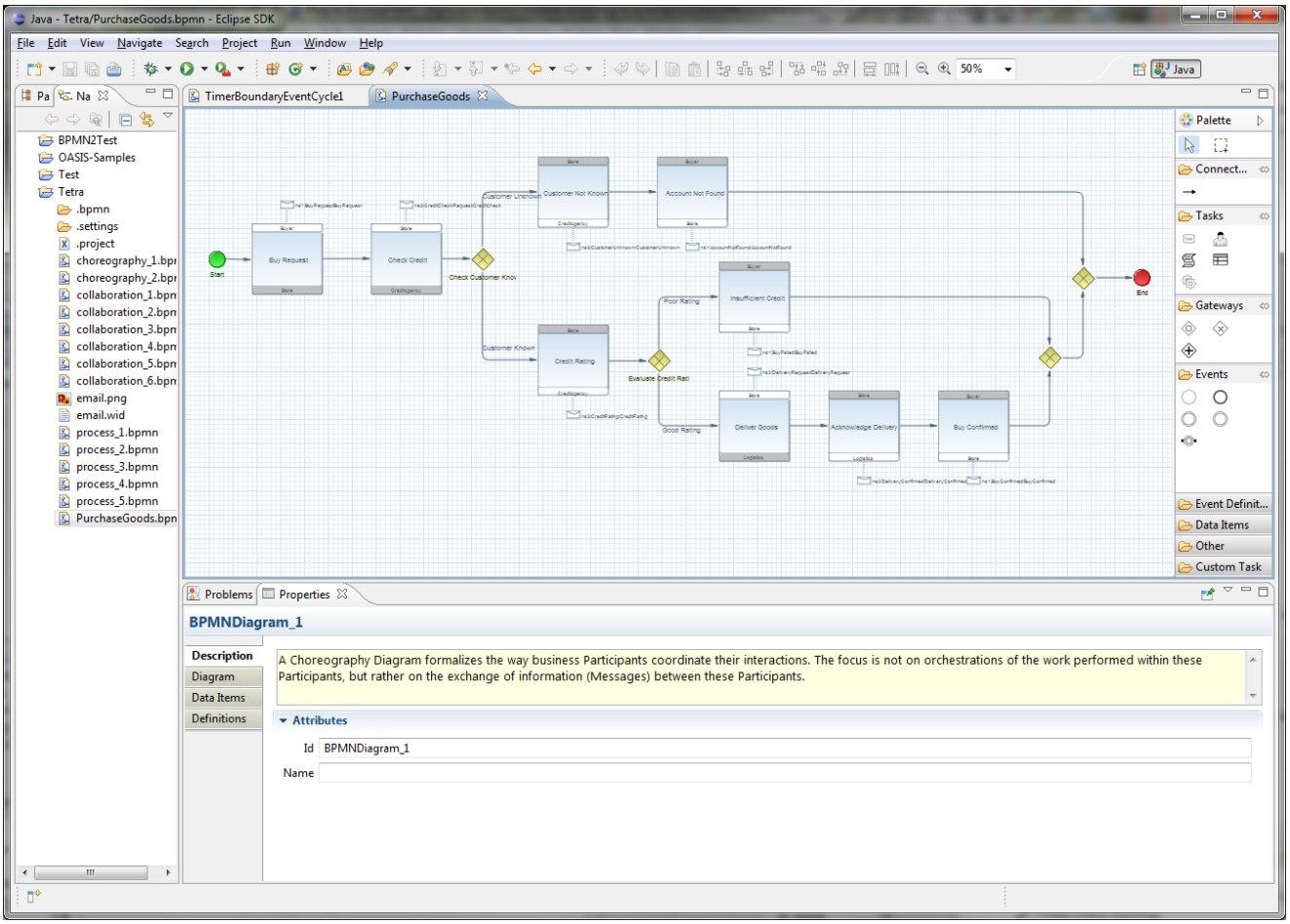


Figure 9.5.

9.2.1. Filtering elements and attributes

You can define which of the BPMN 2.0 elements and attributes you want to use when describing your BPMN 2.0 diagrams. Since the BPMN 2.0 specification is rather complex and includes a very large set of different node types and attributes for each of those nodes, you may not want to use all of these elements and attributes in your project. Elements and attributes can be enablement / disabled at the project level using the BPMN2 preferences category (right-click your project folder and select Properties ... which will open up a new dialog). The BPMN2 preferences contain an entry for all supported elements and attributes (per node type) and you can enable or disable each of those by (un)checking the box for each of those elements and attributes.

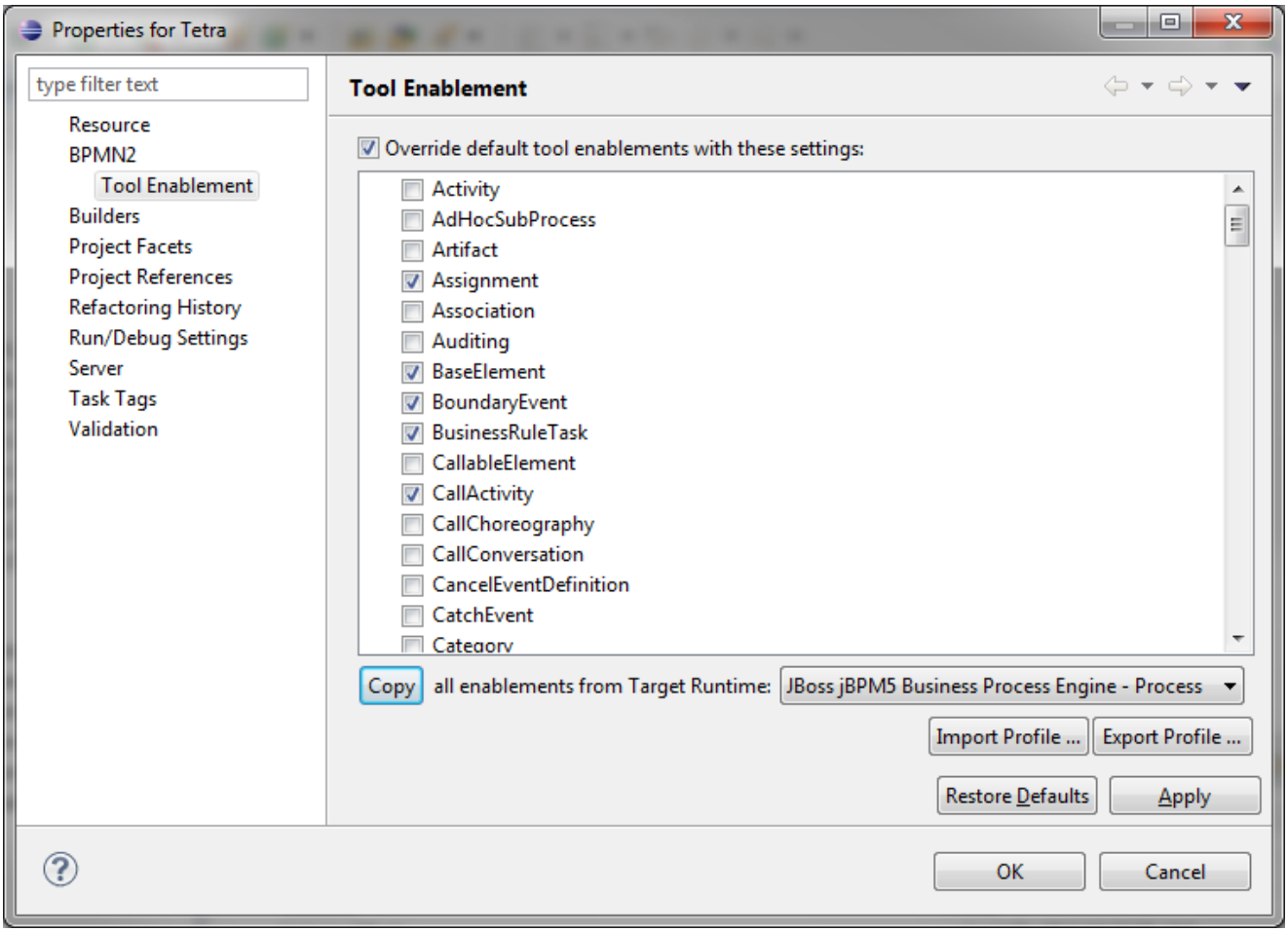


Figure 9.6.

9.2.2. Adding custom task nodes

When creating and adding `<task>` or other `<task>` type nodes to a process, you might want to add input and output parameters to the node. Furthermore, you can configure jBPM to use custom `WorkItemHandler` implementations in conjunction with these nodes. These `WorkItemHandler` instances will then be used when your service node is reached.

The concept of customizing `<task>` and other `<task>`-type nodes and using custom `WorkItemHandler` implementations with these nodes is referred to as creating *custom work items* within jBPM. More information about this can be found in the *Domain-specific processes* chapter.

The following sections cover the following node types:

- Task
- Service Task
- Send Task

- Receive Task
- Manual Task*

Tip

A *Manual Task* should be used to represent a human activity that is *not* managed by the process engine or the human-task component. In this case, you would probably create a custom `WorkItemHandler` implementation that would interface with a technical component that the actor would use to indicate completion of the task.

However, if you were using a human task server, such as the jBPM human-task component, then you would use a *User Task* node instead.

9.2.2.1. Configuring the input and output parameters

WRITE MORE: what they are, example for notification?

9.2.2.2. Configuring the node to be handled by a `WorkItemHandler`

WRITE MORE: add tns:taskname as an attribute to the element

When your process is actually run by the engine, you'll have to first register your `WorkItemHandler` with jBPM. See the *Work Item Handlers* section for an overview and the *???* sections for an example.

9.3. Changing editor behavior

The "General settings" tab in the User Preferences lets you specify a "Target Runtime" which customizes the editor's behavior for a specific BPMN execution environment. Currently only jBPM5 and a generic runtime are defined for the editor, but others are in the works.

This preference page (shown below) also lets you configure default values for BPMN Diagram Interchange (or "DI") attributes.

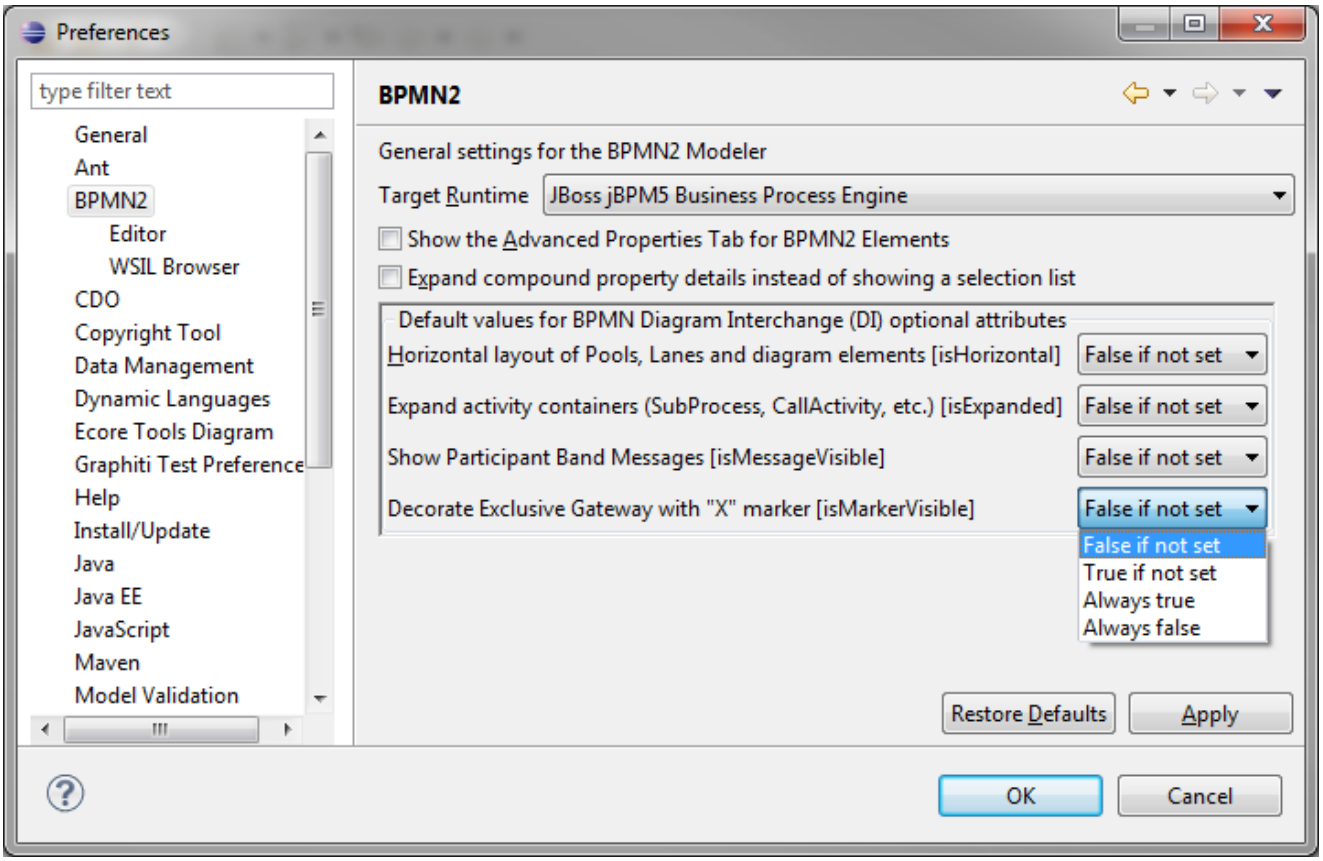


Figure 9.7.

9.4. Changing editor appearance

The preference page shown below lets you customize the appearance (colors and fonts) for all of the different elements that can be placed on the diagram canvas.

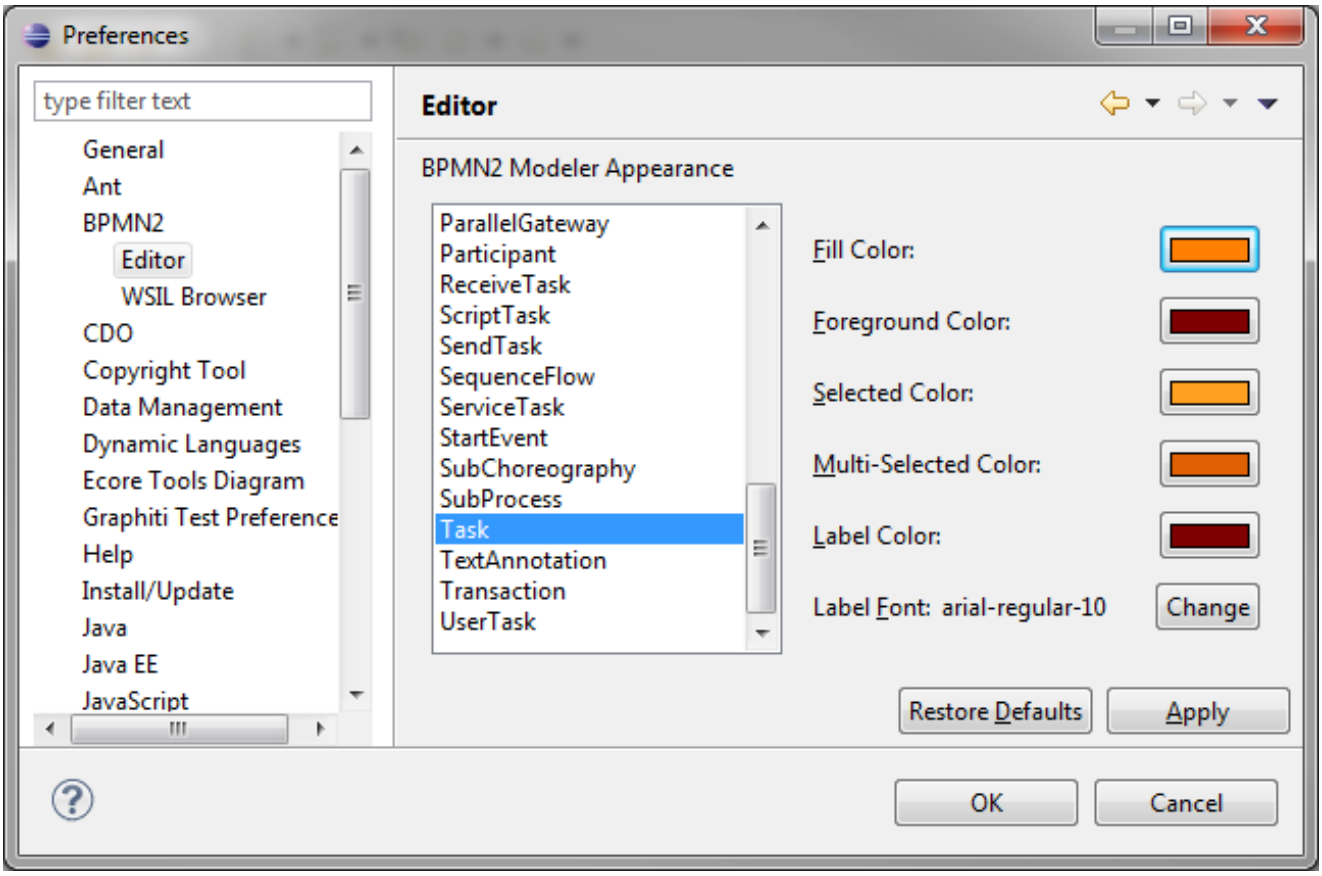

Preferences
type filter text
General
Ant
BPMN2
Editor
WSIL Browser
CDO
Copyright Tool
Data Management
Dynamic Languages
Ecore Tools Diagram
Graphiti Test Preference
Help
Install/Update
Java
Java EE
JavaScript
Editor
BPMN2 Modeler Appearance
ParallelGateway
Participant
ReceiveTask
ScriptTask
SendTask
SequenceFlow
ServiceTask
StartEvent
SubChoreography
SubProcess
Task
TextAnnotation
Transaction
UserTask
Fill Color:
Foreground Color:
Selected Color:
Multi-Selected Color:
Label Color:
Label Font: arial-regular-10
Change
Restore Defaults
Apply
OK
Cancel

Figure 9.8.

Chapter 10. Designer

Web-based process editing is possible using the jBPM Designer. The designer is fully integrated into Drools Guvnor, the knowledge repository where you can store all your BPM assets such as of course your BPMN2 processes as well as rules, process images, workitem configurations, and process forms. The Designer can be used to create, view or update BPMN2 based processes which are executable in the jBPM runtime environment.

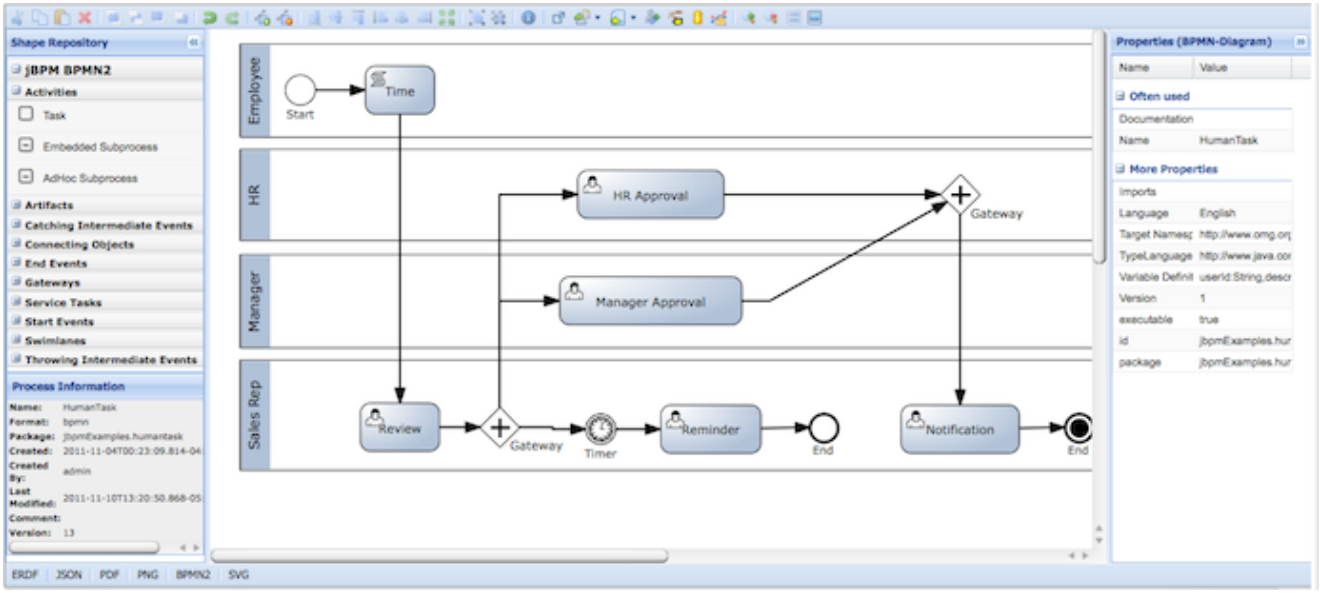


Figure 10.1.

Designer targets the following scenarios:

- View and/or edit existing BPMN2 processes: The designer allows you to open existing BPMN2 processes (for example created using the BPMN2 Eclipse editor or any other tooling that exports BPMN2 XML) in a web context.
- Create fully executable BPMN2 processes: A user can create a new BPMN2 process in the Designer and use the editing capabilities (drag and drop and filling in properties in the properties panel) to fill in the details. This for example allows business users to create complete business processes all inside a a browser. The integration with Drools Guvnor allows for your business processes as wells as other business assets such as business rules, process forms/images, etc. to be stored and versioned inside a content repository.

Designer supports all BPMN2 elements that are also supported by jBPM as well as all jBPM-specific BPMN2 extension elements and attributes.

10.1. Installation

If you are using the jBPM installer, this should automatically download and install the latest version of the designer for you. To manually install the designer, simply drop the designer war

into your application server deploy folder. Currently out-of-the-box designer deployments exist for JBoss 5.1.0 and JBoss AS7. Note: If you want to deploy on other (versions of an) application server, you might have to adjust the dependencies inside the war based on the default libraries provided by your application server. The latest version of the designer can be found *here* [http://sourceforge.net/projects/jbpm/files/designer/].

To start working with the designer, open Guvnor (e.g. *http://localhost:8080/drools-guvnor* [http://localhost:8080/drools-guvnor]) and either open an existing BPMN2 process or create a new one (under the "Knowledge Bases category on the left, select create new BPMN2 process"). This will open up the designer for the selected process in the center panel. You can use the palette on the left to drag and drop node types and the properties tab on the right to fill in the details (if either of these panels is not visible, click the arrow on the side of the editor to make them move forward).

The designer may also be opened stand-alone by using the following link: *http://localhost:8080/designer/editor?profile=jbpm&uuid=123456* (where 123456 should be replaced by the uuid of a process stored in Guvnor). Note that running designer in this way allows you to only view existing processes, and not save any edits nor create new ones. Information on how to integrate designer into your own applications can be found *here* [http://blog.athico.com/2011/04/using-oryx-designer-and-guvnor-in-your.html].

10.2. Source code

The designer source code is available for each release. You can find it *here* [http://sourceforge.net/projects/jbpm/files/designer/].

You can also browse and clone the project on *github* [https://github.com/tsurdilo/process-designer].

10.3. Designer UI Explained

The Designer UI is composed of a number of sections as shown in the screenshot below:

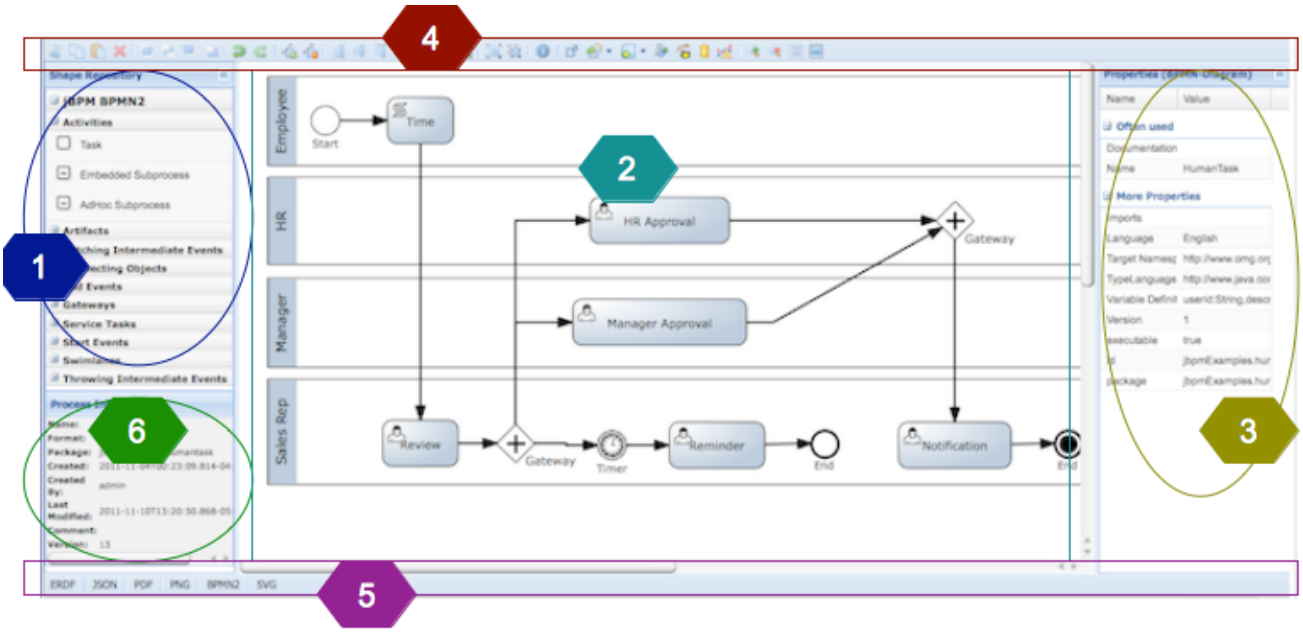


Figure 10.2.

- (1) Shape Repository Panel - the expandable section on the left shows the jBPM BPMN2 (default) shape repository. It includes all shapes of the jBPM BPMN2 stencil set which can be used to assemble your processes. If you expand each section sub-group you can see the BPMN2 elements that can be placed onto the Designer Canvas (2) by dragging and dropping the shape onto it.
- (2) Canvas - this is your process drawing board. After dropping different shapes onto the canvas, you can move them around, connect them, etc. Clicking on a shape on the canvas allows you to set its properties in the expandable Properties Window (3)
- (3) Properties Panel - this expandable section on the right allows you to set both process and shape properties. It is divided in two sections, namely "Often used", and "More Properties" section which is expandable. When clicking on a shape in the Canvas, this panel is reloaded to show properties specific to the shape type. If you click on the canvas itself (not on a shape) the section shows your general process properties.
- (4) Toolbar - the toolbar contains operations which can be performed on shapes present on the Canvas. Individual operations are disabled or enabled depending on what is selected. For example, if no shapes are selected, the Cut/Paste/Delete operations are disabled, and become enabled once you select a shape. Hovering over the icons in the Toolbar displays the description text of the operation.
- (5) Footer - the footer contains operations that allow users to view the source of the process being editor in the Canvas section in various formats such as BPMN2, PNG, JSON, etc.
- (6) Process Information - this section contains information about your process, such as its name, creation date, version, etc

Connecting shapes together in the canvas is realized with the Shape-Menu. The Shape-Menu is displayed by clicking on a shape:

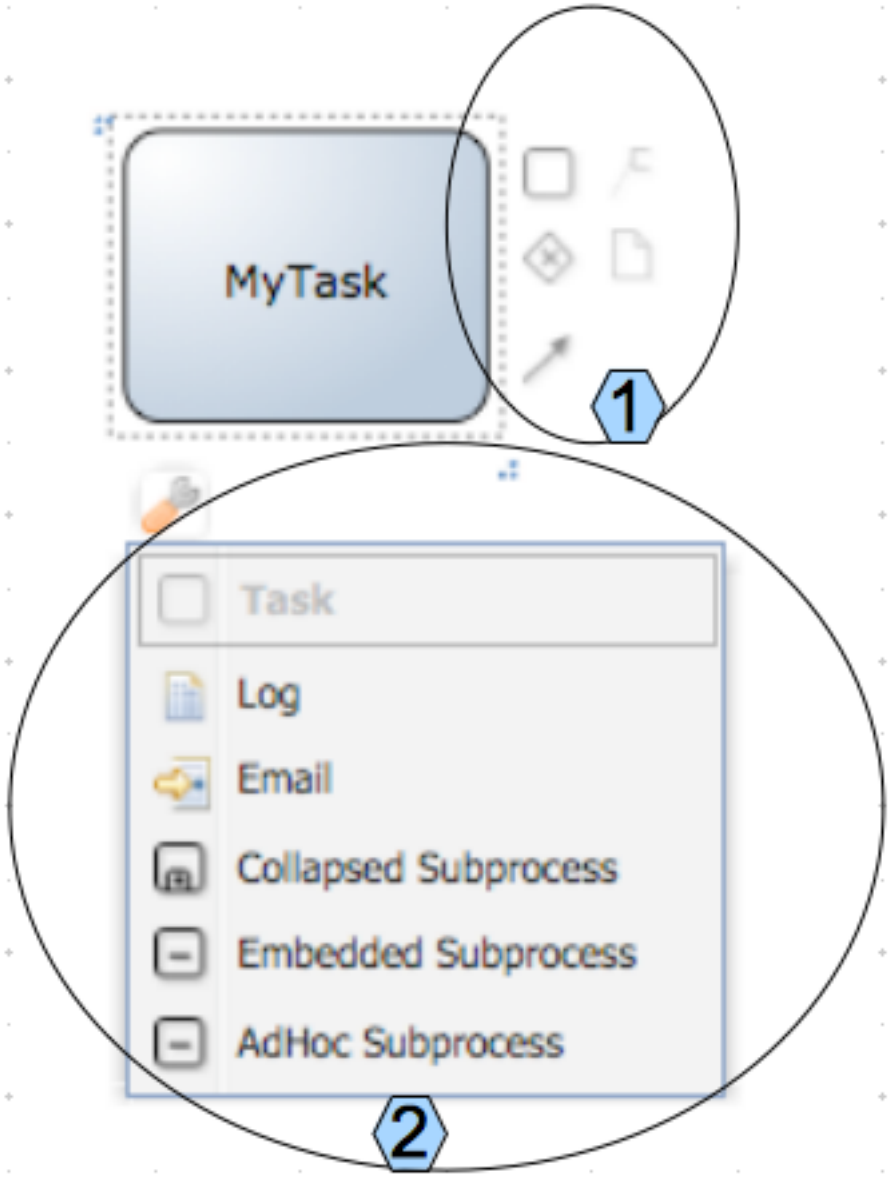


Figure 10.3.

The Shape-Menu is composed of two sections:

- (1) Connection section: allows you to easily connect your shape with a new one. The shapes displayed in this section are based on connection rules of the BPMN2 specification.
- (2) Morphing section: allows you to easily morphe a base shape into any other that extend this base shape.

Following sequence of picture shows how easy it is to quickly create and connect multiple shapes in the canvas:

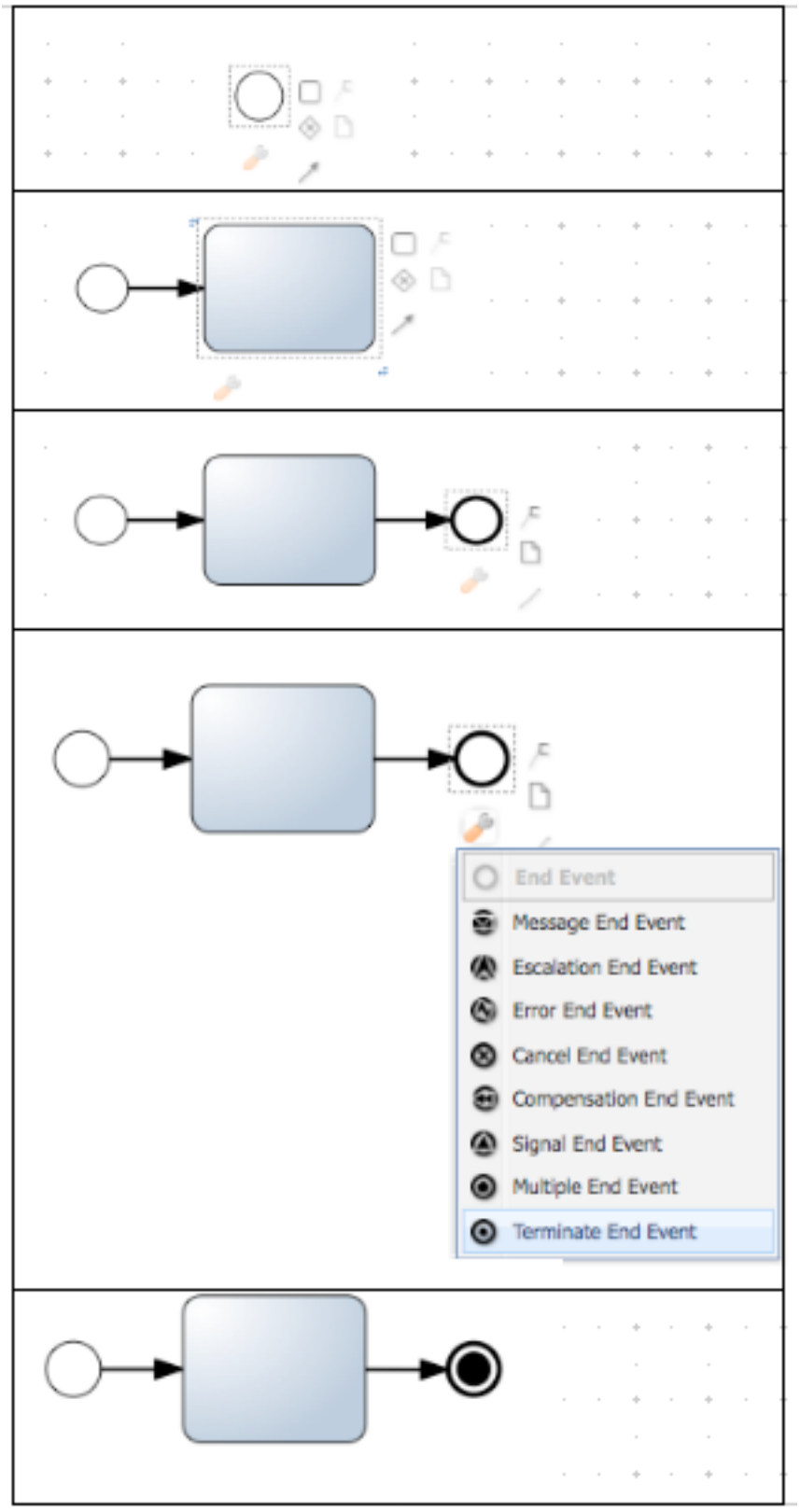

End Event
Message End Event
Escalation End Event
Error End Event
Cancel End Event
Compensation End Event
Signal End Event
Multiple End Event
Terminate End Event

Figure 10.4.

You can also name your shapes by double-clicking on the shape in the canvas. This sets the name attribute of the particular shape:

Figure 10.5.

10.4. Support for domain-specific service nodes

Designer has full support for jBPM domain-specific service nodes. To include your service nodes in the Designer jBPM BPMN2 stencil set, you can either upload your existing service node definitions into Guvnor, or use the the new service node configuration editor which we added to Guvnor to create new configurations.

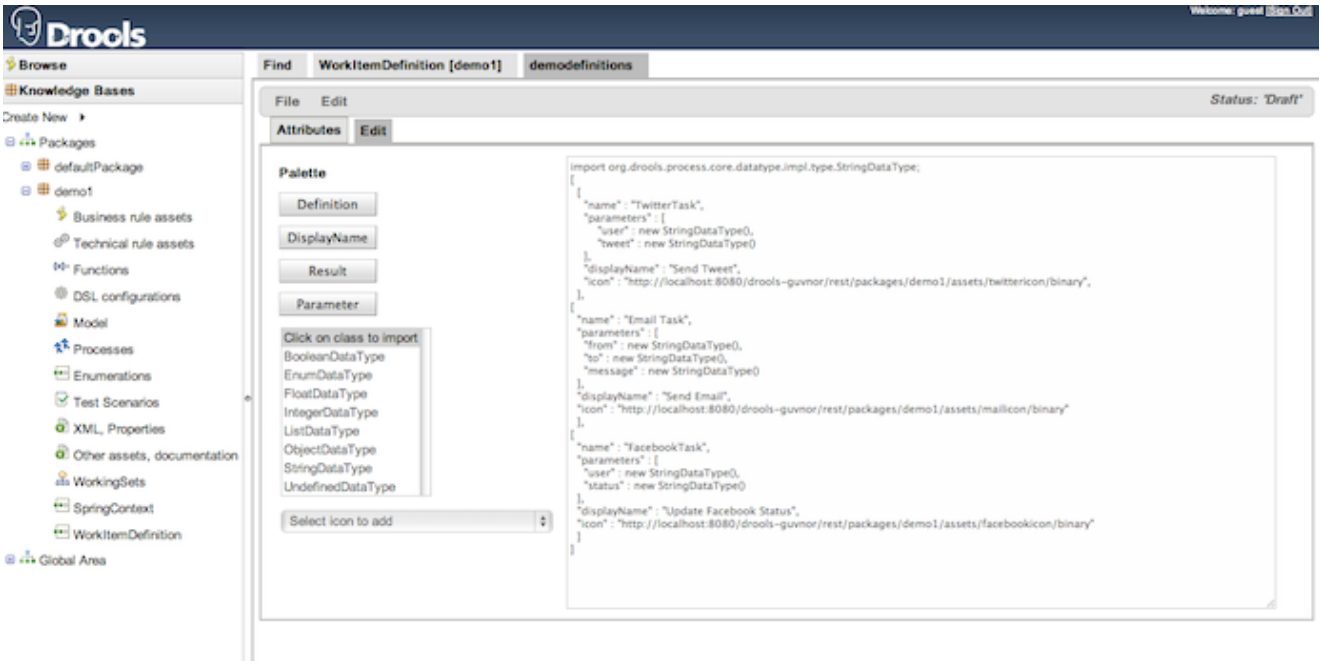


Figure 10.6.

