JBoss Ruleflow 1.1

Ruleflow Guide

A User Guide for Ruleflow

David Le Sage (Product Version)

JBoss Ruleflow 1.1 Ruleflow Guide
A User Guide for Ruleflow
Edition 1

Author David Le Sage (Product Version)

A guide to using Ruleflow with BRMS 5.1

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

[1] https://fedorahosted.org/liberation-fonts/

> **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

`Mono-spaced Bold Italic` or ***`Proportional Bold Italic`***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **`ssh`** ***`username@domain.name`*** at a shell prompt. If the remote machine is **`example.com`** and your username on that machine is john, type **`ssh john@example.com`**.
>
> The **`mount -o remount`** ***`file-system`*** command remounts the named file system. For example, to remount the **`/home`** file system, the command is **`mount -o remount /home`**.
>
> To see the version of a currently installed package, use the **`rpm -q`** ***`package`*** command. It will return a result as follows: ***`package-version-release`***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **`mono-spaced roman`** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **`mono-spaced roman`** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

```
public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

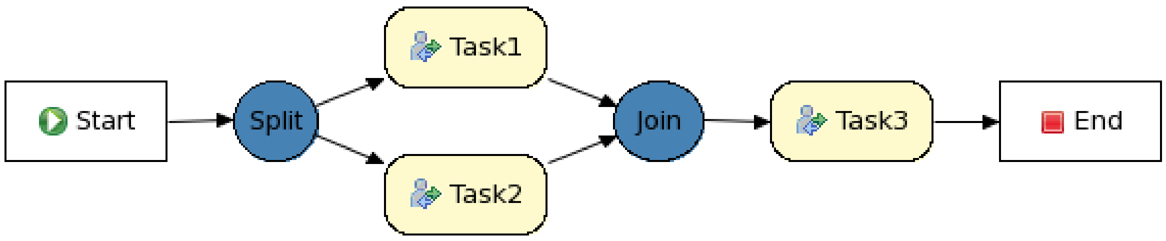
If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: *http://bugzilla.redhat.com/bugzilla/* against the product **Documentation.**

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

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

Introduction

JBoss Rules Flow is a workflow or process engine that allows advanced integration of processes and rules. A process or a workflow describes the order in which a series of steps need to be executed, using a flow chart. For example, the following figure shows a process where first Task1 and Task2 need to be executed in parallel. After completion of both, Task3 needs to be executed.



The following chapters will teach you everything you need to know about JBoss Rules Flow. Its distinguishing characteristics are:

1. Advanced integration of processes and rules: Processes and rules are usually considered as two different paradigms when it comes to defining business logic. While loose coupling between a processes and rules is possible by integrating both a process and a rules engine, we provide advanced integration of processes and rules out-of-the-box. This allows users to use rules to define part of their business logic when defining their business processes and vice versa.

2. Unification of processes and rules: We consider rules, processes and event processing all as different types of knowledge. Not only do we allow the advanced integration of these three types, we also offer a unified API and unified tooling so that users should not have to learn three different products but can easily combine these three types using our knowledge-based API. The tooling also allows seamless integration of these different kinds of knowledge, including things like a unified knowledge repository, audit logs, debugging, etc.

3. Declarative modelling: JBoss Rules Flow tries to keep processes as declarative as possible, i.e., focussing on what should happen instead of how. As a result, we try to avoid having to hardcode details into your process but offer ways to describe your work in an abstract way (e.g., by using pluggable work items, a business scripting language, etc.). We also allow users to easily create domain-specific extensions, making it much simpler to read, update or create these processes as they are using domain-specific concepts that are closely related to the problem at hand and can be understood by domain experts.

4. Generic process engine supporting multiple process languages: We do not believe that there is one process language that fits all purposes. Therefore, the JBoss Rules Flow engine is based on a generic process engine that allows the definition and execution of different types of process languages, like for example our RuleFlow language, WS-BPEL (a standard targeted towards web service orchestration), OSWorkflow (another existing workflow language), jPDL (the process language defined by the jBPM project), etc. All these languages are based on the same set of core building blocks, making it easier to implement your own process language by reusing and combining these low-level building blocks the way you want to.

Installer

JBoss Rules now comes with an installer.

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: drools-{version}-install.zip

You can for example find the latest snapshot release here.

http://hudson.jboss.org/hudson/job/drools/lastSuccessfulBuild/artifact/trunk/target/

2.3. Download the installer

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 Guvnor into JBoss AS
- Install the gwt-console into JBoss AS
- Install the Eclipse plugins

Once the demo setup has finished (this could take a while as it might have to download the various components), 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*[1] 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
- How to complete human tasks using the test human task client in Eclipse

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

2.5. Using Guvnor repository

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*[2] gives an overview of how to manage your repository. It shows you:

- How to import an existing sample repository, containing the performance evaluation process as shown in the previous section
- How to look up the processes that are part of a package
- How to build a package so it can be used for creating a session (like for example in the gwt-console as shown in the next section)

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

2.6. Using web management consoles

First make sure you have imported the sample repository and built the defaultPackage in Guvnor first (see previous section). Open up the process management console:

http://localhost:8080/gwt-console

Log in, using krisv / krisv as username / password. The *following screencast*[3] 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

[1] http://people.redhat.com/kverlaen/install-eclipse.swf
[2] http://people.redhat.com/kverlaen/install-guvnor.swf
[3] http://people.redhat.com/kverlaen/install-gwt-console.swf

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

Once you're done playing:

