jBPM User Guide


jBPM

Chapter 1. Overview

This chapter will give an overview of the various components that are offered as part of the jBPM project. It will walk through the different phases in the life cycle of a business process, from modeling and deployment to execution and monitoring.

1.1. Modeling

jBPM currently allows users to create and modify business processes using graphical flow charts. We target both developers and business users, using an Eclipse editors and a web-based editor respectively. Currently, we offer three options to model your BPMN2 processes.

1.1.1. Drools Flow Eclipse Plugin for creating BPMN2 processes

This Eclipse plugin allows developers to create, test and debug BPMN2 processes.

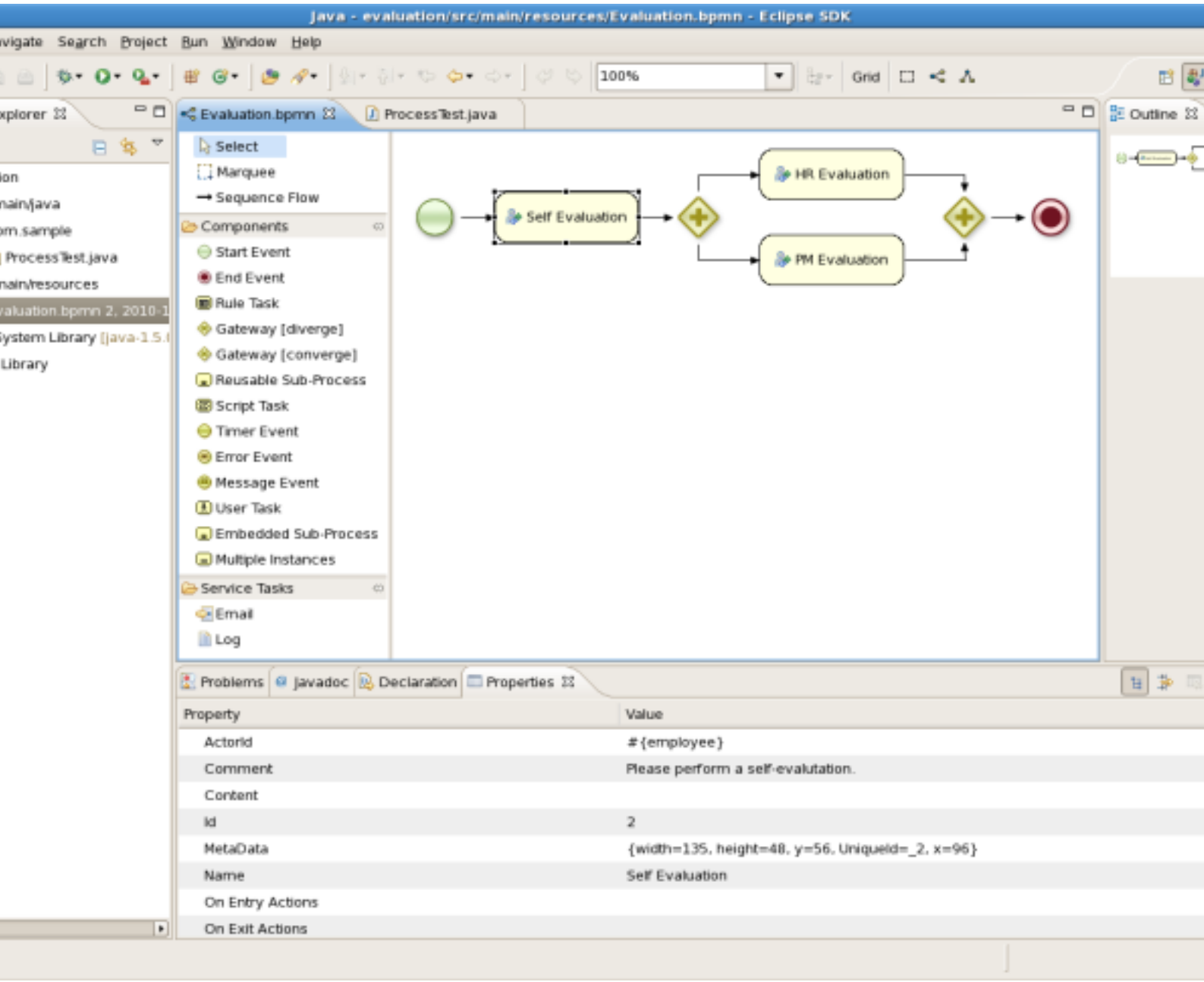


Figure 1.1. The Drools Flow editor for creating BPMN2 processes

1.1.2. jBPM5 Eclipse Plugin for creating BPMN2 processes

This new 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.

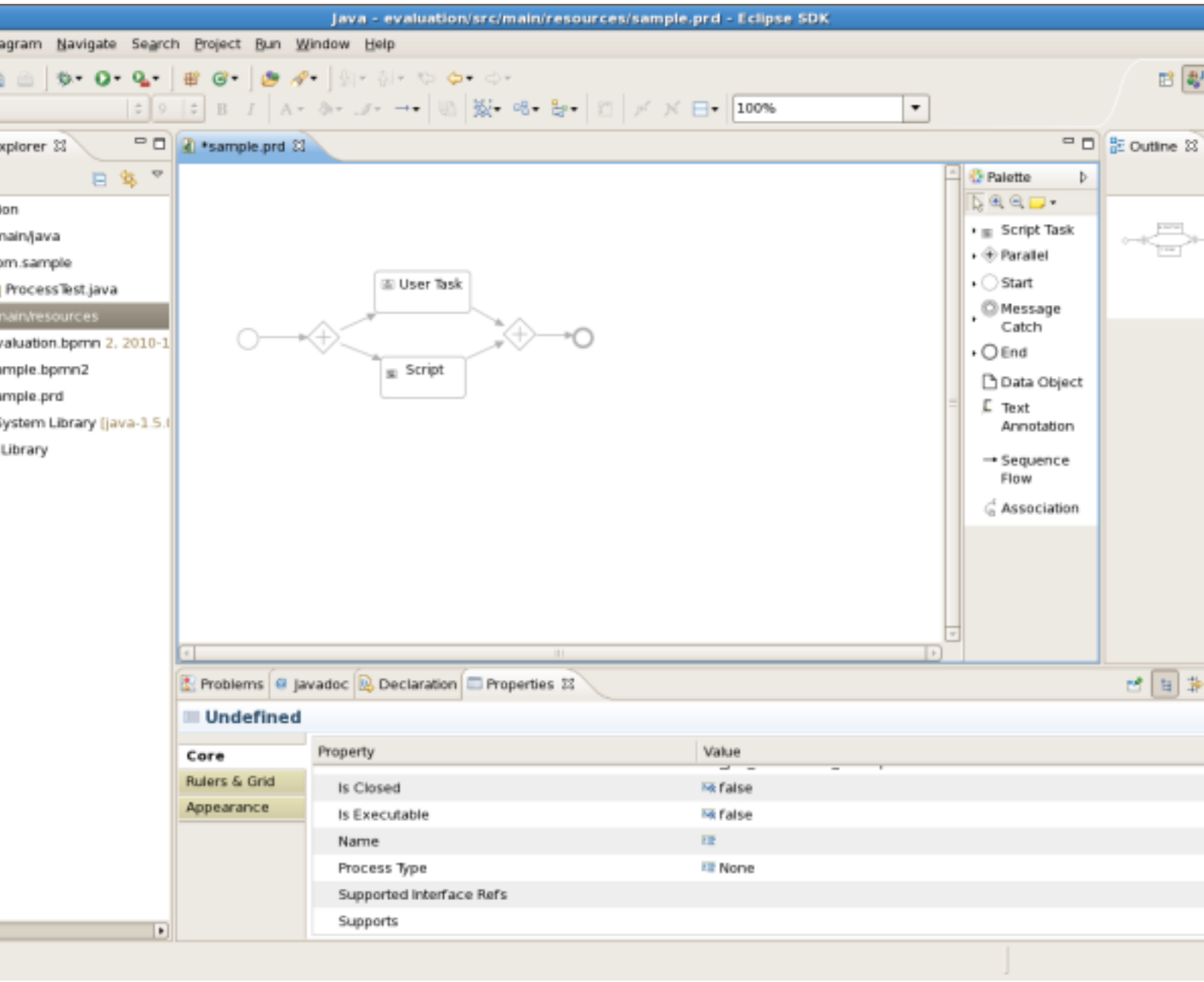


Figure 1.2. The jBPM editor for creating BPMN2 processes

1.1.3. Web-based process modeling using Oryx Designer

Web-based editing is possible using the Oryx Designer. This designer is also integrated into Guvnor, the knowledge repository where you can store your processes. The Designer can be used to view, update or create BPMN2 processes.

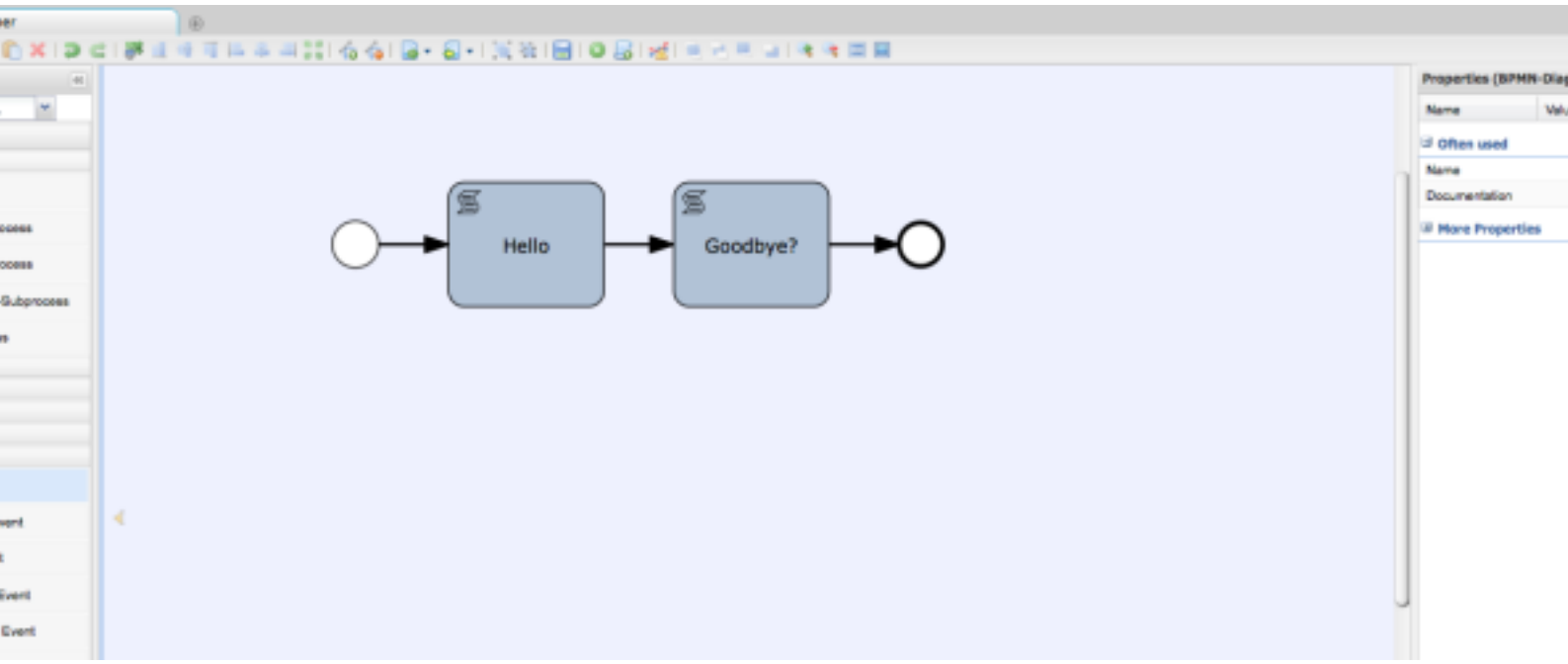


Figure 1.3. The web-based Oryx editor for creating BPMN2 processes

1.2. Deployment

Guvnor can be used as a knowledge repository, for storing your processes, domain model, business rules, etc. Guvnor provides a web-based management console for inspecting, modifying and testing your knowledge. The Oryx Designer is integrated with the Guvnor console. Guvnor also supports collaboration between different users, scenario testing, packaging, etc.

1.3. Execution

1.3.1. Process engine

The core engine is a lightweight workflow engine in Java:

- native BPMN2 process execution
- pluggable persistence and transactions
- audit and history logging
- based on a generic process engine

We do yet implement all node types and attributes as defined in the BPMN 2.0 specification, but we already support a very significant subset, which includes all common node types. The following list gives an overview of the various elements that can already be executed using the BPMN 2.0 XML format:

- *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 (Java or MVEL expression language)
  - 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 (Java, MVEL or XPath expression language)
    - Inclusive (Java, MVEL or XPath expression language)
    - Parallel
    - Event-Based

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

1.3.2. Human task service

Human tasks are an important part of a BPM solution. While some of the work performed in a process can be executed automatically, some tasks need to be executed with the interaction of human actors. This includes task lists for users, claiming and/or assigning tasks, etc. The process engine itself is not tied to one specific, internal implementation but allows for other implementations to be plugged in. By default jBPM currently supports a WS-HT service as the default implementation. The life cycle as supported by the Human Task specification is shown below. Note that the WS-HT service also includes features like group assignment, escalation, assignment rules, etc.