Once you have some service node configurations present, you can see them being included in Designer stencil set by re-opening an existing or creating a new process. Your service nodes will be now available under the "Service Task" section of the jBPM BPMN2 stencil set.

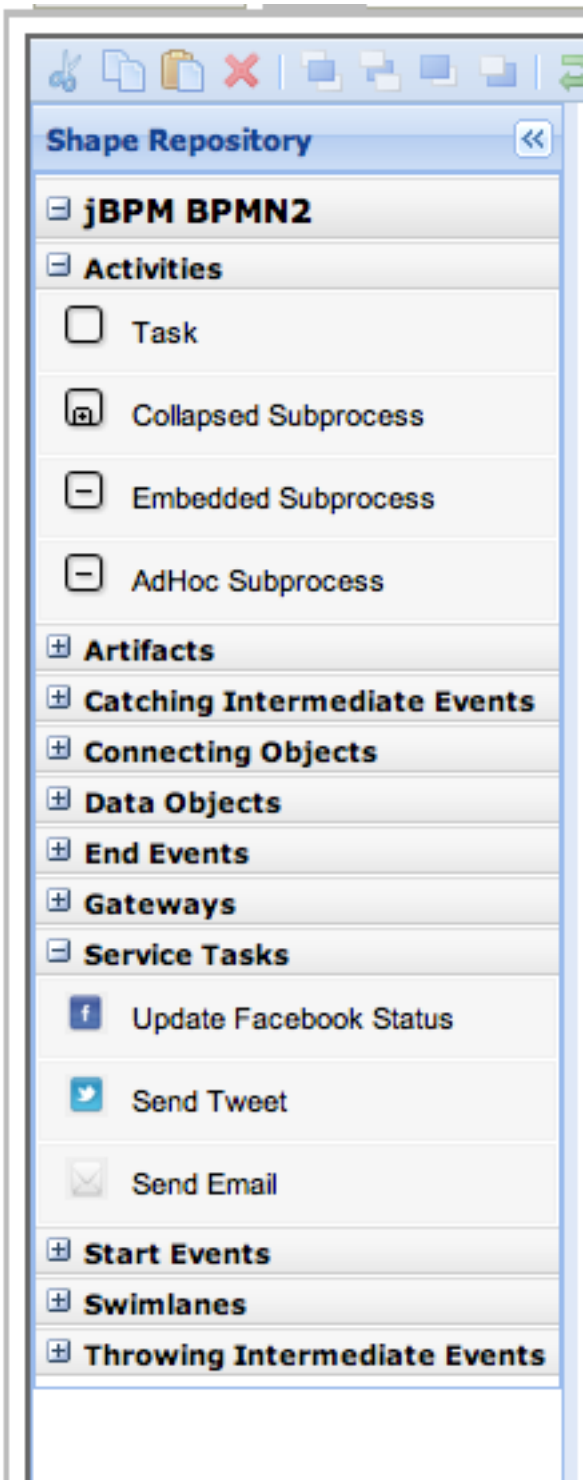


Figure 10.7.

Service nodes are fully usable within your processes. Please note that the service node configurations are package-specific in Guvnor. If you want to re-use your service nodes across multiple Guvnor packages, you have to copy their configurations to each indidual package you would like to use them in.

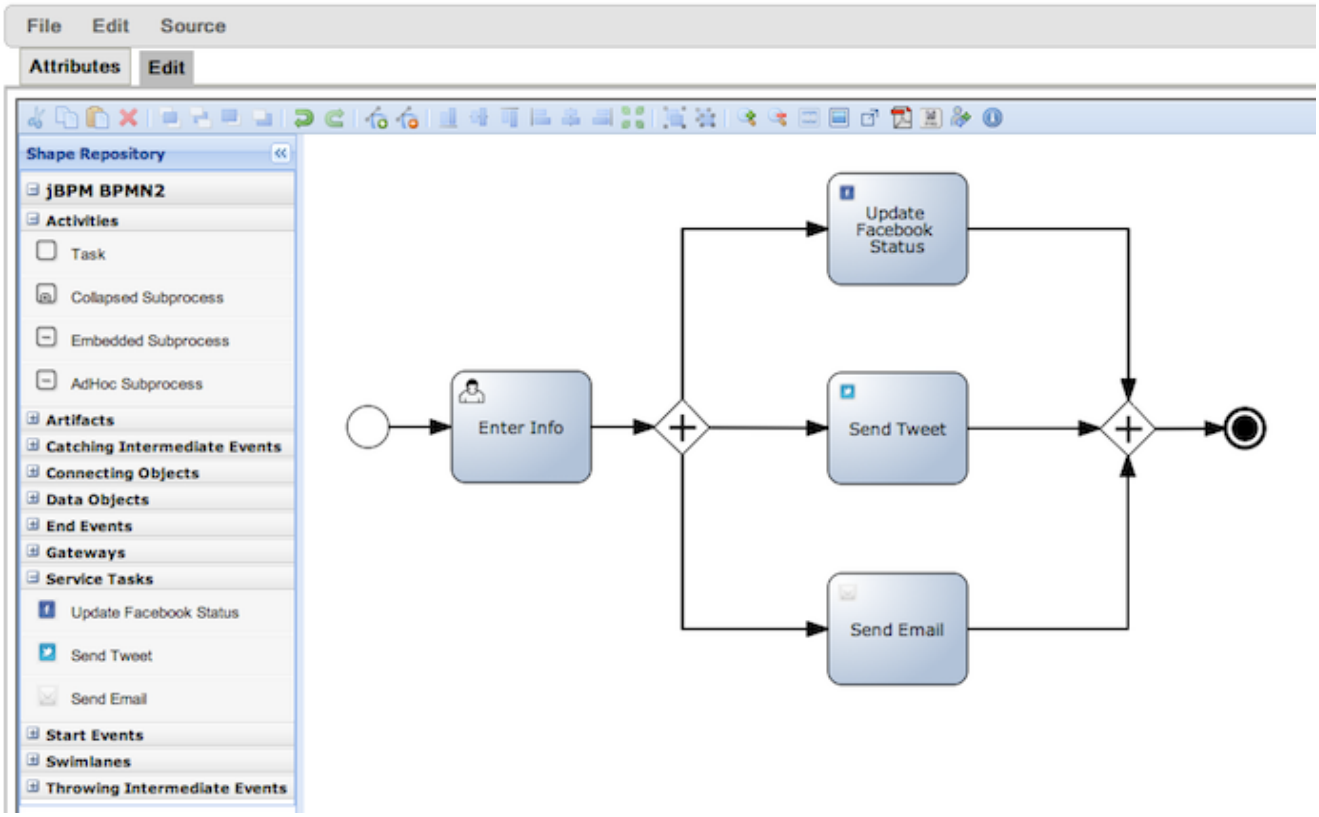


Figure 10.8.

For more information on this feature please view *this* [http://vimeo.com/26126678], and *this* [http://vimeo.com/24288229] video.

10.5. Configuring Designer

Designer is tightly integrated with Guvnor. By default Designer expects to find a Guvnor instance on http://localhost:8080/drools-guvnor/. Guvnor, by default, expects to find the Designer on http://localhost:8080/designer. Here we show how to configure both Designer and Guvnor to be able to change these default settings when needed.

10.5.1. Changing the default configuration in Designer

In cases where Guvnor is configured to use https, or is running on a different host/port/domain/subdomain you have to configure Designer to reflect these settings. in order to change Designer configurations you have to deploy it as an exploded war. In $designer.war/profiles/jbpm.xml notice the section on the bottom:

```
      <externalloadurl protocol="http" host="localhost:8080" subdomain="drools-guvnor/org.drools.guvnor.Guvnor/oryxeditor" usr="admin" pwd="admin"/>
```

The configuration attributes include:

- protocol: the protocol to use (http/https)
- host: includes both the host and the port that Guvnor is running on
- subdomain: in some situations Guvnor subdomain is not drools-guvnor. You should leave the path to the servlet as-is.
- usr: if you have set up JAAS authentication in Guvnor, provide a Guvnor user name here. Note that this user should have admin privileges in Guvnor
- pwd: password for the Guvnor user

Alernative you can specify these configrations via system properties:

- oryx.external.protocol
- oryx.external.host
- oryx.external.usr
- oryx.external.pwd

If you choose to use system properties you do not have to deploy the designer war as exploded.

10.5.2. Changing the default configuration in Guvnor

To configure Guvnor to reflect the host/port/domain/subdomain and the default profile settings of the Designer, we need to edit $drools-guvnor.war/WEB-INF/preferences.properties:

```
#Designer configuration
designer.url=http://localhost:8080
#Do not change this unless you know what are you doing
designer.context=designer
designer.profile=jbpm
```

The configuration attributes include:

- designer.url: set the protocol, host, and port where Designer is located at
- designer.context: this sets the configured subdomain of Designer. Should not change unless you deploy it under some other subdomain
- designer.profile: Designer can have multiple profiles defined. Profiles determine the used stencil set, the saving/loading strategy of processes, etc. The default profile name used is "jbpm" and this should not be changed unless you create a custom profile to be used

Note that in order to be able to edit $drools-guvnor.war/WEB-INF/preferences.properties, you have to deploy Guvnor as an exploded archive.

10.6. Generation of process and task forms

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

Figure 10.9.

Designer will iterate through your process BPMN2 and create forms for your process, and each of the human tasks in your process. It uses the defined process variables and human task data input/output parameters and associations to create form fields. The generated forms are stored in Guvnor, and the user is presented with a page which shows each of the forms created as well as a link to their sources in Guvnor:

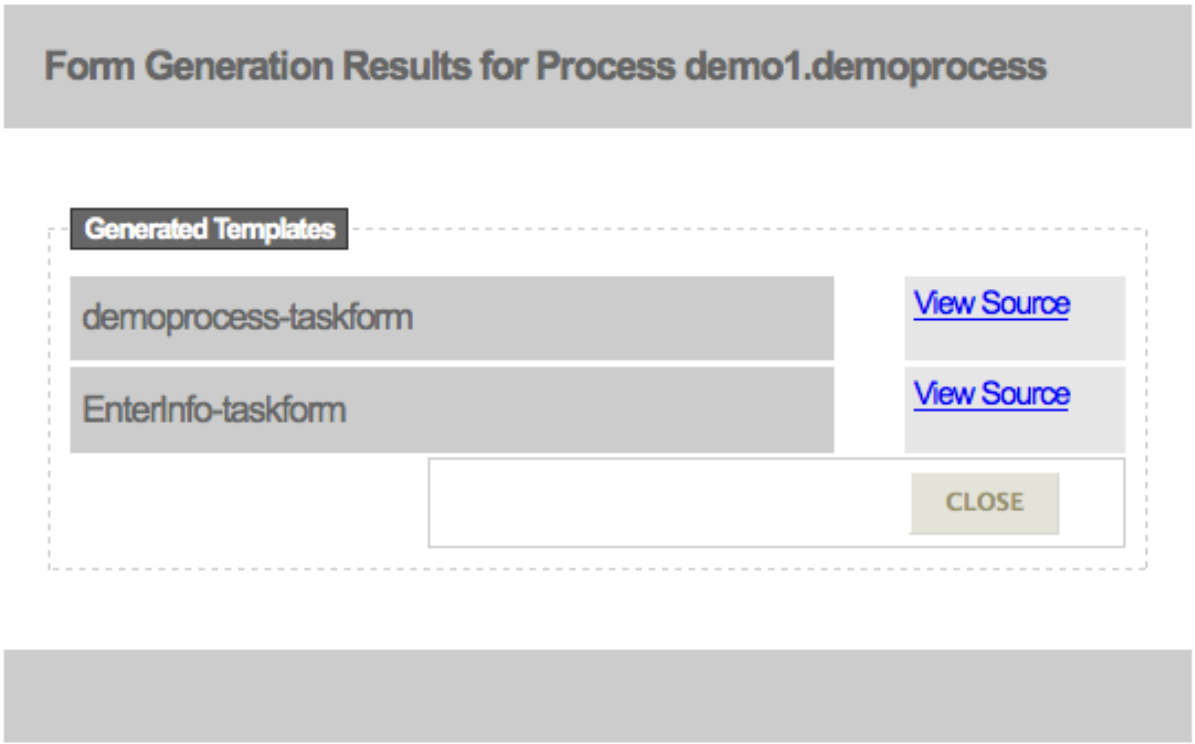


Figure 10.10.

As mentioned, all forms are fully usable inside jBPM console. In addition each form includes basic JavaScript form validation which is determined based on the type of the process variables, and/

or human task data input/output association definitions. Here is an example generated human task form.

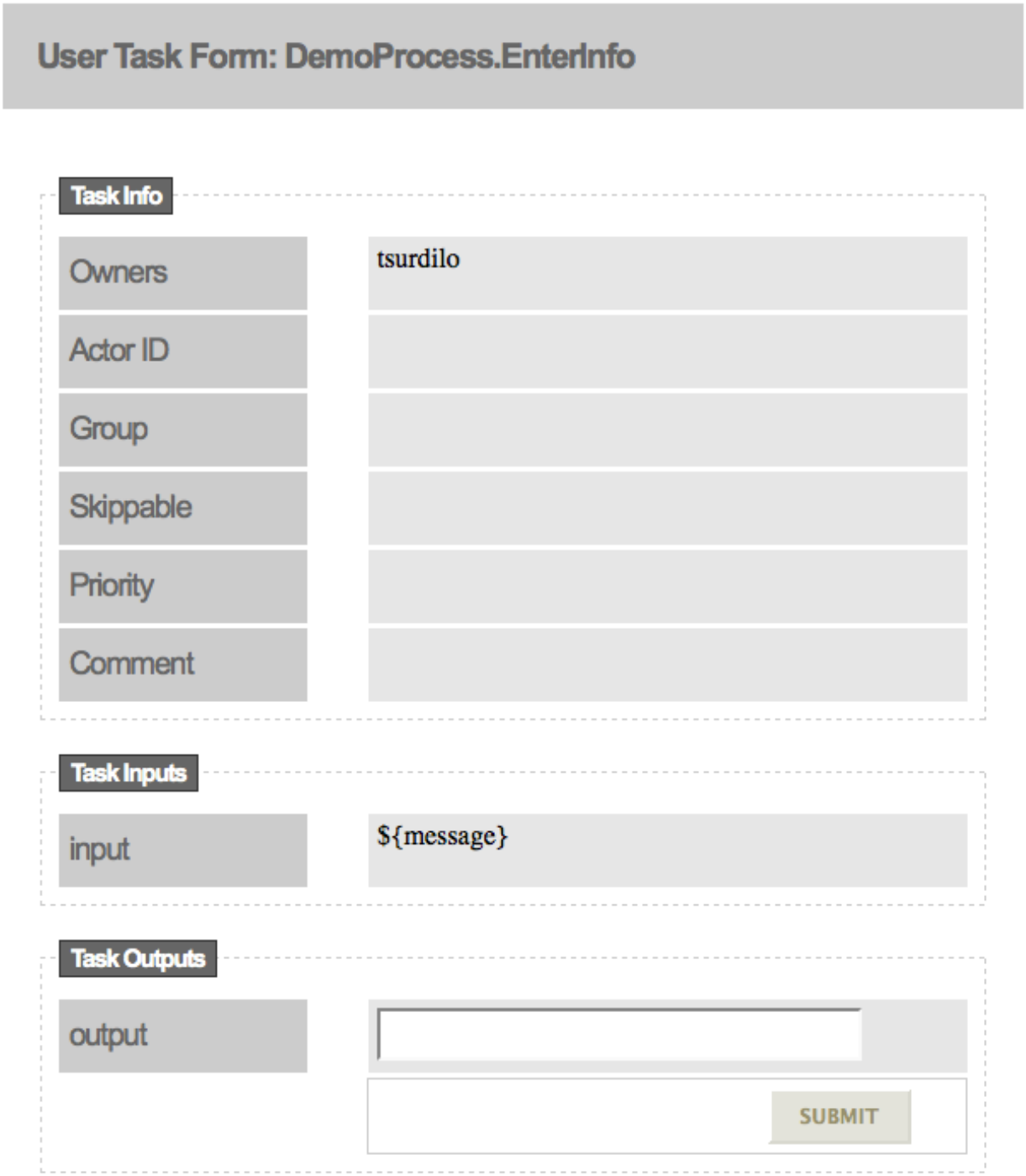


Figure 10.11.

In order for process and task forms to be generated you have to make sure that your process has its id parameter set, as well that each of your human tasks have the TaskName parameter set. Task forms contain pure HTML, CSS, and JavaScript, so they are easily editable in any HTML

editor. Please note that there is no edit feature available currently in Designer, so each time you generate forms, existing ones will be overwritten.

For more information on this feature please view *this* [http://vimeo.com/26126678] video.

10.7. View processes as PDF and PNG

Any process created in Designer can be easily viewed in PDF and PNG formats. In the Designer footer section locate the "Convert to PDF" and "Convert to PNG" buttons. Both PDF and PNG formats are also stored in Guvnor, making it easily accessible.

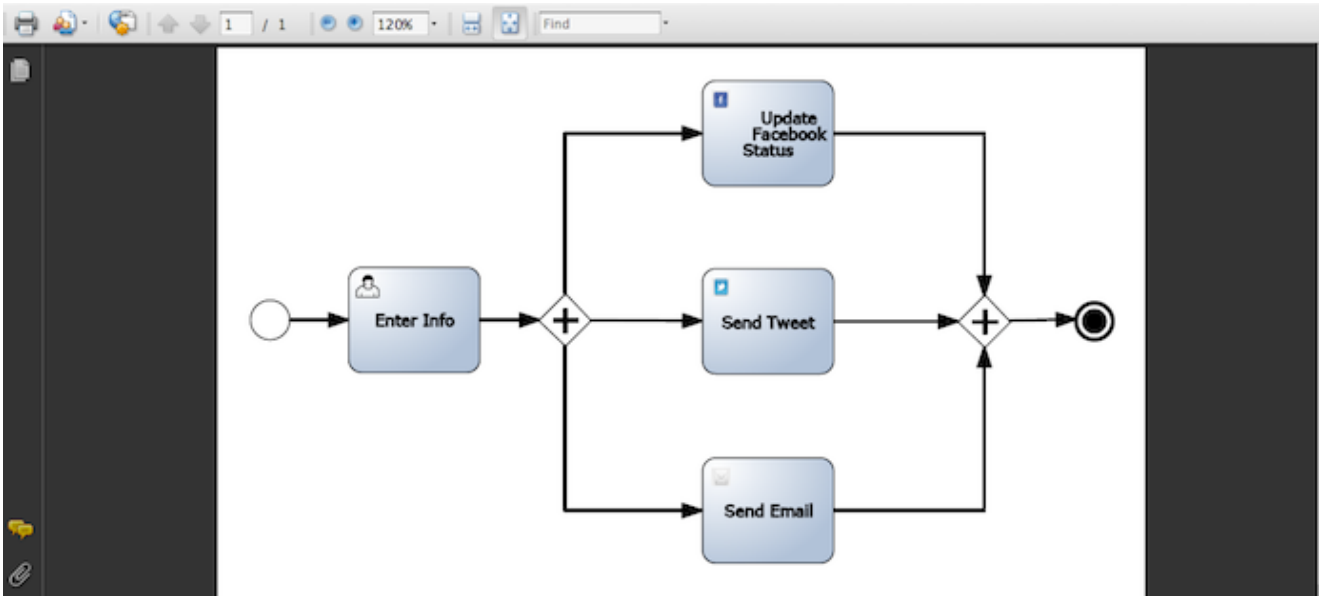


Figure 10.12.

The footer section also includes buttons to view the process sources in ERDF, JSON, SVG, and BPMN2 formats.

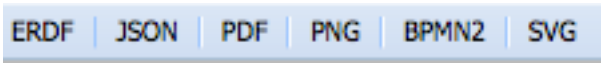


Figure 10.13.

10.8. Viewing process BPMN2 source

At any time you can view your process's BPMN2 source by selecting the Source->View Source link in the Guvnor toolbar above the designer frame. The source generated by designer is fully BPMN2 compliant and can be used in any BPMN2 compliant editor.

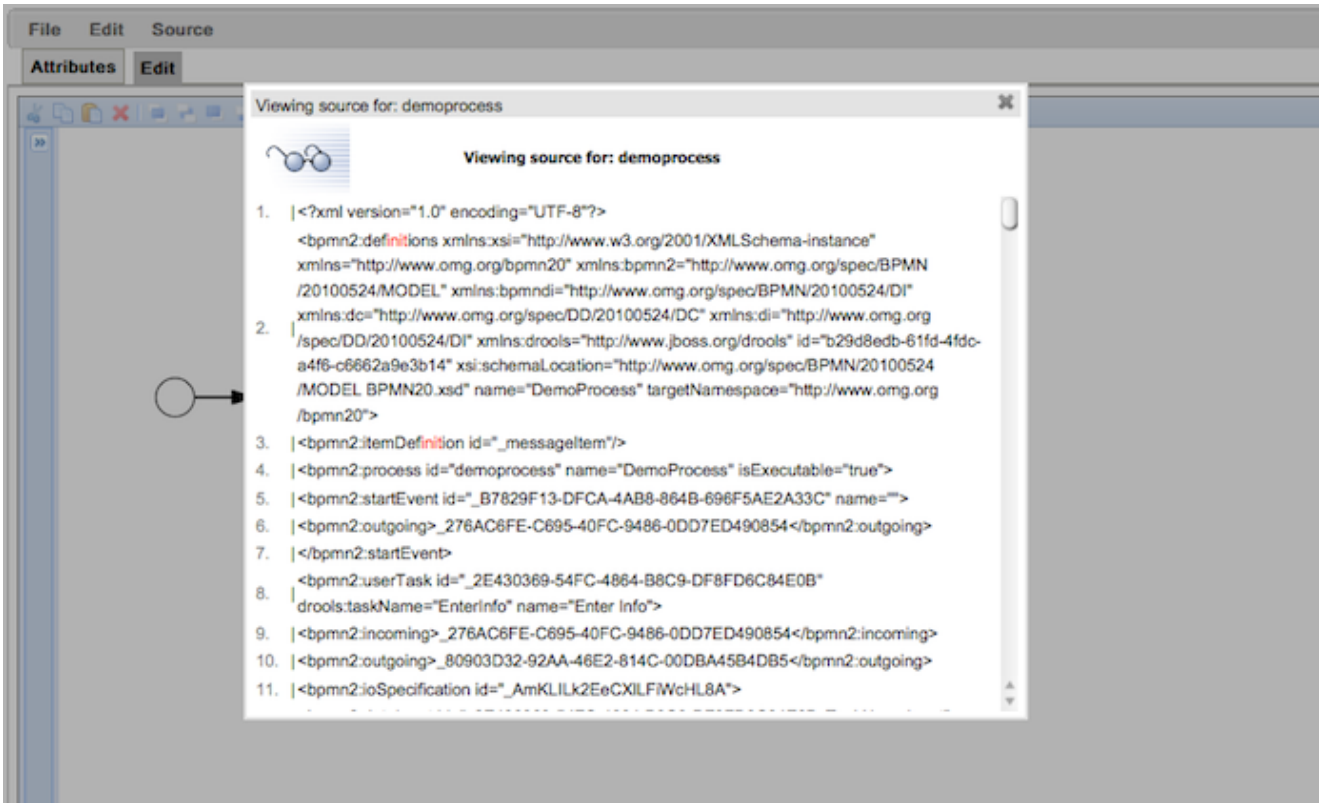


Figure 10.14.

Same can be done by clicking on the BPMN2 button in the footer section of the designer:

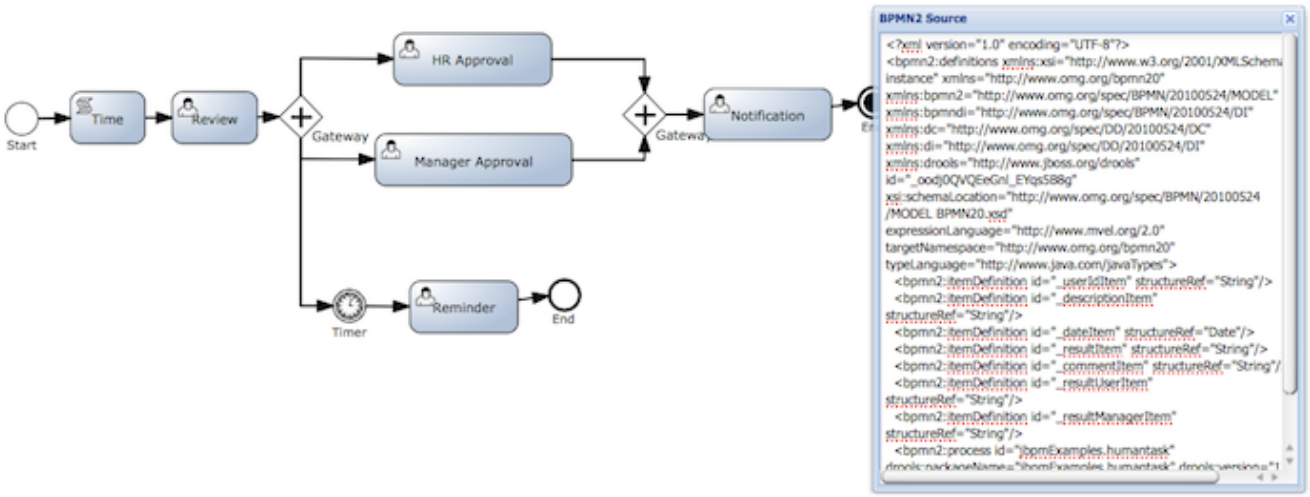


Figure 10.15.

10.9. Embedding designer in your own application

It is possible to embed the designer in your own application and still be able to utilize Guvnor as the asset repository for all of your process assets. For more information on this feature please view *this video* [http://vimeo.com/22033817].

10.10. Migrating existing jBPM 3.2 based processes to BPMN2

To migrate your existing jBPM 3.2 based processes to BPMN2 locate the migration button in the toolbar section of the designer:

Figure 10.16.

The feature allows users to select the location of their processdefinition file, and the location of its gpd.xml file. Designer then uses the *jbpmmigration tool* [https://github.com/droolsjbpm/jbpmmigration] to convert the jBPM 3.2 based processes to BPMN2 and displays it onto the designer canvas:

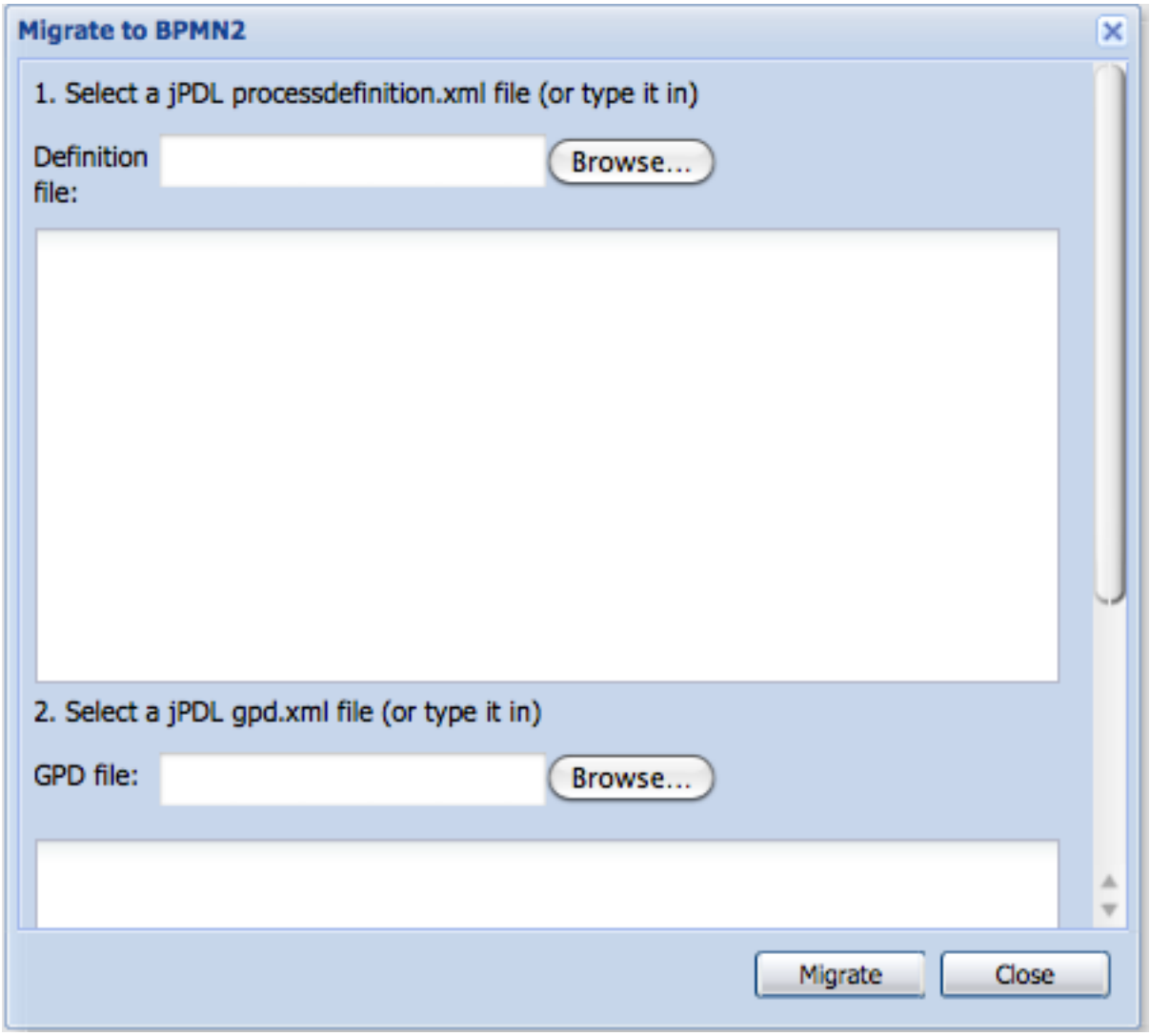


Figure 10.17.

For more information on this feature please view *this* [http://vimeo.com/30857949] video.

10.11. Visual Process Validation

To run process validation against the process you are developing in the designer, locate the validation button in the designer toolbar section:

Figure 10.18.

In case of validation errors, designer presents a red "X" mark next to process nodes that contain them. Mouse-over this red "X" presents a tooltip with the descriptions of validation errors. Note that since the process node is not visually displayed, designer will merge all process-node-specific validation errors with those of the very first node of the BPMN2 process. Following is a screenshot of the visual process validation feature in use:

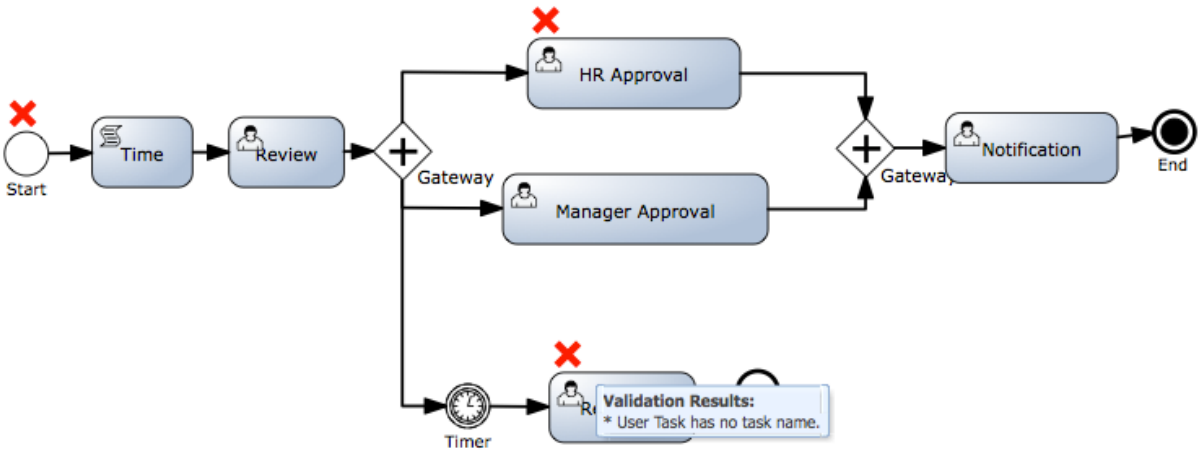


Figure 10.19.

For more information on this feature please view *this* [http://vimeo.com/30857949] video.

10.12. Integration with the jBPM Service Repository

Designer integrates with the *jBPM Service Repository and allows users to install and use assets from the repository.* [http://kverlaen.blogspot.com/2011/10/introducing-service-repository.html]. To connect to the Service Repository from designer, click on the service repository button in the designer toolbar:

Figure 10.20.

Designer will present you with all assets located in the jBPM service repository in table format. Colums of this table show information about the specific asset in the repo. To install the item to your local Guvnor package, simply double-click on the item row. You will have to save and re-open your process in order to be able to start using the installed items.

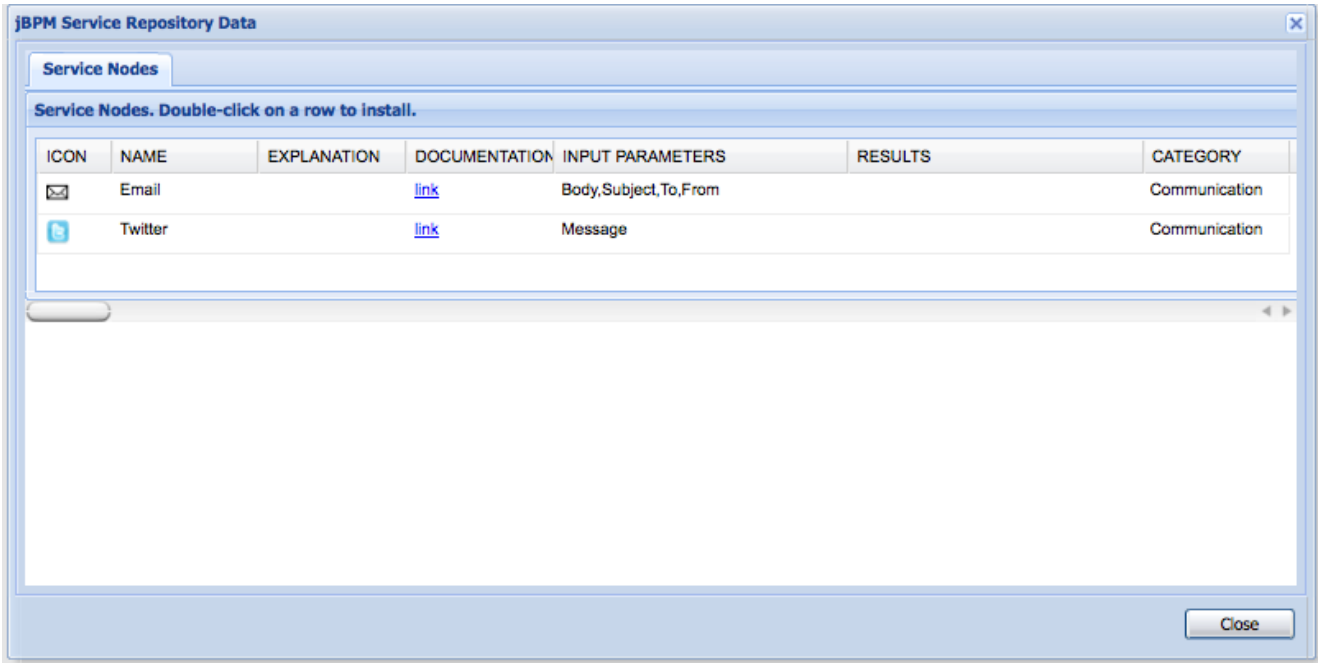


Figure 10.21.

For more information on this feature please view *this* [http://vimeo.com/30857949] video.

10.13. Generating code to share the process image, PDF, and embedded process editor

It is important to be able to share your process with users who do not have access to your running designer instance. For these cases designer allows code generation of "sharable" image, PDF and embedded editor code of your processes. To use this feature locate the following dropdown in the designer toolbar section:

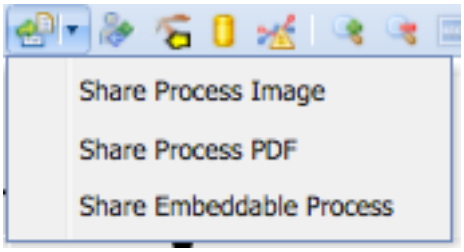


Figure 10.22.