```
ant stop.demo
```

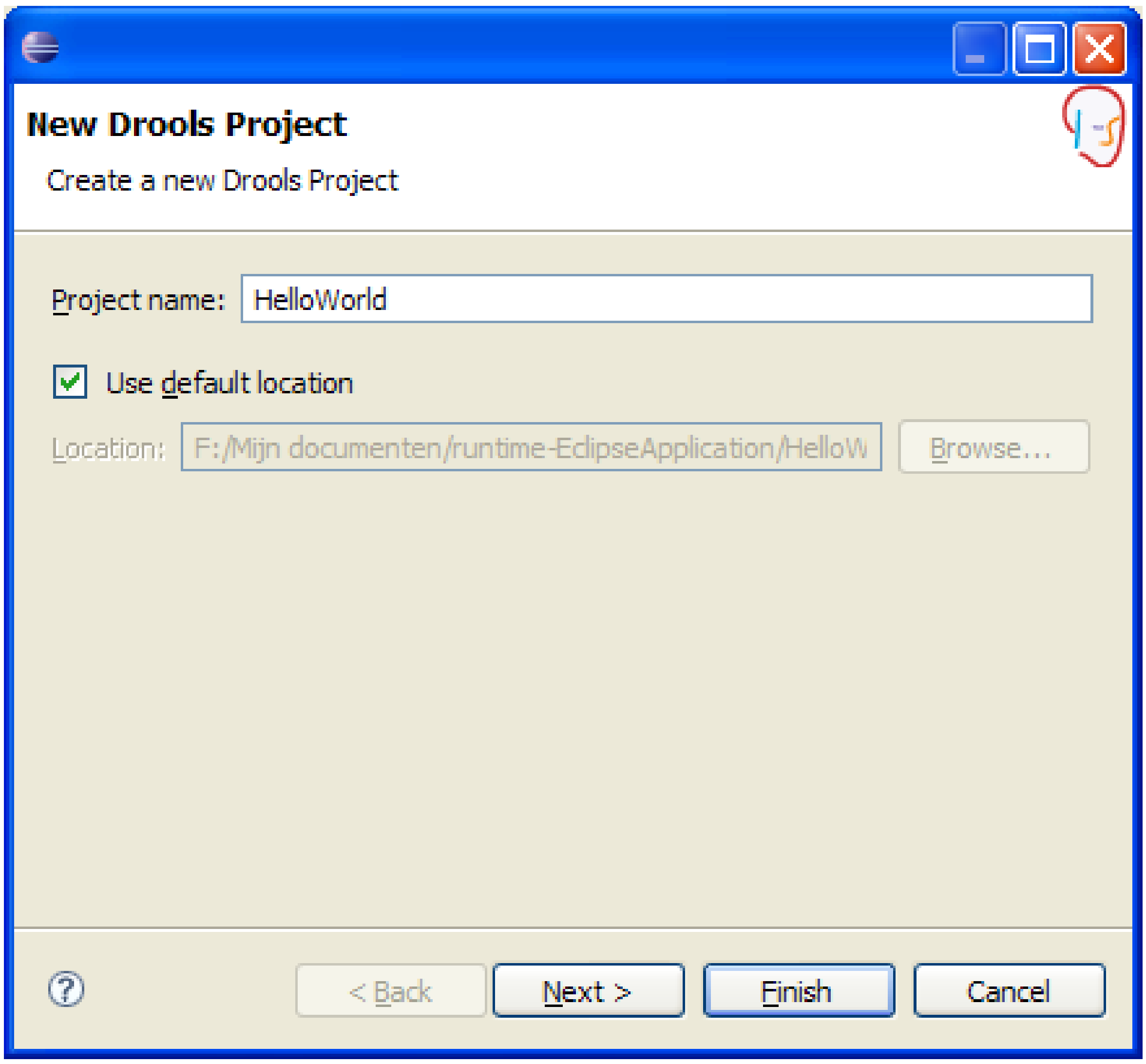
and simply close all the rest.

Getting Started

This section describes how to get started with JBoss Rules Flow. It will guide you to create and execute your first JBoss Rules Flow process.

3.1. Creating Your First Process

The JBoss Rules project wizard can be used to set up an executable project that contains the necessary files to get started easily with defining and executing processes. This wizard will set up a basic project structure, the classpath, a sample process and execution code to get you started. To create a new JBoss Rules project, simply left-click on the JBoss Rules action button (with the JBoss Rules