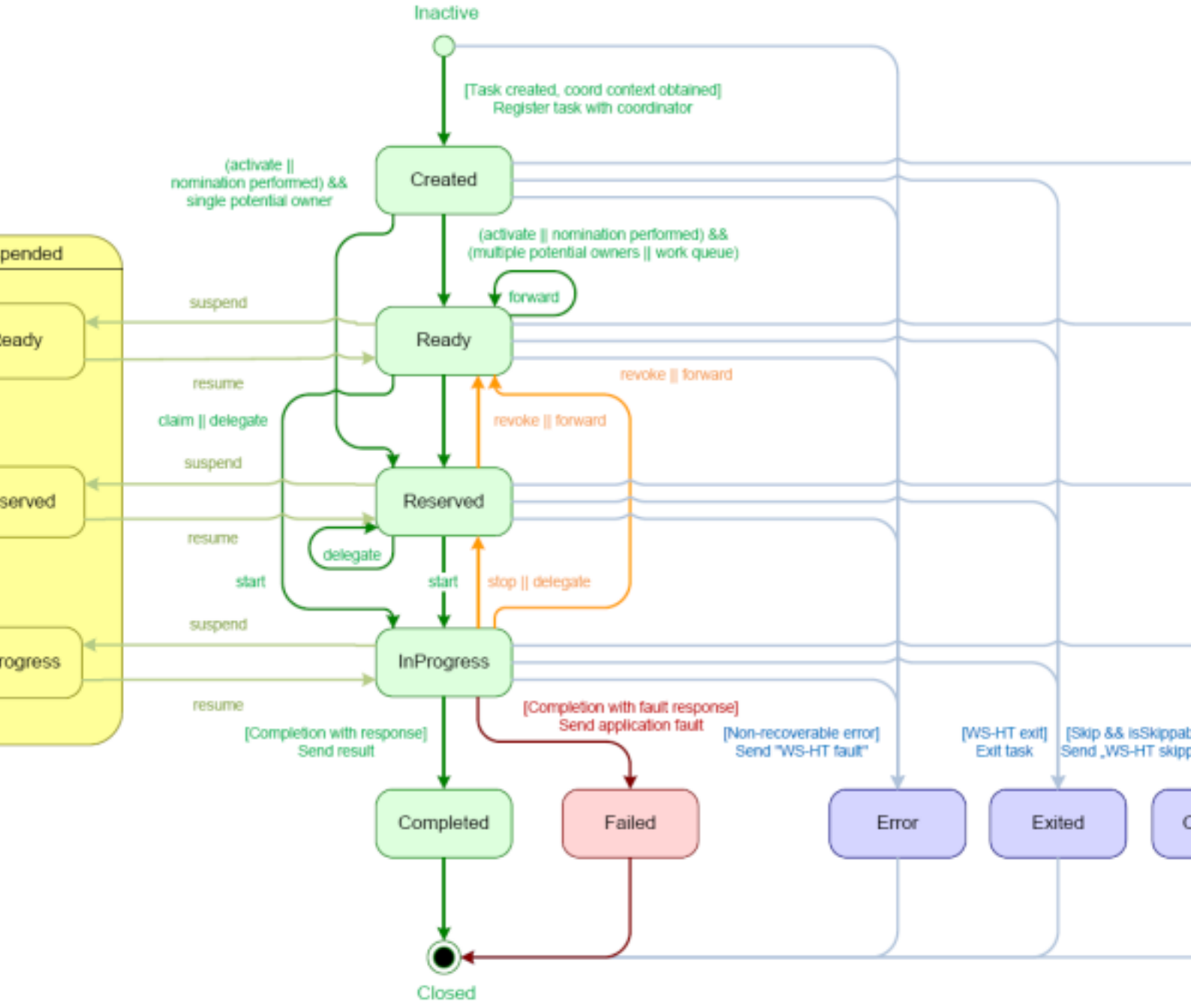


Figure 1.4. The WS-HT human task life cycle

User-defined human task forms are also supported using a template-based approach, while a other task form solutions can easily be integrated.

1.4. Monitoring

To monitoring your processes, you first need the ability to know what happening at runtime. Process listeners can be used to listen to different kinds of events occuring at runtime, like process instances being started or node instances being completed, etc. This information could then be

used to create history logs (for reporting or static analysis purposes), or to dymanically analyze and respond using Business Activity Monitoring (BAM).

1.4.1. Web-based management console

Processes can be managed through a web console. This includes features like managing your process instances (starting/stopping/inspecting), ...

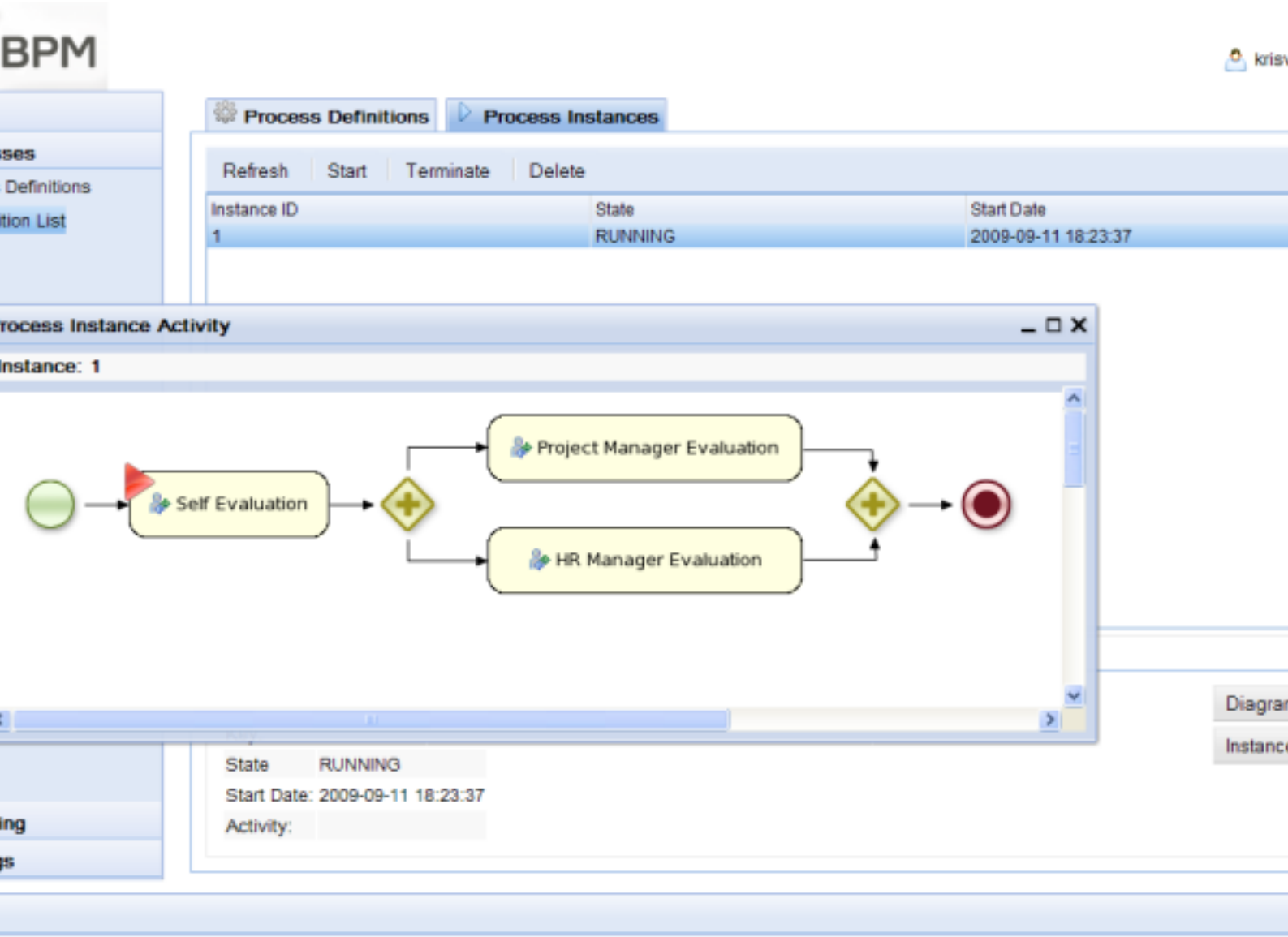


Figure 1.5. Managing your process instances

... inspecting your (human) task list and executing those tasks, ...

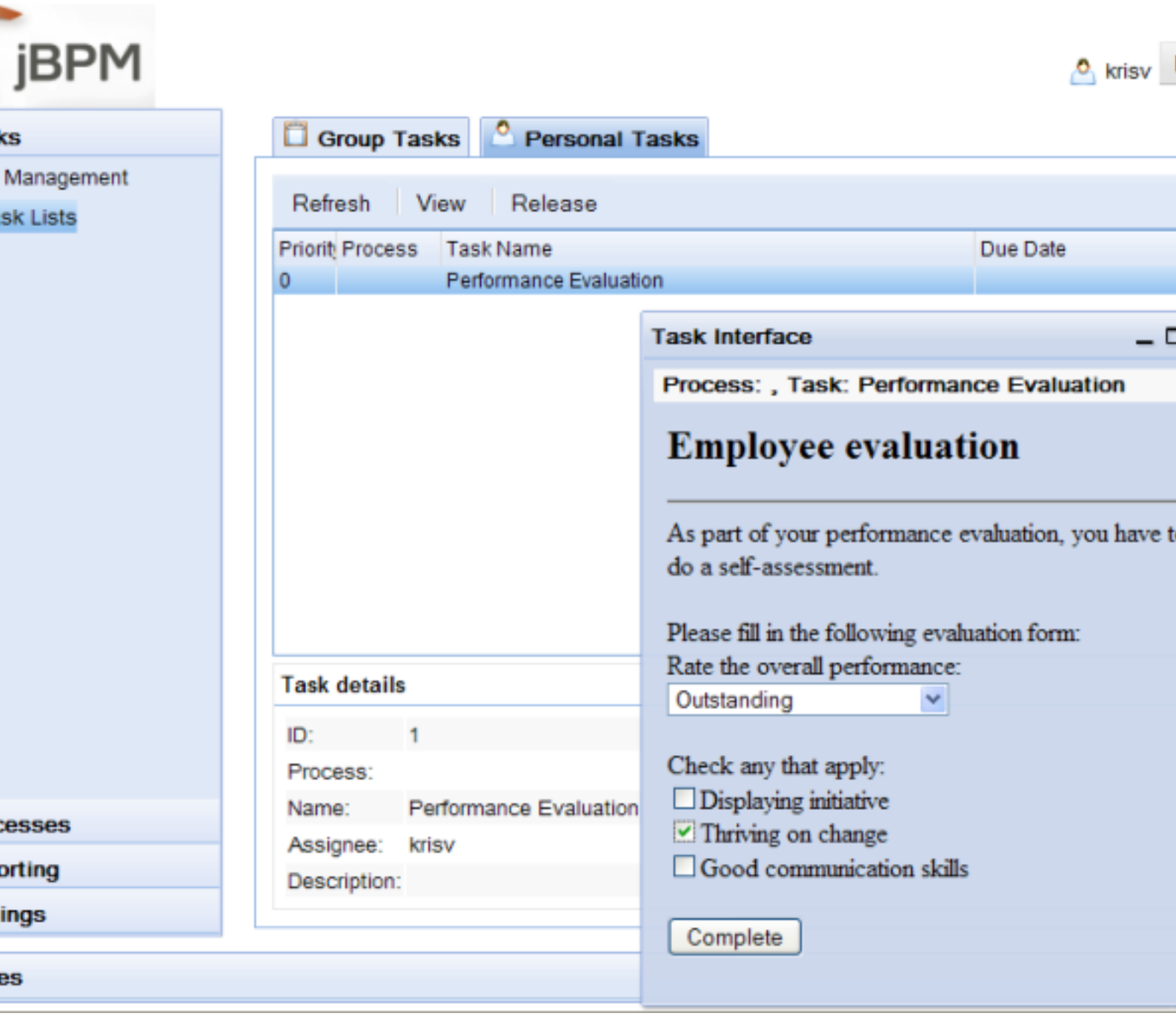


Figure 1.6. Managing your tasks

... and generating reports.

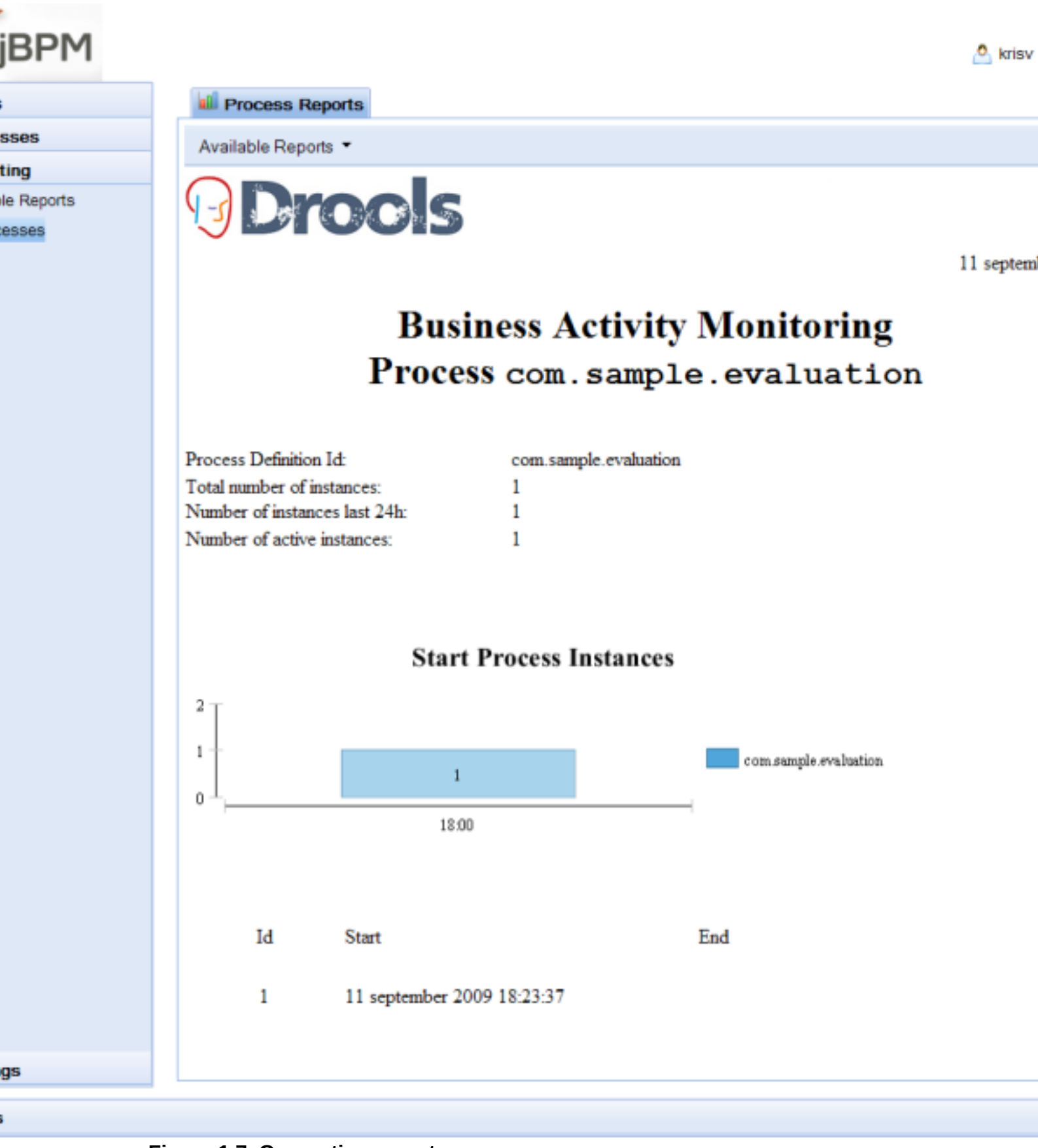


Figure 1.7. Generating reports

Chapter 2. 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.