10.14. Importing existing BPMN2 processes

You can easily import your existing BPMN2 processes into the designer by locating and clicking on the following dropdown selection list in the toolbar section:

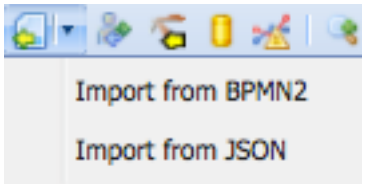


Figure 10.23.

You will be able to either select an existing file on your filesystem or paste existing BPMN2 XML. The designer canvas will automatically import and display your process without a page refresh.

10.15. Viewing Process Information

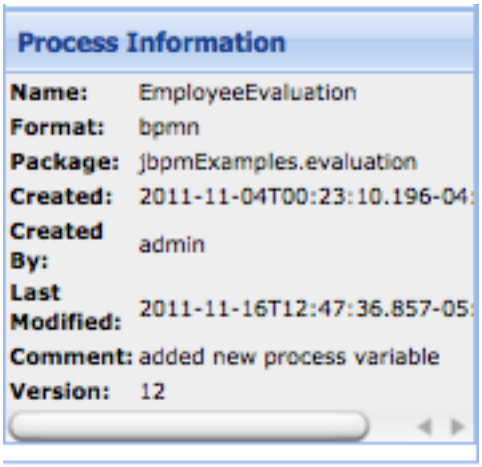


Figure 10.24.

The Process Information section displays important information about your process. These include the process:

- name

- format
- Guvnor package name the process belongs to
- creation date
- name of user that created the process
- last modification date
- last check-in comment
- version number

10.16. Requirements

Java:

- Java 6

Browsers:

- Mozilla Firefox (including 6)
- Google Chrome

JBoss AS:

- Designer war is currently compatible with JBoss AS 4.x, 5.1, and 7

Chapter 11. Console

Business processes can be managed through a web console. This includes features like managing your process instances (starting/stopping/inspecting), inspecting your (human) task list and executing those tasks, and generating reports.

The jBPM console consists of two wars that must be deployed in your application server and contains the necessary libraries, the actual application, etc. One jar contains the server application, the other one the client.

11.1. Installation

The easiest way to get started with the console is probably to use the installer. This will download, install and configure all the necessary components to get the console running, including an in-memory database, a human task service, etc. Check out the chapter on the installer for more information.

The console is a separate sub-project that is shared across different projects, like for example jBPM and RiftSaw. The source code of the version that jBPM5 is currently using can be found on SVN *here* [http://anonsvn.jboss.org/repos/soag/bpm-console/tags/bpm-console-2.1/]. The latest version of the console has been moved to Git and can be found *here* [https://github.com/bpmc].

11.1.1. Authorization

The console requires users to log in before being to use the application. The console uses normal username / password authentication. When using JBossAS for example, this can be specified in the users.properties file in the server/{profile}/conf folder. There you can specify the combination of users that can log into the console and their password.

When using the jBPM installer, a predefined users.properties file (located in the auth folder) is copied to the jbossas/server/default/conf folder automatically. This file can be edited and contains a few predefined users: admin, krisv, john, mary, and sales-rep (as these are commonly used in examples). The password associated with these users is the same as their username.

11.1.2. User and group management

The human task service requires you to define which groups a user is part of, so that he can then claim the tasks that are assigned to one of the groups he is part of. The console uses username / group association for that. When using JBossAS for example, this can be specified in the roles.properties file in the server/{profile}/conf folder. There you can specify the combination of users and the groups they are part of.

When using the jBPM installer, a predefined roles.properties file (located in the auth folder) is copied to the jbossas/server/default/conf folder automatically. This file can be edited and contains the groups the predefined users are part of (as these are commonly used in examples): all

users are part of the admin, manager and user group but john is also part of the PM (project management) group, mary is part of HR (human resources) and sales-rep is part of sales.

11.1.3. Registering your own service handlers

As explained in the chapter on domain-specific services, jBPM allows you to register your own domain-specific services as custom service tasks. The process only contains a high-level description of the service that needs to be executed, and a handler is responsible for the actual implementation, i.e. invoking the service.

You must register your handlers to be able to execute domain-specific services. You can register your handlers by dropping a configuration file in the classpath that specifies the implementation class for each of the handlers. You can specify which configuration files must be loaded in the drools.session.conf file, using the drools.workItemHandlers property (as a list of space-separated file names). These file names should contain a Map of entries, the name and the corresponding WorkItemHandler instance that should be used to execute the service. The configuration file is using the MVEL script language to specify a map of type Map<String,WorkItemHandler>.

You should also make sure that the implementation classes (and dependencies) are also available on the classpath of the server war, for example by dropping the necessary wars in the server/{profile}/lib directory of your JBossAS installation.

For example, suggest you want to use the "Email" service task (that is provided out-of-the-box as an example in the jbpm-workitems module). You should put the jbpm-workitems, javax.mail and javax.activation jars in the lib folder of the AS and the include the following two configuration files in the META-INF folder in the WEB-INF/classes folder of the server war. The drools.session.conf simply refers to the CustomWorkItemHandlers.conf file that contains the actual handlers:

```
drools.workItemHandlers = CustomWorkItemHandlers.conf
```

This configuration file then specifies which handler to register for each of the domain-specific services that are being used, using MVEL to specify a Map<String,WorkItemHandler> (with host, port, username and password replaced by a meaningful value of course):

```
[
  "Email": new org.jbpm.process.workitem.email.EmailWorkItemHandler(
             "host", "port", "username", "password"),
]
```

The installer simplifies registering your own work item handlers significantly by offering these configuration files in the jbpm-installer/conf folder already and automatically copying them to the right location when installing the demo. Simply update these files with your own entries before running ant install.demo.

11.1.4. Configure management console

Management console can be configured to suit deployment needs of the environment. Its main configuration is done via property file - default.jbpm.console.properties, which can be found in jbpm-gwt-console-server.war/WEB-INF/classes. This configuration is sample setup for default installation if there is a need to configure it differently a custom file should be provided: jbpm.console.properties that can be placed on any directory of the file system where console will have access to. Console by default will look for it inside JBoss AS configuration directory that is given as jboss.server.config.dir system property. If jBPM console is deployed to other servers or default location is not acceptable custom location can be provided as jbpm.conf.dir system property. It allows administrators to configure following aspects of management console:

- task server connectivity
- Guvnor connectivity
- console host and port numer
- console resource directory (for local process, rules, etc repository)

Each of mentioned aspects can have one or more attributes that drive their behavior, following is a complete list of supported properties for every aspect.

Management console configuration

- jbpm.console.server.host : host/ip address used to bind management console (default localhost)
- jbpm.console.server.port : port used to bind management console (default 8080)
- jbpm.console.server.context : context root that is used to bind console server web application (default gwt-console-server)
- jbpm.console.directory : local directory used as process/rules repository

Task server connectivity

- jbpm.console.task.service.strategy : transport used to connect to task server (default HornetQ and accepts Mina|HornetQ|JMS)
- jbpm.console.task.service.host : host where Task Server is deployed (default localhost) applies to all transports
- jbpm.console.task.service.port : port where Task Server is deployed (default 5153) applies to all transports

- JMSTaskClient.connectionFactory : JNDI name of connection factory only for JMS (no default)
- JMSTaskClient.acknowledgeMode : acknowledgment mode only for JMS (no default)
- JMSTaskClient.transactedQueue : transacted queue name only for JMS (no default)
- JMSTaskClient.queueName : queue name only for JMS (no default)
- JMSTaskClient.responseQueueName : response queue name only for JMS (no default)

Guvnor connectivity

- guvnor.protocol : protocol to access Guvnor (default http)
- guvnor.host : host and port number where Guvnor is deployed (default localhost:8080)
- guvnor.subdomain : subdomain/context root of Guvnor (default drools-guvnor)
- guvnor.usr : user id to authenticate in Guvnor (default admin)
- guvnor.pwd : password to authenticate in Guvnor (default admin)
- guvnor.packages : comma separated list of packages to load from Guvnor
- guvnor.connect.timeout : connect timeout (default 10000)
- guvnor.read.timeout : read timeout (default 10000)
- guvnor.snapshot.name : configure package snapshot name (default LATEST)

Once the overall configuration is done, next step is to be able to control runtime behavior of management console that consists of:

- knowledge base setup
- stateful session setup

These runtime components are configured via dedicated managers that are extensible and can be configured with system properties, note that configuration of managers is optional and required only if default managers are not suitable for particular environment

- knowledge base manager: -Djbpm.knowledgebase.manager=com.company.CustomKnowledgeBaseManager
- stateful session manager: -Djbpm.session.manager=com.company.CustomSessionManager

Be default knowledge base manager will build knowledge base according to configuration given in jbpm.console.properties (or default.jbpm.console.properties) file and stateful session will be build based on session template, that is MVEL file named session.template (default.session.template

that is bundled in jBPM console). session.teplate file, same as jbpm.console.properties is an extension point to configure jBPM console without changing its internal files and can be placed on any directory on the file system. Session template is intended to provide following configuration for stateful session:

- businessKey - a unique key that will be used to get session from JNDI
- persistenceUnit - name of the persistence unit to be used
- properties - list of key value pairs of session configuration
- workItemHandlers - list of key value pairs (work item name: class name of work item handler)
- eventListeners - list of event listener classes to be registered on the session
- environmentEntries - list of key value pairs of environment entires to be put before session is created
- imported - true|false if set to true session will be looked up from JNDI using business key instead of creating new one - it means that session should be build by another application and console will use it as well

session template is dedicated to default session manager implementation and can be substituted with anyother mechanism together with custom implementation of SessionManager interface. See next section about custom managers.

```
new SessionTemplate().{
    businessKey = "jbpmConsole",
    imported = false,
    persistenceUnit = "org.jbpm.persistence.jpa",

    properties = ["drools.processInstanceManagerFactory":"org.jbpm.persistence.processinstance.
                  "drools.processSignalManagerFactory" : "org.jbpm.persistence.processinstance.
                    ],

                    workItemHandlers     =     ["Human     Task"     :     "new

\", taskClient, ksession, org.jbpm.task.utils.OnErrorAction.LOG)",
                                                        "Service   Task"   :   "new
 org.jbpm.process.workitem.bpmn2.ServiceTaskHandler(ksession)"],

                                          eventListeners          =          ["new
 org.jbpm.process.audit.JPAWorkingMemoryDbLogger(ksession)",
                                                                              "new
 org.jbpm.integration.console.listeners.TriggerRulesEventListener(ksession)"  ]
};
```

Default session template is present above and configures most important elements of the environment. As you can see there are option to refer to some already existing object when registering work item handlers and event listeners:

- ksession - session instance that is being built
- taskClient - talk client that is configured based on settings given in jbpm.console.properties (default.jbpm.console.properties)

Note

Important to note is that if someone provides custom implementation of work item handler for Human Task, keep in mind that it is important to connect handler as soon as session is created to be able to receive task events and move process forward. Default manager invokes three methods on human task handler:

- setIpAddress with single String argument
- setPort with single int argument
- connect no arguments

so ensure you have them to be properly initialized

To sum up, jbpm console comes with two files inside its server component (gwt-console-server.war), these are default.jbpm.console.properties and default.session.template. These two files should not be modified but in case a change to configuration is required they should be copied and renamed to jbpm.console.properties and session.template respectively. Location of these custom files can be decided by administrator but recommended for JBoss AS is to put them into jboss configuration directory (jboss_home/standalone/configuration for AS 7). If custom location is used it must be provided as system property -Djbpm.conf.dir. Any changes applied to custom configuration will be preserved between jbpm upgrades as they do not reside inside jbpm applications.

11.1.4.1. Implementing custom managers

To implement custom managers that are responsible for building knowledge base and session certain requirements must be met: Knolwedge Base Manager

- Custom class must implement org.jbpm.integration.console.kbase.KnowledgeBaseManager
- it must be configured with -Djbpm.knowledgebase.manager=[classname]

Session Manager

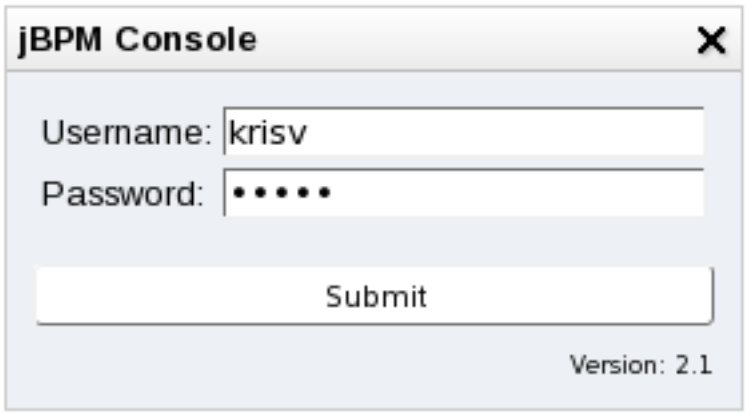
- Custom class must implement org.jbpm.integration.console.session.SessionManager

- Custom class must provide constructor that accepts KnowledgeBase argument
- it must be configured with -Djbpm.session.manager=[classname]

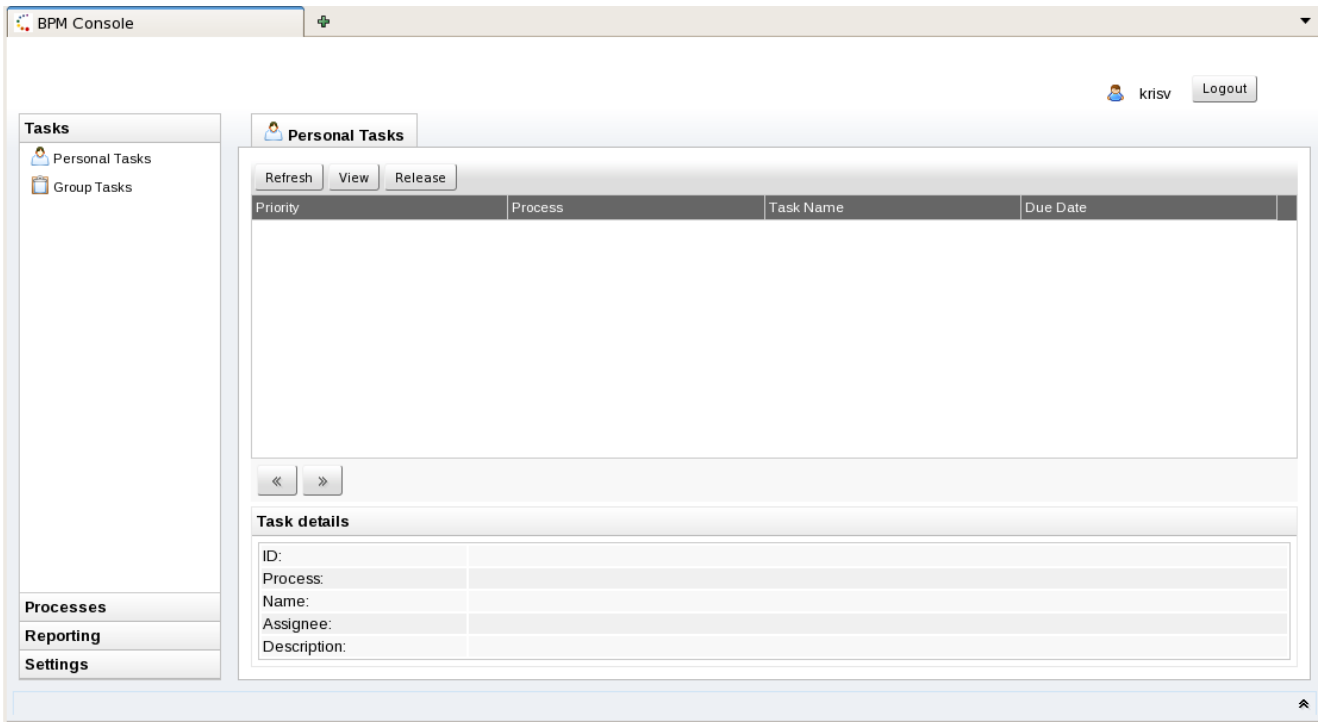
11.2. Running the process management console

Now navigate to the following URL (replace the host and/or port depending on how the application server is configured): *http://localhost:8080/jbpm-console*

A login screen should pop up, asking for your user name and password. By default, the following username/password configurations are supported: krisv/krisv, admin/admin, john/john and mary/mary.



After filling these in, the process management workbench should be opened, as shown in the screenshot below. On the right you will see several tabs, related to process instance management, human task lists and reporting, as explained in the following sections.

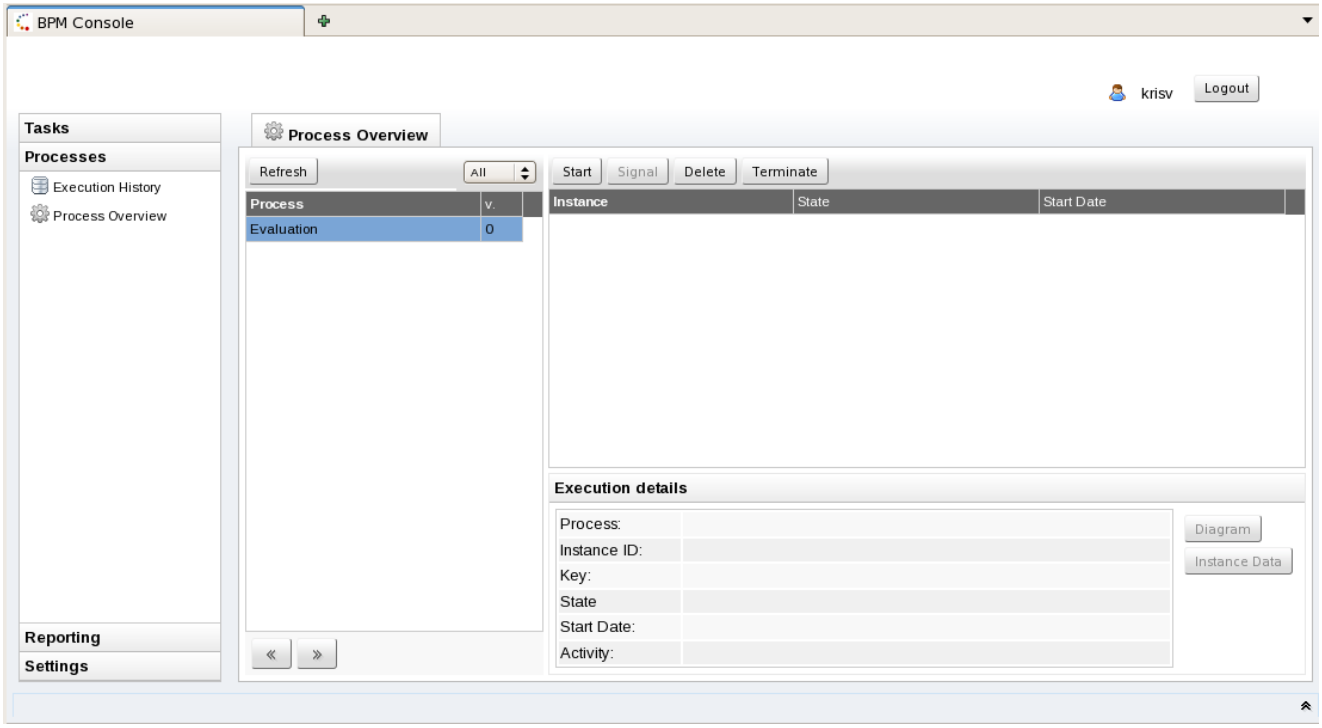


11.2.1. Managing process instances

The "Processes" section allows you to inspect the process definitions that are currently part of the installed knowledge base, start new process instances and manage running process instances (which includes inspecting their state and data).

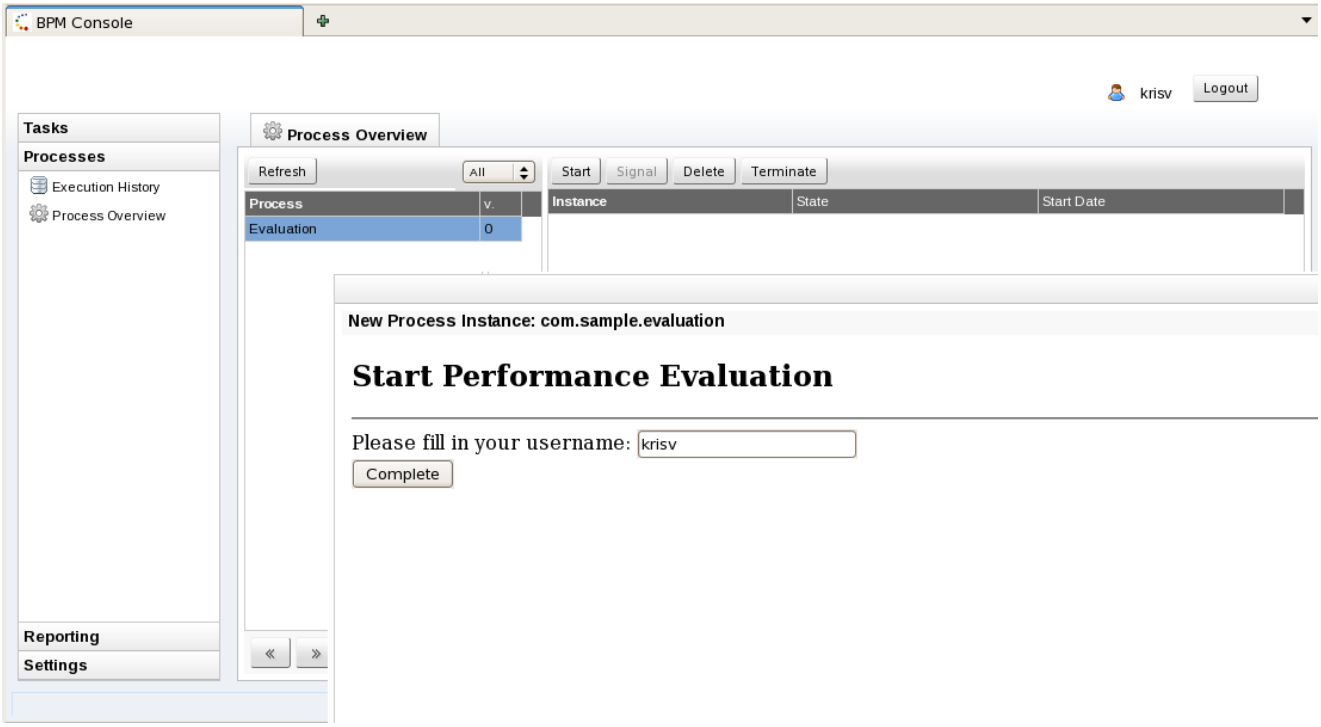
11.2.1.1. Inspecting process definitions

When you open the process definition list, all known process definitions are shown. You can then either inspect process instances for one specific process or start a new process instance.



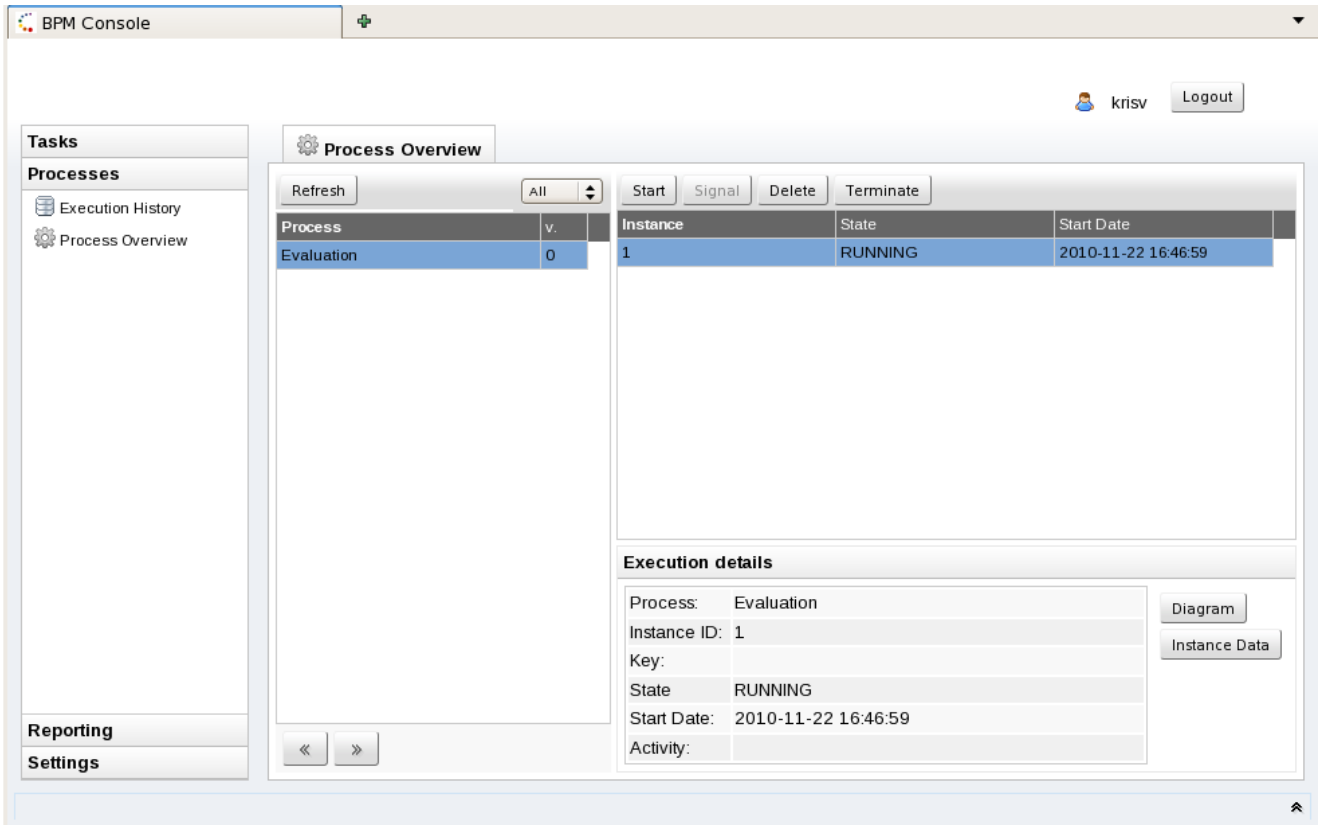
11.2.1.2. Starting new process instances

To start a new process instance for one specific process definition, select the process definition in the process definition list. Click on the "Start" button in the instances table to start a new instance of that specific process. When a form is associated with this particular process (to ask for additional information before starting the process), this form will be shown. After completing this form, the process will be started with the provided information.



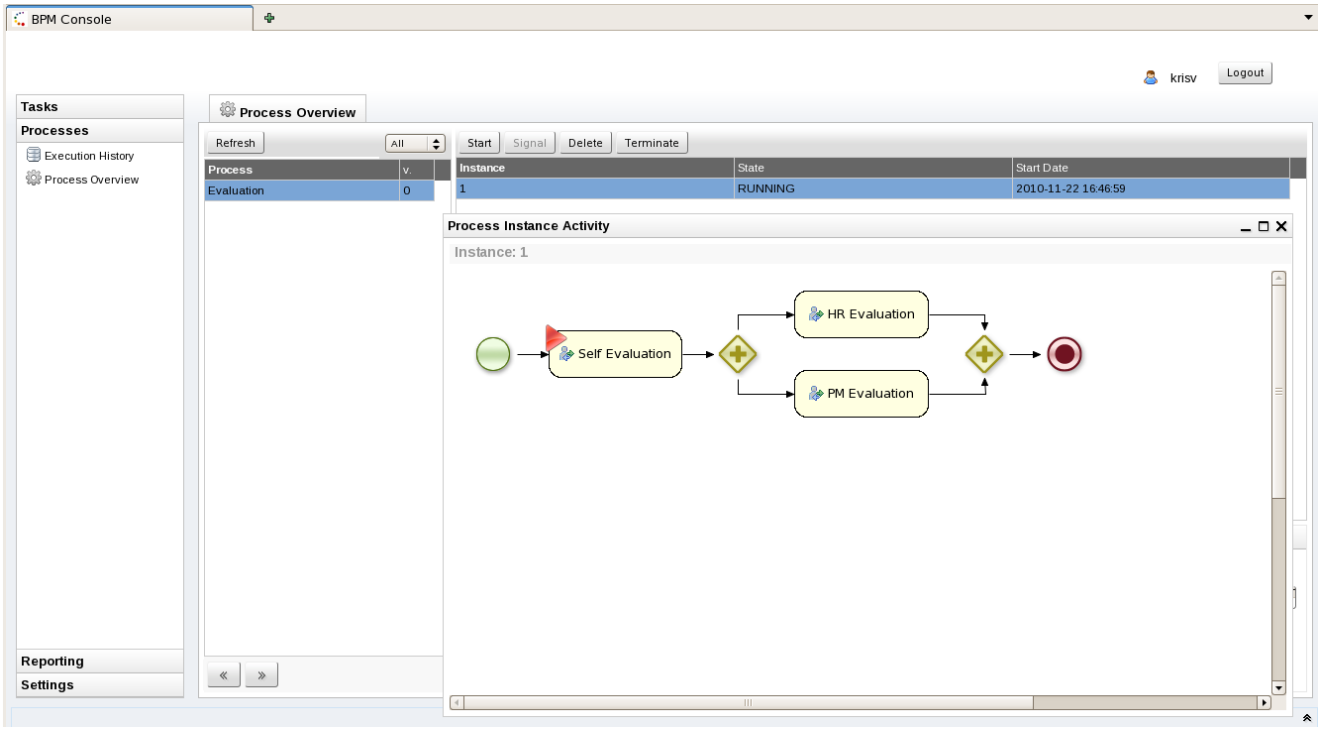
11.2.1.3. Managing process instances

The process instances table shows all running instances of that specific process definition. Select a process instance to show the details of that specific process instance.



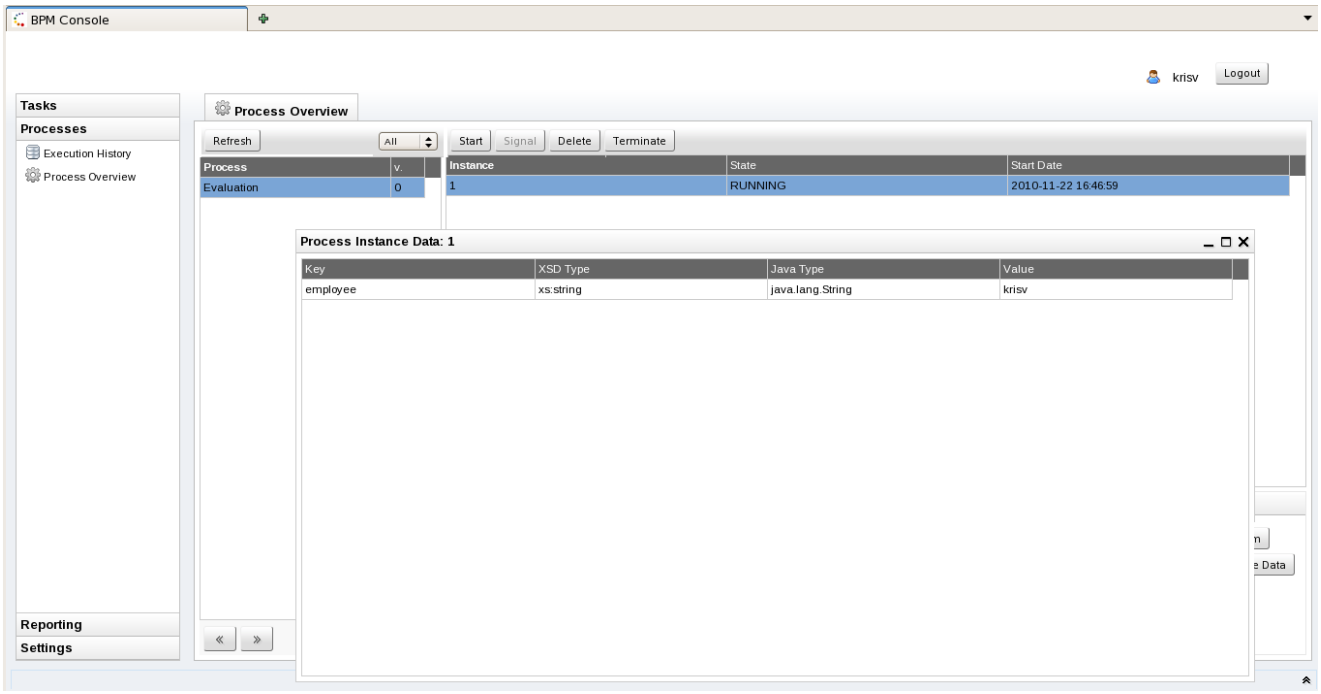
11.2.1.4. Inspecting process instance state

You can inspect the state of a specific process instance by clicking on the "Diagram" button. This will show you the process flow chart, where a red triangle is shown at each node that is currently active (like for example a human task node waiting for the task to be completed or a join node waiting for more incoming connections before continuing). [Note that multiple instances of one node could be executing simultaneously. They will still be shown using only one red triangle.]



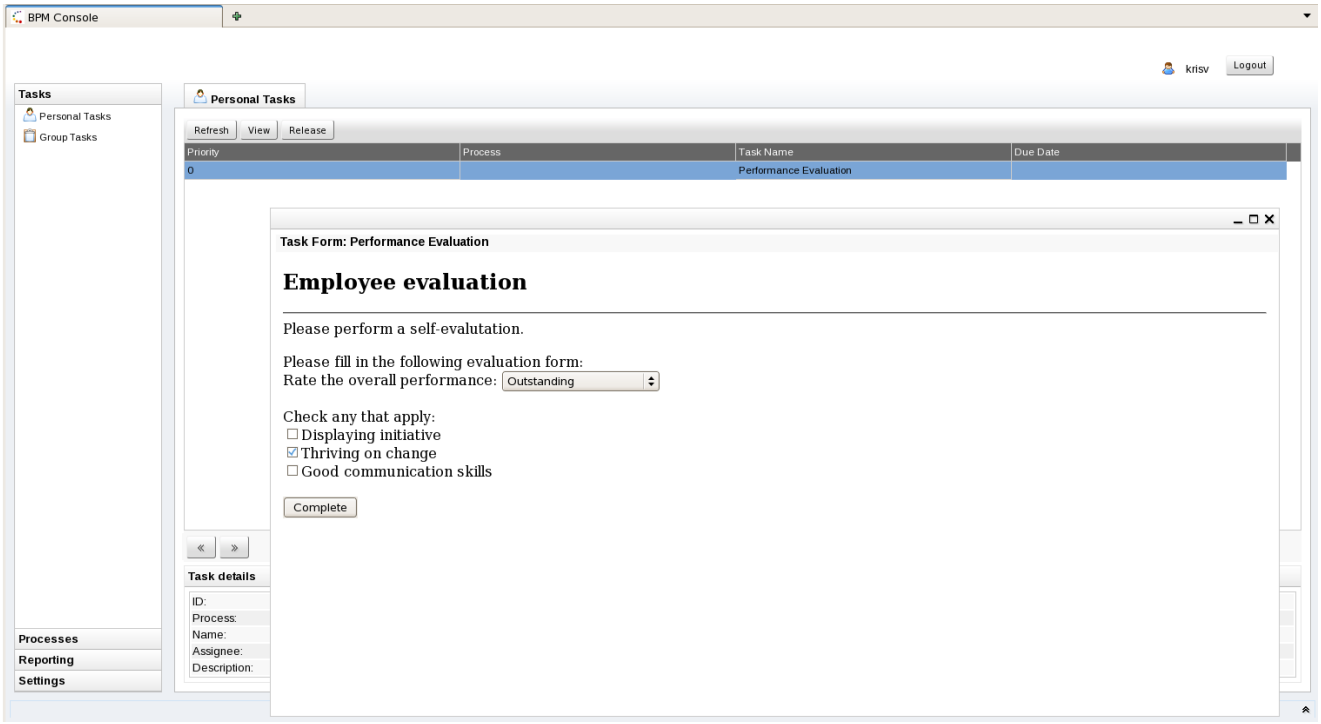
11.2.1.5. Inspecting process instance variables

You can inspect the (top-level) variables of a specific process instance by clicking on the "Instance Data" button. This will show you how each variable defined in the process maps to its corresponding value for that specific process instance.