head) in the IDE toolbar and select "New JBoss Rules Project". (Note that the JBoss Rules action button only shows up in the JBoss Rules perspective. To open the JBoss Rules perspective (if you haven't done so already), click the "Open Perspective" button in the top right corner of your IDE window, select "Other..." and pick the JBoss Rules perspective.) Alternatively, you could also select "File", then "New" followed by "Project ...", and in the JBoss Rules folder, select "JBoss Rules Project". This should open the following dialog:



New Drools Project

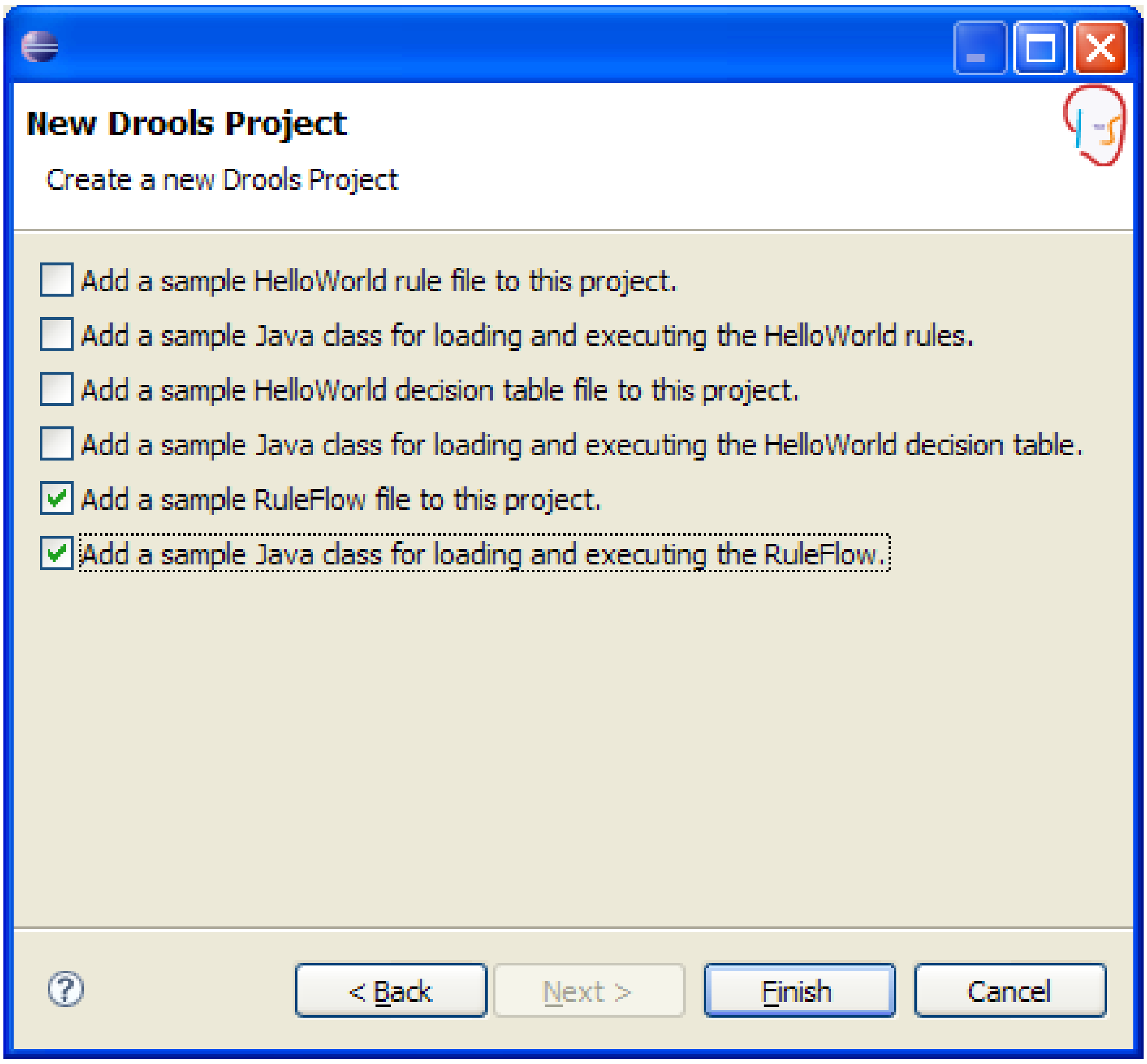
Create a new Drools Project

Project name: HelloWorld

[✓] Use default location