2.1. 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*

2.2. Download the installer

First of all, you need to download the installer: jBPM-{version}-install.zip

You can for example find the latest snapshot release here.

http://hudson.jboss.org/hudson/job/jBPM5/lastSuccessfulBuild/artifact/target/

2.3. 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 Oryx Designer into JBoss AS
- Install the jBPM gwt-console into JBoss AS
- Install the jBPM Eclipse plugin
- Install the Drools Eclipse plugin

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

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

```
ant start.demo
```

This will:

- Start 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 and gwt-console, as explained in the next three sections.

2.4. Using Eclipse Tooling

The *following screencast* [http://people.redhat.com/kverlaen/install-eclipse-jbpm.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

Once Eclipse has opened, simple 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 by double-clicking it. To execute the process, right-click on ProcessTest.java in the com.sample package (under "src/main/java") and select "Run As - Java Application". 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. We will show you later how you could complete human tasks like this using a human task client like the jbpm-console.

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. Simple select "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.ProcessTest" Java class. You can open the BPMN2 process by double-clicking it. To execute the process, right-click on ProcessTest.java and select "Run As - Java Application". You should see a "Hello World" statement in the output console.

2.5. Using web management consoles

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/install-gwt-console-jbpm.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

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 process 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 click 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", 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 click 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 happened so far. Under "Reporting", select "Report Templates". By default, the console has two report templates, one for generating a generic overview for all processes and one for inspecting once specific process definition. If you select the latter, make sure to enter "com.sample.evaluation" as the process definition id to see the activity related to the evaluation process. 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.

2.6. Using Guvnor repository

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, using any non-empty username / password (we disabled authentication for demo purposes). The *following screencast* [http://people.redhat.com/kverlaen/install-guvnor-jbpm.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

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.

2.7. 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]

2.8. 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 in the jbpm-installer/jboss-4.2.3.GA/server/default/log directory.

For all other questions, try contacting the jBPM community as described above.

Chapter 3. Processes

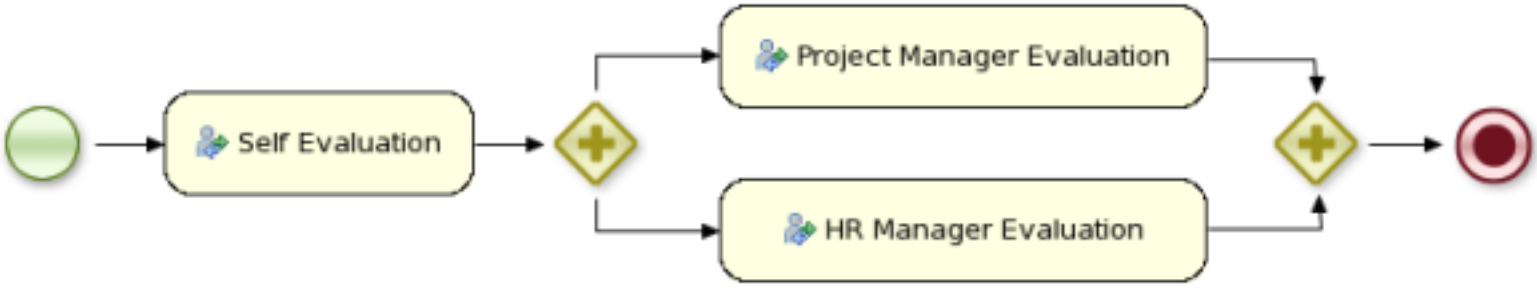


Figure 3.1. Process

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.

3.1. 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.

3.1.1. Using the graphical BPMN2 Editor

The graphical BPMN2 editor is a 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 an 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.

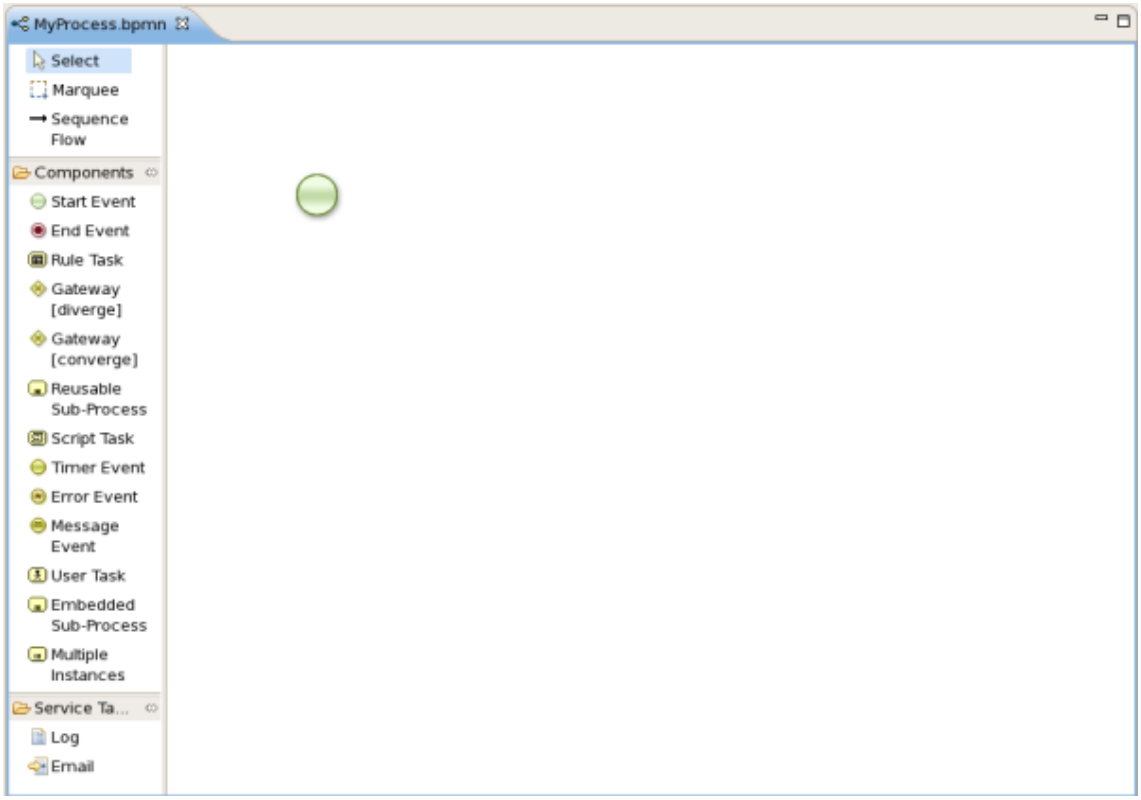


Figure 3.2. New process

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.

3.1.2. Defining processes using XML

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

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"
             targetNamespace="http://www.jboss.org/drools"
             typeLanguage="http://www.java.com/javaTypes"
             expressionLanguage="http://www.mvel.org/2.0"
```

```
          xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"Rule Task
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL BPMN20.xsd"
          xmlns:g="http://www.jboss.org/drools/flow/gpd"
          xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
          xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
          xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
          xmlns:tns="http://www.jboss.org/drools">

  <process processType="Private" isExecutable="true" id="com.sample.hello" name="Hello
Process" >

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

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

 </process>

 <bpmndi:BPMNDiagram>
  <bpmndi:BPMNPlane bpmnElement="com.sample.hello" >
    <bpmndi:BPMNShape bpmnElement="_1" >
      <dc:Bounds x="16" y="16" width="48" height="48" />
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="_2" >
      <dc:Bounds x="96" y="16" width="80" height="48" />
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="_3" >
      <dc:Bounds x="208" y="16" width="48" height="48" />
    </bpmndi:BPMNShape>
    <bpmndi:BPMNEdge bpmnElement="_1-_2" >
      <di:waypoint x="40" y="40" />
      <di:waypoint x="136" y="40" />
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge bpmnElement="_2-_3" >
      <di:waypoint x="136" y="40" />
```

```
      <di:waypoint x="232" y="40" />
    </bpmndi:BPMNEdge>
  </bpmndi:BPMNPlane>
</bpmndi:BPMNDiagram>

</definitions>
```

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

3.1.3. 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 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. Some examples about how to build processes using this fluent API can be found in the junit tests.

3.2. Using a Process in Your Application

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.rf"),
              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*: Processes are only started if you explicitly state that they should be started. This is because you could potentially define a lot of processes in your Knowledge Base and the engine has no way to know when you would like to start each of these. To activate a particular process, you will need to start it by calling the `startProcess` method on your session. For example:

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
ksession.startProcess("com.sample.hello");
```

The parameter of the `startProcess` method represents the id of the process that needs to be started. This process id needs to be specified as a property of the process, shown in the Properties View when you click the background canvas of your process.

You may specify additional parameters that are used to pass on input data to the process, using the `startProcess(String processId, Map parameters)` method, which takes an additional set of parameters as name-value pairs. These parameters are then copied to the newly created process instance as top-level variables of the process.

3.3. Detailed Explanation of the Different Node Types

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 actor responsible for the execution of human tasks. See chapter “*Human Tasks*” for details.

- *Connection Layout*: Specify how the connections are visualized on the canvas using the connection layout property:
  - 'Manual' always draws your connections as lines going straight from their start to end point (with the possibility to use intermediate break points).
  - 'Shortest path' is similar, but it tries to go around any obstacles it might encounter between the start and end point, to avoid lines crossing nodes.
  - 'Manhattan' draws connections by only using horizontal and vertical lines.

A BPMN2 process supports different types of nodes:

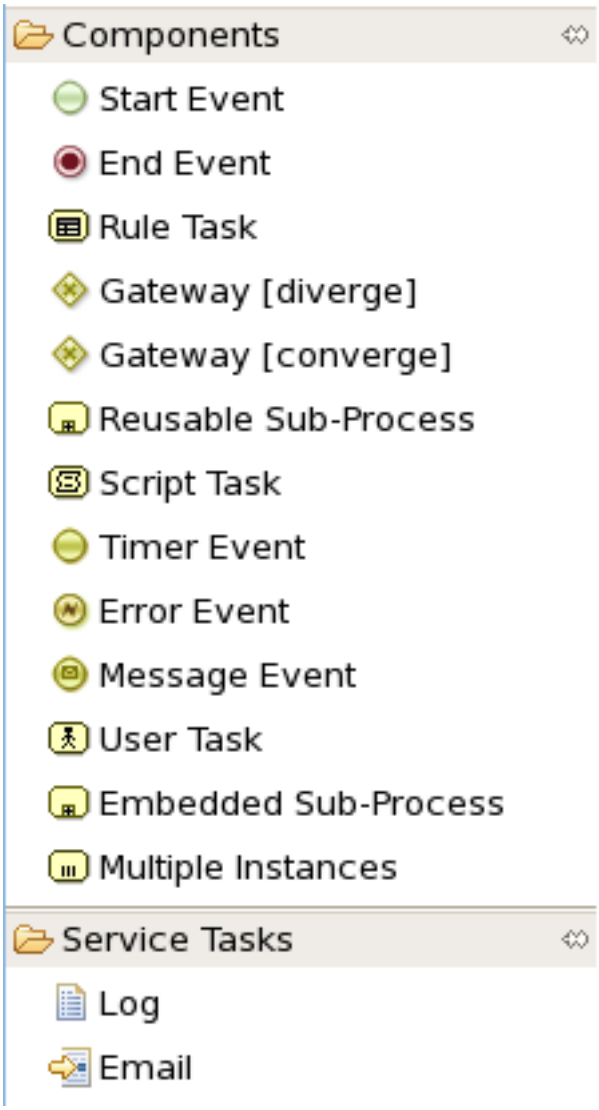


Figure 3.3. The different types of BPMN2 nodes

1. **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.

2. **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 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 be terminating for the entire process or just for the path. If the process is terminated, all nodes that are still active (on parallel paths) in this process are cancelled. Non-terminating End Events are simply ends for some path, while other parallel paths still continue.
3. **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. This means that, during the execution of a ruleflow group, it is possible that new activations belonging to the currently active ruleflow group are added to the Agenda 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.
4. **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. 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 (X)OR split).

5. **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. **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*: If this property is true, the SubFlow node will only continue if that SubFlow process has terminated its execution (completed or aborted); otherwise it will continue immediately after starting the subprocess.
- *Independent*: If this property is true, the subprocess is started as an independent process, which means that the SubFlow process will not be terminated if this process reaches an end node; otherwise the active sub-process will be cancelled on termination (or abortion) of the process.
- *On-entry and on-exit actions*: Actions that are executed upon entry or exit of this node, respectively.
- *Parameter in/out mapping*: A SubFlow node can also define in- and out-mappings for variables. The value of variables in this process with variable names given in the "in" mapping will be used as parameters (with the associated parameter name) when starting the process. The value of the variables in the subprocess with the given variable name in the "out" mappings will be copied to the variables of this process when the subprocess has been completed. Note that you can use "out" mappings only when "Wait for completion" is set to true.

7. **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 globals, the 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 associated with this action node.
8. **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 (in milliseconds) 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 process). 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 (in milliseconds) that the node should wait before triggering the first time.

9. **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 exception handler that is capable of handling this kind of fault. If no fault 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 appriopriate exception handlers that is 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).

10. **Message Event**: A Message Event can be used to respond to internal or external events during the execution of the process. A Message 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.
    - *Scope*: An event could be used to listen to internal events only, i.e., events that are signaled to this process instance directly, using `processInstance.signalEvent(String type, Object data)`. When an event node is defined as external, it will also be listening to external events that are signaled to the process engine directly, using `ksession.signalEvent(String type, Object event)`.

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*: Actions 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.

12 **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. 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.

13 **Multiple Instances**: A Multiple Instance node 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 node should have one incoming connection and one outgoing connection. A Multiple Instance node awaits the completion of the embedded segment for each of the collection''s elements 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 of type `java.util.Collection`.
- *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.

14 **Service Task (or Work Item node)**: 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. Refer to 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 item 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}`.

3.4. Data

While the flow chart focusses 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, you use variables. 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.
- Actions can access variables directly, simply by using the name of the variable as a parameter name.

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

Changing the value of a variable can be done through the Knowledge Context:

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

- Service Tasks and Reusable Sub-Processes can pass the value of parameters to the outside world by mapping the variable to one of the work item parameters, either by using a parameter mapping or by injecting it into a String parameter, using `#{expression}`. The results of a WorkItem can also be copied 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.