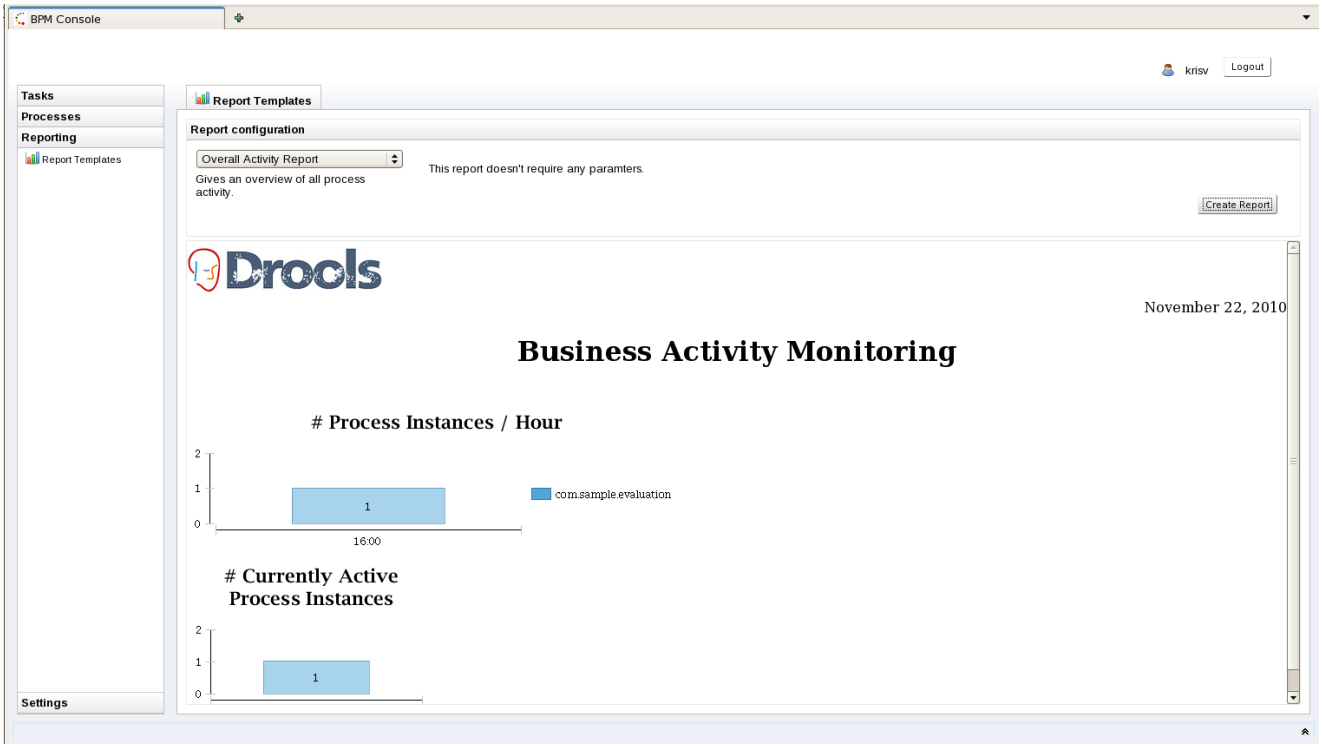
11.2.2. Human task lists

The task management section allows a user to see his/her current task list. The group task list shows all the tasks that are not yet assigned to one specific user but that the currently logged in user could claim. The personal task list shows all tasks that are assigned to the currently logged in user. To execute a task, select it in your personal task list and select "View". If a form is associated with the selected task (for example to ask for additional information), this form will be shown. After completing the form, the task will also be completed.

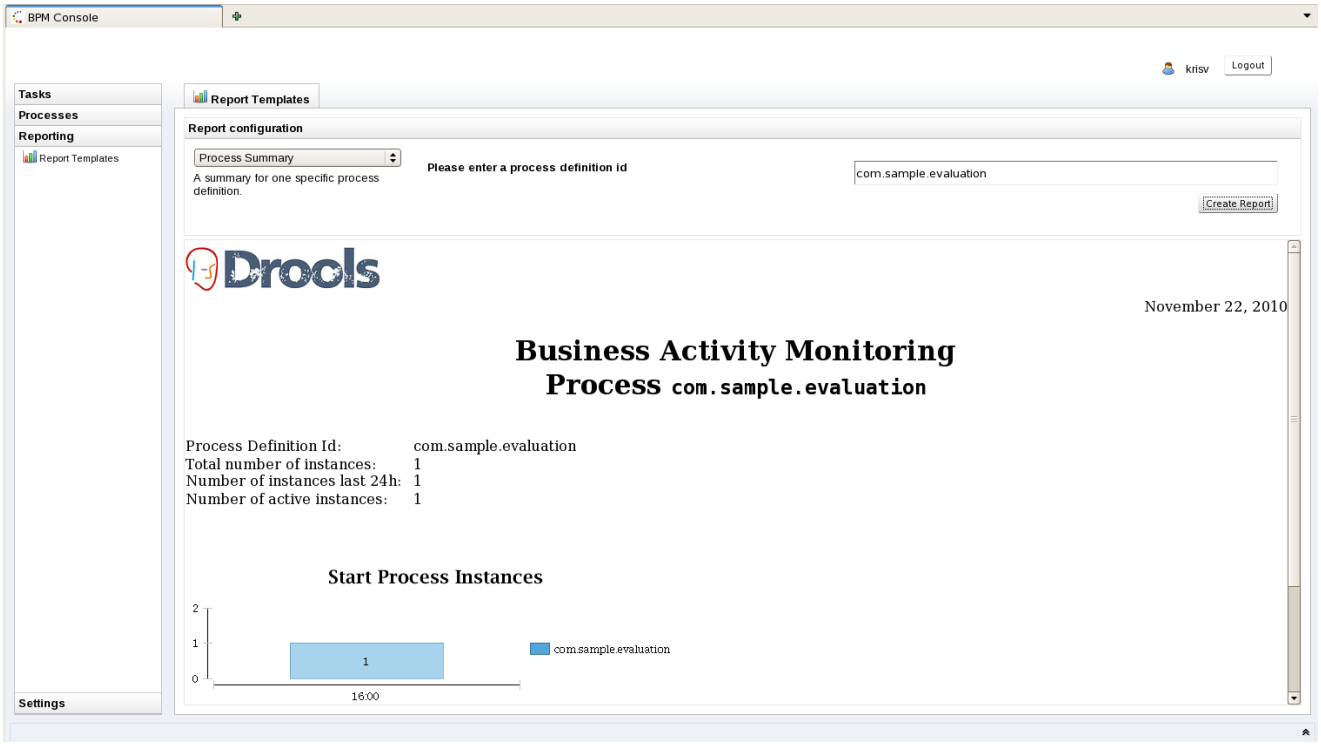


11.2.3. Reporting

The reporting section allows you to view reports about the execution of processes. This includes an overall report showing an overview of all processes, as shown below.



A report regarding one specific process instance can also be generated.



jBPM provides some sample reports that could be used to visualize some generic execution characteristics like the number of active process instances per process etc. But custom reports could be generated to show the information your company thinks is important, by replacing the report templates in the report directory.

The jBPM installer by default does not install the reporting engine (to limit the size of the download). If you want to try out reporting, make sure to put the jBPM.birt.download property in the build.properties file to true before running the installer. If you get an exception that the report engine was not initialized correctly, please run the installer again after making sure that reporting is enabled.

11.3. Adding new process / task forms

Forms can be used to (1) start a new process or (2) complete a human task. We use freemarker templates to dynamically create forms. To create a form for a specific process definition, create a freemarker template with the name {processId}.ftl. The template itself should use HTML code to model the form. For example, the form to start the evaluation process shown above is defined in the com.sample.evaluation.ftl file:

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

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

```
<html>
<body>
<h2>Employee evaluation</h2>
<hr>
${task.descriptions[0].text}<br/>
```

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

Task forms also have access to the additional task parameters that might be mapped in the user task node from process variable using parameter mapping. Check out the chapter on human tasks for more details. These task parameters are also directly accessible inside the task form. For example, imagine that you want to make a task form for review customer requests. The user task node copies the userId (of the customer that performed the request), the comment (the description of the request) and the date (the actual date and time of the request) from the process into the task as task parameters. In that case, these parameters will then be accessible directly in the task form, as shown below:

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

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

Forms should either be available on the classpath (for example inside a jar in the jbossas/server/default/lib folder or added to the set of sample forms in the jbpm-gwt-form.jar in the jbpm console server war), or you could use the Guvnor process repository to store your forms as well. Check out the chapter on the process repository to get more information on how to do that.

11.4. REST interface

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

The list URLS that the REST interface exposes can be inspected if you navigate to the following URL (after installing and starting the console):

http://localhost:8080/gwt-console-server/rs/server/resources/jbpm

For example, this allows you to close a task using

/gwt-console-server/rs/task/{taskId}/close

or starting a new process instance using

/gwt-console-server/rs/process/definition/{id}/new_instance

Chapter 12. Human Tasks

An important aspect of business processes is human task management. While some of the work performed in a process can be executed automatically, some tasks need to be executed by human actors.

jBPM supports a special human task node inside processes for modeling this interaction with human users. This human task node allows process designers to define the properties related to the task that the human actor needs to execute, like for example the type of task, the actor(s), or the data associated with the task.

jBPM also includes a so-called human task service, a back-end service that manages the life cycle of these tasks at runtime. The jBPM implementation is based on the WS-HumanTask specification. Note however that this implementation is fully pluggable, meaning that users can integrate their own human task solution if necessary.

In order to have human actors participate in your processes, you first need to (1) include human task nodes inside your process to model the interaction with human actors, (2) integrate a task management component (like for example the WS-HumanTask based implementation provided by jBPM) and (3) have end users interact with a human task client to request their task list and claim and complete the tasks assigned to them. Each of these three elements will be discussed in more detail in the next sections.

12.1. Human tasks inside processes

jBPM supports the use of human tasks inside processes using a special user task node (as shown in the figure above). A user task node represents an atomic task that needs to be executed by a human actor.

[Although jBPM has a special user task node for including human tasks inside a process, human tasks are considered the same as any other kind of external service that needs to be invoked and are therefore simply implemented as a domain-specific service. See the chapter on *domain-specific processes* to learn more about this.]

A user task node contains the following properties:

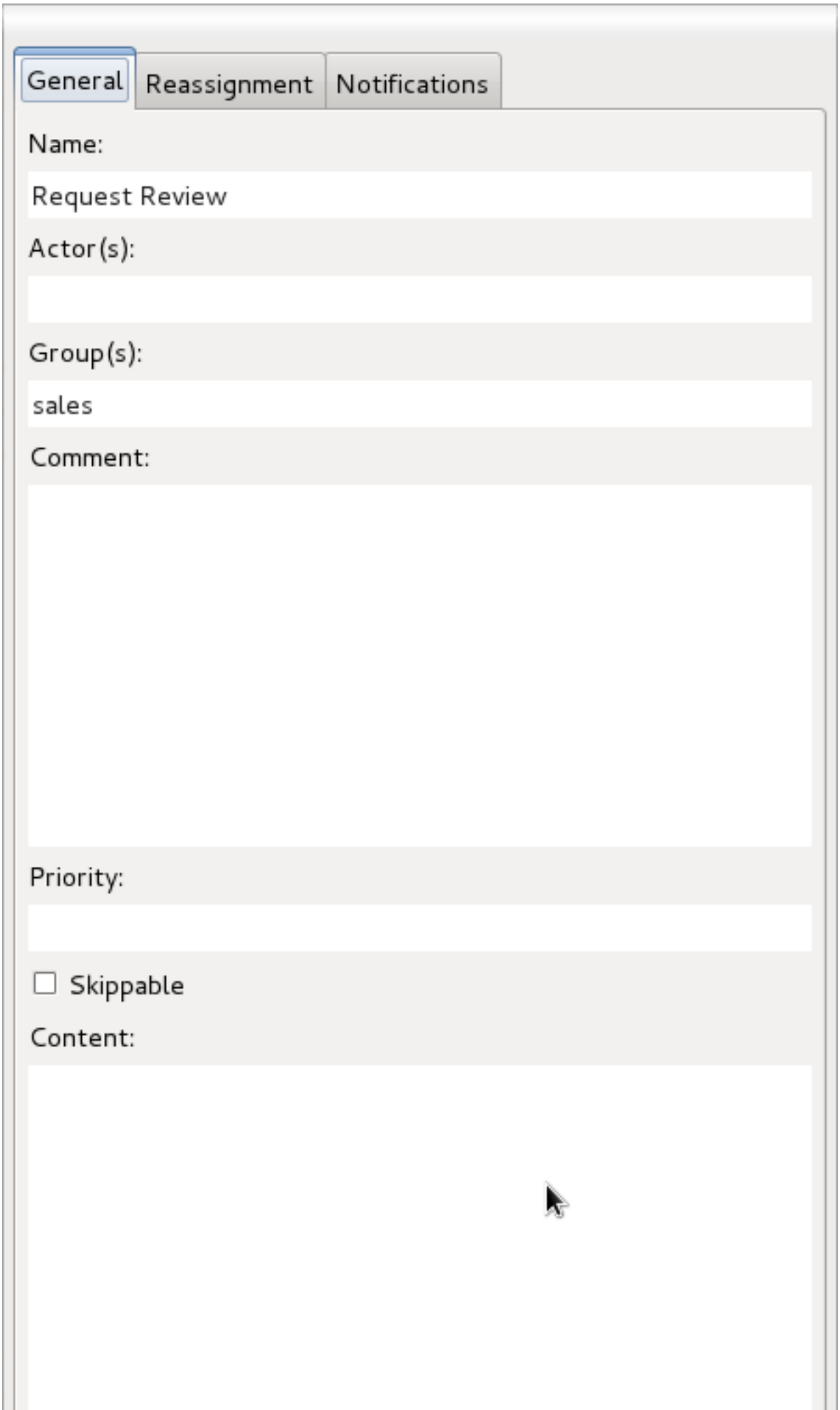
- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.

- *TaskName*: The name of the human task.
- *Priority*: An integer indicating the priority of the human task.
- *Comment*: A comment associated with the human task.
- *ActorId*: The actor id that is responsible for executing the human task. A list of actor id's can be specified using a comma (',') as separator.
- *GroupId*: The group id that is responsible for executing the human task. A list of group id's can be specified using a comma (',') as separator.
- *Skippable*: Specifies whether the human task can be skipped, i.e., whether the actor may decide not to execute the task.
- *Content*: The data associated with this task.
- *Swimlane*: The swimlane this human task node is part of. Swimlanes make it easy to assign multiple human tasks to the same actor. 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.

You can edit these variables in the properties view (see below) when selecting the user task node, or the most important properties can also be edited by double-clicking the user task node, after which a custom user task node editor is opened, as shown below as well.

Properties

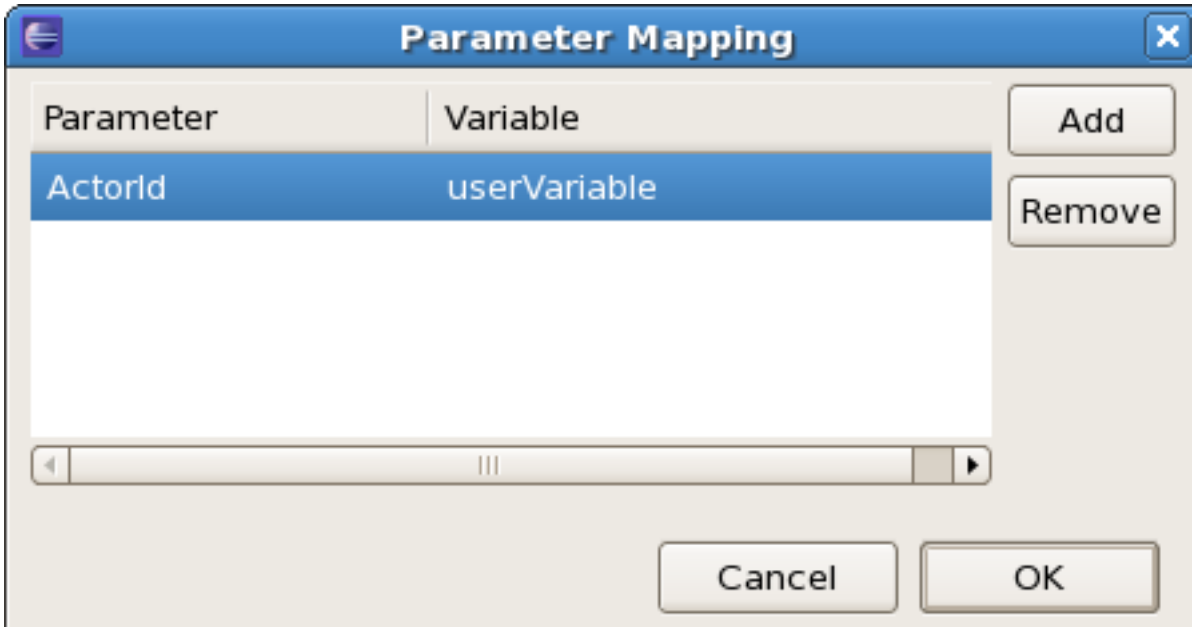
Property	Value
ActorId	Sales Representative
Comment	You should call #{customer.name} to confirm the order.
Content	
Id	4
Name	Human Task
On Entry Actions	
On Exit Actions	
Parameter Mapping	{}
Priority	3
Result Mapping	{}
Skippable	true
Swimlane	
TaskName	Call customer
Timers	
Wait for completion	true


General
Reassignment
Notifications
Name:
Request Review
Actor(s):
Group(s):
sales
Comment:
Priority:
Skippable
Content:

In many cases, the parameters of a user task (like for example the task name, actorId, or priority) can be defined when creating the process. You simply fill in the value of these properties in the property editor. It is however likely that some of the properties of the human task are dependent on some data related to the process instance this task is being requested in. For example, if a business process is used to model how to handle incoming sales requests, tasks that are assigned to a sales representative could include information related to that specific sales request, like its unique id, the name of the customer that requested it, etc. You can make your human task properties dynamic in two ways:

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

- Parameter mapping: You can map the value of a process variable (or a value derived from a variable) to a task parameter. For example, if you need to assign a task to a user whose id is a variable in your process, you can do so by mapping that variable to the parameter ActorId, as shown in the following screenshot. [Note that, for parameters of type String, this would be identical to specifying the ActorId using #{userVariable}, so it would probably be easier to use #{expression} in this case, but parameter mapping also allow you to assign a value to properties that are not of type String.]



12.1.1. User and group assignment

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

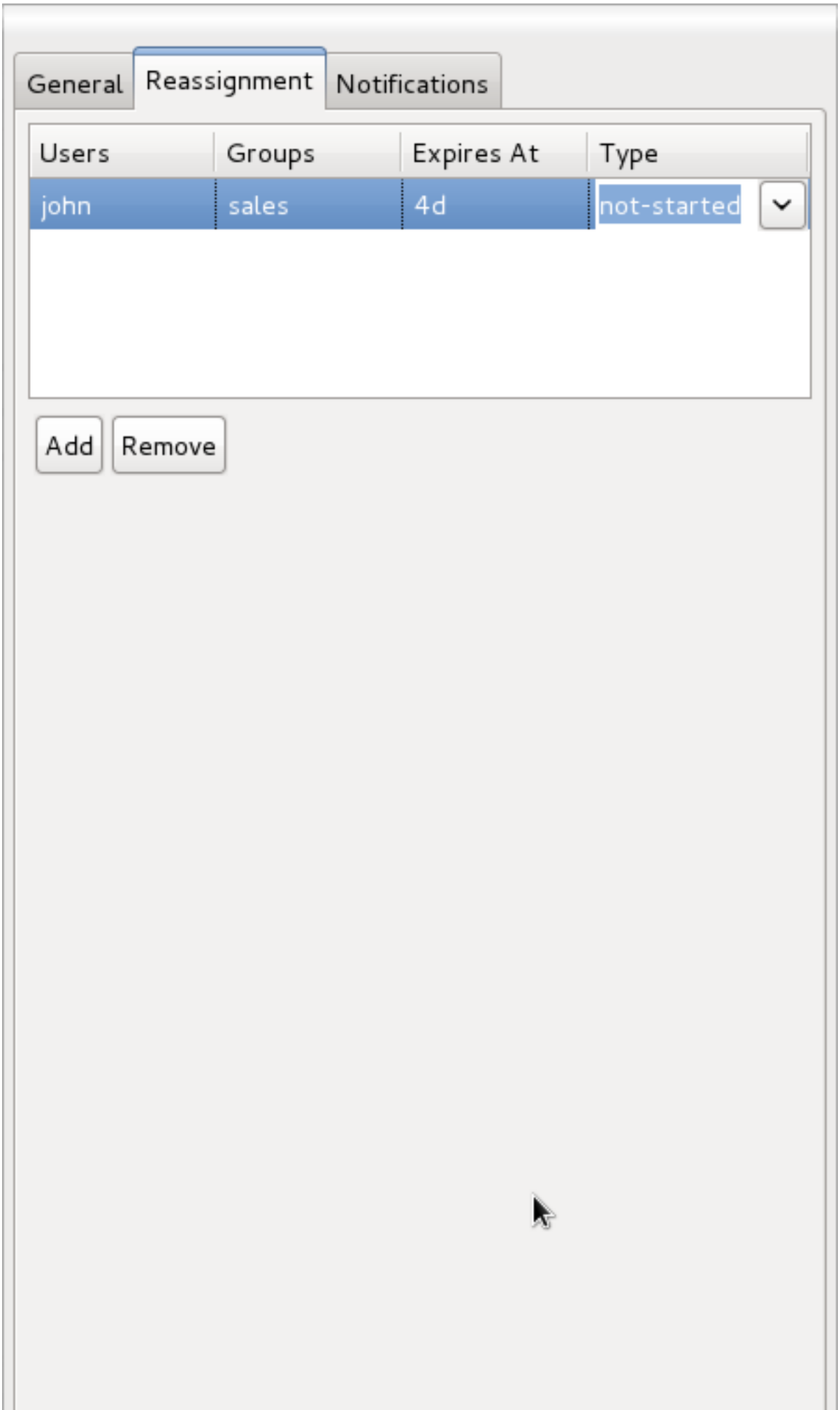
Tasks can also be assigned to one or more groups. This means that any user that is part of the group can claim and execute the task. For more information on how user and group management is handled in the default human task service, check out the user and group assignment.

12.1.2. Task escalation and notification

There are number of situations that can raise a need for escalation of a task, for instance - user assigned to a task can be on vacation or too busy with other work. In such cases task should be automatically reassigned to another actor or group. Escalation can be defined for tasks that are in following statuses:

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

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

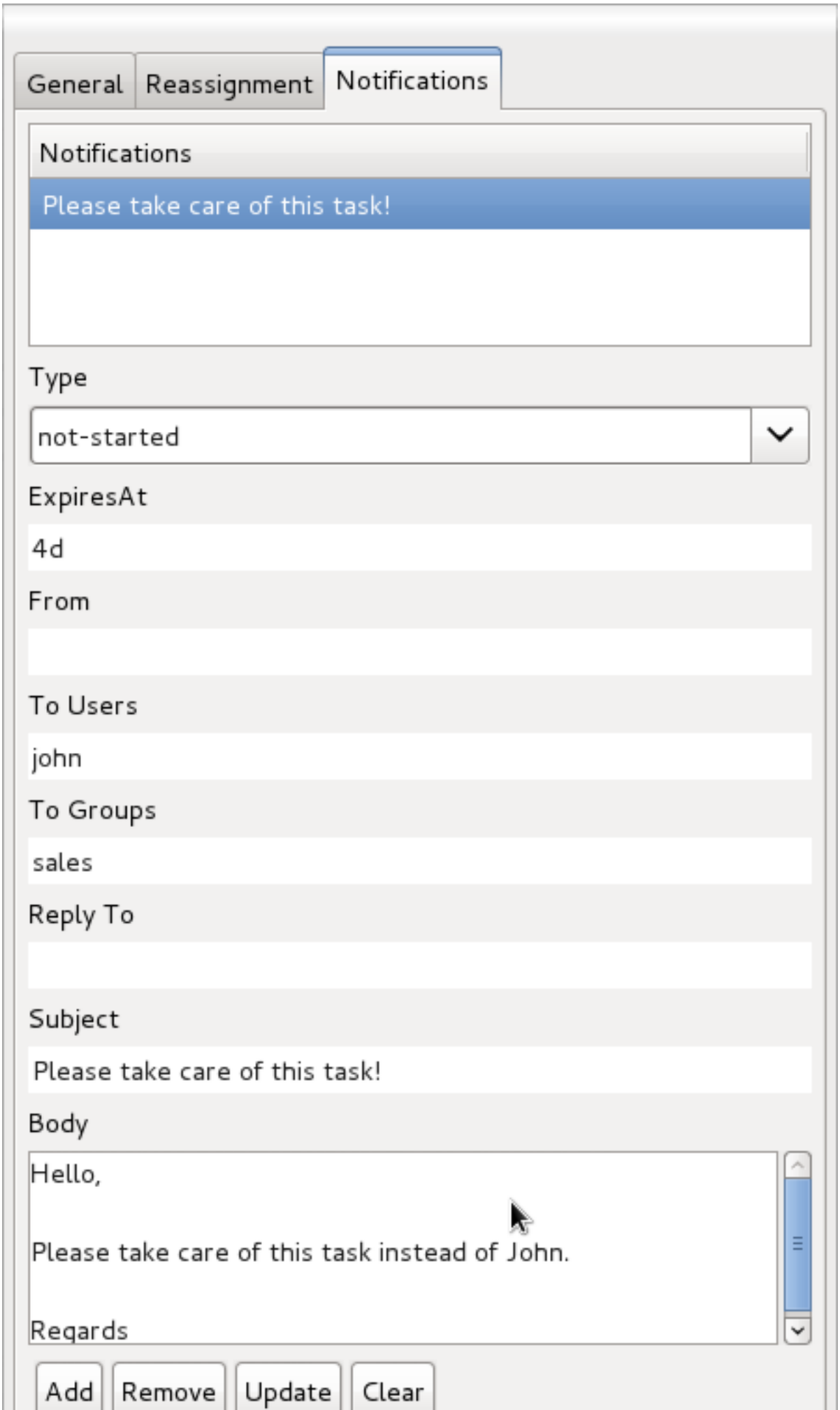

General
Reassignment
Notifications
Users
Groups
Expires At
Type
john
sales
4d
not-started
Add
Remove

Following is a list of attributes that can be specified:

- *Users*: comma spearated list of user ids that should be assigned to the task on escalation. Acceptable are String values and expressions #{user-id}
- *Groups*: comma spearated list of group ids that should be assigned to the task on escalation. Acceptable are String values and expressions #{group-id}
- *Expires At*: time definition about when escalation should take place. It should be defined as time defintion (2m, 4h, 6d, etc.), in same way as for timers. Acceptable are String values and expressions #{expiresAt}
- *Type*: identifies type of task state on which escalation should take place (not-started | not-completed)

In addition to escalation, email notifications can be sent out as well. It is very similar to escalation in terms of definition, allows notification to be sent for tasks that are in following statuses:

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


General
Reassignment
Notifications
Notifications
Please take care of this task!
Type
not-started
ExpiresAt
4d
From
To Users
john
To Groups
sales
Reply To
Subject
Please take care of this task!
Body
Hello,
Please take care of this task instead of John.
Regards
Add
Remove
Update
Clear

Email notification has following properties:

- *Type*: identifies type of task state on which escalation should take place (not-started | not-completed)
- *Expires At*: time definition about when escalation should take place. It should be defined as time defintion (2m, 4h, 6d, etc.), in same way as for timers. Acceptable are String values and expressions #{expiresAt}
- *From*: (Optional) user or group id that will be used as From field for email message - accepts String and expression
- *To Users*: comman separated list of user ids that will become reciepients of the notification
- *To Groups*: comman separated list of group ids that will become reciepients of the notification
- *Reply To*: (Optional) user or group id that should receive replies to the notification
- *Subject*: Subject of the notification - accepts String and expression
- *Body*: Body of the notification - accepts String and expression

Notification can reference process variables by *#{processVariable}* and task variables *${taskVariable}*. Main difference between those two is that process variables will be resolved at task creation time and task variables will be resolved at notification time. There are several task variables (besides regular ones) that can be used while working with notifications:

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

An example that illustrates a simple notification message (its body) that shows how different variables can be accessed:

```
<html>
 <body>
  <b>${owners[0].id} you have been assigned to a task (task-id ${taskId})</b><br>
```

```
  You can access it in your task
                       <a           href="http://localhost:8080/jbpm-console/
app.html#errai_ToolSet_Tasks;Group_Tasks.3">inbox</a><br/>
  Important technical information that can be of use when working on it<br/>
  - process instance id - ${processInstanceId}<br/>
  - work item id - ${workItemId}<br/>

  <hr/>

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

  <i>Regards from jBPM team</i>
 </body>
</html>
```

12.1.3. Data mapping

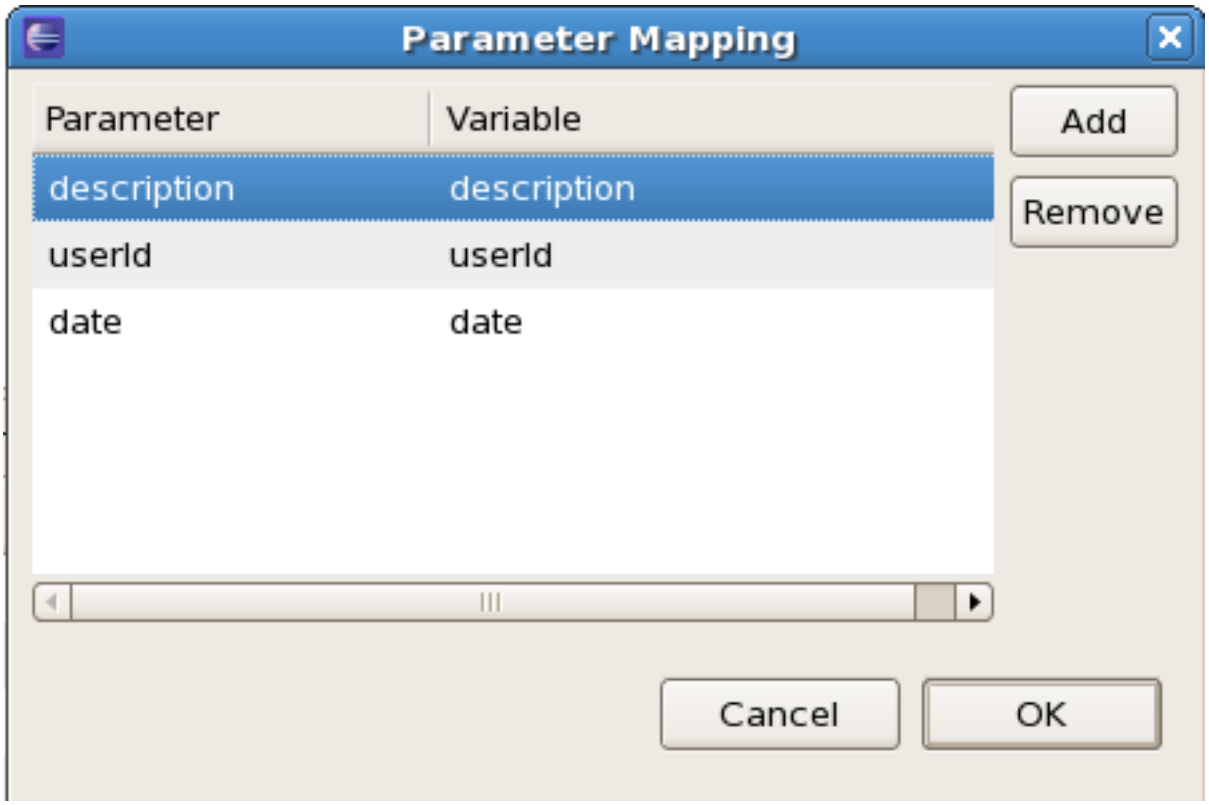
Human tasks typically present some data related to the task that needs to be performed to the actor that is executing the task and usually also request the actor to provide some result data related to the execution of the task. Task forms are typically used to present this data to the actor and request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

12.1.3.1. Task parameters

Data that needs to be displayed in a task form should be passed to the task, using parameter mapping. Parameter mapping allows you to copy the value of a process variable to a task parameter (as described above). This could for example be the customer name that needs to be displayed in the task form, the actual request, etc. To copy data to the task, simply map the variable to a task parameter. This parameter will then be accessible in the task form (as shown later, when describing how to create task forms).

For example, the following human task (as part of the humantask example in jbpm-examples) is assigned to a sales representative that needs to decide whether to accept or reject a request from a customer. Therefore, it copies the following process variables to the task as task parameters:

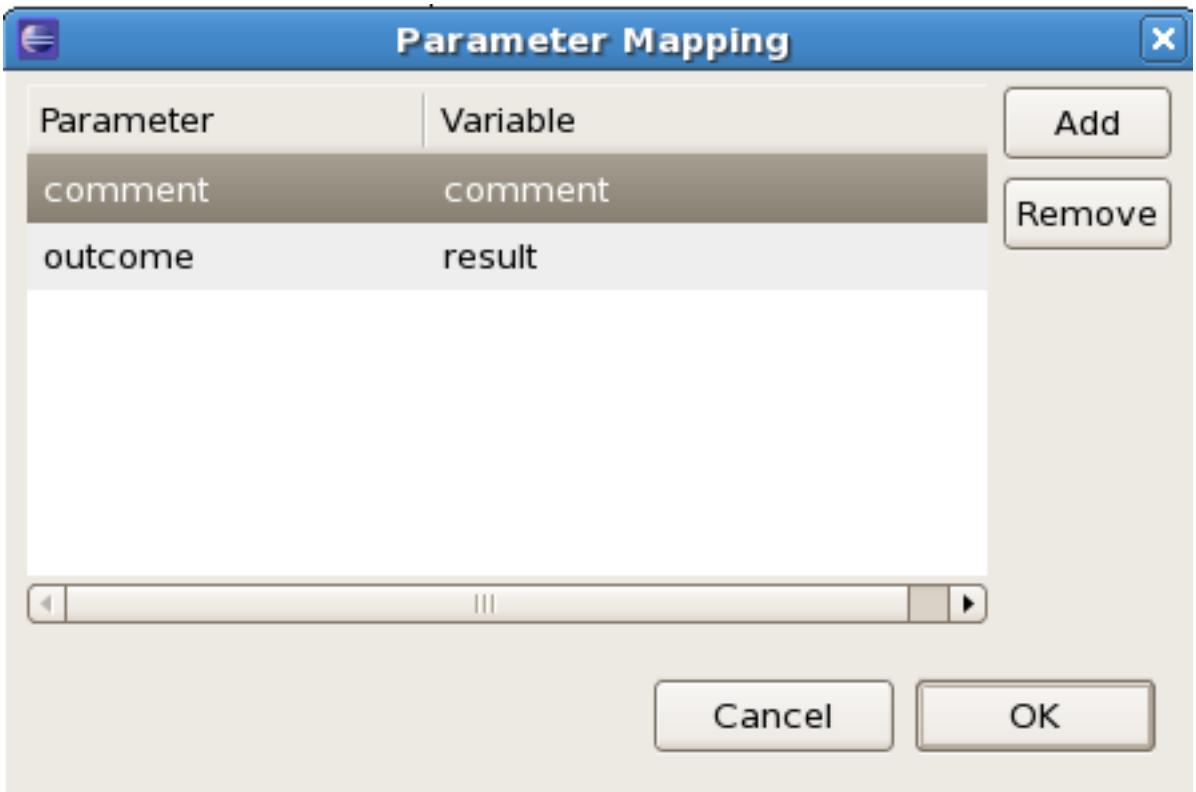
the userId (of the customer doing the request), the description (of the request), and the date (of the request).



12.1.3.2. Task results

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

For example, the following human task (as part of the humantask example in jbpm-examples) is assigned to a sales representative that needs to decide whether to accept or reject a request from a customer. Therefore, it copies the following task results back to the process: the outcome (the decision that the sales representative has made regarding this request, in this case "Accept" or "Reject") and the comment (the justification why).



12.1.4. Swimlanes

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

Whenever a human task that is part of a swimlane is completed, the actorId of that swimlane is set to the actorId that executed that human task. This allows for example to assign a human task to a group of users, and to assign future tasks of that swimlame to the user that claimed the first task. This will also automatically change the assignment of tasks if at some point one of the tasks is reassigned to another user.

To add a human task to a swimlane, simply specify the name of the swimlane as the value of the "Swimlane" parameter of the user task node. A process must also define all the swimlanes that it contains. To do so, open the process properties by clicking on the background of the process and click on the "Swimlanes" property. You can add new swimlanes there.

The new BPMN2 Eclipse editor will support a visual representation of swimlanes (as horizontal lanes), so that it will be possible to define a human task as part of a swimlane simply by dropping the task in that lane on the process model.

12.1.5. Examples

The jbpm-examples module has some examples that show human tasks in action, like the evaluation example and the humantask example. These examples show some of the more advanced features in action, like for example group assignment, data passing in and out of human tasks, swimlanes, etc. Be sure to take a look at them for more details and a working example.

12.2. Human task service

As far as the jBPM engine is concerned, human tasks are similar to any other external service that needs to be invoked and are implemented as a domain-specific service. (For more on domain-specific services, see the chapter on them *here*.) Because a human task is an example of such a domain-specific service, the process itself only contains a high-level, abstract description of the human task to be executed and a work item handler that is responsible for binding this (abstract) task to a specific implementation.

Users can plug in any human task service implementation, such as the one that's provided by jBPM, or they may register their own implementation. In the next paragraphs, we will describe the human task servcie implementation provided by jBPM.