Location: F:/Mijn documenten/runtime-EclipseApplication/HelloW

Give your project a name and click "Next". In the following dialog you can select which elements are added to your project by default. Since we are creating a new process, deselect the first two checkboxes and select the last two. This will generate a sample process and a Java class to execute this process.

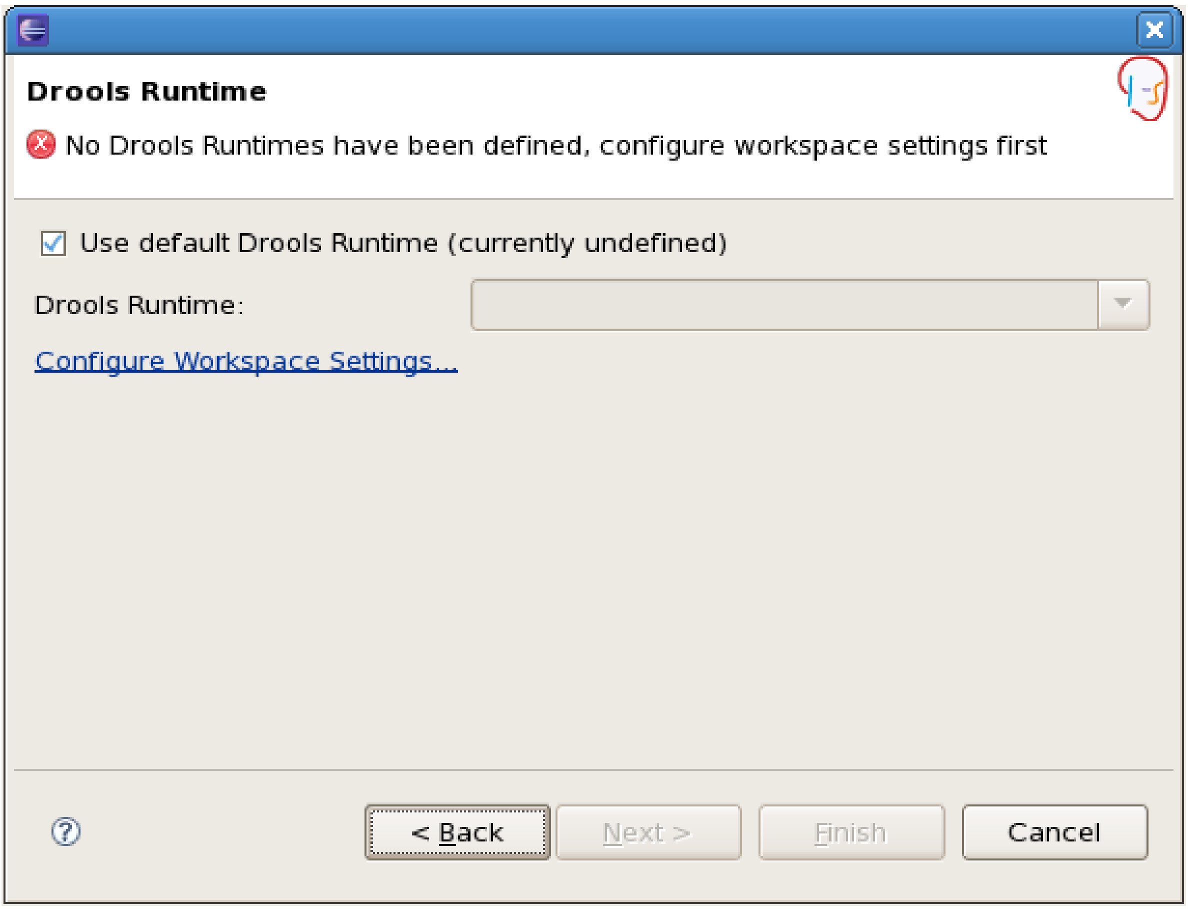

New Drools Project

Create a new Drools Project

- [ ] Add a sample HelloWorld rule file to this project.
- [ ] Add a sample Java class for loading and executing the HelloWorld rules.
- [ ] Add a sample HelloWorld decision table file to this project.