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. The Knowledge Session can be accessed in actions using the Knowledge Context:

```
kcontext.getKnowledgeRuntime().insert( new Person(...) );
```

3.5. 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 ...
```

3.6. Actions

Actions 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.drools.runtime.process.ProcessContext` and can be used for several tasks:

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

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

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

```
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 );
```

3.7. Events

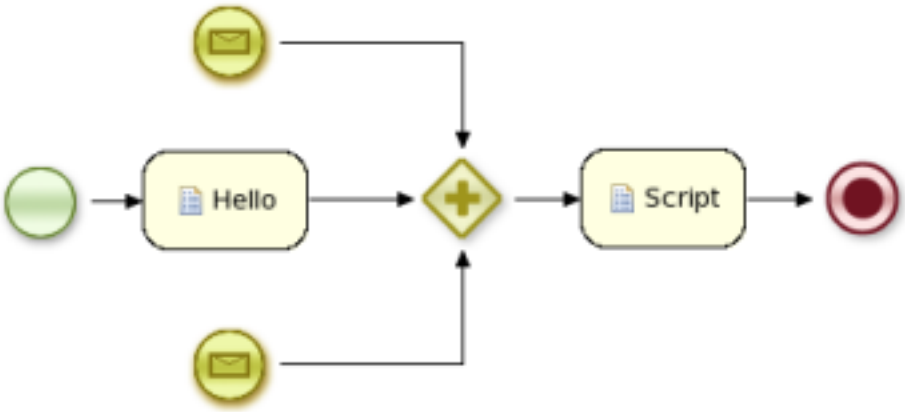


Figure 3.4. A sample process using events

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 occurence 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.

3.8. Timers

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

A Timer Event is set up with a delay. The delay specifies the amount of time (in milliseconds) to wait after node activation before triggering the timer the first time.

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 its triggers, once or repeatedly, activate the Timer node's successor. This means that the outgoing connection of a timer with a positive perios is triggered multiple times. Cancelling a Timer node also cancels the associated timer, whereafter no more triggerings will occur.
- Timers can be associated with a Sub-Process as a boundary event. This is however currently only possible by doing this in XML directly. We will be adding support for graphically specifying this in the new BPMN2 editor.

3.9. 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.

3.9.1. 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 accross 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 used 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 sofisticated 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 initiazed 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 allow 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.

Chapter 4. BPMN 2.0

4.1. 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 all the different jBPM5 components 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 and collaboration. The jBPM project however focuses on that part of the specification that can be used to specify executable processes.

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">

<processprocessType="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.

- Oryx 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 Oryx editor (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.
- You can always manually create your BPMN 2.0 process files by writing the XML directly.

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");
```

4.2. 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/main/resources folder for the jbpm-bpmn2 module like *here* [http://github.com/krisv/jbpm/tree/master/jbpm-bpmn2/src/test/resources/].

4.3. Supported elements / attributes

Table 4.1. Keywords

Element	Supported attributes	Supported elements	Extension attributes
definitions		rootElement BPMNDiagram	
process	processType isExecutable name id	property laneSet flowElement	packageName
sequenceFlow	sourceRef targetRef isImmediate name id	conditionExpression	bendpoints
interface	name id	operation	
operation	name id	inMessageRef	
laneSet		lane	
lane	name id	flowNodeRef	
Events			
startEvent	name id	dataOutput dataOutputAssociation outputSet eventDefinition	x y width height
endEvent	name id	dataInput dataInputAssociation inputSet eventDefinition	x y width height
intermediateCatchEvent	name id	dataOutput dataOutputAssociation outputSet eventDefinition	x y width height
intermediateThrowEvent	name id	dataInput dataInputAssociation inputSet eventDefinition	x y width height
boundaryEvent	cancelActivity attachedToRef name id	eventDefinition	x y width height
terminateEventDefinition			
cancelEventDefinition			
compensateEventDefinition	activityRef	documentation extensionElements	

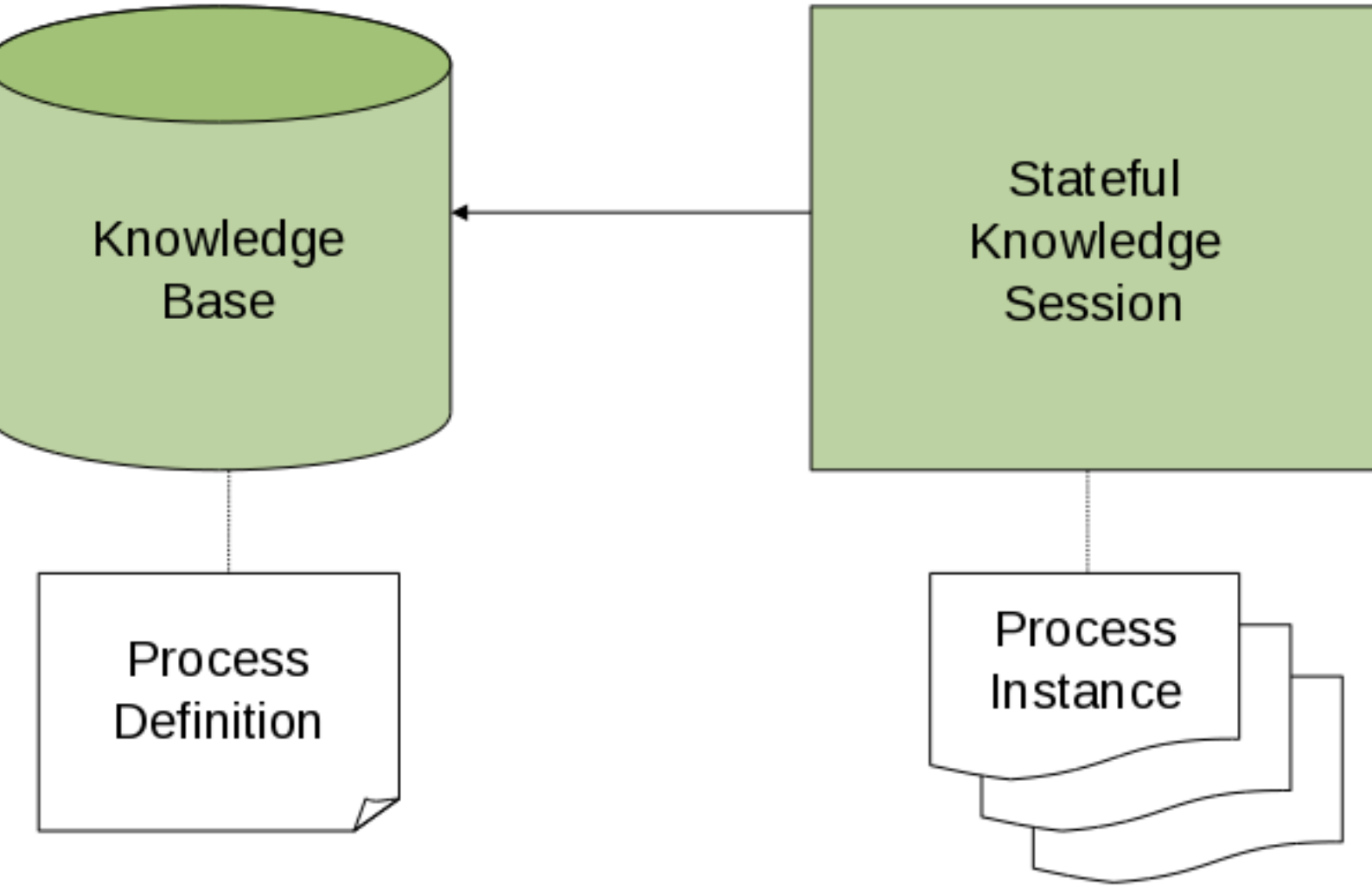
Element	Supported attributes	Supported elements	Extension attributes
conditionalEventDefinition		condition	
errorEventDefinition	errorRef		
error	errorCode id		
escalationEventDefinition	escalationRef		
escalation	escalationCode id		
messageEventDefinition	messageRef		
message	itemRef id		
signalEventDefinition	signalRef		
timerEventDefinition		timeCycle	
Activities			
task	name id	ioSpecification dataInputAssociation dataOutputAssociation	taskName x y width height
scriptTask	scriptFormat name id	script	x y width height
script		text[mixed content]	
userTask	name id	ioSpecification dataInputAssociation dataOutputAssociation resourceRole	x y width height
potentialOwner		resourceAssignmentExpression	
resourceAssignmentExpression		expression	
businessRuleTask	name id		x y width height ruleFlowGroup
manualTask	name id		x y width height
sendTask	messageRef name id	ioSpecification dataInputAssociation	x y width height
receiveTask	messageRef name id	ioSpecification dataOutputAssociation	x y width height
serviceTask	operationRef name id	ioSpecification dataInputAssociation dataOutputAssociation	x y width height
subProcess	name id	flowElement property loopCharacteristics	x y width height
adHocSubProcess	cancelRemainingInstances name id	completionCondition flowElement property	x y width height

Element	Supported attributes	Supported elements	Extension attributes
callActivity	calledElement name id	ioSpecification dataInputAssociation dataOutputAssociation	x y width height waitForCompletion independent
multiInstanceLoopCharacteristics		loopDataInputRef inputDataItem	
Gateways			
parallelGateway	gatewayDirection name id		x y width height
eventBasedGateway	gatewayDirection name id		x y width height
exclusiveGateway	default gatewayDirection name id		x y width height
inclusiveGateway	default gatewayDirection name id		x y width height
Data			
property	itemSubjectRef id		
dataObject	itemSubjectRef id		
itemDefinition	structureRef id		
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			

Element	Supported attributes	Supported elements	Extension attributes
BPMNDiagram		BPMNPlane	
BPMNPlane	bpmnElement	BPMNEdge BPMNShape	
BPMNShape	bpmnElement	Bounds	
BPMNEdge	bpmnElement	waypoint	
Bounds	x y width height		
waypoint	x y		

Chapter 5. API

To interact with the process engine (to for example start a process), you need to set up a session. This session will be used to communicate with the process engine. A session also needs to have a reference to a knowledge base. This knowledge base is used to look up the process definitions whenever necessary. 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, image 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 then comes in, a new process instance is started for that sales order.

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 much independent session as you want and creating a session is considered relatively lightweight. How much sessions you create is up to you, but in general you could for example create one session and direct all calls in your application to that one session. You could decide to create multiple sessions if for example you want to have multiple independent processing units (for example, you want all processes from one customer be completely independent of processes of another customer so 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 one create session that you then use to execute all your processes.

5.1. The jBPM API

The 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 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. The following code snippet shows how to create a knowledge base consisting of only one process definition (using a knowledge builder, in this case to add a resource from the classpath).

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

5.1.2. Session

Next, you should create a session to interact with the engine. The following code snippet shows how easy it is to create a session based on the earlier 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 instance
 * @return the ProcessInstance that represents the instance of the process that was started
 */
ProcessInstance startProcess(String processId,
                             Map<String, Object> parameters);

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

```
     */
    void signalEvent(String type,
                     Object event);

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

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

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

    /**
     * Aborts the process instance with the given id.  If the process instance has been completed
```

```
    * (or aborted), or the process instance cannot be found, this method will throw an
    * IllegalArgumentException.
    *
    * @param id the id of the process instance
    */
   void abortProcessInstance(long processInstanceId);

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

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.

```
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 );