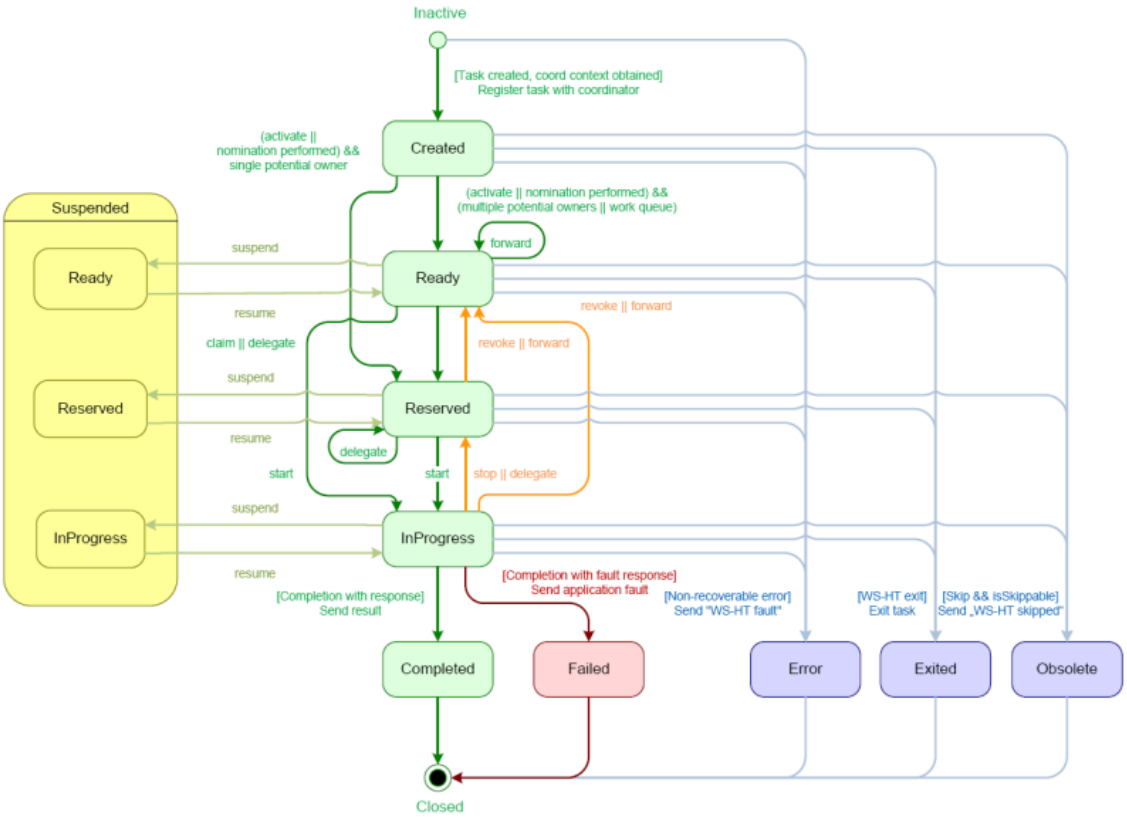
The jBPM project provides a default implementation of a human task service based on the WS-HumanTask specification. If you do not need to integrate jBPM with another existing implementation of a human task service, you can use this service. The jBPM implementation manages the life cycle of the tasks (creation, claiming, completion, etc.) and stores the state of all the tasks, task lists, and other associated information. It also supports features like internationalization, calendar integration, different types of assignments, delegation, escalation and deadlines. The code for the implementation itself can be found in the jbpm-human-task module.

The jBPM task service implementation is based on the WS-HumanTask (WS-HT) specification. This specification defines (in detail) the model of the tasks, the life cycle, and many other features. It is very comprehensive and the first version can be found *here* [http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf].

12.2.1. Task life cycle

From the perspective of a process, when a user task node is encountered during the execution, a human task is created. The process will then only leave the user task node when the associated human task has been completed or aborted.

The human task itself usually has a complete life cycle itself as well. For details beyond what is described below, please check out the WS-HumanTask specification. The following diagram is from the WS-HumanTask specification and describes the human task life cycle.



A newly created task starts in the "Created" stage. Usually, it will then automatically become "Ready", after which the task will show up on the task list of all the actors that are allowed to execute the task. The task will stay "Ready" until one of these actors claims the task, indicating that he or she will be executing it.

When a user then eventually claims the task, the status will change to "Reserved". Note that a task that only has one potential (specific) actor will automatically be assigned to that actor upon creation of the task. When the user who has claimed the task starts executing it, the task status will change from "Reserved" to "InProgress".

Lastly, once the user has performed and completed the task, the task status will change to "Completed". In this step, the user can optionally specify the result data related to the task. If the task could not be completed, the user could also indicate this by using a fault response, possibly including fault data, in which case the status would change to "Failed".

While the life cycle explained above is the normal life cycle, the specification also describes a number of other life cycle methods, including:

- Delegating or forwarding a task, so that the task is assigned to another actor
- Revoking a task, so that it is no longer claimed by one specific actor but is (re)available to all actors allowed to take it
- Temporarily suspending and resuming a task

- Stopping a task in progress
- Skipping a task (if the task has been marked as skippable), in which case the task will not be executed

12.2.2. Linking the human task service to the jBPM engine

Just like any other external service, the human task service can be integrated with the jBPM engine by registering a work item handler that translates the abstract work item (in this case a human task) to a specific invocation of a service (in this case, the jBPM implementation of the human task service). There are several implementations of a work item handler available that can be selected depending on following factors:

- transport used (HornetQ, Mina, JMS)
- local interaction - same transaction boundary as the engine
- mode of interaction - synchronous or asynchronous

Here is a list of all available work item handlers for human tasks:

Table 12.1. Work item handlers for human task

Class name	Module	Mode
`org.jbpm.process.workitem.wsht.LocalHTWorkItemHandler`	jbpm-human-task-core	Local
`org.jbpm.process.workitem.wsht.AsyncHornetQHTWorkItemHandler`	jbpm-human-task-hornetq	Async
`org.jbpm.process.workitem.wsht.HornetQHTWorkItemHandler`	jbpm-human-task-hornetq	Sync
`org.jbpm.process.workitem.wsht.AsyncMinaHTWorkItemHandler`	jbpm-human-task-mina	Async
`org.jbpm.process.workitem.wsht.MinaHTWorkItemHandler`	jbpm-human-task-mina	Sync

Once you select the one that meets your needs you can register this work item handler like this:

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

By default, this handler will connect to the human task service on the local machine on port 5153 via hornetq. You can easily change connection details of the human task service by either building

TaskClient yourself and pass it as handler constructor argument or by setting ip address and port number after handler is created.

Note

Important to note is that when there is requirement to use multiple knowledge sessions (meaning every session will have a dedicated work item handler for human tasks) you must configure handler to react only to tasks that were initiated by that session that is attached to the handler to avoid duplicated activations.

```
new AsyncHornetQHTWorkItemHandler(ksession, true))
```

The communication between the human task service and the process engine, or any task client, is message based. While the client/server transport mechanism is pluggable (allowing different implementations), the default is HornetQ. An alternative implementation using Mina (*http://mina.apache.org/*) is also available.

12.2.3. Interacting with the human task service

The human task service exposes a Java API for managing the life cycle of tasks. This allows clients to integrate (at a low level) with the human task service. Note that end *users* should probably not interact with this low-level API directly, but use one of the more user-friendly task clients (see below) instead. These clients offer a graphical user interface to request task lists, claim and complete tasks, and manage tasks in general. The task clients listed below use the Java API to internally interact with the human task service. Of course, the low-level API is also available so that developers can use it in their code to interact with the human task service directly.

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

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

If you take a look at the method signatures you will notice that almost all of these methods take the following arguments:

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

When you invoke a message on the TaskClient, a message is created that will be sent to the server. The server then executes the operation requested in the message.

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

```
TaskClient client = new TaskClient(new MinaTaskClientConnector("client 1",
    new MinaTaskClientHandler(SystemEventListenerFactory.getSystemEventListener())));
client.connect("127.0.0.1", 9123);

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

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

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

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

12.2.4. User and group assignment

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

and execute this task. Tasks can also be assigned to one or more groups. This means that any user that is part of the group can claim and execute the task.

The human task service needs to know about valid user and group ids (to make sure tasks are assigned to existing users and/or groups to avoid errors and tasks that end up assigned to non-existing users). User and group registration has to be done before tasks can be assigned to them. One possible registration method is to dynamically adding users and groups to the task service session:

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

The human task service itself does not maintain the relationship between users and groups. This is considered outside the scope of the human task service: in general, businesses already have existing services that manage this information (i.e. an LDAP service). The human task service does allow you to specify the list of groups that a user is part of, so that this information can also be taken into account when managing tasks.

For example, if a task is assigned to the group "sales" and the user "sales-rep-1", who is a member of "sales", wants to claim that task, then that user needs to pass the fact that he is a member of "sales" when requesting the list of tasks that he is assigned to as potential owner:

```
List<String> groups = new ArrayList<String>();
groups.add("sales");
taskClient.getTasksAssignedAsPotentialOwner("sales-rep",      groups,      "en-
UK", taskSummaryHandler);
```

The WS-HumanTask specification also introduces the role of an administrator. An administrator can manipulate the life cycle of the task, even though he might not be assigned as a potential owner of that task. By default, jBPM registers a special user with userId "Administrator" as the administrator of each task. You should therefor make sure that you always define at least a user "Adminstrator" when registering the list of valid users at the task service.

It is often necessary to hook into existing systems and/or services (such as LDAP) where users and groups are maintained in order to perform validation without having to manually register all users and group with the task service. jBPM provides the UserGroupCallback interface which allows you to create your own implementation for user and group management:

```
public interface UserGroupCallback {
```

```
    /**
     * Resolves existence of user id.
     * @param userId    the user id assigned to the task
     * @return true if userId exists, false otherwise.
     */
    boolean existsUser(String userId);

    /**
     * Resolves existence of group id.
     * @param groupId   the group id assigned to the task
     * @return true if groupId exists, false otherwise.
     */
    boolean existsGroup(String groupId);

    /**
     * Returns list of group ids for specified user id.
     * @param userId    the user id assigned to the task
     * @param groupIds  list of group ids assigned to the task
     * @param allExistingGroupIds   list of all currently known group ids
     * @return List of group ids.
     */
    List<String> getGroupsForUser(String userId, List<String> groupIds, List<String> allExistin
}
```

If you register your own implementation of the UserGroupCallback interface, the human task service will call it whenever it needs to perform user and group validation. Here is a very simple example implementation which treats all users and groups as being valid:

```
public class DefaultUserGroupCallbackImpl implements UserGroupCallback {

    public boolean existsUser(String userId) {
        // accept all by default
        return true;
    }

    public boolean existsGroup(String groupId) {
        // accept all by default
        return true;
    }

    public List<String> getGroupsForUser(String userId, List<String> groupIds,
            List<String> allExistingGroupIds) {
        if(groupIds != null) {
            List<String> retList = new ArrayList<String>(groupIds);
            // merge all groups
            if(allExistingGroupIds != null) {
```

```
                for(String grp : allExistingGroupIds) {
                    if(!retList.contains(grp)) {
                        retList.add(grp);
                    }
                }
            }
            return retList;
        } else {
            // return empty list by default
            return new ArrayList<String>();
        }
    }
}
```

You can register your own implementation of the UserGroupCallback interface in a properties file called jbpm.usergroup.callback.properties which should be available on the classpath, for example:

```
jbpm.usergroup.callback=org.jbpm.task.service.DefaultUserGroupCallbackImpl
```

or via a system property, for example - `Djbpm.usergroup.callback=org.jbpm.task.service.DefaultUserGroupCallbackImpl`. If you are using the jBPM installer, you can also modify `$jbpm-installer-dir$/task-service/resources/org/jbpm/jbpm.usergroup.callback.properties` directly to register your own callback implementation.

12.2.4.1. Connecting Human Task server to LDAP

jBPM comes with a dedicated UserGroupCallback implementation for LDAP servers that allows task server to retrieve user and group/role information directly from LDAP. To be able to use this callback it must be configured according to specifics of LDAP server and its structure to collect proper information.
LDAP UserGroupCallback properties

- ldap.bind.user : username used to connect to the LDAP server (optional if LDAP server accepts anonymous access)
- ldap.bind.pwd : password used to connect to the LDAP server(optional if LDAP server accepts anonymous access)
- ldap.user.ctx : context in LDAP that will be used when searching for user information (mandatory)
- ldap.role.ctx : context in LDAP that will be used when searching for group/role information (mandatory)

- ldap.user.roles.ctx : context in LDAP that will be used when searching for user group/role membership information (optional, if not given ldap.role.ctx will be used)
- ldap.user.filter : filter that will be used to search for user information, usually will contain substitution keys {0} to be replaced with parameters (mandatory)
- ldap.role.filter : filter that will be used to search for group/role information, usually will contain substitution keys {0} to be replaced with parameters (mandatory)
- ldap.user.roles.filter : filter that will be used to search for user group/role membership information, usually will contain substitution keys {0} to be replaced with parameters (mandatory)
- ldap.user.attr.id : attribute name of the user id in LDAP (optional, if not given 'uid' will be used)
- ldap.roles.attr.id : attribute name of the group/role id in LDAP (optional, if not given 'cn' will be used)
- ldap.user.id.dn : is user id a DN, instructs the callback to query for user DN before searching for roles (optional, default false)
- java.naming.factory.initial : initial conntext factory class name (default com.sun.jndi.ldap.LdapCtxFactory)
- java.naming.security.authentication : authentication type (none, simple, strong where simple is default one)
- java.naming.security.protocol : specifies security protocol to be used, for instance ssl
- java.naming.provider.url : LDAP url to be used default is ldap://localhost:389, or if protocol is set to ssl ldap://localhost:636

Depending on how human task server is started LDAP callback can be configured in two ways:

- programatically - build property object with all required attributes and register new callback

```
Properties properties = new Properties();
properties.setProperty(LDAPUserGroupCallbackImpl.USER_CTX,  "ou=People,dc=my-
domain,dc=com");
properties.setProperty(LDAPUserGroupCallbackImpl.ROLE_CTX,   "ou=Roles,dc=my-
domain,dc=com");
properties.setProperty(LDAPUserGroupCallbackImpl.USER_ROLES_CTX,  "ou=Roles,dc=my-
domain,dc=com");
properties.setProperty(LDAPUserGroupCallbackImpl.USER_FILTER, "(uid={0})");
properties.setProperty(LDAPUserGroupCallbackImpl.ROLE_FILTER, "(cn={0})");
properties.setProperty(LDAPUserGroupCallbackImpl.USER_ROLES_FILTER, "(member={0})");

UserGroupCallback ldapUserGroupCallback = new LDAPUserGroupCallbackImpl(properties);

UserGroupCallbackManager.getInstance().setCallback(ldapUserGroupCallback);
```

- declaratively - create property file (jbpm.usergroup.callback.properties) with all required attributes, place it on the root of the classpath and declare LDAP callback to be registered (see section Starting the human task server for deatils). Alternatively, location of jbpm.usergroup.callback.properties can be specified via system property - Djbpm.usergroup.callback.properties=FILE_LOCATION_ON_CLASSPATH

```
#ldap.bind.user=
#ldap.bind.pwd=
ldap.user.ctx=ou\=People,dc\=my-domain,dc\=com
ldap.role.ctx=ou\=Roles,dc\=my-domain,dc\=com
ldap.user.roles.ctx=ou\=Roles,dc\=my-domain,dc\=com
ldap.user.filter=(uid\={0})
ldap.role.filter=(cn\={0})
ldap.user.roles.filter=(member\={0})
#ldap.user.attr.id=
#ldap.roles.attr.id=
```

12.2.5. Starting the human task service

The human task service is a completely independent service that the process engine communicates with. We therefore recommend that you start it as a separate service as well. The jBPM installer contains a command to start the task server (in this case using Mina as transport protocol), or you can use the following code fragment:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("org.jbpm.task");
TaskService taskService = new TaskService(emf, SystemEventListenerFactory.getSystemEventListene
MinaTaskServer server = new MinaTaskServer( taskService );
Thread thread = new Thread( server );
thread.start();
```

The task management component uses the Java Persistence API (JPA) to store all task information in a persistent manner. To configure the persistence, you need to modify the persistence.xml configuration file accordingly. We refer to the JPA documentation on how to do that. The following fragment shows for example how to use the task management component with hibernate and an in-memory H2 database:

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

```
    xsi:schemaLocation=
      "http://java.sun.com/xml/ns/persistence
       http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
       http://java.sun.com/xml/ns/persistence/orm
       http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
    xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/persistence">

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

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

```
</persistence>
```

The first time you start the task management component, you need to make sure that all the necessary users and groups are added to the database. Our implementation requires all users and groups to be predefined before trying to assign a task to that user or group. So you need to make sure you add the necessary users and group to the database using the taskSession.addUser(user) and taskSession.addGroup(group) methods. Note that you at least need an "Administrator" user as all tasks are automatically assigned to this user as the administrator role.

The jbpm-human-task module contains a org.jbpm.task.RunTaskService class in the src/test/java source folder that can be used to start a task server. It automatically adds users and groups as defined in LoadUsers.mvel and LoadGroups.mvel configuration files.

The jBPM installer automatically starts a human task service (using an in-memory H2 database) as a separate Java application. This task service is defined in the task-service directory in the jbpm-installer folder. You can register new users and task by modifying the LoadUsers.mvel and LoadGroups.mvel scripts in the resources directory.

12.2.5.1. Configure escalation and notifications

To allow Task Server to perform escalations and notification a bit of configuration is required. Most of the configuration is for notification support as it relies on external system (mail server) but as they are handled by EscalatedDeadlineHandler implementation so configuration apply to both.

```
// configure email service
Properties emailProperties = new Properties();
emailProperties.setProperty("from", "jbpm@domain.com");
emailProperties.setProperty("replyTo", "jbpm@domain.com");
emailProperties.setProperty("mail.smtp.host", "localhost");
emailProperties.setProperty("mail.smtp.port", "2345");

// configure default UserInfo
Properties userInfoProperties = new Properties();
// : separated values for each org entity email:locale:display-name
userInfoProperties.setProperty("john", "john@domain.com:en-UK:John");
userInfoProperties.setProperty("mike", "mike@domain.com:en-UK:Mike");
userInfoProperties.setProperty("Administrator",          "admin@domain.com:en-
UK:Admin");

// build escalation handler
DefaultEscalatedDeadlineHandler handler = new DefaultEscalatedDeadlineHandler(emailProperties);
// set user info on the escalation handler
handler.setUserInfo(new DefaultUserInfo(userInfoProperties));

EntityManagerFactory emf = Persistence.createEntityManagerFactory("org.jbpm.task");
// when building TaskService provide escalation handler as argument
```

```
TaskService taskService = new TaskService(emf, SystemEventListenerFactory.getSystemEventListene
MinaTaskServer server = new MinaTaskServer( taskService );
Thread thread = new Thread( server );
thread.start();
```

Note that default implementation of UserInfo is just for demo purposes to have a fully operational task server. Custom user info classes can be provided that implement following interface:

```
public interface UserInfo {
    String getDisplayName(OrganizationalEntity entity);

    Iterator<OrganizationalEntity> getMembersForGroup(Group group);

    boolean hasEmail(Group group);

    String getEmailForEntity(OrganizationalEntity entity);

    String getLanguageForEntity(OrganizationalEntity entity);
}
```

If you are using the jBPM installer, just drop your property files into `$jbpm-installer-dir$/task-service/resources/org/jbpm/`, make sure that they are named email.properties and userinfo.properties.

12.2.5.1.1. User information retrieved from LDAP server

More production alike configuration would be to use LDAP server as user information repository and to achieve that a dedicated UserInfo implementation is shipped with jBPM - LDAPUserInfoImpl. This is especially useful when configuring task server to use LDAP based user group callback, with this complete user/group information are externalized to LDAP server.
LDAP UserGroupCallback properties

- ldap.bind.user : username used to connect to the LDAP server (optional if LDAP server accepts anonymous access)
- ldap.bind.pwd : password used to connect to the LDAP server(optional if LDAP server accepts anonymous access)
- ldap.user.ctx : context in LDAP that will be used when searching for user information (mandatory)
- ldap.role.ctx : context in LDAP that will be used when searching for group/role information (mandatory)

- ldap.user.filter : filter that will be used to search for user information, usually will contain substitution keys {0} to be replaced with parameters (mandatory)
- ldap.role.filter : filter that will be used to search for group/role information, usually will contain substitution keys {0} to be replaced with parameters (mandatory)
- ldap.role.members.filter : filter that will be used to search for user group/role membership information, usually will contain substitution keys {0} to be replaced with parameters (optional default same as ldap.role.filter)
- ldap.email.attr.id : attribute id that contains email address in LDAP (default mail)
- ldap.name.attr.id : attribute id that contians display name in LDAP (default displayName)
- ldap.lang.attr.id : attribute id that contians language information (default locale)
- ldap.member.attr.id : attribute id on group/role object in LDAP that contains members (default member)
- ldap.user.attr.id : attribute id that contains user id in LDAP server (default uid)
- ldap.role.attr.id : attribute id that contains group/role id in LDAP server (default cn)
- ldap.entity.id.dn : instructs if the organizational entity is (or can be) DN, especially important when members of a group will be returned as DN instead of user ids (default false)
- java.naming.factory.initial : initial conntext factory class name (default com.sun.jndi.ldap.LdapCtxFactory)
- java.naming.security.authentication : authentication type (none, simple, strong where simple is default one)
- java.naming.security.protocol : specifies security protocol to be used, for instance ssl
- java.naming.provider.url : LDAP url to be used default is ldap://localhost:389, or if protocol is set to ssl ldap://localhost:636

Depending on how human task server is started LDAP user info can be configured in two ways:

- programatically - build property object with all required attributes and register new user info on escalation handler

```
Properties properties = new Properties();
properties.setProperty(LDAPUserInfoImpl.USER_CTX, "ou=People,dc=jbpm,dc=org");
properties.setProperty(LDAPUserInfoImpl.ROLE_CTX, "ou=Roles,dc=jbpm,dc=org");
properties.setProperty(LDAPUserInfoImpl.USER_FILTER, "(uid={0})");
properties.setProperty(LDAPUserInfoImpl.ROLE_FILTER, "(cn={0})");
properties.setProperty(LDAPUserInfoImpl.IS_ENTITY_ID_DN, "true");
```

```
UserInfo ldapUserInfo = new LDAPUserInfoImpl(properties);

DefaultEscalatedDeadlineHandler handler = new DefaultEscalatedDeadlineHandler(emailProperties
handler.setUserInfo(ldapUserInfo);
```

- declaratively - create property file (jbpm.user.info.properties) with all required attributes, place it on the root of the classpath and declare LDAP user info implementation to be registered (see section Starting the human task server for deatils). Alternatively, location of jbpm.user.info.properties can be specified via system property -Djbpm.user.info.properties=FILE_LOCATION_ON_CLASSPATH

```
#ldap.bind.user=
#ldap.bind.pwd=

ldap.user.ctx=ou\=People,dc\=my-domain,dc\=com
ldap.role.ctx=ou\=Roles,dc\=my-domain,dc\=com

ldap.user.filter=(uid\={0})
ldap.role.filter=(cn\={0})
#ldap.role.members.filter=

#ldap.email.attr.id
#ldap.name.attr.id
#ldap.lang.attr.id
#ldap.member.attr.id
#ldap.user.attr.id
#ldap.role.attr.id

ldap.entity.id.dn=true
```

12.2.6. Starting the human task service as web application

Human task service can be started as web application to simplify deployment. As part of application configuration user can select number of settings to be applied on startup. Configuration is done via web.xml of jbpm-human-task-war application by setting init parameters of the HumanTaskServiceServlet.Following is a complete list of supported parameters and their meaning:

General settings

- task.persistence.unit : name of persistence unit that will be used to build EntityManagerFactory (default org.jbpm.task)

- user.group.callback.class : implementation of UserGroupCallback interface to be used to resolve users and groups (default DefaultUserGroupCallbackImpl)
- escalated.deadline.handler.class : implementation of EscalatedDeadlineHandler interface to be used to hadnle escalations and notifications (default DefaultEscalatedDeadlineHandler)
- user.info.class : implementation of UserInfo interface to be used to resolve user/group information such as email address, prefered language
- load.users : allows to specify location of a file that will be used to initially populate task server db with users. Accepts two types of files: MVEL and properties; must be suffixed with .mvel or .properties. Location of the file can be either on classpath (with prefix classpath:) or valid URL. NOTE: that with custom users files Administrator user must always be present
- load.groups : allows to specify location of a file that will be used to initially populate task server db with groups. Accepts two types of files: MVEL and properties;file must be suffixed with .mvel or .properties. Location of the file can be either on classpath (with prefix classpath:) or valid URL.

Transport settings

- active.config : main parameter that controls what transport is configured for Task Server, by default set to HornetQ and accepts Mina, HornetQ, JMS

Apache Mina

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

HornetQ

- hornetq.host : host/ip address used to bind HornetQ server (default localhost)
- hornetq.port : port used to bind HornetQ server (default 5153)

JMS

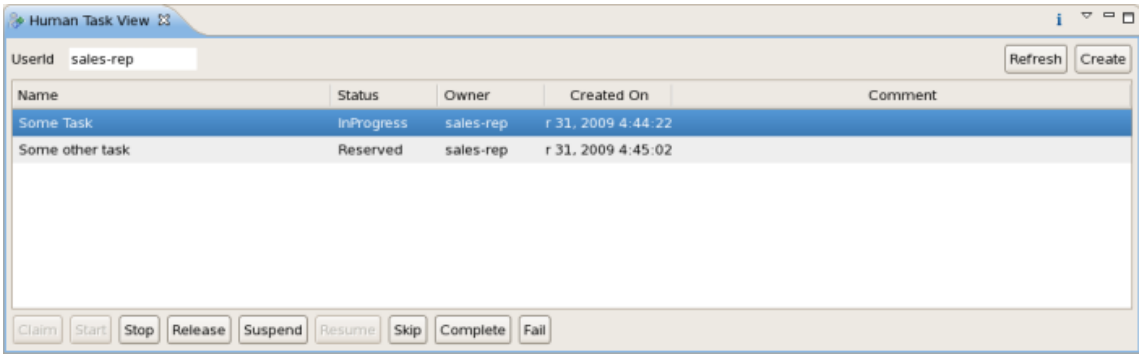
- JMSTaskServer.connectionFactory : JNDI name of QueueConnectionFactory to look up (no default)
- JMSTaskServer.transacted : boolean flag that indicates if jms session will be transacted or not (no default)
- JMSTaskServer.acknowledgeMode : acknowledgment mode (default DUPS_OK_ACKNOWLEDGE)
- JMSTaskServer.queueName : name of JMS queue (no default)

- JMSTaskServer.responseQueueName : name of JMS response queue (no default)

12.3. Human task clients

12.3.1. Eclipse demo task client

The Drools IDE contains a org.drools.eclipse.task plugin that allows you to test and/or debug processes using human tasks. In contains a Human Task View that can connect to a running task management component, request the relevant tasks for a particular user (i.e. the tasks where the user is either a potential owner or the tasks that the user already claimed and is executing). The life cycle of these tasks can then be executed, i.e. claiming or releasing a task, starting or stopping the execution of a task, completing a task, etc. A screenshot of this Human Task View is shown below. You can configure which task management component to connect to in the Drools Task preference page (select Window -> Preferences and select Drools Task). Here you can specify the url and port (default = 127.0.0.1:9123).



Notice that this task client only supports a (small) sub-set of the features provided the human task service. But in general this is sufficient to do some initial testing and debugging or demoing inside the Eclipse IDE.

12.3.2. Web-based task client in jBPM Console

The jBPM console also contains a task view for looking up task lists and managing the life cycle of tasks, task forms to complete the tasks, etc. See the chapter on the jBPM console for more information.

12.4. Human task persistence

The folowing entity relationship diagram (ERD) shows the persitent entities used by the Human Task service. (Clicking on the image below will take you to an enlarged view of the image.)

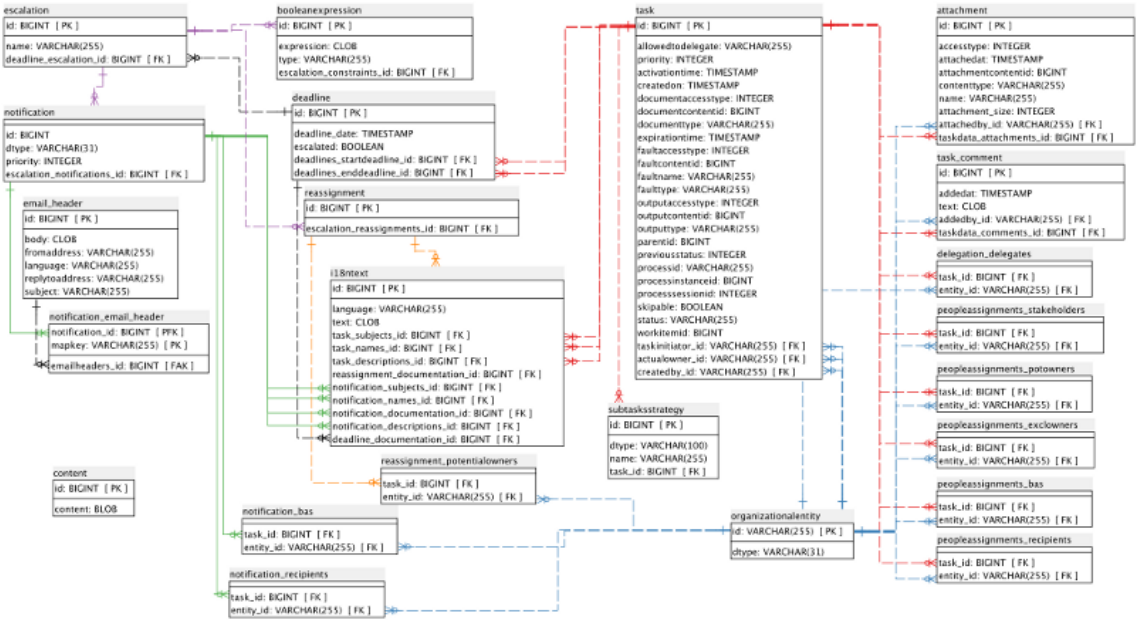


Figure 12.1. Human Task service data model

[images/Chapter-HumanTasks/human_task_schema.png]

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

- The `task` entity which represents the main information for a task. (See the righthand side of the ERD above.)
- The `deadline`, `escalation` and `notification` entities which represent deadlines and escalations for a task as well as any notifications associated with those deadlines. (See the lefthand side of the ERD above.)

Two other main entities in the data model are the `i18ntext` and `organizationalentity`.

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

The following paragraphs and tables describe the group of entities including and associated with the `task` entity. These entities are shown on the right hand side of the ERD. (See below for information about the `deadline`, `escalation` and `notification` group of entities).

The column “FK” in the tables below, indicates whether or not a column in a database table has a foreign key constraint on it. If the “Nullable” column is empty, then the described database table column *is* nullable.

While a number of foreign key columns of different tables are specified as non-nullable, many of these columns will simply contain the value `-1` or `0` if there is no associated entity.

12.4.1. Task related entities

The `task` entity contains much of the essential information for describing a task. Although a number of columns are not nullable, many of them are simply set to "-1" if the value used in the column hasn't been set by the task service.

Table 12.2. Task

Field	Description	Nullable	FK
`id`	The primary key of the task identity	NOT	
`priority`	The priority of the task	NOT	
`allowedtodelegate`	The group to whom this task may be delegated		
`status`	The status of the task		
`previousstatus`	The previous status of the task		
`actualowner_id`	The id of the organizational entity who owns the task	NOT	FK
`createdby_id`	The id of the organizational entity who created the task	NOT	FK
`createdon`	The timestamp describing when this task was created		
`activationtime`	The timestamp describing when this task was activated		
`expirationtime`	The timestamp describing when this task will expire		
`skipable`	Whether or not this task may be skipped	NOT	
`workitemid`	The id of the work item associated with this task (see jBPM core schema)	NOT	

Field	Description	Nullable	FK
`processinstanceid`	The id of the process instance associated with this task (see jBPM core schema)	NOT	
`documentaccesstype`	How a document associated with the task can be accessed		
`documenttype`	The type of data in the document		
`documentcontentid`	The id of the `content` entity containing the document data	NOT	
`outputaccesstype`	How the output document associated with the task can be accessed		
`outputtype`	The type of data in the output document		
`outputcontentid`	The id of the `content` entity containing the output document data	NOT	
`faultname`	The name of the fault generated, if a fault occurs		
`faultaccesstype`	How the document associated with the fault can be accessed		
`faulttype`	The type of data in the fault document		
`faultcontentid`	The id of the `content` entity containing the fault document data	NOT	
`parentid`	This is the id of the parent task	NOT	
`processid`	The name (id) of the associated process		
`processsessionid`	The id of the associated (knowledge) session	NOT	

Field	Description	Nullable	FK
`taskinitiator_id`	The id of the organizational entity who created the task	NOT	FK

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

Table 12.3. SubTasksStrategy

Field	Description	Nullable	FK
`id`	The primary key	NOT	
`dtype`	A discriminator column	NOT	
`name`	The name of the strategy		
`task_id`	The primary key of the associated `task`	NOT	FK

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

Table 12.4. OrganizationalEntity

Field	Description	Nullable
`id`	The primary key	NOT
`dtype`	The discriminator column	NOT

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

Table 12.5. Attachment

Field	Description	Nullable	FK
`id`	The primary key	NOT	
`name`	The (file) name of the attachment		
`accesstype`	How the attachment can be accessed		
`attachedat`	When the attachment was attached to the task		
`attachment_size`	The size (in bytes) of the attachment		

Field	Description	Nullable	FK
attachmentcontentid	The id of the content entity storing the raw data of the attachment	NOT	
contenttype	The MIME type of the attachment data		
attachedby_id	The id of the organizationalentity entity that attached the attachment	NOT	FK
taskdata_attachments_id	The id of the task entity to which this attachment belongs	NOT	FK

The task_comment entity describes comments added to tasks.

Table 12.6. task_comment

Field	Description	Nullable	FK
id	The primary key	NOT	
addedat	The timestamp of when the comment was added to the task		
text	The text of the comment		
addedby_id	The primary key of the associated organizationalentity entity	NOT	FK
taskdata_comments_id	The primary key of the associated task entity	NOT	FK

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

Table 12.7. delegation_delegates

Field	Description	Nullable	FK
task_id	The primary key of the associated task	NOT	FK

Field	Description	Nullable	FK
`entity_id`	The primary key of the associated `organizationalentity`	NOT	FK

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

Table 12.8. peopleassignments_stakeholders

Field	Description	Nullable	FK
`task_id`	The primary key of the associated `task` entity	NOT	FK
`entity_id`	The primary key of the associated `organizationalentity` entity	NOT	FK

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

Table 12.9. peopleassignments_potowners

Field	Description	Nullable	FK
`task_id`	The primary key of the associated `task` entity	NOT	FK
`entity_id`	The primary key of the associated `organizationalentity` entity	NOT	FK

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

Table 12.10. peopleassignments_exclowners

Field	Description	Nullable	FK
`task_id`	The primary key of the associated `task` entity	NOT	FK
`entity_id`	The primary key of the associated	NOT	FK

Field	Description	Nullable	FK
	`organizationalentity` entity		

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

Table 12.11. peopleassignments_bas

Field	Description	Nullable	FK
`task_id`	The primary key of the associated `task` entity	NOT	FK
`entity_id`	The primary key of the associated `organizationalentity` entity	NOT	FK

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

Table 12.12. peopleassignments_recipients

Field	Description	Nullable	FK
`task_id`	The primary key of the associated `task` entity	NOT	FK
`entity_id`	The primary key of the associated `organizationalentity` entity	NOT	FK

12.4.2. Deadline, Escalation and Notification related entities

The following paragraphs and tables describe the group of entities having to do with deadline, escalation, and notification information. These entities are shown on the left hand side of the ERD diagram above.

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

Table 12.13. deadline

Field	Description	Nullable	FK
`id`	The primary key	NOT	

Field	Description	Nullable	FK
deadline_date	The deadline date		
escalated	Whether or not the deadline has been escalated	NOT	
deadlines_startdeadline_id	The id of the associated task entity which uses this deadline as its start deadline.	NOT	FK
deadlines_enddeadline_id	The id of the associated task entity which uses this deadline as its end deadline.	NOT	FK

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

Table 12.14. escalation

Field	Description	Nullable	FK
id	The primary key	NOT	
name	The name of the escalation event		
deadline_escalation_id	The id of the associated deadline entity	NOT	FK

The booleanexpression entity represents an expression that evaluates to a boolean. These expressions are used in order to determine whether or not a constraint should be applied.

Table 12.15. booleanexpression

Field	Description	Nullable	FK
id	The primary key	NOT	
expression	The expression text		
type	The type of expression		
escalation_constraints_id	The id of the escalation entity for which this expression is used as a constraint	NOT	FK

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

Table 12.16. notification

Field	Description	Nullable	FK
`id`	The primary key	NOT	
`dtype`	The discriminator column	NOT	
`priority`	The priority of the notification	NOT	
`escalation_notification_id`	The id of the associated `escalation` entity	NOT	FK

The `email_header` entity describes an e-mail that will be sent as part of a notification.

Table 12.17. email_header

Field	Description	Nullable	
`id`	The primary key	NOT	
`fromaddress`	The e-mail address from which the e-mail is sent		
`replytoaddress`	The reply-to address used in the e-mail		
`language`	The language in which the e-mail is written		
`subject`	The subject of the e-mail		
`body`	The body of the e-mail		

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

Table 12.18. notification_email_header

Field	Description	Nullable	FK
`notification_id`	Together with the `mapkey`, this field is part of the primary key. This field refers	NOT	FK

Field	Description	Nullable	FK
	to the `notification` entity that the `email_header` is associated with.		
`mapkey`	Together with the `mapkey`, this field is part of the primary key. This field describes what the type is of the associated `email_header`.	NOT	
`emailheaders_id`	The id of the associated `email_header` entity	NOT	FK

The `reassignment` entity describes reassignments associated with escalations.

Table 12.19. reassignment

Field	Description	Nullable	FK
`id`	The primary key	NOT	
`escalation_reassignments_id`	The id of the associated `escalation` entity	NOT	FK

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

Table 12.20. reassignment_potentialowners

Field	Description	Nullable	FK
`task_id`	The primary key of the associated `reassignment` entity	NOT	FK
`entity_id`	The primary key of the associated `organizationalentity` entity	NOT	FK

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

Table 12.21. notification_bas

Field	Description	Nullable	FK
`task_id`	The primary key of the associated `notification` entity	NOT	FK
`entity_id`	The primary key of the associated `organizationalentity` entity	NOT	FK

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

Table 12.22. notification_recipients

Field	Description	Nullable	FK
`task_id`	The primary key of the associated `notification` entity	NOT	FK
`entity_id`	The primary key of the associated `organizationalentity` entity	NOT	FK

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

Table 12.23. content

Field	Description	Nullable
`id`	The primary key	NOT
`content`	The content data	NOT

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

Although all foreign keys are not nullable, they will be set to 0 if they are not being used.

Table 12.24. i18ntext

Field	Description	Nullable	FK
`id`	The primary key	NOT	
`language`	The language that the text is in.		

Field	Description	Nullable	FK
`text`	The text		
`task_subjects_id`	The id of the `task` entity for which this is a subject	NOT	FK
`task_names_id`	The id of the `task` entity for which this is a name	NOT	FK
`task_descriptions_id`	The id of the `task` entity for which this is a description	NOT	FK
`reassignment_documentation_id`	The id of the `reassignment` entity for which this is documentation	NOT	FK
`notification_subjects_id`	The id of the `notification` entity for which this is a subject	NOT	FK
`notification_names_id`	The id of the `notification` entity for which this is a name	NOT	FK
`notification_documentation_id`	The id of the `notification` entity for which this is documentation	NOT	FK
`notification_descriptions_id`	The id of the `notification` entity for which this is a description	NOT	FK
`deadline_documentation_id`	The id of the `deadline` entity for which this is documentation	NOT	FK

Chapter 13. Domain-specific processes

13.1. Introduction

jBPM provides the ability to create and use domain-specific task nodes in your business processes. This simplifies development when you're creating business processes that contain tasks dealing with other technical systems.

When using jBPM, we call these domain-specific task nodes "*custom work items*" or (custom) "*service nodes*". There are two separate aspects to creating and using custom work items:

- Adding a node with a custom work item to a process definition using the eclipse editor or jBPM designer.
- Creating a custom *work item handler* that the jBPM engine will use when executing the custom work item in a running process.

With regards to a BPMN2 process, custom work items are certain types of `<task>` nodes. In most cases, custom work items are `<task>` nodes in a BPMN2 process definition, although they can also be used with certain other task type nodes such as, among others, `<serviceTask>` or `<sendTask>` nodes.

Tip

When creating custom work items, it's important to separate the data associated with the work item, from how the work item should be handled. In other words, separate the *what* from the *how*. That means that custom work items should be:

- declarative (what, not how)
- high-level (no code)

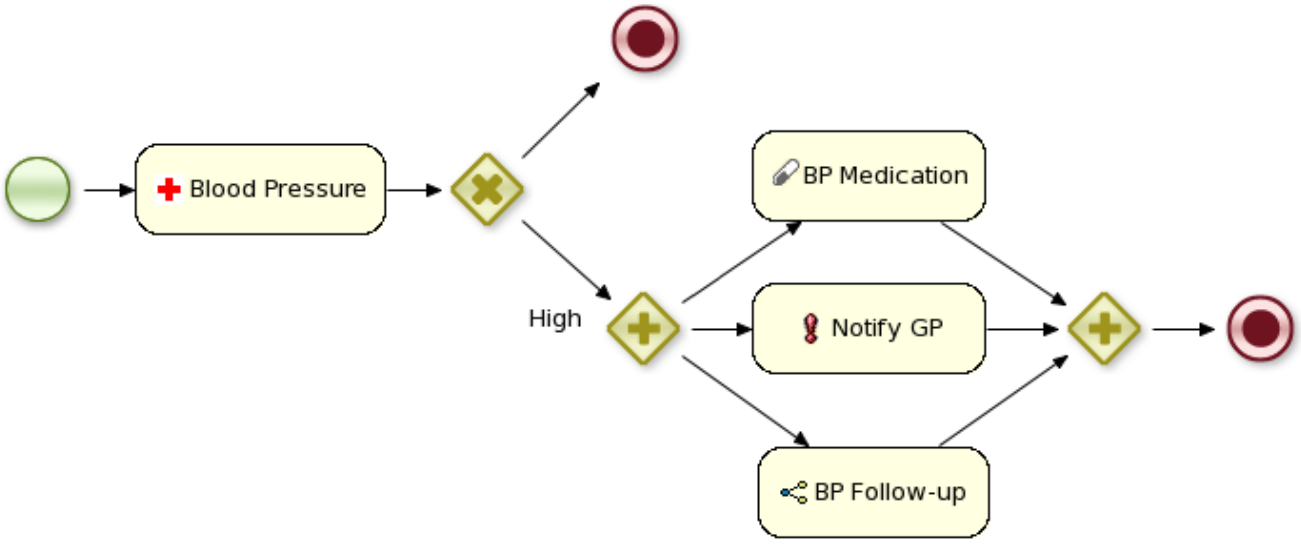
On the other hand, custom work item *handlers*, which are java classes, should be:

- procedural (how, not what)
- low-level (because it's code!)

Work item handlers should almost never contain any data.

Users can thus easily define their own set of domain-specific service nodes and integrate them with the process language. For example, the next figure shows an example of a healtchare-

related BPMN2 process. The process includes domain-specific service nodes for measuring blood pressure, prescribing medication, notifying care providers and following-up on the patient.



13.2. Overview

Before moving on to an example, this section explains what custom work items and custom work item handlers are.

13.2.1. Work Item Definitions

In short, we use the term *custom work item* when we're describing a node in your process that represents a domain-specific task and as such, contains extra properties and is handled by a `WorkItemHandler` implementation.

Because it's a domain-specific *task*, that means that a *custom work item* is equivalent to a `<task>` or `<task>`-type node in BPMN2. However, a *`WorkItem`* is also Java class instance that's used when a `WorkItemHandler` instance is called to complete the task or work item.

Depending on the BPMN2 editor you're using, you can create a custom work item definition in one of two ways:

- If you're using *Designer*, then this means creating a MVEL based definition and adding the definition in Designer itself. A description of this can be found in the *Support for domain-specific service nodes* section in the *Designer* chapter. Once this is done, a new service node will appear on the BPMN 2.0 palette.
- If you're using the *eclipse BPMN 2.0 modeler plugin* (which can be found *here* [http://eclipse.org/bpmn2-modeler/]), then you'll can modify the BPMN2 `<task>` or `<task>`-type element to work with `WorkItemHandler` implementations. See the *Adding custom task nodes* section in the *Eclipse BPMN 2.0 Plugin* chapter.

13.2.2. Work Item Handlers

A *work item handler* is a Java class used to execute (or abort) work items. That also means that the class implements the `org.kie.runtime.instance.WorkItemHandler` interface. While jBPM provides some custom `WorkItemHandler` instances (listed below), a Java developer with a minimal knowledge of jBPM can easily create a new work item handler class with it's own custom business logic.

Among others, jBPM offers the following `WorkItemHandler` implementations:

- In the `jbpm-bpmn2` module, `org.jbpm.bpmn2.handler` package:
  - ReceiveTaskHandler (for use with BPMN element `<receiveTask>`)
  - SendTaskHandler (for use with BPMN element `<sendTask>`)
  - ServiceTaskHandler (for use with BPMN element `<serviceTask>`)
- In the `jbpm-workitems` module, in various packages under the `org.jbpm.process.workitem` package:
  - ArchiveWorkItemHandler

There are a many more `WorkItemHandler` implementations present in the `jbpm-workitems` module. If you're looking for specific integration logic with Twitter, for example, we recommend you take a look at the classes made available there.

In general, a `WorkItemHandler`'s `.executeWorkItem(...)` and `.abortWorkItem(...)` methods will do the following:

1. Extract information about the task being executed (or aborted) from the `WorkItem` instance
2. Execute the necessary business logic. This might be mean interacting with a web service, database, or other technical component.
3. Inform the process engine that the work item has been completed (or aborted) by calling one of the following two mtehods on the `WorkItemManager` instance passed to the method:

```
WorkItemManager.completeWorkItem(long workItemId, Map<String, Object> results)
WorkItemManager.abortWorkItem(long workItemId)
```

In order to make sure that your custom work item handler is used for a particular process instance, it's necessary to register the work item handler before starting the process. This makes the engine aware of your `WorkItemHandler` so that the engine can use it for the proper node. For example:

```
ksession.getWorkItemManager().registerWorkItemHandler("Notification", new NotificationWorkItemH
```

The `ksession` variable above is a `StatefulKnowledgeSession` (and also a `KieSession`) instance. The example code above comes from the example that we will go through in the next session.

Tip

You can use different work item handlers for the same process depending on the system on which it runs: by registering different work item handlers on different systems, you can customize how a custom work item is processed on a particular system. You can also substitute mock `WorkItemHandler` instances when testing.

13.3. Example: Notifications

Let's start by showing you how to include a simple work item for sending notifications. A work item is defined by a unique name and includes additional parameters that describe the work in more detail. Work items can also return information after they have been executed, specified as results.

Our notification work item could be defined using a work definition with four parameters and no results. For example:

- Name: "Notification"
- Parameters:
  - From [String type]
  - To [String type]
  - Message [String type]
  - Priority [String type]

13.3.1. The Notification Work Item Definition

13.3.1.1. Creating the work item definition

In our example we will create a MVEL work item definition that defines a "Notification" work item. Using MVEL is the default way to This file will be placed in the project classpath in a directory called `META-INF`. The work item configuration file for this example, `MyWorkDefinitions.wid`, will look like this:

```
import org.drools.core.process.core.datatype.impl.type.StringDataType;
[
// the Notification work item
```

```
  [
  "name" : "Notification",
  "parameters" : [
  "Message" : new StringDataType(),
  "From" : new StringDataType(),
  "To" : new StringDataType(),
  "Priority" : new StringDataType(),
  ],
  "displayName" : "Notification",
  "icon" : "icons/notification.gif"
  ]

  ]
```

The project directory structure could then look something like this:

```
project/src/main/resources/META-INF/MyWorkDefinitions.wid
```

We also want to *add* a specific icon to be used in the process editor with the work item. To add this, you will need `.gif` or `.png` images with a pixel size of 16x16. We put them in a directory outside of the `META-INF` directory, for example, here:

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

13.3.1.2. Registering the work definition

The jBPM eclipse editor uses the configuration mechanisms supplied by Drools to register work item definition files. That means adding a `drools.workDefinitions` property to the `drools.rulebase.conf` file in the `META-INF`.

The `drools.workDefinitions` property represents a list of files containing work item definitions, separated usings spaces. If you want to *exclude* all other work item definitions and only use your definition, you could use the following:

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

However, if you only want to add the newly created node definition to the existing palette nodes, you can define the `drools.workDefinitions` property as follows:

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

We recommended that you use the extension `.wid` for your own definitions of domain specific nodes. The `.conf` extension used with the default definition file, `WorkDefinitions.conf`, for backward compatibility reasons.

13.3.1.3. Using your new work item in your processes

We've created our work item definition and configured it, so now we can start using it in our processes. The process editor contains a separate section in the palette where the different service nodes that have been defined for the project appear.

Using drag and drop, a notification node can be created inside your process. The properties can be filled in using the properties view.

Besides any custom properties, the following three properties are available for all work items:

1. `Parameter Mapping`: Allows you to map the value of a variable in the process to a parameter of the work item. This allows you to customize the work item based on the current state of the actual process instance (for example, the priority of the notification could be dependent of some process-specific information).

2. `Result Mapping`: Allows you to map a result (returned once a work item has been executed) to a variable of the process. This allows you to use results in the remainder of the process.

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

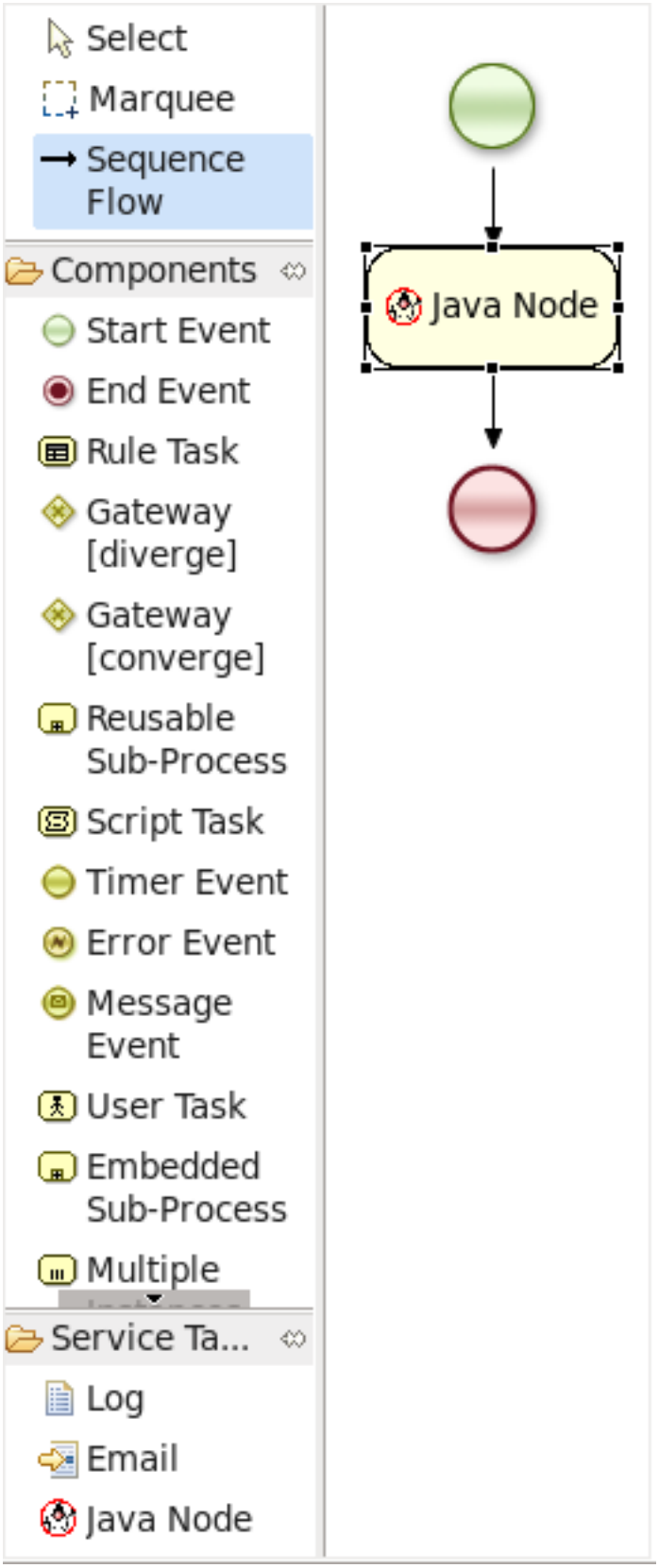
Here is an example that creates a domain specific node to execute Java, asking for the class and method parameters. It includes a custom `java.gif` icon and consists of the following files and resulting screenshot:

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

]
```

```
// located in: project/src/main/resources/META-INF/drools.rulebase.conf
//
  drools.workDefinitions = JavaNodeDefinition.wid WorkDefinitions.conf

// icon for java.gif located in:
//  project/src/main/resources/icons/java.gif
```


Select
Marquee
Sequence Flow
Components
Start Event
End Event
Rule Task
Gateway [diverge]
Gateway [converge]
Reusable Sub-Process
Script Task
Timer Event
Error Event
Message Event
User Task
Embedded Sub-Process
Multiple
Service Ta...
Log
Email
Java Node
Java Node

13.3.2. The `NotificationWorkItemHandler`

13.3.2.1. Creating a new work item handler

Once we've created our `Notification` work item definition (see the sections above), we can then create a custom implementation of a *work item handler* that will contain the logic to send the notification.

In order to execute our *Notification* work items, we first create a `NotificationWorkItemHandler` that implements the `WorkItemHandler` interface:

```
package com.sample;

import org.kie.api.runtime.process.WorkItem;
import org.kie.api.runtime.process.WorkItemHandler;
import org.kie.api.runtime.process.WorkItemManager;

public class NotificationWorkItemHandler implements WorkItemHandler {

  public void executeWorkItem(WorkItem workItem, WorkItemManager manager) {
    // extract parameters
    String from = (String) workItem.getParameter("From");
    String to = (String) workItem.getParameter("To");
    String message = (String) workItem.getParameter("Message");
    String priority = (String) workItem.getParameter("Priority");

    // send email
   EmailService service = ServiceRegistry.getInstance().getEmailService();
    service.sendEmail(from, to, "Notification", message);

    // notify manager that work item has been completed
    manager.completeWorkItem(workItem.getId(), null);
  }

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

}
```

1. The `ServiceRegistry` class is simply a made-up class that we're using for this example. In your own `WorkItemHandler` implementations, the code containing your domain-specific logic would go here.
2. Notifying the *`WorkItemManager`* instance when your a work item has been completed is crucial. For many synchronous actions, like sending an email in this

case, the `WorkItemHandler` implementation will notify the `WorkItemManager` in the `executeWorkItem(...)` method.

This `WorkItemHandler` sends a notification as an email and then notifies the WorkItemManager that the work item has been completed.

Note that not all work items can be completed directly. In cases where executing a work item takes some time, execution can continue *asynchronously* and the work item manager can be notified later.

In these situations, it might also be possible that a work item is *aborted* before it has been completed. The `WorkItemHandler.abortWorkItem(...)` method can be used to specify how to abort such work items.

Tip

Remember, if the `WorkItemManager` is not notified about the completion, the process engine will never be notified that your service node has completed.

13.3.2.2. Registering the work item handler

`WorkItemHandler` instances need to be registered with the `WorkItemManager` in order to be used. In this case, we need to register an instance of our `NotificationWorkItemHandler` in order to use it with our process containing a `Notification` work item. We can do that like this:

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
ksession.getWorkItemManager().registerWorkItemHandler(
  "Notification",                                                    (1)
  new NotificationWorkItemHandler()                                  (2)
);
```

1. This is the drools name of the `<task>` (or other task type) node. See below for an example.
2. This is the instance of our custom work item handler instance!

If we were to look at the BPMN2 syntax for our process with the `Notification` process, we would see something like the following example. Note the use of the `tns:taskName` attribute in the `<task>` node. This is necessary for the `WorkItemManager` to be able to see which `WorkItemHandler` instance should be used with which task or work item.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"
```

```
            xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
               xs:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
 BPMN20.xsd"
...
            xmlns:tns="http://www.jboss.org/drools">

...

    <process  isExecutable="true"  id="myCustomProcess"  name="Domain-Specific
 Process" >

...

    <task id="_5" name="Notification Task" tns:taskName="Notification" >

...
```

Tip

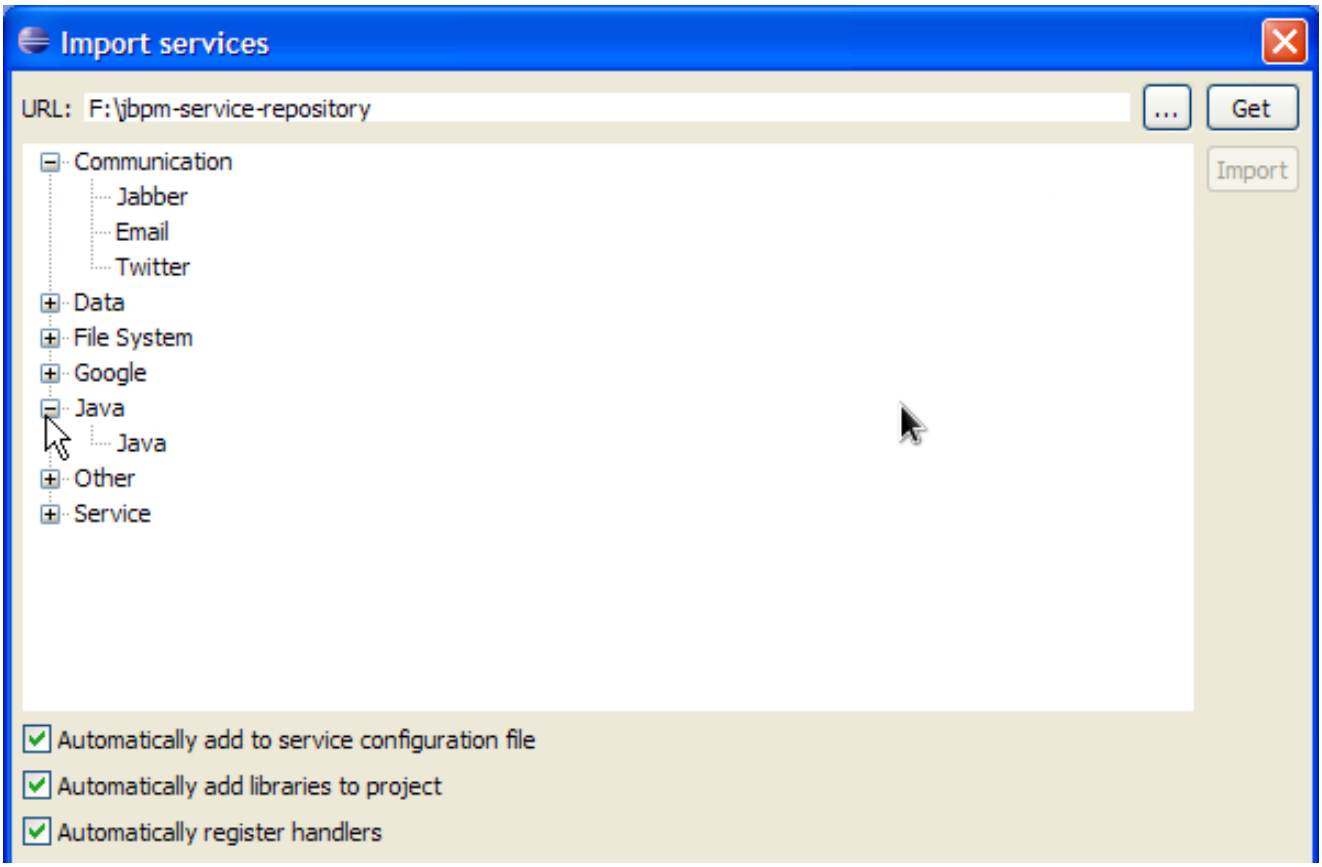
Different work item handlers could be used depending on the context. For example, during testing or simulation, it might not be necessary to actually execute the work items. In this case specialized dummy work item handlers could be used during testing.

13.4. Service repository

A lot of these domain-specific services are generic, and can be reused by a lot of different users. Think for example about integration with Twitter, doing file system operations or sending email. Once such a domain-specific service has been created, you might want to make it available to other users so they can easily import and start using it.

A service repository allows you to import services by browsing the repository looking for services you might need and importing these services into your workspace. These will then automatically be added to your palette and you can start using them in your processes. You can also import additional artefacts like for example an icon, any dependencies you might need, a default handler that will be used to execute the service (although you're always free to override the default, for example for testing), etc.

To browse the repository, open the wizard to import services, point it to the right location (this could be to a directory in your file system but also a public or private URL) and select the services you would like to import. For example, in Eclipse, right-click your project that contains your processes and select "Configure ... -> Import jBPM services ...". This will open up a repository browser. In the URL field, fill in the URL of your repository (see below for the URL of the public jBPM repository that hosts some common service implementations out-of-the-box), or use the "..." button to browse to a folder on your file system. Click the Get button to retrieve the contents of that repository.



Select the service you would like to import and then click the Import button. Note that the Eclipse wizard allows you to define whether you would like to automatically configure the service (so it shows up in the palette of your processes), whether you would also like to download any dependencies that might be needed for executing the service and/or whether you would like to automatically register the default handler, so make sure to mark the right checkboxes before importing your service (if you are unsure what to do, leaving all check boxes marked is probably best).

After importing your service, (re)open your process diagram and the new service should show up in your palette and you can start using it in your process. Note that most services also include documentation on how to use them (e.g. what the different input and output parameters are) when you select them browsing the service repository.

Click on the image below to see a screencast where we import the twitter service in a new jBPM project and create a simple process with it that sends an actual tweet. Note that you need the necessary twitter keys and secrets to be able to programatically send tweets to your twitter account. How to create these is explained *here* [http://people.redhat.com/kverlaen/repository/Twitter/], but once you have these, you can just drop them in your project using a simple configuration file.

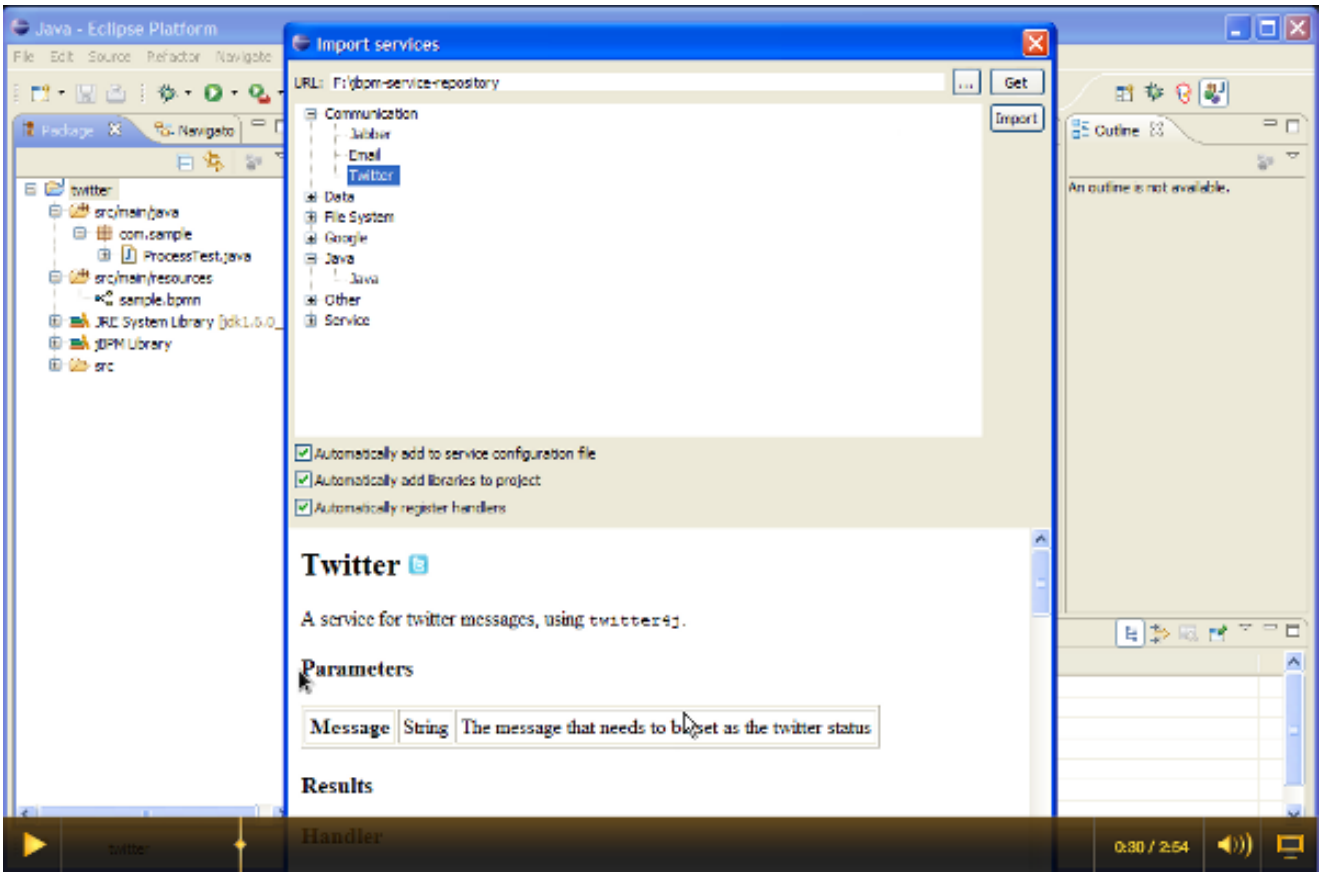


Figure 13.1.

[http://people.redhat.com/kverlaen/twitter-repository.swf]

13.4.1. Public jBPM service repository

We are building a public service repository that contains predefined services that people can use out-of-the-box if they want to:

http://people.redhat.com/kverlaen/repository

This repository contains some integrations for common services like Twitter integration or file system operations that you can import. Simply point the import wizard to this URL to start browsing the repository.

If you have an implementation of a common service that you would like to contribute to the community, do not hesitate to contact someone from the development team. We are always looking for contributions to extend our repository.

13.4.2. Setting up your own service repository

You can set up your own service repository and add your own services by creating a configuration file that contains the necessary information (this is an extended version of the normal work

definition configuration file as described earlier in this chapter) and putting the necessary files (like an icon, dependencies, documentation, etc.) in the right folders.

The extended configuration file contains the normal properties (like name, parameters, results and icon), with some additional ones. For example, the following extended configuration file describes the Twitter integration service (as shown in the screencast above):

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

- The icon property should refer to a file with the given file name in the same folder as the extended configuration file (so it can be downloaded by the import wizard and used in the process diagrams). Icons should be 16x16 GIF files.
- The category property defines the category this service should be placed under when browsing the repository.
- The defaultHandler property defines the default handler implementation (i.e. the Java class that implements the `WorkItemHandler` interface and can be used to execute the service). This can automatically be registered as the handler for that service when importing the service from the repository.
- The documentation property defines a documentation file that describes what the service does and how it works. This property should refer to a HTML file with the given name in the same folder as the extended configuration file (so it can be shown by the import wizard when browsing the repository).

- The dependencies property defines additional dependencies that are necessary to execute this service. This usually includes the handler implementation jar, but could also include additional external dependencies. These dependencies should also be located on the repository on the given location (relative to the folder where the extended configuration file is located), so they can be downloaded by the import wizard when importing the service.

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

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

You can create your own hierarchical structure, because if one of those folders also contains an `index.conf` file, that will be used to scan additional sub-folders. Note that the hierarchical structure of the repository is not shown when browsing the repository using the import wizard, as the category property in the configuration file is used for that.

Chapter 14. Exception Management

14.1. Overview

This chapter will describe how to deal with unexpected behavior in your business processes using both BPMN2 and technical mechanisms.

The first section (Introduction) will define and explain the types of exceptions that can happen or be used in a business process (Business Exceptions and Technical Exceptions).

The next section will explain Technical Exceptions: we'll go through an example that uses both BPMN2 and `WorkItemHandler` implementations in order to isolate and handle exceptions caused by a technical component. We will also explain how to modify the example to suit other use cases.

14.2. Introduction

What happens to a business process when something unexpected happens during the process? Most of the time, when you create and design a new process definition, you'll begin by describing the *normative* or desirable behaviour. However, a process definition that only describes all of the normal tasks and their execution order is incomplete.

The next step is to think about what might go *wrong* when the business process is run. What would happen if any of the human or technical actors in the process do *not* respond in unexpexected ways? Will any of the technical systems that the process interacts with return unexpected results -- or not return any results at all?

Deviations from the normative or "happy" flow of a business process are called *exceptions*. In some cases, exceptions might not be that unusual, such as trying to debit an empty bank account. However, some processes might contain many complex situations involving exceptions, all of which must be handled correctly.

Note

The rest of chapter assumes that you know how to create custom `<task>` nodes and how to implement and register `WorkItemHandler` implementations. More information about these topics can be found in the *Domain-specific processes* chapter.

14.3. Business Exceptions

Business Exceptions are exceptions that are designed and managed in the BPMN2 specification of a business process. In other words, Business Exceptions are exceptions which happen at the process or workflow level, and are not related to the technical components.

Many of the elements in BPMN2 related to Business Exceptions are related to *Compensation* and *Business Transactions*. Compensation, in particular, is complexer than many other parts of the BPMN2 specfication.

Full support for *compensation* and *business transactions* is expected with the release of jBPM 6.1 or 6.2. Once that has been implemented, this section will contain more information about using those BPMN2 features with jBPM.

14.3.1. Business Exceptions elements in BPMN2

The following attempts to briefly describe Compensation and Business Transaction related elements in BPMN2. For more complete information about these elements and their uses, see the BPMN2 specification, Bruce Silver's book `BPMN Method and Style` or any of the other available books about the use of BPMN2.

Table 14.1. BPMN2 Exception Handling Elements

BPMN2 Element types	Description
Errors and Error Events	Error Events can be used to signal when a process has encountered an unexpected situation: signalling an error is often called *throwing* an error. Boundary Error Events in a different part of the process can then be used to *catch* the error and initiate a sequence of activities to handle the exception. Errors themselves can be extended with extra information that is passed from the throwing to catching event. This is done with the use of an Item Definition.
Business Transactions	A Business Transaction in BPMN2 is a subprocess which can be used with *compensation*. Grouping activities in a Business Transaction lets the process designer easily add exception handling to specific activities in the subprocess. Using a Business Transaction guarantees that all activities in the transaction will have completed successfully if the Business Transaction completes successfully. When a Business Transaction is interrupted or otherwise not completed successfully, there is a guarantee that all activities in

BPMN2 Element types	Description
	the Business Transaction that have been initiated will be compensated if compensating activities are defined for those activities.
Compensation	Exception handling activities *associated* with the normal activies in a Business Transaction are triggered by *Compensation Events*. Compensation Events may only be used within Business Transactions. There are 3 types of compensation events: Intermediate (a.k.a. Boundary) (catch) events, Start (catch) events, and Intermediate or End (throw) events. Compensation Boundary (catch) events are attached to activites (e.g. tasks) that could cause an exception. They may only be attached to activites inside a Business Transaction. If a Business Transaction fails, possibly because of the failure of one of the activities inside it, then the activities associated with Boundary (catch) events will be triggered. Only *one* activity or node may be associated with a Compensation Boundary Event! Start (catch) events are used when defining an *Compensation Event SubProcess* inside a Business Transaction. Compensation Event SubProcesses are often used when a subprocess is needed to compensate for the Business Transaction as a whole (as opposed to defining compensating activities per node in the Business Transaction. This subprocess is triggered when a Business Transaction fails, just like activities attached to Compensation Boundary (catch) events. Compensation Intermediate and End events are used within Business Transactions in order to throw Compensation Events. Often, logic in the Business Transaction subprocesses will determine whether or not

BPMN2 Element types	Description
	the Business Transaction has succeeded or failed. If the subprocess has failed, then the process will proceed to an Intermediate or End Compensation Event in order to trigger compensation for the Business Transaction subprocess.
Cancel Events	Cancel Events trigger *cancellation* of a Business Transaction and can thus only be used with a Business Transaction. When a Cancel Event is thrown, this indicates that the Business Transaction should be cancelled. Entities involved in the Business Transaction are then informed (via a *TransactionProtocol Cancel Message*) that the Business Transaction has been cancelled. *Cancellation* of a Business Transaction implicitly triggers *compensation* of the Business Transaction. See the sources mentioned above for the differences between Error Events (abortion of a process), Cancel Events (cancellation) and Compensate Events (compensation).

14.4. Technical Exceptions

Technical exceptions happen when a technical component of a business process acts in an unexpected way. When using Java based systems, this often results in a literal Java Exception being thrown by the system.

Technical components used in a process can fail in a way that can not be described using BPMN2. In this case, it's important to handle these exceptions in expected ways.

The following types of code might throw exceptions:

- Any code that is present in the process definition itself
- Any code that is executed during a process and is not part of jBPM
- Any code that interacts with a technical component outside of the process engine

However, those are somewhat abstract defintions. We can narrow down the places at which an exception might be thrown. Technical exceptions can occur at the following points:

1. Code present in `<scriptTask>` nodes or in the jbpm-specific `<onEntry>` and `<onExit>` elements

2. Code executed in `WorkItemHandlers` associated with `<task>` and task-type nodes

It is *much easier* to ensure correct exception handling for `<task>` and other task-type nodes that use `WorkItemHandler` implementations, than for code executed directly in a `<scriptTask>`.

Exceptions thrown by `<scriptTask>` can cause the process to fail in an unrecoverable fashion. While there are certain things that you can do to contain the damage, a process that has failed in this way can not be restarted or otherwise recovered. This also applies for other nodes in a process definition that contain script code in the node definition, such as the `<onEntry>` and `<onExit>` elements.

When jBPM engine does throw an exception generated by the code in a `<scriptTask>` the exception thrown is a special Java exception called the `WorkflowRuntimeException` that contains information about the process.

Warning

Again, exceptions generated by a `<scriptTask>` node (and other nodes containing script code) will leave the process *unrecoverable*. In fact, often, the code that starts the process itself will end up throwing the exception generated by the business process, without returning a reference to the process instance.

For this reason, it's important to limit the scope of the code in these nodes to operations dealing with process variables. Using a `<scriptTask>` to interact with a different technical component, such as a database or web service has *significant risks* because any exceptions thrown will corrupt or abort the process.

`<task>` nodes, `<serviceTask>` nodes and the rest of the `task`-type nodes are explictly meant for interacting with other systems -- not `<scriptTask>` nodes! Use `<task>`-type nodes to interact with other technical components.

14.4.1. Handling exceptions in `WorkItemHandler` instances

`WorkItemHandler` classes are used when your process interacts with other technical systems. For an introduction to them and how to use them in processes, please see the *Domain-specific processes* chapter.