}
```

An audit log can be created based on the information provided by these process listeners. We provide various default logger implementations:

1. Console logger: This logger writes out all the events to the console.

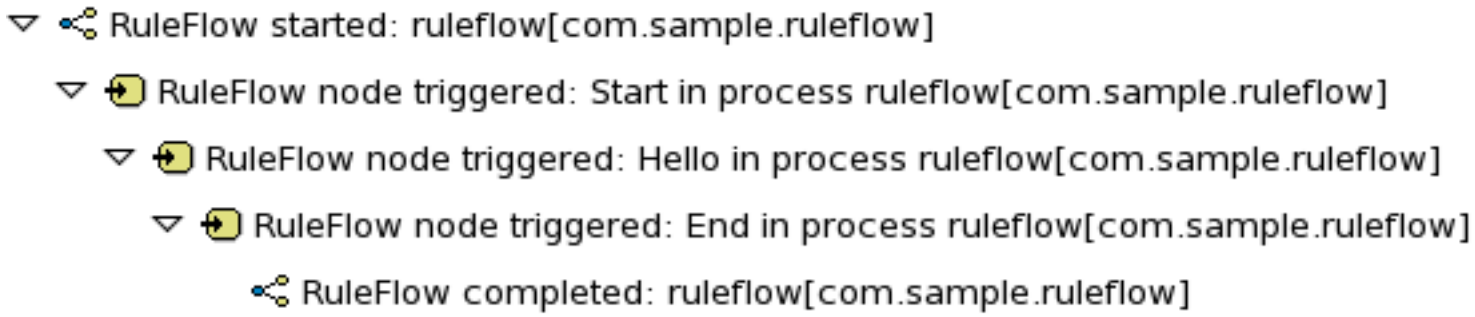
2. File logger: This logger writes out all the events to a file using an XML representation. This log file might then be used in the IDE to generate a tree-based visualization of the events that occurred during execution.

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.

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

The log file 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 only 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. Human Tasks

An important aspect of work flow and BPM is human task management. While some of the work performed in a process can be executed automatically, some tasks need to be executed with the interaction of human actors. jBPM supports the use of human tasks inside processes using a special user task node that will represent this interaction. This node allows process designers to define the type of task, the actor(s), the data associated with the task, etc. We also have implemented a task service that can be used to manage these user tasks. Users are however open to integrate any other solution if they want to, as this is fully pluggable.

To start using human tasks inside your processes, you first need to (1) include user task nodes inside your process, (2) integrate a task management component of your choice (e.g. the WS-HT implementation provided by us) and (3) have end users interact with the human task management component using some kind of user interface. These elements will be discussed in more detail in the next sections.

6.1. 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 simply considered as any other kind of external service that needs to be invoked and are therefore simply implemented as a special kind of work item. The only thing that is special about the user task node is that we have added support for swimlanes, making it easier to assign tasks to users (see below). A user task node contains the following properties:

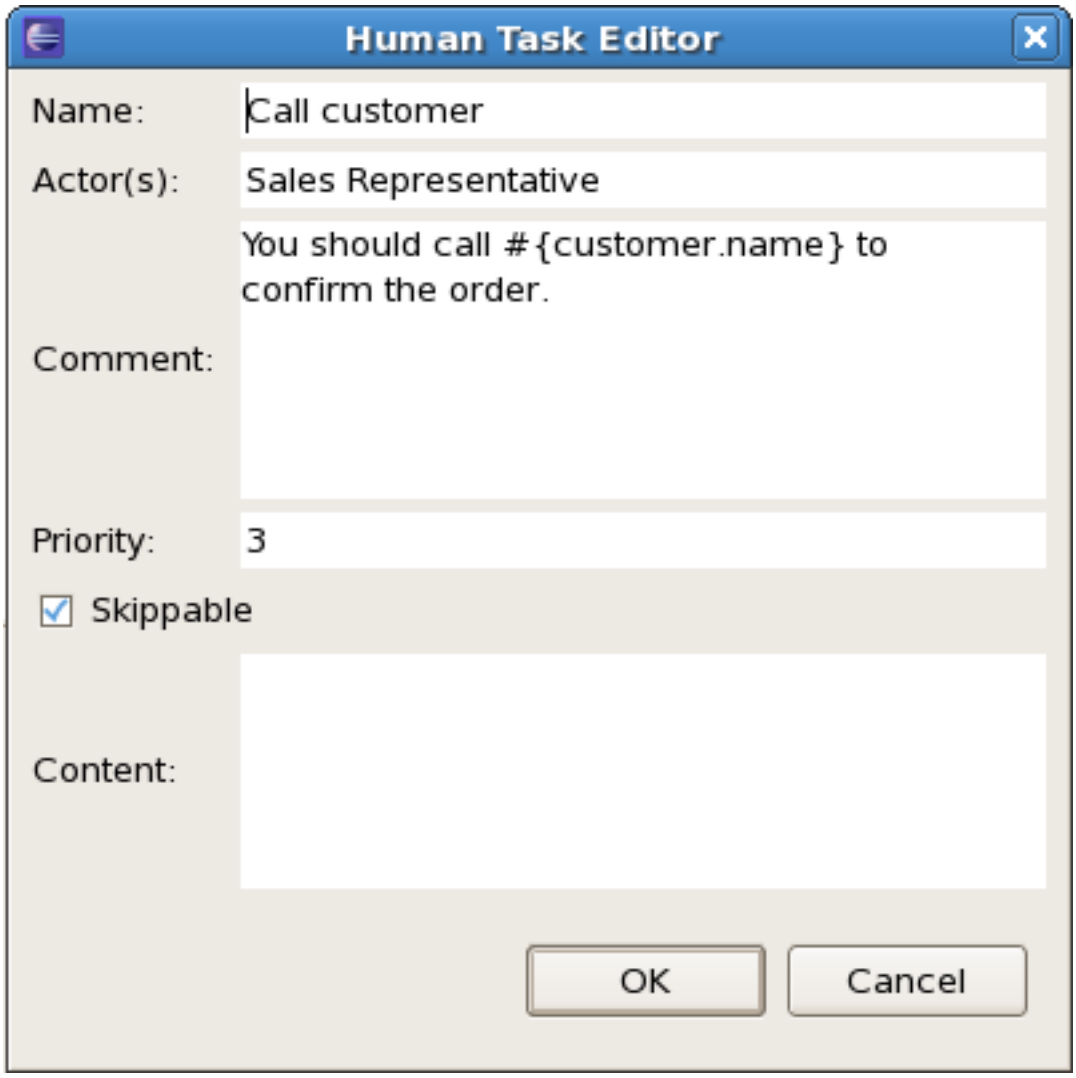
- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *TaskName*: The name of the human task.
- *Priority*: An integer indicating the priority of the human task.
- *Comment*: A comment associated with the human task.
- *ActorId*: The actor id that is responsible for executing the human task. A list of actor id's can be specified using a comma (',') as separator.
- *Skippable*: Specifies whether the human task can be skipped (i.e. the actor decides not to execute the human 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 below for more detail on how to use swimlanes.
- *Wait for completion*: If this property is true, the human task node will only continue if the human task has been terminated (i.e. completed or any other terminal state); otherwise it will continue immediately after creating the human task.
- *On-entry and on-exit actions*: Actions that are executed upon entry and exit of this node.
- *Parameter mapping*: Allows copying the value of process variables to parameters of the human task. Upon creation of the human tasks, the values will be copied.
- *Result mapping*: Allows copying the value of result parameters of the human task to a process variable. Upon completion of the human task, the values will be copied. Note that can only use result mappings when "Wait for completion" is set to true. 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.
- *Timers*: Timers that are linked to this node (see the 'timers' section for more details).
- *ParentId*: Allows to specify the parent task id, in the case that this task is a sub task of another. (see the 'sub task' section for more details)

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



Note that you could either specify the values of the different parameters (actorId, priority, content, etc.) directly (in which case they will be the same for each execution of this process), or make them context-specific, based on the data inside the process instance. For example, 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 #{...} will be replaced by the toString() value of the variable. 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 #{person.name.firstname}. For example, when sending an email, the body of the email could contain something like "Dear #{customer.name}, ...". For other types of variables, it is possible to map the value of a variable to a parameter using the parameter mapping.

6.1.1. Swimlanes

User task nodes can be used in combination with swimlanes to assign multiple human tasks to the similar actors. Tasks in the same swimlane will be assigned 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 the swimlane as well. All other tasks that will be created in that swimlane will use that actorId as well, even if an actorId has been specified for the task as well.

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.

6.2. Human task management component

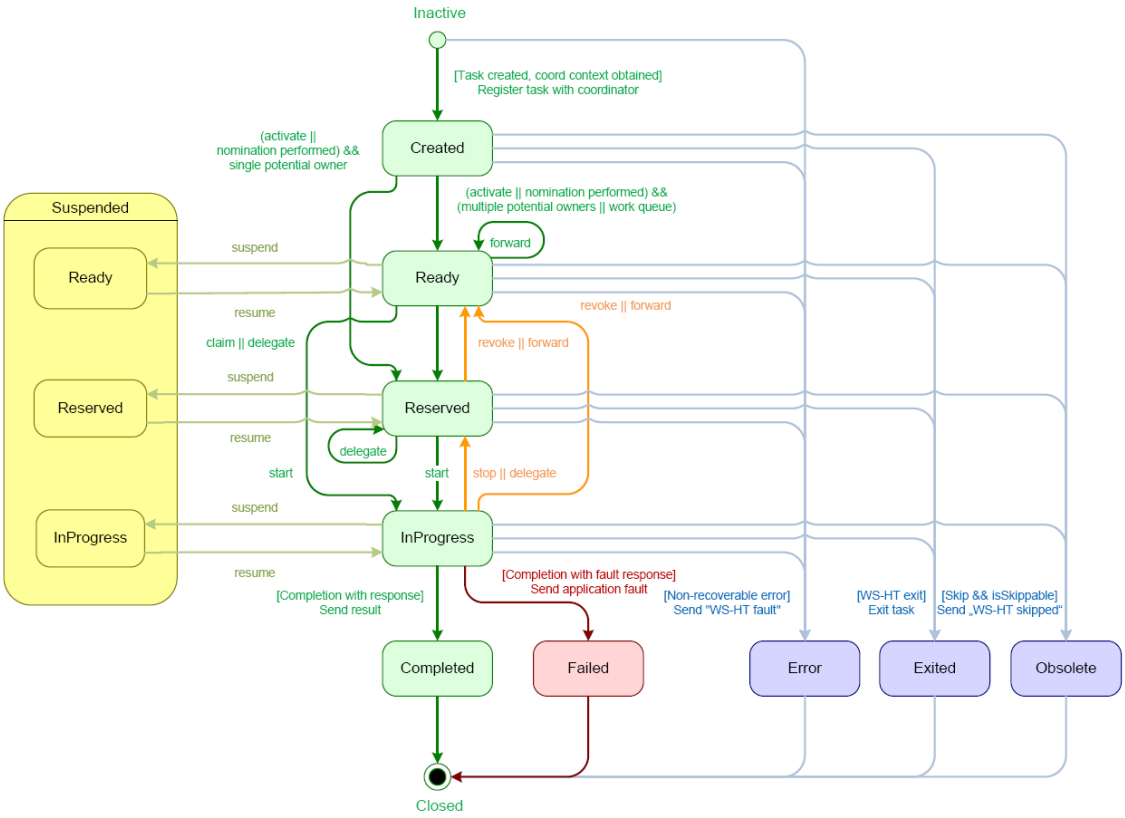
As far as the jBPM engine is concerned, human tasks are similar to any other external service that needs to be invoked and are implemented as an extension of normal work items. As a result, the process itself only contains an abstract description of the human tasks that need to be executed, and a work item handler is responsible for binding this abstract tasks to a specific implementation. Using our pluggable work item handler approach (see the chapter on domain-specific processes for more details), users can plug in any back-end implementation.

We do however provide an implementation of such a human task management component based on the WS-HumanTask specification. If you do not have the requirement to integrate a specific human task component yourself, you can use this service. It manages the task life cycle of the tasks (creation, claiming, completion, etc.) and stores the state of the task persistently. It also supports features like internationalization, calendar integration, different types of assignments, delegation, deadlines, etc.

Because we did not want to implement a custom solution when a standard is available, we chose to implement our service based on the WS-HumanTask (WS-HT) specification. This specification defines in detail the model of the tasks, the life cycle, and a lot of other features as the ones mentioned above. It is pretty comprehensive and can be found *here* [http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf].

6.2.1. Task life cycle

Looking from the perspective of the process, whenever a user task node is triggered during the execution of a process instance, a human task is created. The process will only continue from that point when that human task has been completed or aborted (unless of course you specify that the process does not need to wait for the human task to complete, by setting the "Wait for completion" property to true). However, the human task usually has a separate life cycle itself. We will now shortly introduce this life cycle, as shown in the figure below. For more details, check out the WS-HumanTask specification.



Whenever a task is created, it starts in the "Created" stage. It usually automatically transfers to the "Ready" state, at which point the task will show up on the task list of all the actors that are allowed to execute the task. There, it is waiting for one of these actors to claim the task, indicating that he or she will be executing the task. Once a user has claimed a task, the status is changed to "Reserved". Note that a task that only has one potential actor will automatically be assigned to that actor upon creation of that task. After claiming the task, that user can then at some point decide to start executing the task, in which case the task status is changed to "InProgress". Finally, once the task has been performed, the user must complete the task (and can specify the result data related to the task), in which case the status is changed to "Completed". If the task could not be completed, the user can also indicate this using a fault response (possibly with fault data associated), in which case the status is changed to "Failed".

The life cycle explained above is the normal life cycle. The service also allows a lot of other life cycle methods, like:

- Delegating or forwarding a task, in which case it is assigned to another actor
- Revoking a task, so it is no longer claimed by one specific actor but reappears on the task list of all potential actors
- Temporarly 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

6.2.2. Linking the task component to the jBPM engine

The task management component needs to be integrated with the jBPM engine just like any other external service, by registering a work item handler that is responsible for translating the abstract work item (in this case a human task) to a specific invocation. We have implemented such a work item handler (org.jbpm.process.workitem.wsht.WSHumanTaskHandler in the jbpm-human-task module) so you can easily link this work item handler like this:

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

By default, this handler will connect to the human task management component on the local machine on port 9123, but you can easily change that by invoking the setConnection(ipAddress, port) method on the WSHumanTaskHandler.

If you are using persistence for the session (check out the chapter on persistence for more information), you should use the org.jbpm.process.workitem.wsht.CommandBasedWSHumanTaskHandler as that makes sure that the state of the process instances is persisted correctly after interacting with the process engine.

The communication between the human task service and the process engine, or any task client, is done using messages being sent between the client and the server. The implementation allows different transport mechanisms being plugged in, but by default, Mina *(http://mina.apache.org/)* [http://mina.apache.org/] is used for client/server communication. An alternative implementation using HornetQ is also available.

A task client offers the following methods for managing the life cycle of human tasks:

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

```
...
```

You can either use these methods directly, or probably use some kind of GUI that the end user will use to lookup and execute the tasks that are assigned to them. If you take a look a the method signatures you will notice that almost all of this method takes the following arguments:

- **taskId**: The id of the task that we are working with. This is usually extract 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 usually asynchronous, so you should use a response handler that will be notified when the results are available.

As you can imagine all the methods create a message that will be sent to the server, and the server will execute the logic that implements the correct action.

6.2.3. Starting the Task Management Component

The task management component is a completely independent service that the process engine communicates with. We therefore recommend to start it as a separate service as well. The 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.getSystemEventListener());
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.drools.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.

6.2.4. Interacting With the Task Management Component

The task management component exposes various methods to manage the life cycle of the tasks through a Java API. This allows clients to integrate (at a low level) with the task management component. Note that end users should probably not interact with this low-level API directly but rather use one of the task list clients (see below). These clients interact with the task management component using this API. The following code sample shows how to create a task client and interact with the task service to create, start and complete a task.

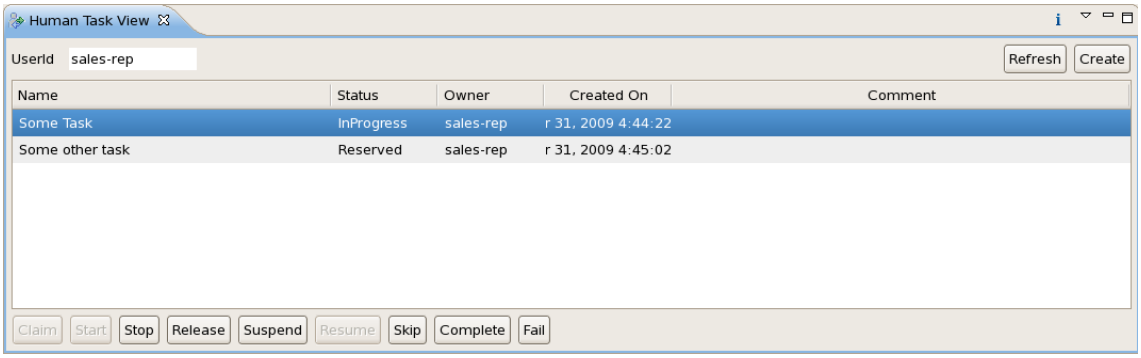
```
TaskClient client = new TaskClient(new MinaTaskClientConnector("client 1",
    new MinaTaskClientHandler(SystemEventListenerFactory.getSystemEventListener())));
client.connect("127.0.0.1", 9123);

// 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 );

// completing a task
```

```
responseHandler = new BlockingTaskOperationResponseHandler();
client.complete( taskId, users.get( "bobba" ).getId(), null, responseHandler );
```

6.3. Human Task Management Interface

6.3.1. Eclipse integration

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).



6.3.2. Web-based Task View

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

Chapter 7. Domain-specific processes

7.1. Introduction

One of the goals of jBPM is to allow users to extend the default process constructs with domain-specific extensions that simplify development in a particular application domain. This tutorial describes how to take your first steps towards domain-specific processes. Note that you don't need to be a jBPM expert to define your own domain-specific nodes, this should be considered integration code that a normal developer with some experience in jBPM can do himself.

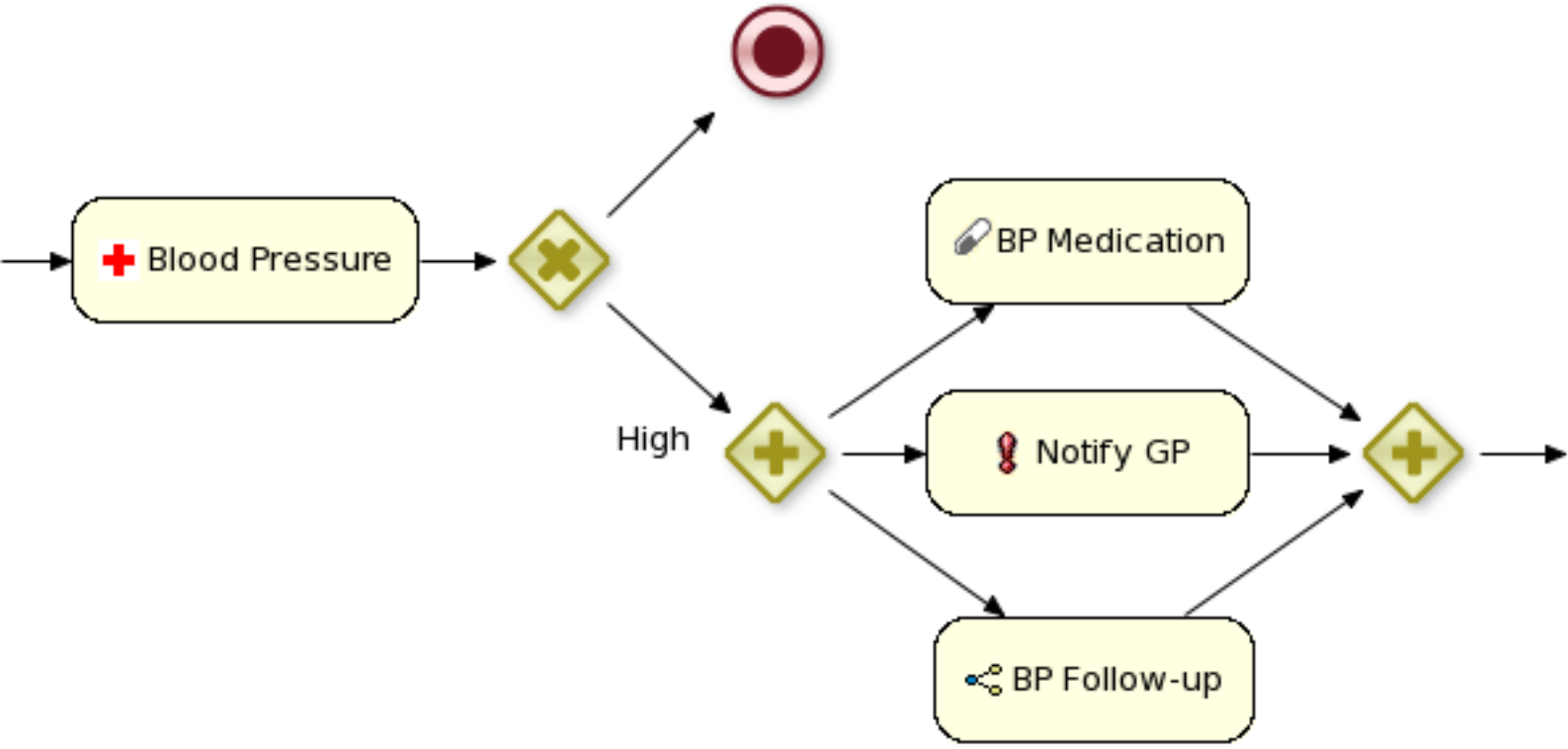
Most process languages offer some generic action (node) construct that allows plugging in custum user actions. However, these actions are usually low-level, where the user is required to write custom code to implement the work that should be incorporated in the process. The code is also closely linked to a specific target environment, making it difficult to reuse the process in different contexts.

Domain-specific languages are targeted to one particular application domain and therefore can offer constructs that are closely related to the problem the user is trying to solve. This makes the processes and easier to understand and self-documenting. We will show you how to define domain-specific work items (also called service nodes), which represent atomic units of work that need to be executed. These service nodes specify the work that should be executed in the context of a process in a declarative manner, i.e. specifying what should be executed (and not how) on a higher level (no code) and hiding implementation details.

So we want service nodes that are:

1. domain-specific
2. declarative (what, not how)
3. high-level (no code)
4. customizable to the context

Users can easily define their own set of domain-specific service nodes and integrate them in our process language. For example, the next figure shows an example of a process in a healthcare context. The process includes domain-specific service nodes for ordering nursing tasks (e.g. measuring blood pressure), prescribing medication and notifying care providers.



7.2. Example: Notifications

Let's start by showing you how to include a simple work item for sending notifications. A work item represent an atomic unit of work in a declarative way. It is defined by a unique name and additional parameters that can be used to 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 thus be defined using a work definition with four parameters and no results:

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

7.2.1. Creating the work definition

All wprk definitions must be specified in one or more configuration files in the project classpath, where all the properties are specified as name-value pairs. Parameters and results are maps where each parameter name is also mapped to the expected data type. Note that this configuration file also includes some additional user interface information, like the icon and the display name

of the work item. (We use MVEL for reading in the configuration file, which allows us to do more advanced configuration files). Our MyWorkDefinitions.conf file looks like this:

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

]
```

7.2.2. Registering the work definition

The configuration API can be used to register work definition files for your project using the drools.workDefinitions property, which represents a list of files containing work definitions (separated usings spaces). For example, include a drools.rulebase.conf file in the META-INF directory of your project and add the following line:

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

7.2.3. Using your new work item in your processes

Once our work definition has been created and registered, 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.

Apart from the properties defined by for this work item, all work items also have these three properties:

1. Parameter Mapping: Allows you 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.

7.2.4. Executing service nodes

The Drools engine contains a WorkItemManager that is responsible for executing work items whenever necessary. The WorkItemManager is responsible for delegating the work items to WorkItemHandlers that execute the work item and notify the WorkItemManager when the work item has been completed. For executing notification work items, a NotificationWorkItemHandler should be created (implementing the WorkItemHandler interface):

```
package com.sample;

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

public class NotificationWorkItemHandler implements WorkItemHandler {

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

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

}
```

This WorkItemHandler sends a notification as an email and then immediate 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 being aborted before it has been completed. The abort method can be used to specify how to abort such work items.

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

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

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

1. The process is more declarative, specifying what should be executed, not how.
2. Changes to the environment can be implemented by adapting the work item handler. The process itself should not be changed. It is also possible to use the same process in different environments, where the work item handler is responsible for integrating with the right services.
3. It is easy to share work item handlers across processes and projects (which would be more difficult if the code would be embedded in the process itself).
4. 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.

Chapter 8. Persistence

jBPM allows the persistent storage of certain information, i.e., the process runtime state, the history information, etc.

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.

8.1.1. Binary Persistence

jBPM provides a binary persistence mechanism that allows you to save the state of a process instance as a binary dataset. This way, the state of all running process instances can always be stored in a persistent location. Note that these binary datasets usually are 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, possibly, some variable values.

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, after its start or continuation from a wait state, the engine proceeds until no more actions can be performed. 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. It is, however, pretty straightforward to configure the engine to do this, by adding a configuration file and the necessary dependencies. Persistence itself is based on the Java Persistence API (JPA) and can thus work with several persistence mechanisms. We are using Hibernate by default, but feel free to employ alternatives.

A H2 database is used underneath to store the data, but you might choose your own alternative for this, too.

First of all, you need to add the necessary dependencies to your classpath. If you're using the Eclipse IDE, you can do that by adding the jar files to your Drools runtime directory, or by manually adding these 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, the H2 database and Bitronix for JTA-based transaction management, the following list of additional dependencies is needed:

1. jbpm-persistence-jpa (org.jbpm)
2. drools-persistence-jpa (org.drools)
3. persistence-api (javax.persistence)
4. hibernate-entitymanager (org.hibernate)
5. hibernate-annotations (org.hibernate)
6. hibernate-commons-annotations (org.hibernate)
7. hibernate-core (org.hibernate)
8. dom4j (dom4j)
9. jta (javax.transaction)
10. btm (org.codehaus.btm)
11. javassist (javassist)
12. slf4j-api (org.slf4j)
13. slf4j-jdk14 (org.slf4j)
14. h2 (com.h2database)
15. commons-collections (commons-collections)

Next, you need to configure the jbpm engine to save the state of the engine whenever necessary. The easiest way to do this is to use `JPAKnowledgeService` to create your knowledge session, 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 yse 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 );
```

Note that we only save the minimal state that is needed to continue execution of the process instance at some later point. This means, for example, that it does not contain information about already executed nodes if that information is no longer relevant, or that process instances that have been completed or aborted are removed from the database. If you want to search for history-related information, you should use the history log, as explained later.

You need to add a persistence configuration to your classpath to configures JPA to use Hibernate and the H2 database (or your 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">
   <provider>org.hibernate.ejb.HibernatePersistence</provider>
   <jta-data-source>jdbc/processInstanceDS</jta-data-source>
   <class>org.drools.persistence.session.SessionInfo</class>
   <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
   <class>org.jbpm.persistence.processinstance.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/processInstanceDS". The following Java fragment could be used to set up this data source, where we are using the file-based H2 database.

```
PoolingDataSource ds = new PoolingDataSource();
ds.setUniqueName("jdbc/testDS1");
ds.setClassName("org.h2.jdbcx.JdbcDataSource");
ds.setMaxPoolSize(3);
ds.setAllowLocalTransactions(true);
ds.getDriverProperties().put("user", "sa");
ds.getDriverProperties().put("password", "sasa");
ds.getDriverProperties().put("URL", "jdbc:h2:file:/NotBackedUp/data/process-instance-db");
ds.init();
```

If you're deploying to an application server, you can usually 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/testDS1</jndi-name>
   <connection-url>jdbc:h2:file:/NotBackedUp/data/process-instance-db</connection-url>
   <driver-class>org.h2.jdbcx.JdbcDataSource</driver-class>
   <user-name>sa</user-name>
```

```
    <password>sasa</password>
  </local-tx-datasource>
</datasources>
```