While you can build exception handling into your own `WorkItemhandler` implementations, there are also two “handler decorator” classes that you can use to *wrap* a `WorkItemhandler` implementation.

These two wrapper classes include logic that is executed when an exception is thrown during the execution (or abortion) of a work item.

Table 14.2. Exception Handling `WorkItemHandler` wrapper classes

Decorator classes in the `org.jbpm.bpmn2.handler` package	Description
`SignallingTaskHandlerDecorator`	This class wraps an existing `WorkItemHandler` implementation. When the `.executeWorkItem(...)` or `.abortWorkItem(...)` methods of the original `WorkItemHandler` instance throw an exception, the `SignallingTaskHandlerDecorator` will catch the exception and signal the process instance using a configurable event type. The exception thrown will be passed as part of the event. This functionality can be used to signal an *Event SubProcess* defined in the process definition.
`LoggingTaskHandlerDecorator`	This class reacts to all exceptions thrown by the `.executeWorkItem(...)` or `.abortWorkItem(...)` `WorkItemHandler` methods by logging the errors. It also saves any exceptions thrown so to an internal list so that they can be retrieved later for inspection or further logging. Lastly, the content and format of the message logged upon an exception are configurable.

While the two classes described above should cover most cases involving exception handling, a Java developer with some experience with jBPM should be able to create a `WorkItemHandler` that executes custom code upon an exception.

If you do decide to write a custom `WorkItemHandler` that includes exception handling logic, keep the following checklist in mind:

1. Are you catching all possible exceptions that you want to (and no more, or less)?
2. Are you making sure to either complete or abort the work item after an exception has been caught? If not, are there mechanisms to retry the process later? Or are incomplete process instances acceptable?

3. What other actions should be taken when an exception is caught? Do you want to simply log the exception, or is it also important to interact with other technical systems? Do you want to trigger a (BPMN2) subprocess that will handle the exception?

Important

When you use the `WorkItemManager` to signal that the work item has been completed or aborted, make sure to do that *after you've sent any signals* to the process instance. Depending on how you've defined your process, calling `WorkItemManager.completeWorkItem(...)` or `WorkItemManager.abortWorkItem(...)` will trigger the completion of the process instance. This is because the these methods trigger the jBPM process engine to continue the process flow.

In the next section, we'll describe an example that uses the `SignallingTaskHandlerDecorator` to signal an *event subprocess* when a work item handler throws an exception.

14.5. Technical Exception Examples

14.5.1. Example: service task handlers

We'll go through one example in this section, and then look quickly at how you can change it to get the behavior you want. The example involves an `<error>` event that's caught by an *(Error) Event SubProcess*.

When an *Error Event* is thrown, the containing process will be interrupted. This means that after the process flow attached to the error event has executed, the following will happen:

1. process execution will stop, and no other parts of the process will execute
2. the process instance will end up in an aborted state (instead of completed)

The example we'll go through contains an `<error>`, but at the end of the secion, we'll show how you can change the process to use a `<signal>` instead.

Tip

The code and BPMN2 process definition shown in the next section are available in the `jbpm-examples` module. See the `org.jbpm.examples.exceptions.ExceptionHandlingErrorExample` class for the java code. The BPMN2 process definition is available in the `exceptions/ExceptionHandlingWithError.bpmn2` file in the `src/main/resources` directory of the `jbpm-examples` module.

14.5.1.1. BPMN2 configuration

Let's look at the BPMN2 process definition first. Besides the definition of the process, the BPMN2 elements defined before the actual process definition are also important. Here's an image of the BPMN2 process that we'll be using in the example:

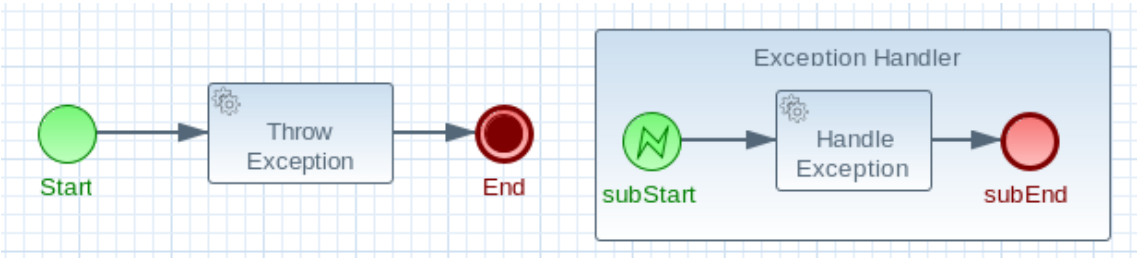


Figure 14.1.

The BPMN2 process fragment below is part of the process shown above, and contains some notes on the different BPMN2 elements.

Note

If you're viewing this on a web browser, you may need to widen your browser window in order to see the "callout" or note numbers on the righthand side of the code.

```
<itemDefinition id="_stringItem" structureRef="java.lang.String"/>                (1)
<message id="_message" itemRef="_stringItem"/>                                    (2)

                              <interface               id="_serviceInterface"
name="org.jbpm.examples.exceptions.service.ExceptionService">
   <operation id="_serviceOperation" name="throwException">
     <inMessageRef>_message</inMessageRef>                                        (2)
   </operation>
 </interface>

<error id="_exception" errorCode="code" structureRef="_exceptionItem"/>           (3)

                            <itemDefinition              id="_exceptionItem"
structureRef="org.kie.api.runtime.process(4).WorkItem"/>
<message id="_exceptionMessage" itemRef="_exceptionItem"/>                        (4)

                         <interface            id="_handlingServiceInterface"
name="org.jbpm.examples.exceptions.service.ExceptionService">
   <operation id="_handlingServiceOperation" name="handleException">
     <inMessageRef>_exceptionMessage</inMessageRef>                               (4)
```

```
   </operation>
 </interface>

    <process  id="ProcessWithExceptionHandlingError"  name="Service  Process"
isExecutable="true" processType="Private">
   <!-- properties -->
   <property id="serviceInputItem" itemSubjectRef="_stringItem"/>
   <property id="exceptionInputItem" itemSubjectRef="_exceptionItem"/>

   <!-- main process -->
   <startEvent id="_1" name="Start" />
       <serviceTask  id="_2"  name="Throw  Exception"  implementation="Other"
operationRef="_serviceOperation">

   <!-- rest of the serviceTask element and process definition... -->

   <subProcess id="_X" name="Exception Handler" triggeredByEvent="true" >
     <startEvent id="_X-1" name="subStart">
       <dataOutput id="_X-1_Output" name="event"/>
       <dataOutputAssociation>
         <sourceRef>_X-1_Output</sourceRef>
        <targetRef>exceptionInputItem</targetRef>
       </dataOutputAssociation>
      <errorEventDefinition id="_X-1_ED_1" errorRef="_exception" />
     </startEvent>

     <!-- rest of the subprocess definition... -->

   </subProcess>

 </process>
```

1. This `<itemDefinition>` element defines a data structure that we then use in the `serviceInputItem` property in the process.
2. This `<message>` element (1rst reference) defines a *message* that has a String as its content (as defined by the `<itemDefintion>` element on line above). The `<interface>` element below it refers to it (2nd reference) in order to define what type of content the service (defined by the `<interface>`) expects.
3. This `<error>` element (1rst reference) defines an error for use later in the process: an *Event SubProcess* is defined that is triggered by this *error* (2nd reference). The content of the error is defined by the `<itemDefintion>` element defined below the `<error>` element.
4. This `<itemDefintion>` element (1rst reference) defines an item that contains a `WorkItem` instance. The `<message>` element (2nd reference) then defines a *message* that uses this *item definition* to define its content. The `<interface>` element below that refers to the `<message>` definition (3rd reference) in order to define the type of content that the service expects.

In the process itself, a `<property>` element (4th reference) is defined as having the content defined by the initial `<itemDefintion>`. This is helpful because it means that the *Event SubProcess* can then store the *error* it receives in that property (5th reference).

Caution

When you're using a `<serviceTask>` to call a Java class, make sure to double check the class name in your BPMN2 definition! A small typo there can cost you time later when you're trying to figure out what went wrong.

14.5.1.2. `SignallingTaskHandlerDecorator` and `WorkItemHandler` configuration

Now that BPMN2 process definition is (hopefully) a little clearer, we can look at how to set up jBPM to take advantage of the above BPMN2.

In the (BPMN2) process definition above, we define two different `<serviceTask>` activities. The `org.jbpm.bpmn2.handler.ServiceTaskHandler` class is the default task handler class used for `<serviceTask>` tasks. If you don't specify a `WorkItemHandler` implementation for a `<serviceTask>`, the `ServiceTaskHandler` class will be used.

In the code below, you'll see that we actually wrap or decorate the `ServiceTaskHandler` class with a `SignallingTaskHandlerDecorator` instance. We do this in order to define the what happens when the `ServiceTaskHandler` throws an exception.

In this case, the `ServiceTaskHandler` will throw an exception because it's configured to call the `ExceptionService.throwException` method, which throws an exception. (See the `_handlingServiceInterface` `<interface>` element in the BPMN2.)

In the code below, we also configure which (error) event is sent to the process instance by the `SignallingTaskHandlerDecorator` instance. The `SignallingTaskHandlerDecorator` does this when an exception is thrown in a *task*. In this case, since we've defined an `<error>` with the *error code* “code” in the BPMN2, we set the signal to `Error-code`.

Important

When signalling the jBPM process engine with an event of some sort, you should keep in mind the rules for signalling process events.

- Error events can be signalled by sending an "Error-" + <the `errorCode` attribute value> value to the session.
- Signal events can be signalled by sending the name of the signal to the session.

```
import java.util.HashMap;
import java.util.Map;

import org.jbpm.bpmn2.handler.ServiceTaskHandler;
import org.jbpm.bpmn2.handler.SignallingTaskHandlerDecorator;
import org.jbpm.examples.exceptions.service.ExceptionService;
import org.kie.api.KieBase;
import org.kie.api.io.ResourceType;
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.internal.builder.KnowledgeBuilder;
import org.kie.internal.builder.KnowledgeBuilderFactory;
import org.kie.internal.io.ResourceFactory;

public class ExceptionHandlingErrorExample {

    public static final void main(String[] args) {
        runExample();
    }

    public static ProcessInstance runExample() {
        KieSession ksession = createKieSession();


        String eventType = "Error-code";                                    (1)
        SignallingTaskHandlerDecorator signallingTaskWrapper                (2)
            = new SignallingTaskHandlerDecorator(ServiceTaskHandler.class, eventType);
        signallingTaskWrapper.setWorkItemExceptionParameterName(ExceptionServic(3)e.exceptionPar
                ksession.getWorkItemManager().registerWorkItemHandler("Service
 Task", signallingTaskWrapper);

        Map<String, Object> params = new HashMap<String, Object>();
        params.put("serviceInputItem", "Input to Original Service");
        ProcessInstance processInstance = ksession.startProcess("ProcessWithExceptionHandlingEr

        return processInstance;
    }

    private static KieSession createKieSession() {
      KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
                kbuilder.add(ResourceFactory.newClassPathResource("exceptions/
ExceptionHandlingWithError.bpmn2"), ResourceType.BPMN2);
        KieBase kbase = kbuilder.newKnowledgeBase();
        return kbase.newKieSession();
    }
```

1. Here we define the name of the event that will be sent to the process instance if the wrapped `WorkItemHandler` implementation throws an exception. The `eventType` string is used when instantiating the `SignallingTaskHandlerDecorator` class.
2. Then we construct an instance of the `SignallingTaskHandlerDecorator` class. In this case, we simply give it the *class name* of the `WorkItemHandler` implementation class to instantiate, but another constructor is available that we can pass an *instance* of a `WorkItemHandler` implementation to (necessary if the `WorkItemHandler` implementation does not have a no-argument constructor).
3. When an exception is thrown by the wrapped `WorkItemHandler`, the `SignallingTaskHandlerDecorator` saves it as a parameter in the `WorkItem` instance with a parameter name that we configure the `SignallingTaskHandlerDecorator` to give it (see the code below for the `ExceptionService`).

14.5.1.3. `ExceptionService` setup and configuration

In the BPMN2 process definition above, a service interface is defined that references the `ExceptionService` class:

```
<interface id="_handlingServiceInterface" name="org.jbpm.examples.exceptions.service.ExceptionS
    <operation id="_handlingServiceOperation" name="handleException">
```

In order to fill in the blanks a little bit, the code for the `ExceptionService` class has been included below. In general, you can specify any Java class with the default or an other no-argument constructor and have it executed during a `<serviceTask>`

```
public class ExceptionService {

  public static String exceptionParameterName = "my.exception.parameter.name";

  public void handleException(WorkItem workItem) {
          System.out.println(  "Handling  exception  caused  by  work  item
 '" + workItem.getName() + "' (id: " + workItem.getId() + ")");

    Map<String, Object> params = workItem.getParameters();
    Throwable throwable = (Throwable) params.get(exceptionParameterName);
    throwable.printStackTrace();
  }

  public String throwException(String message) {
      throw new RuntimeException("Service failed with input: " + message );
  }

  public static void setExceptionParameterName(String exceptionParam) {
      exceptionParameterName = exceptionParam;
```

```
  }

}
```

14.5.1.4. Changing the example to use a `<signal>`

In the example above, the thrown Error Event interrupts the process: no other flows or activities are executed once the Error Event has been thrown.

However, when a *Signal Event* is processed, the process will continue after the *Signal Event SubProcess* (or whatever other activities that the Signal Event triggers) has been executed. Furthermore, this implies that the the process will *not* end up in an aborted state, unlike a process that throws an Error Event.

In the process above, we use the `<error>` element in order to be able to use an Error Event:

```
<error id="_exception" errorCode="code" structureRef="_exceptionItem"/>
```

When we want to use a Signal Event instead, we remove that line and use a `<signal>` element:

```
<signal id="exception-signal" structureRef="_exceptionItem"/>
```

However, we must also change all references to the "`_exception`" `<error>` so that they now refer to the "`exception-signal`" `<signal>`.

That means that the `<errorEventDefintion>` element in the `<startEvent>`,

```
<errorEventDefinition id="_X-1_ED_1" errorRef="_exception" />
```

must be changed to a `<signalEventDefintion>` which would like like this:

```
<signalEventDefinition id="_X-1_ED_1" signalRef="exception-signal"/>
```

In short, we have to make the following changes to the `<startEvent>` in the Event SubProcess:

1. It will now contain a `<signalEventDefintion>` instead of a `<errorEventDefintion>`
2. The `errorRef` attribute in the `<erroEventDefintion>` is now a `signalRef` attribute in the `<signalEventDefintion>`.
3. The `id` attribute in the `signalRef` is of course now the id of the `<signal>` element. Before it was id of `<error>` element.

4. Lastly, when we signal the process in the Java code, we do not signal "`Error-code`" but simply "`exception-signal`", the `id` of the `<signal>` element.

14.5.2. Example: logging exceptions thrown by bad `<scriptTask>` nodes

In this section, we'll briefly describe what's possible when dealing with `<scriptTask>` nodes that throw exceptions, and then quickly go through an example (also available in the `jbpm-examples` module) that illustrates this.

14.5.2.1. Introduction

If you're reading this, then you probably already have problem: you're either expecting to run into this problem because there are scripts in your process definition that might throw an exception, or you're already running a process instance with scripts that are causing a problem.

Unfortunately, if you're running into this problem, then there is not much you can do. The only thing that you *can* do is retrieve more information about exactly what's causing the problem. Luckily, when a `<scriptTask>` node causes an exception, it's wrapped in a `WorkflowRuntimeException`.

What type of information is available? The `WorkflowRuntimeException` instance will contain the information outlined in the following table. All of the fields listed are available via the normal `get*` methods.

Table 14.3. Information contained in `WorkflowRuntimeException` instances.

Field name	Type	Description
`processInstanceId`	`long`	The id of the `ProcessInstance` instance in which the exception occurred. This `ProcessInstance` may not exist anymore or be available in the database if using persistence!
`processId`	`String`	The id of the process definition that was used to start the process (i.e. "`ExceptionScriptTask`" in `ksession.startProcess("ExceptionScriptTas`)
`nodeId`	`long`	The value of the (BPMN2) id attribute of the node that threw the exception.

Field name	Type	Description
`nodeName`	`String`	The value of the (BPMN2) name attribute of the node that threw the exception.
`variables`	`Map<String, Object>`	The map containing the variables in the process instance (*experimental*).
`message`	`String`	The short message indicating what went wrong.
`cause`	`Throwable`	The original exception that was thrown.

14.5.2.2. Example: Exceptions thrown by a `<scriptTask>`.

The following code illustrates how to extract extra information from a process instance that throws a `WorkflowRuntimeException` exception instance.

```
import org.jbpm.workflow.instance.WorkflowRuntimeException;
import org.kie.api.KieBase;
import org.kie.api.io.ResourceType;
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.internal.builder.KnowledgeBuilder;
import org.kie.internal.builder.KnowledgeBuilderFactory;
import org.kie.internal.io.ResourceFactory;

public class ScriptTaskExceptionExample {

    public static final void main(String[] args) {
        runExample();
    }

    public static void runExample() {
        KieSession ksession = createKieSession();
        Map<String, Object> params = new HashMap<String, Object>();
        String varName = "var1";
        params.put( varName , "valueOne" );
        try {
            ProcessInstance processInstance = ksession.startProcess("ExceptionScriptTask", para
        } catch( WorkflowRuntimeException wfre ) {
            String msg = "An exception happened in "
                    + "process instance [" + wfre.getProcessInstanceId()
                    + "] of process [" + wfre.getProcessId()
                    + "] in node [id: " + wfre.getNodeId()
                    + ", name: " + wfre.getNodeName()
```

```
                            + "] and variable " + varName + " had the value
 [" + wfre.getVariables().get(varName)
                    + "]";
            System.out.println(msg);
        }
    }

    private static KieSession createKieSession() {
      KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
                kbuilder.add(ResourceFactory.newClassPathResource("exceptions/
ScriptTaskException.bpmn2"), ResourceType.BPMN2);
        KieBase kbase = kbuilder.newKnowledgeBase();
        return kbase.newKieSession();
    }

}
```

Chapter 15. Flexible Processes

Case management and its relation to BPM is a hot topic nowadays. There definitely seems to be a growing need amongst end users for more flexible and adaptive business processes, without ending up with overly complex solutions. Everyone seems to agree that using a process-centric approach only in many cases leads to complex solutions that are hard to maintain. The "knowledge workers" no longer want to be locked into rigid processes but wants to have the power and flexibility to regain more control over the process themselves.

The term case management is often used in that context. Without trying to give a precise definition of what it might or might not mean, as this has been a hot topic for discussion, it refers to the basic idea that many applications in the real world cannot really be described completely from start to finish (including all possible paths, deviations, exceptions, etc.). Case management takes a different approach: instead of trying to model what should happen from start to finish, let's give the end user the flexibility to decide what should happen at runtime. In its most extreme form for example, case management doesn't even require any process definition at all. Whenever a new case comes in, the end user can decide what to do next based on all the case data.

A typical example can be found in healthcare (clinical decision support to be more precise), where care plans can be used to describe how patients should be treated in specific circumstances, but people like general practitioners still need to have the flexibility to add additional steps and deviate from the proposed plan, as each case is unique. And there are similar examples in claim management, helpdesk support, etc.

So, should we just throw away our BPM system then? No! Even at its most extreme form (where we don't model any process up front), you still need a lot of the other features a BPM system (usually) provides: there still is a clear need for audit logs, monitoring, coordinating various services, human interaction (e.g. using task forms), analysis, etc. And, more importantly, many cases are somewhere in between, or might even evolve from case management to more structured business process over time (when we for example try to extract common approaches from many cases). If we can offer flexibility as part of our processes, can't we let the users decide how and where they would like to apply it?

Let me give you two examples that show how you can add more and more flexibility to your processes. The first example shows a care plan that shows the tasks that should be performed when a patient has high blood pressure. While a large part of the process is still well-structured, the general practitioner can decide himself which tasks should be performed as part of the sub-process. And he also has the ability to add new tasks during that period, tasks that were not defined as part of the process, or repeat tasks multiple times, etc. The process uses an ad-hoc sub-process to model this kind of flexibility, possibly augmented with rules or event processing to help in deciding which fragments to execute.

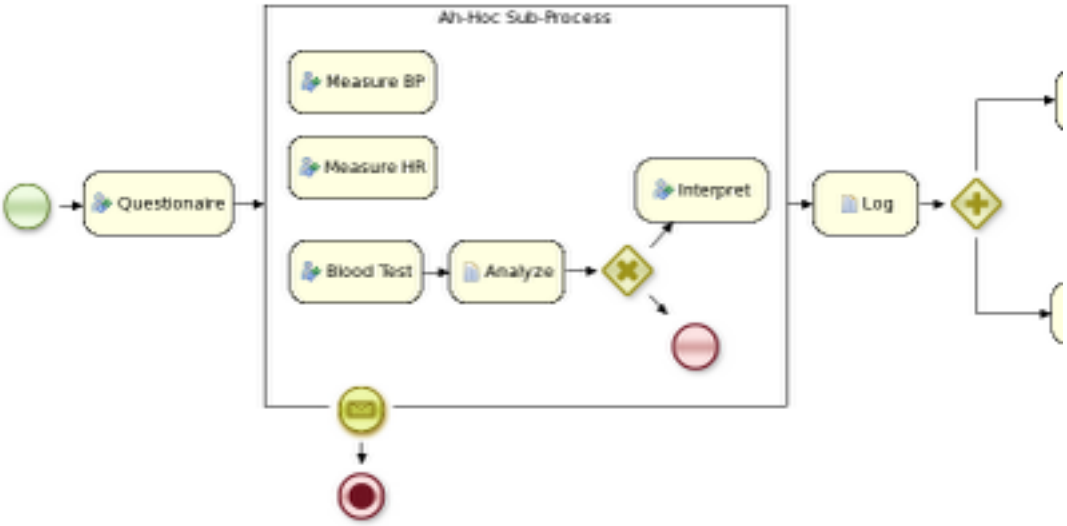


Figure 15.1.

The second example actually goes a lot further than that. In this example, an internet provider could define how cases about internet connectivity problems will be handled by the internet provider. There are a number of actions the case worker can select from, but those are simply small process fragments. The case worker is responsible for selecting what to do next and can even add new tasks dynamically. As you can see, there is not process from start to finish anymore, but the user is responsible for selecting which process fragments to execute.

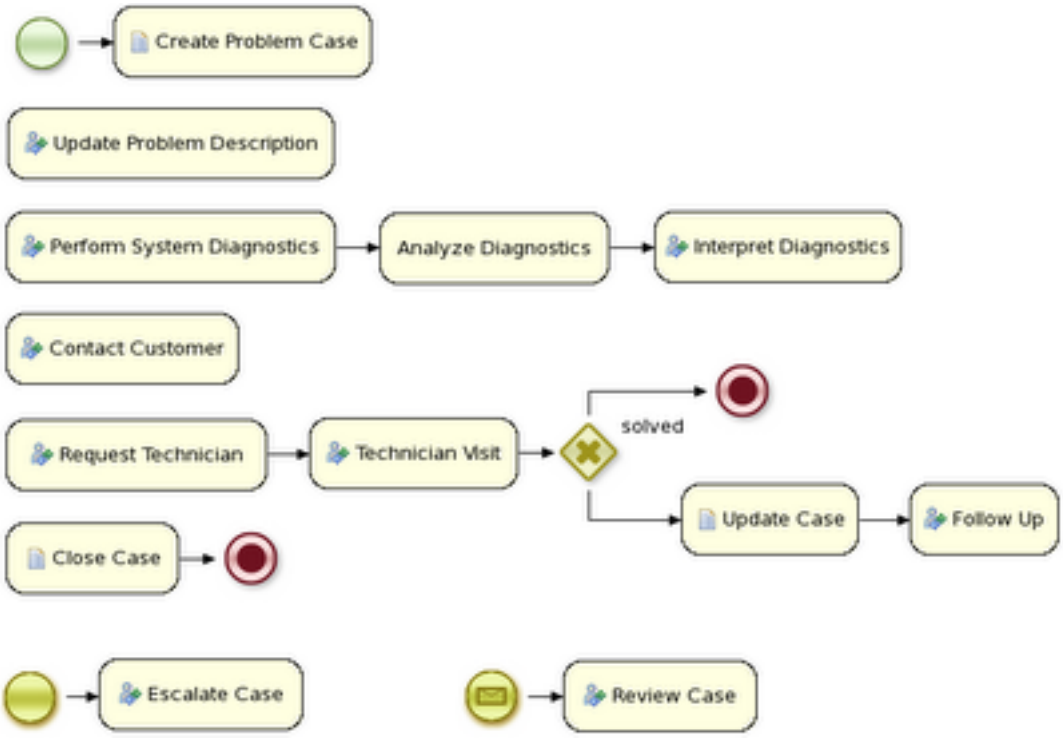


Figure 15.2.

And in its most extreme form, we even allow you to create case instances without a process definition, where what needs to be performed is selected purely at runtime. This however doesn't mean you can't figure out anymore what 's actually happening. For example, meetings can be very adhoc and dynamic, but we usually want a log of what was actually discussed. The following screenshot shows how our regular audit view can still be used in this case, and the end user could then for example get a lot more info about what actually happened by looking at the data associated with each of those steps. And maybe, over time, we can even automate part of that by using a semi-structured process.

Figure 15.3.

Chapter 16. Business Activity Monitoring

You need to actively monitor your processes to make sure you can detect any anomalies and react to unexpected events as soon as possible. Business Activity Monitoring (BAM) is concerned with real-time monitoring of your processes and the option of intervening directly, possibly even automatically, based on the analysis of these events.

jBPM allows users to define reports based on the events generated by the process engine, and possibly direct intervention in specific situations using complex event processing rules (Drools Fusion), as described in the next two sections. Future releases of the jBPM platform will include support for all requirements of Business Activity Monitoring, including a web-based application that can be used to more easily interact with a running process engine, inspect its state, generate reports, etc.

16.1. Reporting

By adding a history logger to the process engine, all relevant events are stored in the database. This history log can be used to monitor and analyze the execution of your processes. We are using the Eclipse BIRT (Business Intelligence Reporting Tool) to create reports that show the key performance indicators. Its easy to define your own reports yourself, using the predefined data sets containing all process history information, and any other data sources you might want to add yourself.

The Eclipse BIRT framework allows you to define data sets, create reports, include charts, preview your reports, and export them on web pages. (Consult the Eclipse BIRT documentation on how to define your own reports.) The following screen shot shows a sample on how to create such a chart.

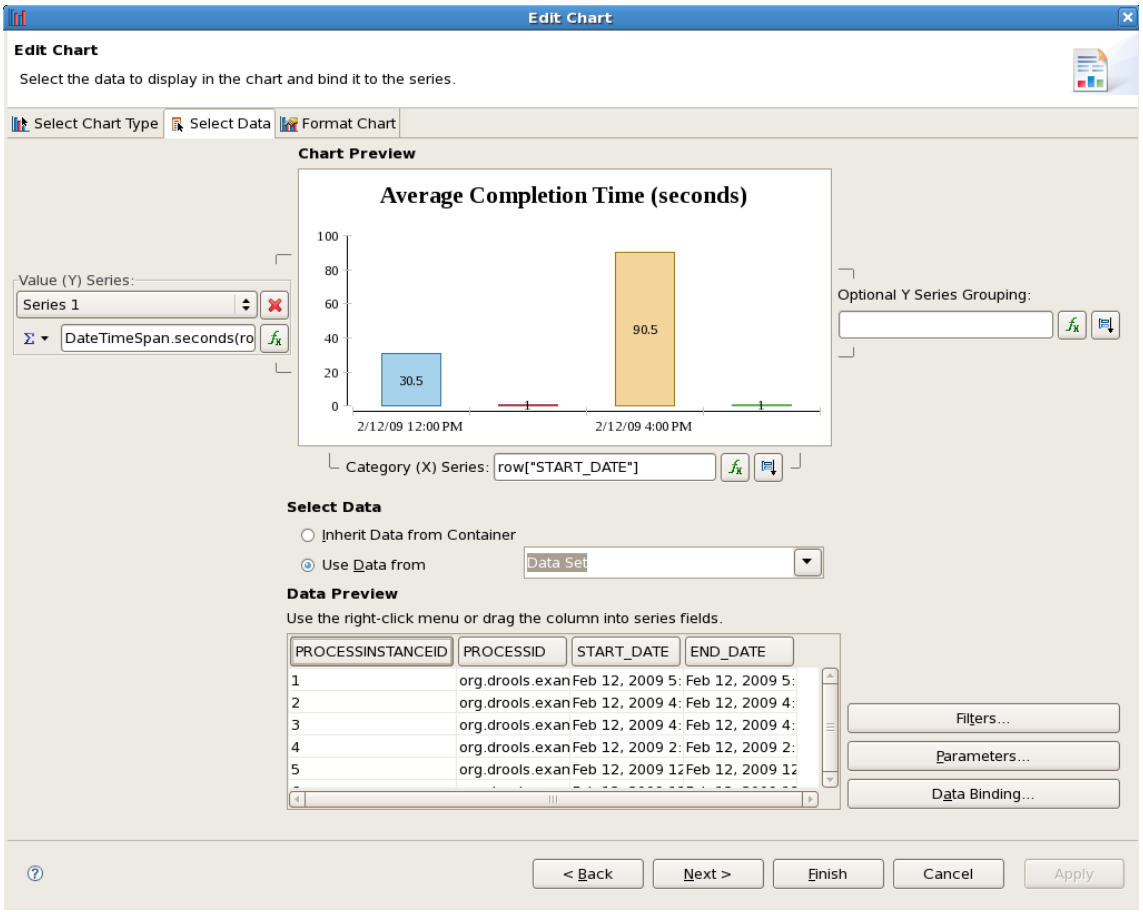


Figure 16.1. Creating a report using Eclipse BIRT

The next figure displays a simple report based on some history data, showing the number of requests per hour and the average completion time of the request during that hour. These charts could be used to check for an unexpected drop or rise of requests, an increase in the average processing time, etc. These charts could signal possible problems before the situation really gets out of hand.

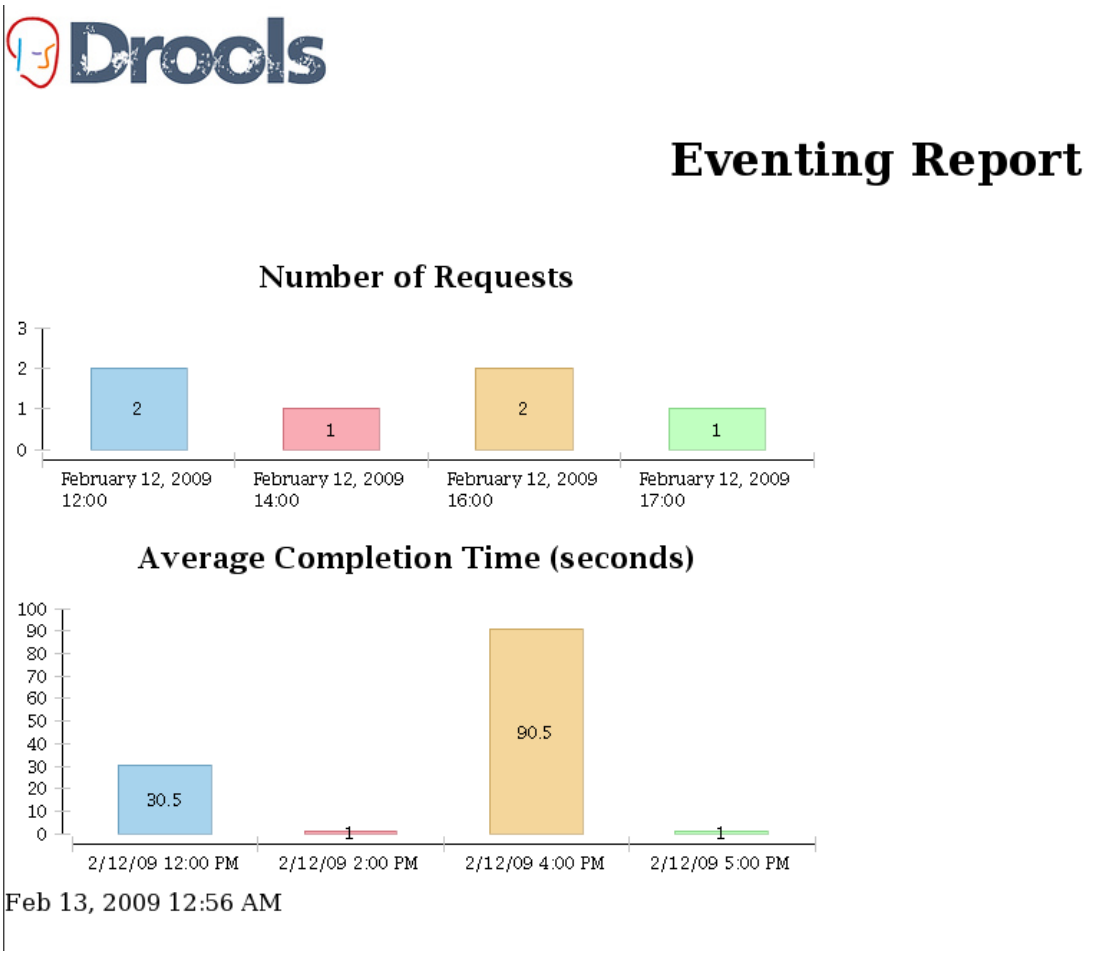


Figure 16.2. The eventing report

16.2. Direct Intervention

Reports can be used to visualize an overview of the current state of your processes, but they rely on a human actor to take action based on the information in these charts. However, we allow users to define automatic responses to specific circumstances.

Drools Fusion provides numerous features that make it easy to process large sets of events. This can be used to monitor the process engine itself. This can be achieved by adding a listener to the engine that forwards all related process events, such as the start and completion of a process instance, or the triggering of a specific node, to a session responsible for processing these events. This could be the same session as the one executing the processes, or an independent session as well. Complex Event Processing (CEP) rules could then be used to specify how to process these events. For example, these rules could generate higher-level business events based on a specific occurrence of low-level process events. The rules could also specify how to respond to specific situations.

The next section shows a sample rule that accumulates all start process events for one specific order process over the last hour, using the "sliding window" support. This rule prints out an error

message if more than 1000 process instances were started in the last hour (e.g., to detect a possible overload of the server). Note that, in a realistic setting, this would probably be replaced by sending an email or other form of notification to the responsible instead of the simple logging.

```
declare ProcessStartedEvent
    @role( event )
end

dialect "mvel"

rule "Number of process instances above threshold"
when
  Number( nbProcesses : intValue > 1000 )
    from accumulate(
                    e:   ProcessStartedEvent(   processInstance.processId   ==
 "com.sample.order.OrderProcess" )
      over window:size(1h),
      count(e) )
then
  System.err.println( "WARNING: Number of order processes in the last hour above
 1000: " +
                      nbProcesses );
end
```

These rules could even be used to alter the behavior of a process automatically at runtime, based on the events generated by the engine. For example, whenever a specific situation is detected, additional rules could be added to the Knowledge Base to modify process behavior. For instance, whenever a large amount of user requests within a specific time frame are detected, an additional validation could be added to the process, enforcing some sort of flow control to reduce the frequency of incoming requests. There is also the possibility of deploying additional logging rules as the consequence of detecting problems. As soon as the situation reverts back to normal, such rules would be removed again.

Chapter 17. Core Engine: Examples

17.1. 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 artefact and click finish. This should import the examples project in your workspace, so you can start looking at the processes and executing the classes.

17.2. 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.
- MultiInstance: 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.

17.3. 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.

Chapter 18. Testing and debugging

Even though business processes aren't code (we even recommend you to make them as high-level as possible and to avoid adding implementation details), they also have a life cycle like other development artefacts. And since business processes can be updated dynamically, testing them (so that you don't break any use cases when doing a modification) is really important as well.

18.1. Unit testing

When unit testing your process, you test whether the process behaves as expected in specific use cases, for example test the output based on the existing input. To simplify unit testing, jBPM includes a helper class called JbpmJUnitTestCase (in the jbpm-bpmn2 test module) that you can use to greatly simplify your junit testing, by offering:

- helper methods to create a new knowledge base and session for a given (set of) process(es)
  - you can select whether you want to use persistence or not
- assert statements to check
  - the state of a process instance (active, completed, aborted)
  - which node instances are currently active
  - which nodes have been triggered (to check the path that has been followed)
  - get the value of variables
  - etc.