8.1.4. Transactions

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();
```

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 provides exactly that. It consists of a repository for storing different kinds of knowledge, not only process definitions but also rules, object models, etc. It allows easy retrieval of this knowledge using WebDAV or by employing a knowledge agent that automatically downloads the information from Guvnor when creating a knowledge base, and provides a web application that allows business users to view and possibly update the information in the knowledge repository. Check out the Drools Guvnor documentation for more information on how to do this.

8.3. History Log

In many cases it is useful (if not necessary) to store information about the execution of process instances, so that this information can be used afterwards, for example, to verify what actions have been executed for a particular process instance, or to monitor and analyze the efficiency of a particular process. Storing history information in the runtime database is usually not a good idea, as this would result in ever-growing runtime data, and monitoring and analysis queries might influence the performance of your runtime engine. That is why history information about the execution of process instances is stored separately.

This history log of execution information is created based on the events generated by the process engine during execution. The Drools runtime engine provides a generic mechanism to listen to different kinds of events. The necessary information can easily be extracted from these events and made persistent, for example in a database. Filters can be used to only store the information you find relevant.

8.3.1. Storing Process Events in a Database

The jbpm-bam module contains an event listener that stores process-related information in a database using JPA or Hibernate directly. The database contains two tables, one for process instance information and one for node instance information (see the figure below):

1. *ProcessInstanceLog:* This lists the process instance id, the process (definition) id, the start date and (if applicable) the end date of all process instances.

2. *NodeInstanceLog:* This table contains more detailed 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. For this, it stores the process instance id and the process id of the process instance it is being executed in, and the node instance id and the corresponding

node id (in the process definition) of the node instance in question. Finally, the type of event (0 = enter, 1 = exit) and the date of the event is stored as well.

ID	PROCESSINSTANCEID	PROCESSID	START_DATE

ID	TYPE	NODEINSTANCEID	NODEID	PROCESSINSTANCEID	PROCESSID	LOG_DATE

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.session.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.jbpm.persistence.processinstance.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. 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.

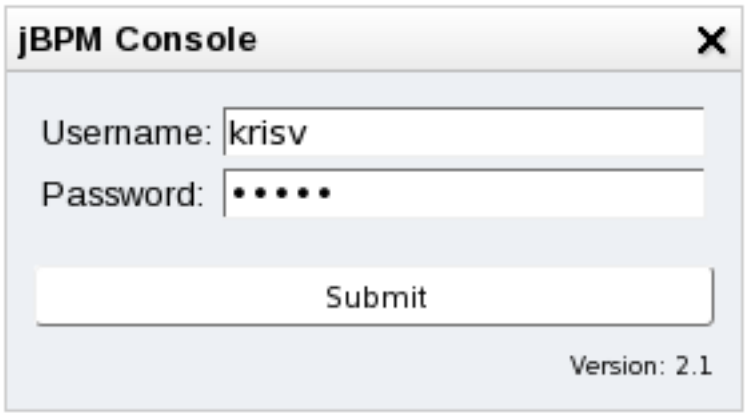
9.1. Installation

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.

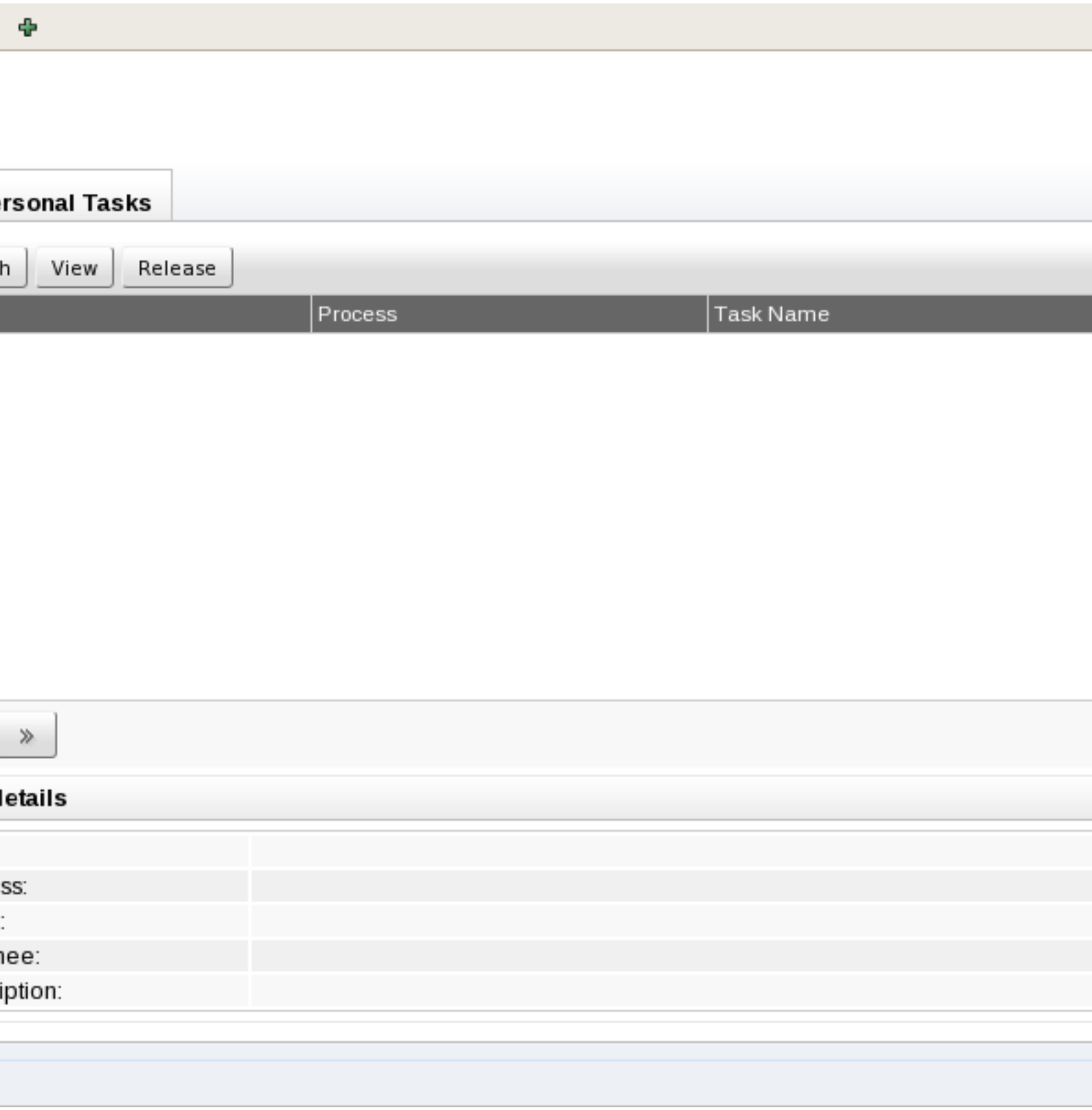
9.2. Running the process management console

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.

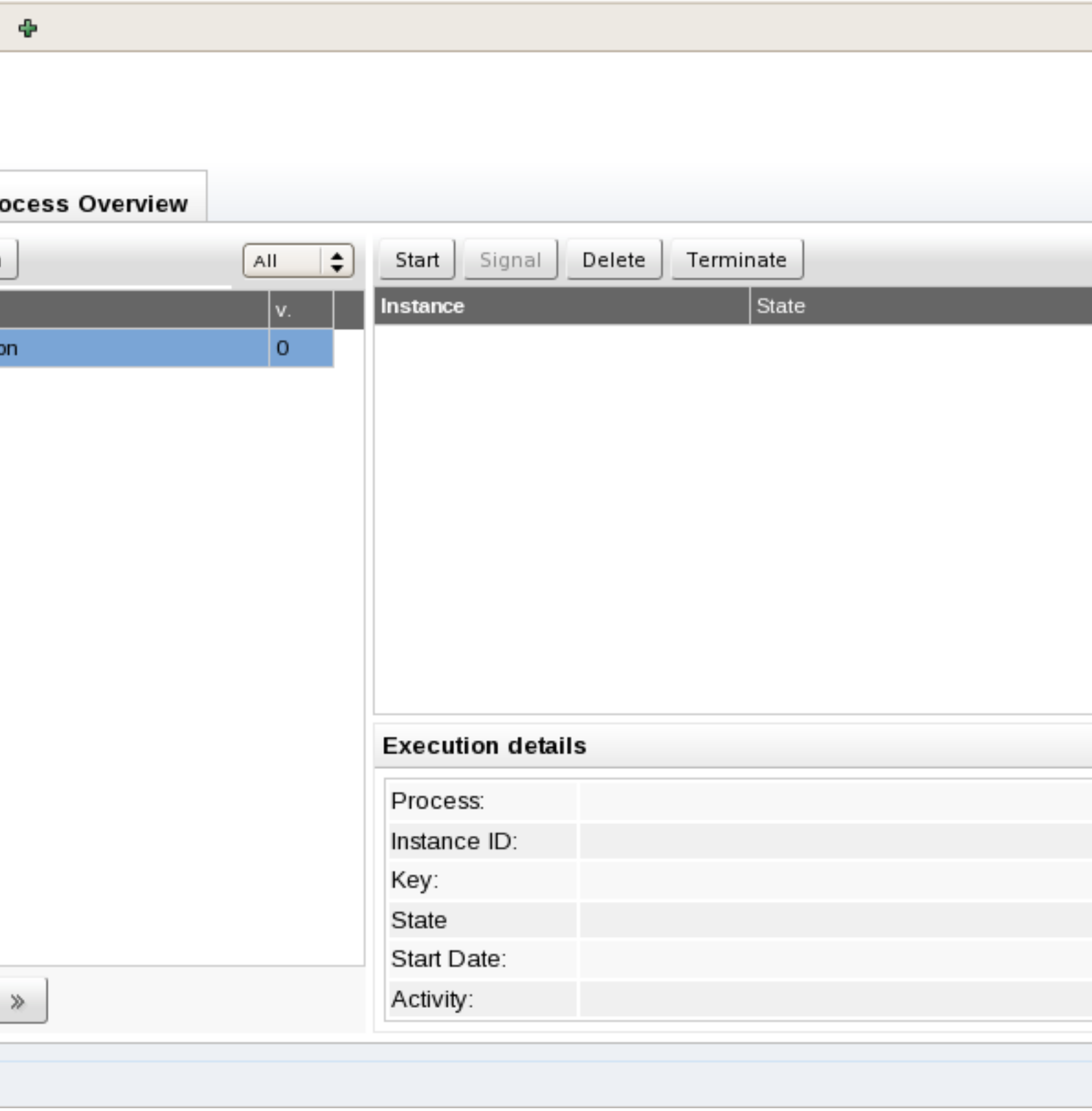

rsonal Tasks
h
View
Release
Process
Task Name
»
letails
ss:
:
nee:
iption:

9.2.1. 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).

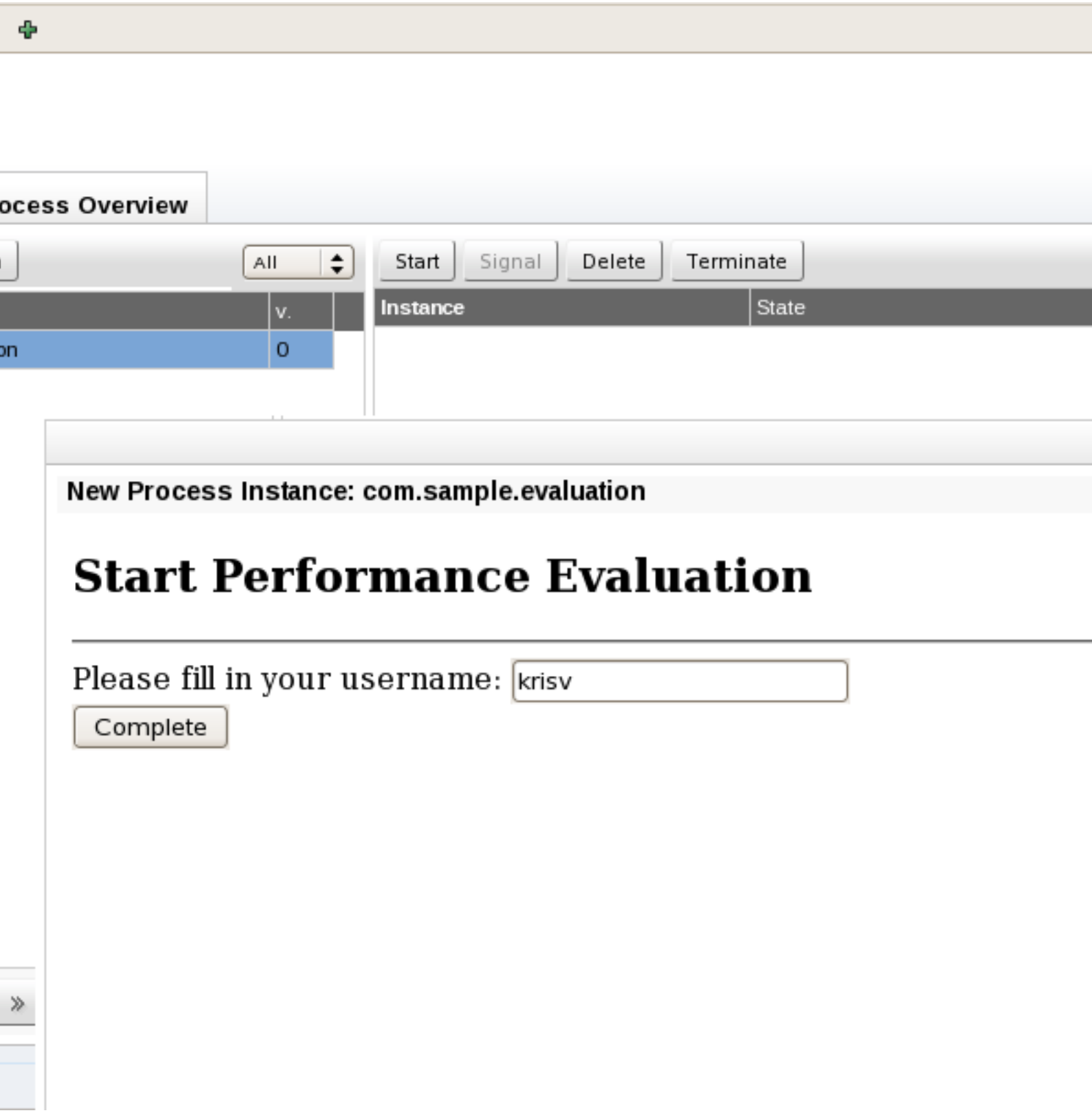
9.2.1.1. Inspecting process definitions

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.


ocess Overview
All
v.
on
0
Start
Signal
Delete
Terminate
Instance
State
Execution details
Process:
Instance ID:
Key:
State
Start Date:
Activity:

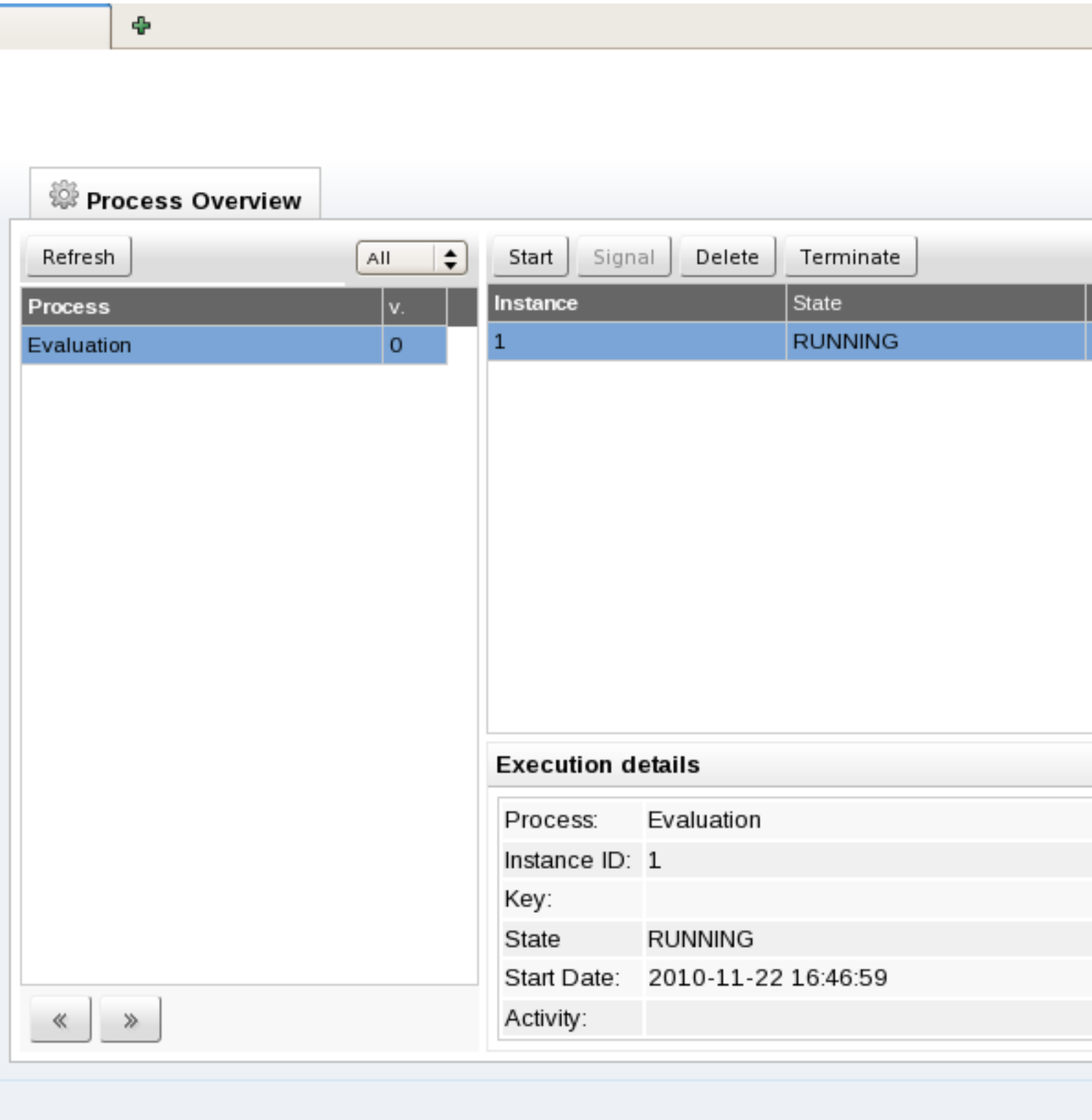
9.2.1.2. Starting new process instances

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.


ocess Overview
All
Start
Signal
Delete
Terminate
v.
Instance
State
0
New Process Instance: com.sample.evaluation
Start Performance Evaluation
Please fill in your username:
krisv
Complete

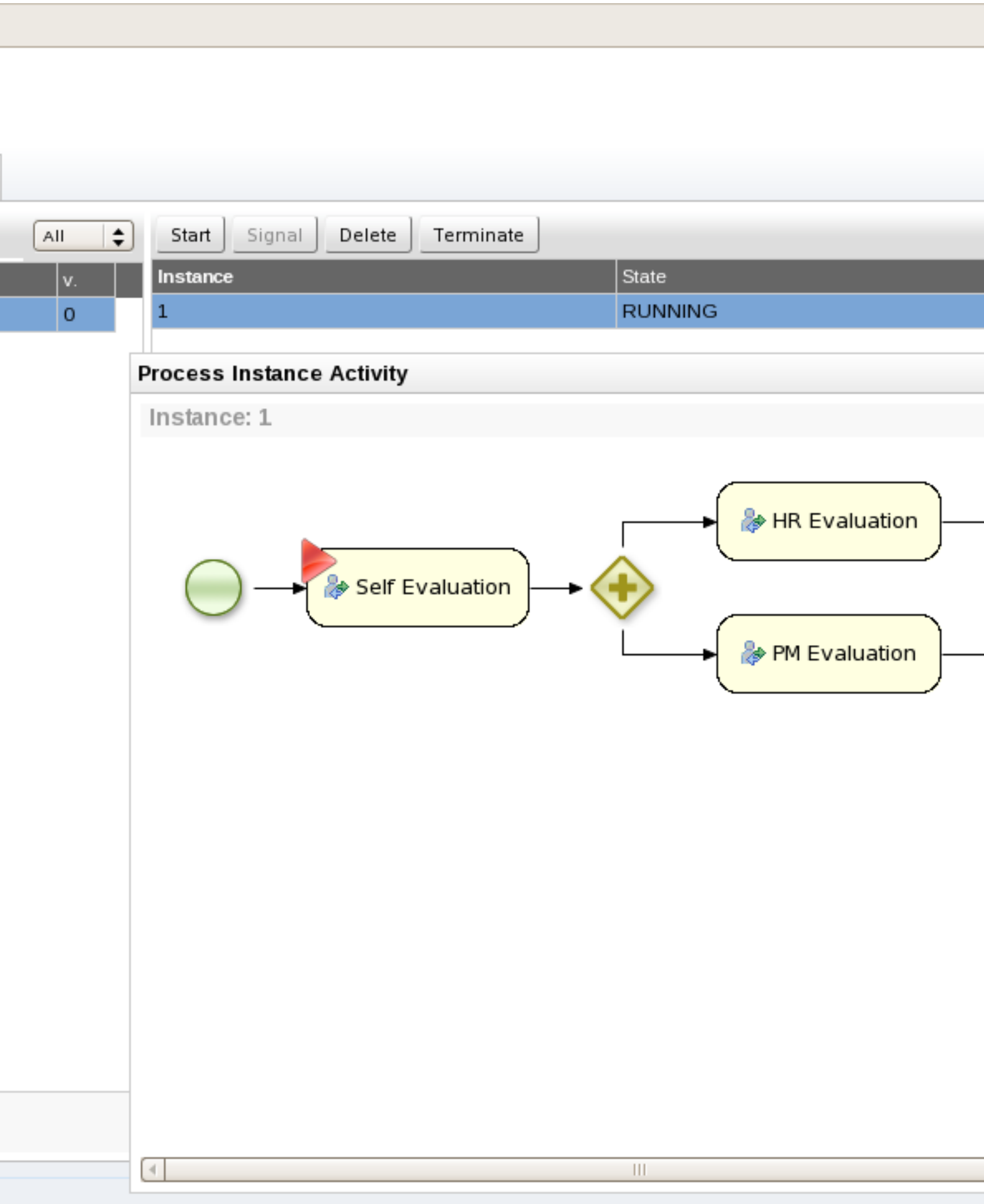
9.2.1.3. Managing process instances

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.


Process Overview
Refresh
All
Start
Signal
Delete
Terminate
Process
v.
Evaluation
0
Instance
State
1
RUNNING
Execution details
Process:
Evaluation
Instance ID:
1
Key:
State
RUNNING
Start Date:
2010-11-22 16:46:59
Activity:

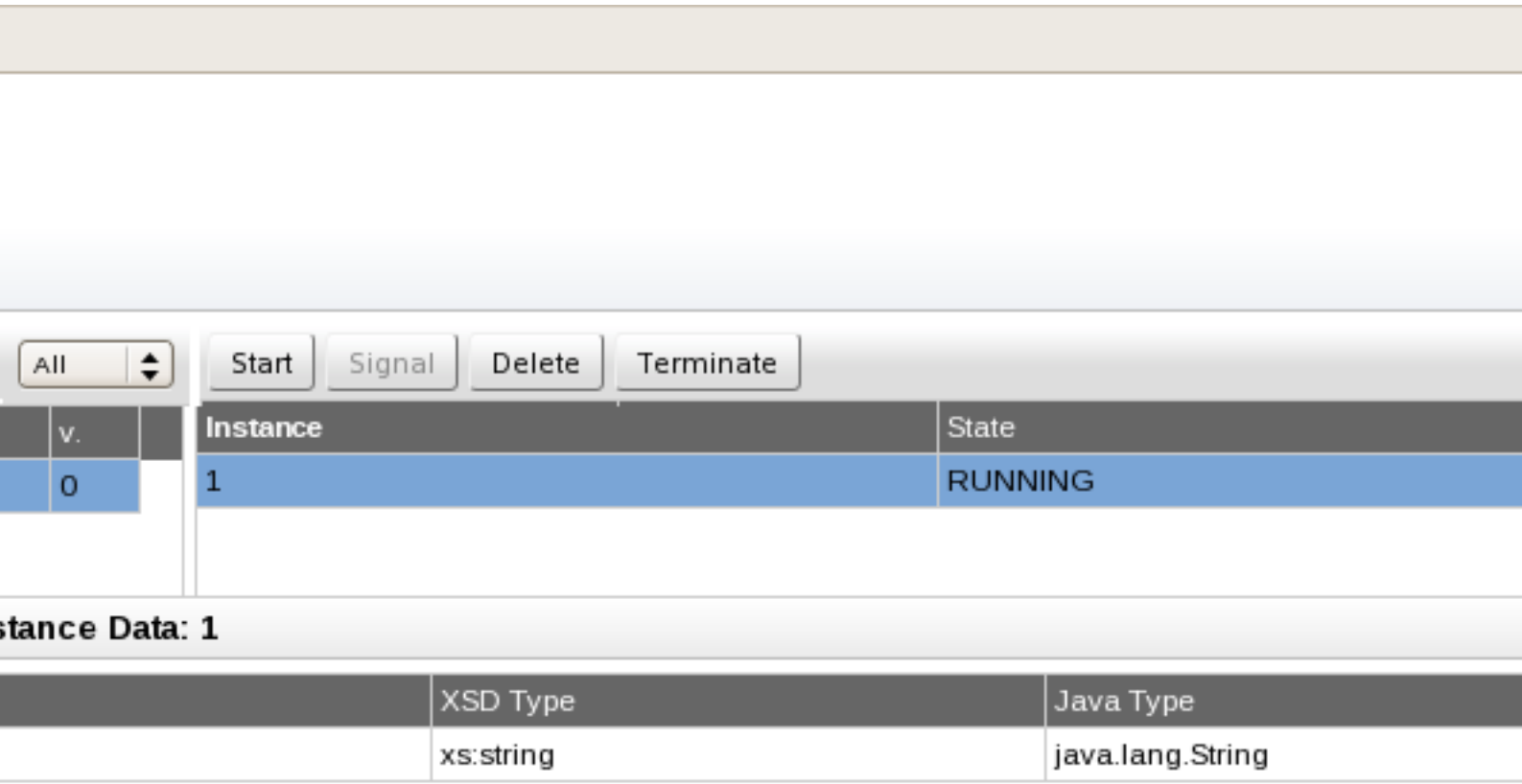
9.2.1.4. Inspecting process instance state

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 traingle 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.]


All
Start
Signal
Delete
Terminate
V.
0
Instance
State
1
RUNNING
Process Instance Activity
Instance: 1
Self Evaluation
HR Evaluation
PM Evaluation

9.2.1.5. 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 it's corresponding value for that specific process instance.


All
Start
Signal
Delete
Terminate
v.
0
Instance
State
1
RUNNING
stance Data: 1
XSD Type
Java Type
xs:string
java.lang.String

9.2.2. 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.

	Process	Task Name
		Performance Evaluation

ance Evaluation

evaluation

a self-evalutation.

following evaluation form:
performance: Outstanding

pply:
tiative
ange
nication skills

9.2.3. 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.

This report doesn't require any paramters.

Business Activity Monitori

Instances / Hour

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

Please enter a process definition id

com.sampl

Business Activity Monitori

Process com.sample.evaluatio

com.sample.evaluation
1
1
1

ocess Instances

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.

9.3. 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 drools-process-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>
```

Data that is provided by the user when filling in the task form will be added as parameters when completing the task. For example, when completing the 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 parameters to process variables.

Forms should be included in the jbpm-gwt-form.jar in the server war.

9.4. 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

For example, this allows you to close a task using

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

or starting a new process instances using

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

Chapter 10. 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.

10.1. Reporting

By adding a history logger to the process engine, all relevent 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.

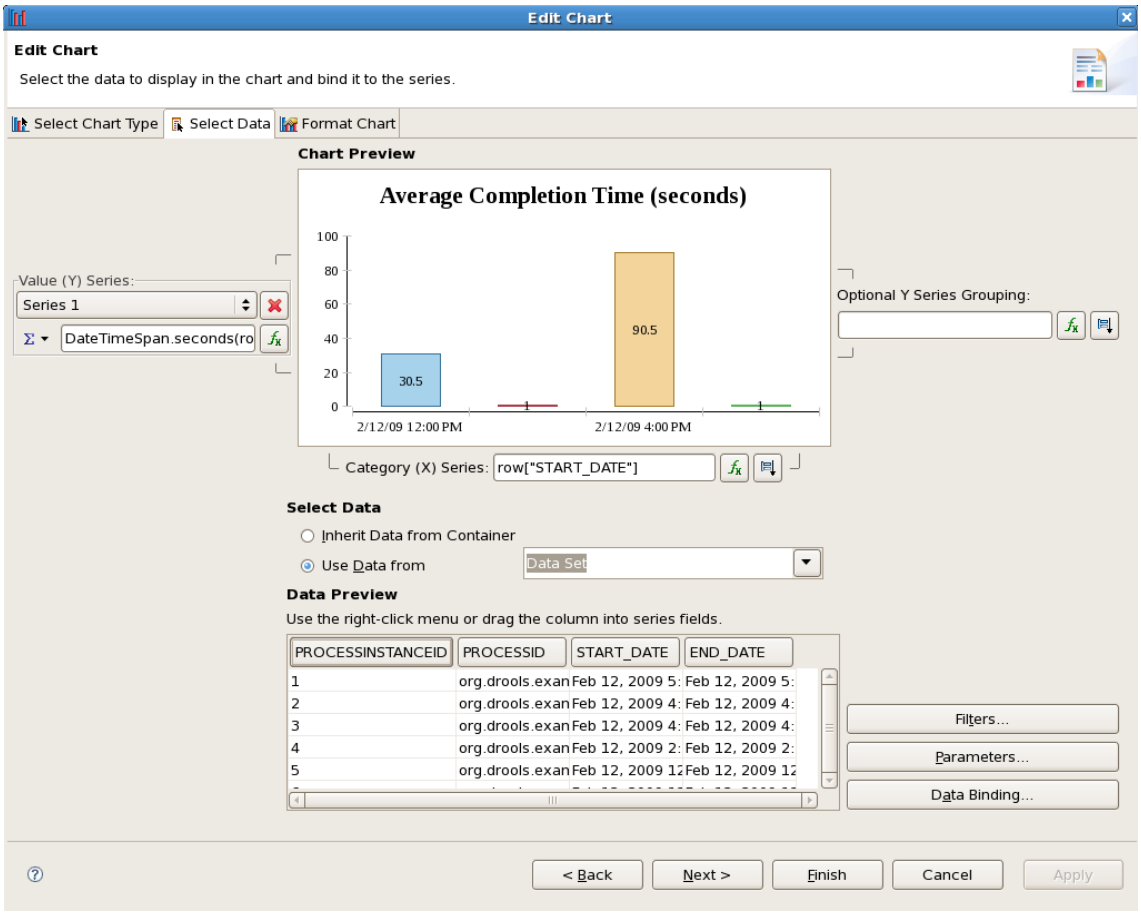


Figure 10.1. Creating a report using Eclipse BIRT

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.

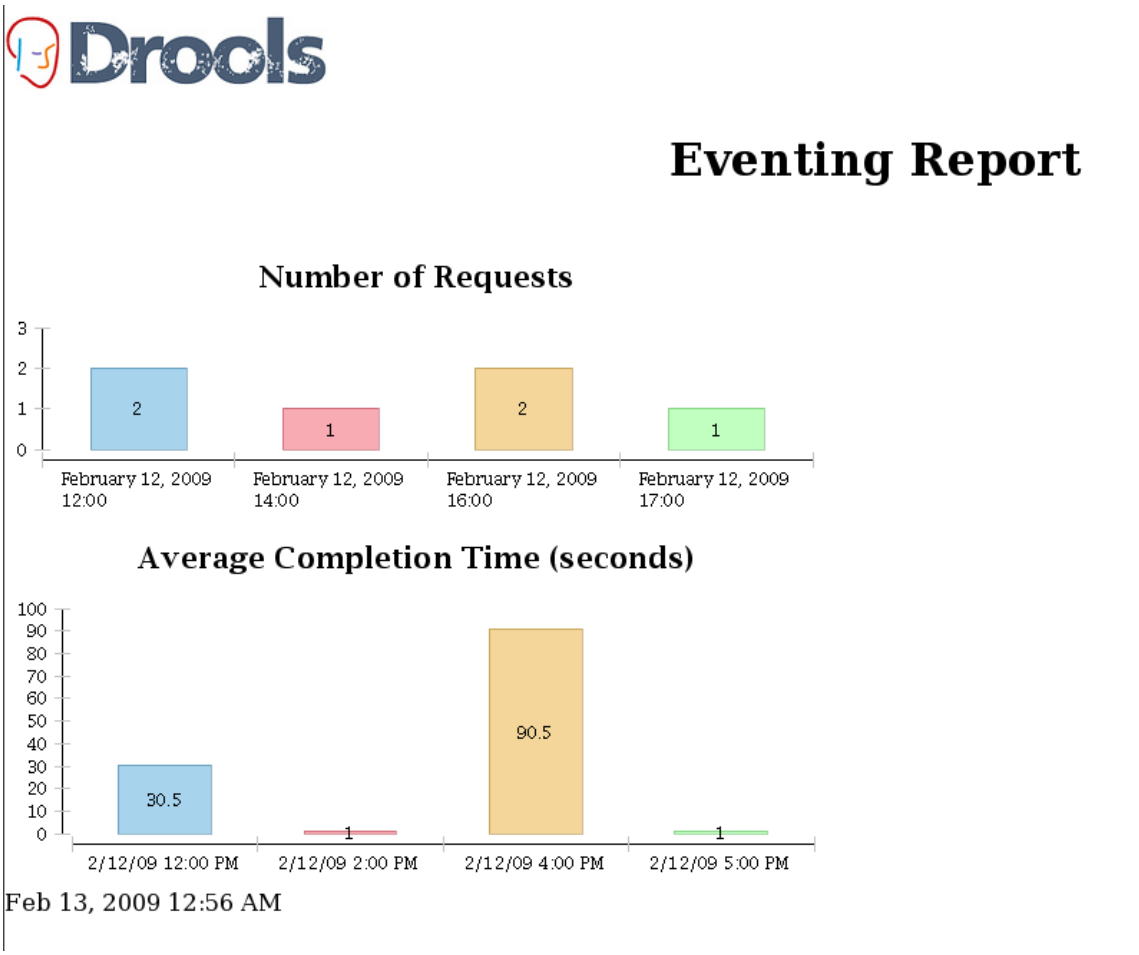


Figure 10.2. The eventing report

10.2. 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 situtation reverts back to normal, such rules would be removed again.

Index