For example, conside the following hello world process containing a start event, a script task and an end event. The following junit test will create a new session, start the process and then verify whether the process instance completed successfully and whether these three nodes have been executed.


```
public class MyProcessTest extends JbpmJUnitTestCase {

   public void testProcess() {
       // create your session and load the given process(es)
     StatefulKnowledgeSession ksession = createKnowledgeSession("sample.bpmn");
```

```
        // start the process
        ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello");
        // check whether the process instance has completed successfully
        assertProcessInstanceCompleted(processInstance.getId(), ksession);
       // check whether the given nodes were executed during the process execution
        assertNodeTriggered(processInstance.getId(), "StartProcess", "Hello", "EndProcess");
    }
}
```

18.1.1. Helper methods to create your session

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

- *createKnowledgeBase(String... process)*: Returns a new knowledge base containing all the processes in the given filenames (loaded from classpath)
- *createKnowledgeBase(Map<String, ResourceType> resources)* :Returns a new knowledge base containing all the resources (not limited to processes but possibly also including other resource types like rules, decision tables, etc.) from the given filenames (loaded from classpath)
- *createKnowledgeBaseGuvnor(String... packages)*: Returns a new knowledge base containing all the processes loaded from Guvnor (the process repository) from the given packages
- *createKnowledgeSession(KnowledgeBase kbase)*: Creates a new statefull knowledge session from the given knowledge base
- *restoreSession(StatefulKnowledgeSession ksession, boolean noCache)* : completely restores this session from database, can be used to recreate a session to simulate a critical failure and to test recovery, if noCache is true, the existing persistence cache will not be used to restore the data

18.1.2. Assertions

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

- *assertProcessInstanceActive(long processInstanceId, StatefulKnowledgeSession ksession)*: check whether the process instance with the given id is still active
- *assertProcessInstanceCompleted(long processInstanceId, StatefulKnowledgeSession ksession)*: check whether the process instance with the given id has completed successfully
- *assertProcessInstanceAborted(long processInstanceId, StatefulKnowledgeSession ksession)*: check whether the process instance with the given id was aborted

- *assertNodeActive(long processInstanceId, StatefulKnowledgeSession ksession, String... name)*: check whether the process instance with the given id contains at least one active node with the given node name (for each of the given names)

- *assertNodeTriggered(long processInstanceId, String... nodeNames)* : check for each given node name whether a node instance was triggered (but not necessarily active anymore) during the execution of the process instance with the given

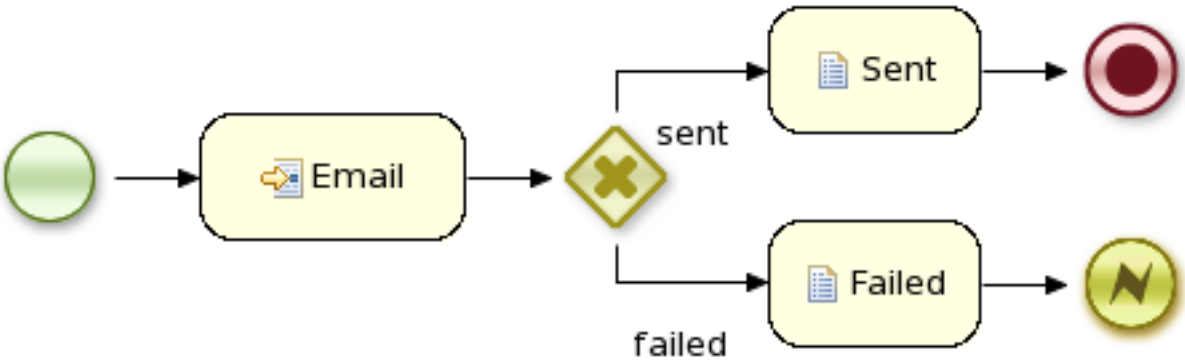
- *getVariableValue(String name, long processInstanceId, StatefulKnowledgeSession ksession)*: retrieves the value of the variable with the given name from the given process instance, can then be used to check the value of process variables

18.1.3. Testing integration with external services

Real-life business processes typically include the invocation of external services (like for example a human task service, an email server or your own domain-specific services). One of the advantages of our domain-specific process approach is that you can specify yourself how to actually execute your own domain-specific nodes, by registering a handler. And this handler can be different depending on your context, allowing you to use testing handlers for unit testing your process. When you are unit testing your business process, you can register test handlers that then verify whether specific services are requested correctly, and provide test responses for those services. For example, imagine you have an email node or a human task as part of your process. When unit testing, you don't want to send out an actual email but rather test whether the email that is requested contains the correct information (for example the right to email, a personalized body, etc.).

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

The following example describes how a process that sends out an email could be tested. This test case in particular will test whether an exception is raised when the email could not be sent (which is simulated by notifying the engine that the sending the email could not be completed). The test case uses a test handler that simply registers when an email was requested (and allows you to test the data related to the email like from, to, etc.). Once the engine has been notified the email could not be sent (using abortWorkItem(..)), the unit test verifies that the process handles this case successfully by logging this and generating an error, which aborts the process instance in this case.




```
public void testProcess2() {
    // create your session and load the given process(es)
    StatefulKnowledgeSession ksession = createKnowledgeSession("sample2.bpmn");
    // register a test handler for "Email"
    TestWorkItemHandler testHandler = new TestWorkItemHandler();
   ksession.getWorkItemManager().registerWorkItemHandler("Email", testHandler);
    // start the process
    ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello2");
    assertProcessInstanceActive(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "StartProcess", "Email");
    // check whether the email has been requested
    WorkItem workItem = testHandler.getWorkItem();
    assertNotNull(workItem);
    assertEquals("Email", workItem.getName());
    assertEquals("me@mail.com", workItem.getParameter("From"));
    assertEquals("you@mail.com", workItem.getParameter("To"));
    // notify the engine the email has been sent
    ksession.getWorkItemManager().abortWorkItem(workItem.getId());
    assertProcessInstanceAborted(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "Gateway", "Failed", "Error");
}
```

18.1.4. Configuring persistence

You can configure whether you want to execute the junit tests using persistence or not. By default, the junit tests will use persistence, meaning that the state of all process instances will be stored in a (in-memory H2) database (which is started by the junit test during setup) and a history log will be used to check assertions related to execution history. When persistence is not used, process instances will only live in memory and an in-memory logger is used for history assertions.

By default, persistence is turned on. To turn off persistence, simply pass a boolean to the super constructor when creating your test case, as shown below:

```
public class MyProcessTest extends JbpmJUnitTestCase {

    public MyProcessTest() {
        // configure this tests to not use persistence in this case
        super(false);
    }

    ...
```

18.2. Debugging

This section describes how to debug processes using the Eclipse plugin. This means that the current state of your running processes can be inspected and visualized during the execution. Note that we currently don't allow you to put breakpoints on the nodes within a process directly. You can however put breakpoints inside any Java code you might have (i.e. your application code that is invoking the engine or invoked by the engine, listeners, etc.) or inside rules (that could be evaluated in the context of a process). At these breakpoints, you can then inspect the internal state of all your process instances.

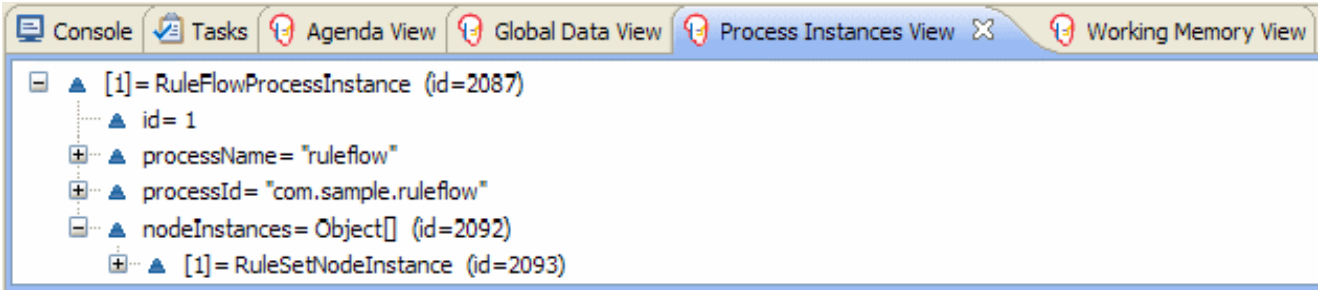
When debugging the application, you can use the following debug views to track the execution of the process:

1. The process instances view, showing all running process instances (and their state). When double-clicking a process instance, the process instance view visually shows the current state of that process instance at that point in time.
2. The human task view, showing the task list of the given user (fill in the user id of the actor and click refresh to view all the tasks for the given actor), where you can then control the life cycle of the task, for example start and complete it.
3. The audit view, showing the audit log (note that you should probably use a threaded file logger if you want to session to save the audit event to the file system on regular intervals, so the audit view can be update to show the latest state).
4. The global data view, showing the globals.
5. Other views related to rule execution like the working memory view (showing the contents (data) in the working memory related to rule execution), the agenda view (showing all activated rules), etc.

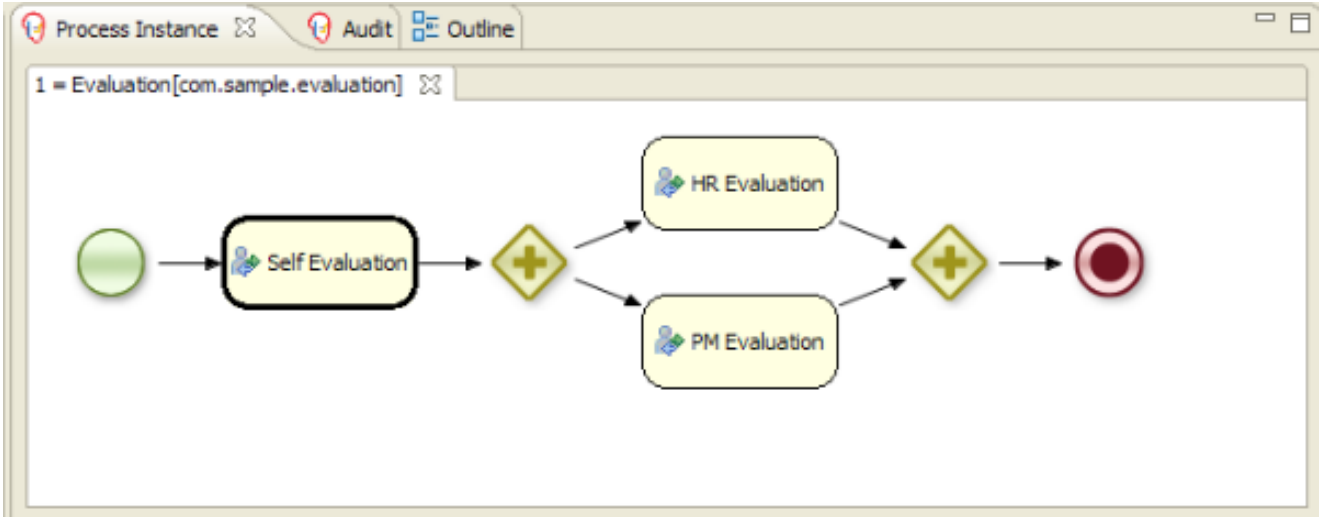
18.2.1. The Process Instances View

The process instances view shows the currently running process instances. The example shows that there is currently one running process (instance), currently executing one node instance, i.e. business rule task. When double-clicking a process instance, the process instance viewer will

graphically show the progress of the process instance. An example where the process instance is waiting for a human actor to perform a self-evaluation task is shown below.

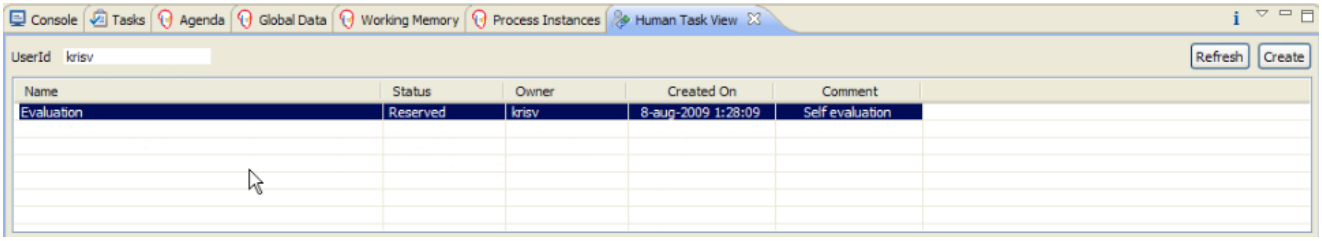


When you double-click a process instance in the process instances view and the process instance view complains that it cannot find the process, this means that the plugin wasn't able to find the process definition of the selected process instance in the cache of parsed process definitions. To solve this, simply change the process definition in question and save again (so it will be parsed) or rebuild the project that contains the process definition in question.



18.2.2. The Human Task View

The Human Task View can connect to a running human task service and request the relevant tasks for a particular user (i.e. the tasks where the user is either a potential owner or the tasks that the user already claimed and is executing). The life cycle of these tasks can then be executed, i.e. claiming or releasing a task, starting or stopping the execution of a task, completing a task, etc. A screenshot of this Human Task View is shown below. You can configure which task service to connect to in the Drools Task preference page (select Window -> Preferences and select Drools Task). Here you can specify the url and port (default = 127.0.0.1:9123).

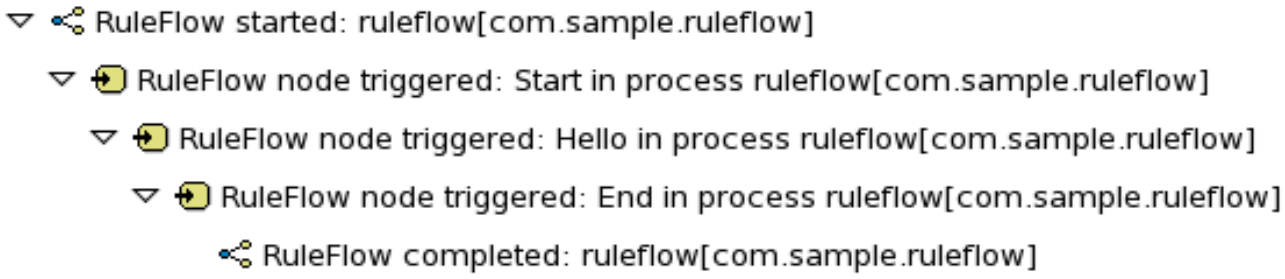


18.2.3. The Audit View

The audit view, showing the audit log, which is a log of all events that were logged from the session. To create a logger, use the KnowledgeRuntimeLoggerFactory to create a new logger and attach it to a session. Note that you should probably use a threaded file logger if you want to session to save the audit event to the file system on regular intervals, so the audit view can be update to show the latest state. When creating a threaded file logger, you can specify the name of the file where the audit log should be created and the interval after which event should be saved to the file (in milliseconds). Be sure to close the logger after usage.

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

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



Chapter 19. Process Repository

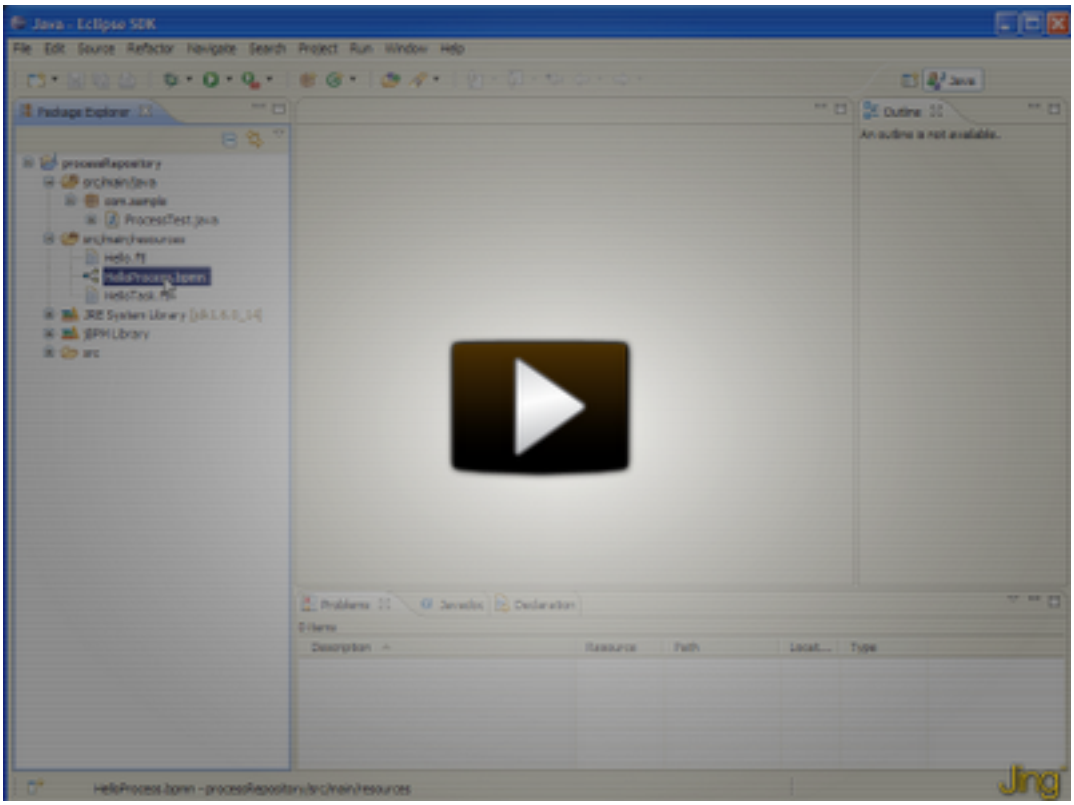
A process repository is an important part of your BPM architecture if you start using more and more business processes in your applications and especially if you want to have the ability to dynamically update them. The process repository is the location where you store and manage your business processes. Because they are not deployed as part of your application, they have their own life cycle, meaning you can update your business processes dynamically, without having to change the application code.

Note that a process repository is a lot more than simply a database to store your process definitions. It almost acts as a combination of a source code management system, content management system, collaboration suite and development and testing environment. These are the kind of features you can expect from a process repository:

- Persistent storage of your processes so the latest version can always easily be accessed from anywhere, including versioning
- Build and deploy selected processes
- User-friendly (web-based) interface to manage, update and deploy your processes (targeted to business users, not just developers)
- Authentication / authorization to make sure only people that have the right role can see and/or edit your processes
- Categorization and searching
- Scenario testing to make sure you don't break anything when you change your process
- Collaboration and other social features like comments, notifications on change, etc.
- Synchronization with your development environment

Actually, it would be better to talk about a knowledge repository, as the repository will not only store your process definitions, but possibly also other related artefacts like task forms, your domain model, associated business rules, etc. Luckily, we don't have to reinvent the wheel for this, as the Guvnor project acts as a generic knowledge repository to store any type of artefacts and already supports most of these features.

The following screencast shows how you can upload your process definition to Guvnor, along with the process form (that is used when you try to start a new instance of that process to collect the necessary data), task forms (for the human tasks inside the process), and the process image (that can be annotated to show runtime progress). The jBPM-console is configured to get all this information from Guvnor whenever necessary and show them in the console.



[http://people.redhat.com/kverlaen/jBPM5-guvnor-integration.swf]

Figure 19.1.

If you use the installer, that should automatically download and install the latest version of Guvnor as well. So simply deploy your assets (for example using the Guvnor Eclipse integration as shown in the screencast, also automatically installed) to Guvnor (taking some naming conventions into account, as explained below), build the package and start up the console.

The current integration of jBPM-console with Guvnor uses the following conventions to find the artefacts it needs:

- jBPM-console looks up artefacts from all available Guvnor packages (it does not look for assets in the Global Area). You can alternatively modify the guvnor.packages property in jBPM.console.properties to limit the lookup to only the packages you need, for example: guvnor.packages=defaultPackage, myPackageA, myPackageB
- A process should define the correct package name attribute, which needs to match the Guvnor package name it belongs to (otherwise you won't be able to build your package in Guvnor)
- Don't forget to build all of your packages in Guvnor before trying to view available processes in the console. Otherwise jBPM-console will not be able to retrieve the pkg from Guvnor.
- Currently, the console will load the process definitions the first time the list of processes is requested in the console. At this point, automatic updating from Guvnor when the package is rebuilt is turned off by default, so you will have to either configure this or restart the application server to get the latest versions.

- Task forms that should be associated with a specific process definition should have the name "{processDefinitionId}.ftl" or "{processDefinitionId}-taskform.ftl"
- Task forms for a specific human task should have the name "{taskName}.ftl" or "{taskName}-taskform.ftl"
- The process diagram for a specific process should have the name "{processDefinitionId}-image.png"
- By default jBPM-console looks up your Guvnor instance under http://localhost:8080/drools-guvnor. To change this, locate jbpm.console.properties and modify the guvnor.protocol, guvnor.host, and guvnor.subdomain property values as needed
- jBPM-console communicates with Guvnor via its REST api. The default connect and read timeouts for this communication are set to 10 seconds via the guvnor.connect.timeout, and guvnor.read.timeout properties in jbpm.console.properties. You can edit values of these properties to set your specific timeout values (in milliseconds)
- If you are using Guvnor with JAAS authentication enabled, jBPM-console uses by default admin/admin credentials. To change this information again locate jbpm.console.properties and change the guvnor.usr, and guvnor.pwd property values.

If you follow these rules, your processes, forms and images should show up without any issues in the jBPM-console.

19.1. Knowledge Agent

When you use the jBPM console, it will be using a knowledge agent to automatically update the knowledge base inside the console based on the resources available on the Guvnor repository. But you could also create a knowledge base from packages built on Guvnor yourself:

```
ResourceChangeScannerConfiguration sconf = ResourceFactory.getResourceChangeScannerService().ne
sconf.setProperty( "drools.resource.scanner.interval", "10" ); // every 10s
ResourceFactory.getResourceChangeScannerService().configure( sconf );
ResourceFactory.getResourceChangeScannerService().start();
ResourceFactory.getResourceChangeNotifierService().start();
KnowledgeAgentConfiguration aconf = KnowledgeAgentFactory.newKnowledgeAgentConfiguration();
aconf.setProperty("drools.agent.newInstance", "false");
KnowledgeAgent    kagent    =    KnowledgeAgentFactory.newKnowledgeAgent("Guvnor
 changeset", aconf);
kagent.applyChangeSet(ResourceFactory.newClassPathResource("changesetGuvnor.xml"));
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

```
<change-set  xmlns='http://drools.org/drools-5.0/change-set'  xmlns:xs='http://
www.w3.org/2001/XMLSchema-instance'       xs:schemaLocation='http://drools.org/
drools-5.0/change-set      http://anonsvn.jboss.org/repos/labs/labs/jbossrules/
trunk/drools-api/src/main/resources/change-set-1.0.0.xsd'>
```

```
   <add>
                      <resource    source='http://localhost:8080/drools-guvnor/
org.drools.guvnor.Guvnor/package/defaultPackage/
LATEST'type='PKG'basicAuthentication='enabled'username='admin'password='admin'/
>
   </add>
</change-set>
```

Chapter 20. Integration with Maven, OSGi, Spring, etc.

jBPM can be integrated with a lot of other technologies. This chapter gives an overview of a few of those that are supported out-of-the-box. Most of these modules are developed as part of the droolsjbpm-integration module, so they work not only for your business processes but also for business rules and complex event processing.

20.1. Maven

By using a Maven pom.xml to define your project dependencies, you can let maven get your dependencies for you. The following pom.xml is an example that could for example be used to create a new Maven project that is capable of executing a BPMN2 process:

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

  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-maven-example</artifactId>
  <name>jBPM Maven Project</name>
  <version>1.0-SNAPSHOT</version>

  <repositories>
    <!--  use this repository for stable releases -->
    <repository>
      <id>jboss-public-repository-group</id>
      <name>JBoss Public Maven Repository Group</name>
      <url>https://repository.jboss.org/nexus/content/groups/public/</url>
      <layout>default</layout>
      <releases>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
    <!-- use this repository for snapshot releases -->
    <repository>
      <id>jboss-snapshot-repository-group</id>
```

```
      <name>JBoss SNAPSHOT Maven Repository Group</name>
      <url>https://repository.jboss.org/nexus/content/repositories/snapshots/</
url>
      <layout>default</layout>
      <releases>
        <enabled>false</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
      </snapshots>
    </repository>

  </repositories>

  <dependencies>
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-bpmn2</artifactId>
      <version>5.4.0.Final</version>
    </dependency>
  </dependencies>

</project>
```

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

20.2. OSGi

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

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

It also supports side by side versioning, so you can have multiple versions of a bundle installed and it'll wire up the correct one. Further to this Bundles can register services for other bundles to use. These services need initialisation, which can cause ordering problems - how do you make sure you don't consume a service before its registered? OSGi has a number of features to help with service composition and ordering. The two main ones are the programmatic ServiceTracker

and the xml based Declarative Services. There are also other projects that help with this; Spring DM, iPOJO, Gravity.

The following jBPM jars are OGSi-enabled:

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

For example, the following code example shows how you can look up the necessary services in an OSGi environment using the service registry and create a session that can then be used to start processes, signal events, etc.

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

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

KnowledgeBaseConfiguration                     kbaseConf                       =
              knowledgeBaseFactoryService.newKnowledgeBaseConfiguration(              null,
 getClass().getClassLoader() );

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

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

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

20.3. Spring

A Spring XML configuration file can be used to easily define and configure knowledge bases and sessions in a Spring environment. This allows you to simply access a session and invoke processes from within your Spring application.

For example, the following configuration file sets up a new session based on a knowledge base with one process definition (loaded from the classpath).

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

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

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

</beans>
```

The following piece of code can be used to load the above Spring configuration, retrieve the session and start the process.

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

Note that you can also inject the session in one of your domain objects, for example by adding the following fragment in the configuration file.

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

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

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

The following example shows a slightly more complex example, where the session is configured to use persistence (JPA using an in-memory database in this case) and transaction (using the Spring transaction manager). When using the Spring transaction manager, you have three options:

- using the JTA transaction manager with a shared entity manager factory (emf)
- using local transactions with a shared entity manager factory (emf)

20.3.1. Spring using the JTA transaction manager

The following code sample shows the Spring configuration file, configured for JTA transactions (using Bitronix in this case) with a shared entity manager factory (emf).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jbpm="http://drools.org/schema/drools-spring"
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans-2.0.xsd
                                http://drools.org/schema/drools-spring http://
drools.org/schema/drools-spring-1.3.0.xsd">

  <bean id="jbpmEMF" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
    <property name="persistenceUnitName" value="org.jbpm.persistence.jta"/>
  </bean>

                          <bean            id="btmConfig"            factory-
method="getConfiguration" class="bitronix.tm.TransactionManagerServices">
  </bean>

  <bean id="BitronixTransactionManager" factory-method="getTransactionManager"
                      class="bitronix.tm.TransactionManagerServices"   depends-
on="btmConfig" destroy-method="shutdown" />
```

```
  <bean id="jbpmTxManager" class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager" ref="BitronixTransactionManager" />
    <property name="userTransaction" ref="BitronixTransactionManager" />
  </bean>

  <jbpm:kbase id="kbase1">
    <jbpm:resources>
      <jbpm:resource type="BPMN2" source="classpath:MyProcess.bpmn"/>
    </jbpm:resources>
  </jbpm:kbase>

  <jbpm:ksession id="ksession1" type="stateful" kbase="kbase1">
    <jbpm:configuration>
      <jbpm:jpa-persistence>
        <jbpm:transaction-manager ref="txManager"/>
        <jbpm:entity-manager-factory ref="emf"/>
      </jbpm:jpa-persistence>
    </jbpm:configuration>
  </jbpm:ksession>
</beans>
```

And the matching persistence.xml file:

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

  <persistence-unit name="org.jbpm.persistence.jta" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>

    <!-- Use this if you are using JPA1 / Hibernate3 -->
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <mapping-file>META-INF/ProcessInstanceInfo.hbm.xml</mapping-file>
    <!-- Use this if you are using JPA2 / Hibernate4 -->
    <!--mapping-file>META-INF/JBPMorm-JPA2.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.process.audit.ProcessInstanceLog</class>
    <class>org.jbpm.process.audit.NodeInstanceLog</class>
    <class>org.jbpm.process.audit.VariableInstanceLog</class>
```

```
    <properties>
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="false"/>
     <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/
>
<property name="hibernate.transaction.manager_lookup_class" value="org.hibernate.transaction.BTMTransactionManagerLookup"
>
    </properties>
  </persistence-unit>
</persistence>
```

20.3.2. Spring using local transactions

To use local transactions (instead of JTA) with a shared entity manager (emf), use:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jbpm="http://drools.org/schema/drools-spring"
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans-2.0.xsd
                                http://drools.org/schema/drools-spring http://
drools.org/schema/drools-spring-1.3.0.xsd">

  <bean id="jbpmEMF" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
    <property name="persistenceUnitName" value="org.jbpm.persistence.local"/>
  </bean>

  <bean id="jbpmTxManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="jbpmEMF"/>
    <property name="nestedTransactionAllowed" value="false"/>
  </bean>

  <jbpm:kbase id="kbase1">
    <jbpm:resources>
      <jbpm:resource type="BPMN2" source="classpath:MyProcess.bpmn"/>
    </jbpm:resources>
  </jbpm:kbase>

  <jbpm:ksession id="ksession1" type="stateful" kbase="kbase1">
    <jbpm:configuration>
      <jbpm:jpa-persistence>
        <jbpm:transaction-manager ref="txManager"/>
        <jbpm:entity-manager-factory ref="emf"/>
```

```
      </jbpm:jpa-persistence>
    </jbpm:configuration>
  </jbpm:ksession>
</beans>
```

And the matching persistence.xml file:

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

        <persistence-unit    name="org.jbpm.persistence.local"    transaction-
type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <non-jta-data-source>jdbc/jbpm-ds</non-jta-data-source>

    <!-- Use this if you are using JPA1 / Hibernate3 -->
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <mapping-file>META-INF/ProcessInstanceInfo.hbm.xml</mapping-file>
    <!-- Use this if you are using JPA2 / Hibernate4 -->
    <!--mapping-file>META-INF/JBPMorm-JPA2.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.process.audit.ProcessInstanceLog</class>
    <class>org.jbpm.process.audit.NodeInstanceLog</class>
    <class>org.jbpm.process.audit.VariableInstanceLog</class>

    <properties>
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="false"/>
     <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/
>
    </properties>
  </persistence-unit>
</persistence>
```

20.3.3. Spring using a shared entity manager

Instead of using a shared entity manager factory (emf), you can also use a shared entity manager instead (both using JTA or local transactions). To do so, create the entity manager in your Spring configuration file:

```
<bean id="jbpmEM" class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
  <property name="entityManagerFactory" ref="jbpmEMF"/>
</bean>
```

You can then create a ksession using the following code:

```
EntityManager em = (EntityManager) context.getBean("jbpmEM");
Environment env = EnvironmentFactory.newEnvironment();
env.set(EnvironmentName.APP_SCOPED_ENTITY_MANAGER, em);
env.set(EnvironmentName.CMD_SCOPED_ENTITY_MANAGER, em);
env.set("IS_JTA_TRANSACTION", false);
env.set("IS_SHARED_ENTITY_MANAGER", true);

AbstractPlatformTransactionManager aptm = (AbstractPlatformTransactionManager) context.getBean(
TransactionManager transactionManager = new DroolsSpringTransactionManager( aptm );
env.set(EnvironmentName.TRANSACTION_MANAGER, transactionManager);

PersistenceContextManager persistenceContextManager = new DroolsSpringJpaManager(env);
env.set(EnvironmentName.PERSISTENCE_CONTEXT_MANAGER, persistenceContextManager);

StatefulKnowledgeSession ksession = JPAKnowledgeService.newStatefulKnowledgeSession( kbase, nul
```

20.3.4. Using a local task service

If you also want to use a local task server, linked to the engine, first of all add the following lines to your persistence.xml:

```
<mapping-file>META-INF/Taskorm.xml</mapping-file>
<class>org.jbpm.task.Attachment</class>
<class>org.jbpm.task.Content</class>
<class>org.jbpm.task.BooleanExpression</class>
<class>org.jbpm.task.Comment</class>
<class>org.jbpm.task.Deadline</class>
<class>org.jbpm.task.Comment</class>
<class>org.jbpm.task.Deadline</class>
<class>org.jbpm.task.Delegation</class>
<class>org.jbpm.task.Escalation</class>
<class>org.jbpm.task.Group</class>
```

```
<class>org.jbpm.task.I18NText</class>
<class>org.jbpm.task.Notification</class>
<class>org.jbpm.task.EmailNotification</class>
<class>org.jbpm.task.EmailNotificationHeader</class>
<class>org.jbpm.task.PeopleAssignments</class>
<class>org.jbpm.task.Reassignment</class>
<class>org.jbpm.task.Status</class>
<class>org.jbpm.task.Task</class>
<class>org.jbpm.task.TaskData</class>
<class>org.jbpm.task.SubTasksStrategy</class>
<class>org.jbpm.task.OnParentAbortAllSubTasksEndStrategy</class>
<class>org.jbpm.task.OnAllSubTasksEndParentEndStrategy</class>
<class>org.jbpm.task.User</class>
```

Next, add the task service configuration to your Spring configuration file, after which you can get your local task service from your Spring context.

```
<bean id="systemEventListener" class="org.kie.internal.SystemEventListenerFactory"
method="getSystemEventListener" />

<bean id="internalTaskService" class="org.jbpm.task.service.TaskService" >
  <property name="systemEventListener" ref="systemEventListener" />
</bean>

<bean id="htTxManager" class="org.drools.container.spring.beans.persistence.HumanTaskSpringTran
  <constructor-arg ref="jbpmTxManager" />
</bean>

<bean id="springTaskSessionFactory" class="org.jbpm.task.service.persistence.TaskSessionSpringF
      init-method="initialize" depends-on="internalTaskService" >
  <property name="entityManagerFactory" ref="jbpmEMF" />
  <property name="transactionManager" ref="htTxManager" />
  <property name="useJTA" value="true" />
  <property name="taskService" ref="internalTaskService" />
</bean>

<bean id="taskService" class="org.jbpm.task.service.local.LocalTaskService" depends-
on="internalTaskService" >
  <constructor-arg ref="internalTaskService" />
</bean>
```

Note that, if you want your session linked to your local task service, you still need to create a synchronous human task handler and register it to the session using:

```
SyncWSHumanTaskHandler humanTaskHandler = new SyncWSHumanTaskHandler(taskService, ksession);
```

```
humanTaskHandler.setLocal(true);
humanTaskHandler.connect();
ksession.getWorkItemManager().registerWorkItemHandler("Human
 Task", humanTaskHandler);
```

20.4. Apache Camel Integration

Camel provides a lightweight bus framework for geting information into and out of jBPM. Additionally Camel provides a way to expose your KnowledgeBases remotely for any sort of client application that can use HTTP, through a SOAP or REST interface.

The following example shows how to setup a remote accessible session:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://camel.apache.org/schema/cxf"
       xmlns:jaxrs="http://cxf.apache.org/jaxrs"
       xmlns:kb="http://drools.org/schema/drools-spring"
       xsi:schemaLocation="
                        http://www.springframework.org/schema/beans    http://
www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://drools.org/schema/drools-spring http://drools.org/schema/drools-
spring.xsd
         http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/
camel-cxf.xsd
       http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd
          http://camel.apache.org/schema/spring http://camel.apache.org/schema/
spring/camel-spring.xsd">

  <!-- jBPM Knowledge Related Config -->

  <kb:grid-node id="node1"/>

  <kb:kbase id="kbase1" node="node1">
    <kb:resources>
      <kb:resource type="BPMN2" source="classpath:MyProcess.bpmn"/>
    </kb:resources>
  </kb:kbase>

  <kb:ksession id="ksession1" type="stateless" kbase="kbase1" node="node1"/>

  <!-- Camel Config -->

  <import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
```

```
  <cxf:rsServer id="rsServer"
                address="/rest"
                serviceClass="org.drools.jax.rs.CommandExecutorImpl">
       <cxf:providers>
           <bean class="org.drools.jax.rs.CommandMessageBodyReader"/>
       </cxf:providers>
  </cxf:rsServer>

  <cxf:cxfEndpoint id="soapServer"
            address="/soap"
             serviceName="ns:CommandExecutor"
             endpointName="ns:CommandExecutorPort"
          wsdlURL="soap.wsdl"
          xmlns:ns="http://soap.jax.drools.org/" >
    <cxf:properties>
      <entry key="dataFormat" value="MESSAGE"/>
      <entry key="defaultOperationName" value="execute"/>
    </cxf:properties>
  </cxf:cxfEndpoint>

  <bean id="kbPolicy" class="org.drools.camel.component.DroolsPolicy" />

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
       <from uri="cxfrs://bean://rsServer"/>
       <policy ref="kbPolicy">
         <unmarshal ref="xstream" />
         <to uri="drools:node1/ksession1" />
         <marshal ref="xstream" />
       </policy>
    </route>

    <route>
      <from uri="cxf://bean://soapServer"/>
      <policy ref="kbPolicy">
        <unmarshal ref="xstream" />
        <to uri="drools:node1/ksession1" />
        <marshal ref="xstream" />
      </policy>
    </route>

  </camelContext>

</beans>
```

To execute the above example you must be sure that you have the following content in your web.xml file:

```
<web-app>
  (...)
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:beans.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <display-name>CXF Servlet</display-name>
    <servlet-name>CXFServlet</servlet-name>
    <servlet-class>
      org.apache.cxf.transport.servlet.CXFServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/kservice/*</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>10</session-timeout>
  </session-config>
</web-app>
```

After deploying the above example, you can test it using any http compatible tool like curl, directly from you command line

```
$ curl -v \
  -H 'Content-Type: text/plain' \
  -d '<batch-execution lookup="ksession1"> \
     <start-process  processId="org.jbpm.sample.my-process"  out-identifier  =
 "processId"/> \
  </batch-execution>' \
  http://localhost:8080/jbpm-camel/kservice/rest/execute
```

The above execution will result in something similar to the following code snippet:

```
HTTP/1.1 200 OK
Content-Length: 131
Server: Apache-Coyote/1.1
Date: Mon, 13 Apr 2012 17:02:42 GMT
Content-Type: text/plain
Connection: close

<?xml        version='1.0'        encoding='UTF-8'?><execution-results><result
 identifier="processId"><long>1</long></result></execution-results>
```