- [ ] Add a sample Java class for loading and executing the HelloWorld decision table.
- [✓] Add a sample RuleFlow file to this project.
- [✓] Add a sample Java class for loading and executing the RuleFlow.

If you have not yet set up a JBoss Rules runtime, you should do this now. A JBoss Rules runtime is a collection of jars on your file system that represent one specific release of the JBoss Rules project jars. To create a runtime, you must either point the IDE to the release of your choice, or you can simply create a new runtime on your file system from the jars included in the JBoss Rules IDE plugin. Since we simply want to use the JBoss Rules version included in this plugin, we will do the latter. Note that you will only have to do this once; next time you create a JBoss Rules project, it will automatically use the default JBoss Rules runtime (unless you specify otherwise).

Unless you have already set up a JBoss Rules runtime, click the "Next" button. The following dialog, as displayed below, shows up, telling you that you have not yet defined a default JBoss Rules runtime and that you should configure the workspace settings first. Do this by clicking on the "Configure Workspace Settings ..." link.


Drools Runtime

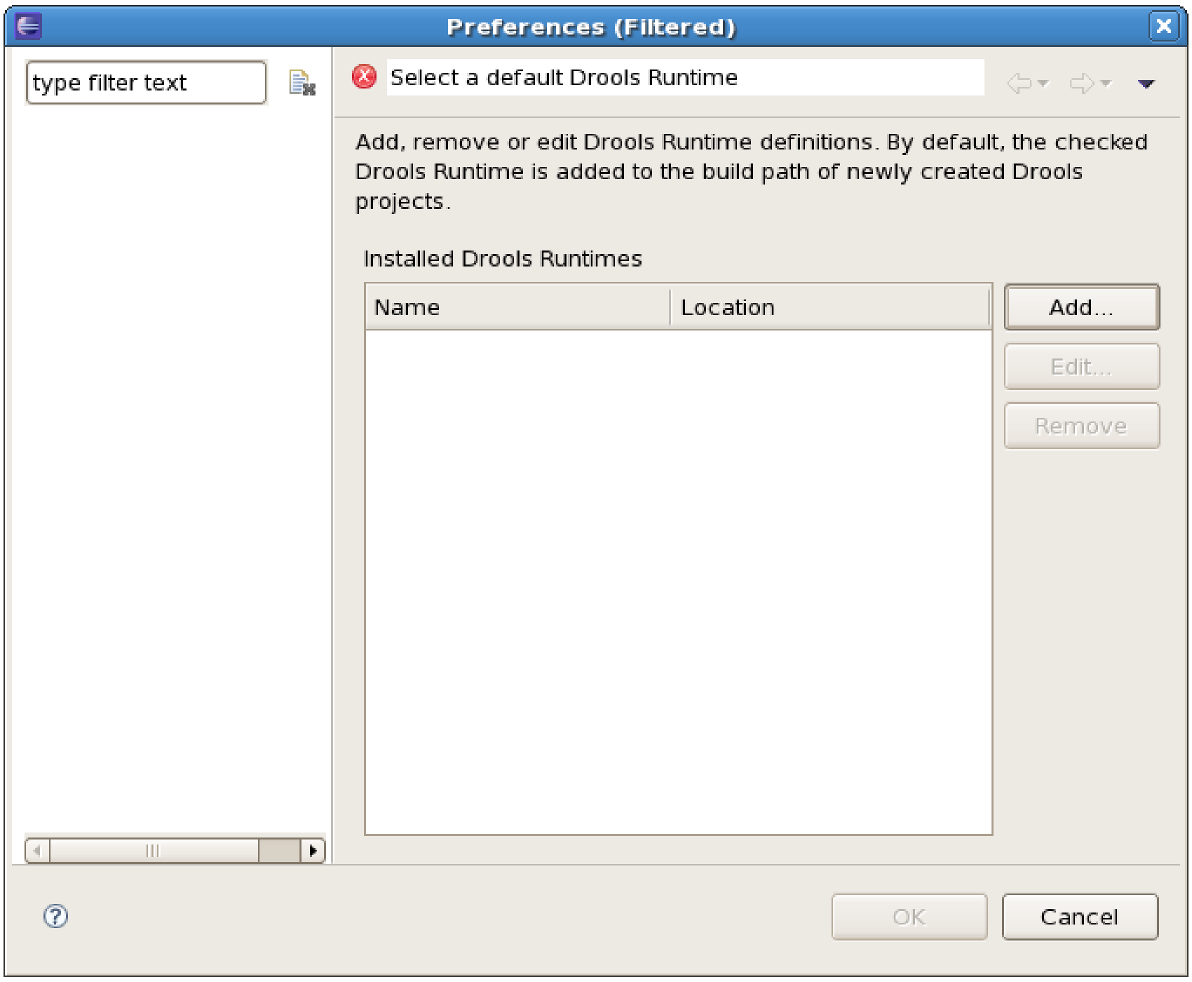
No Drools Runtimes have been defined, configure workspace settings first

☑ Use default Drools Runtime (currently undefined)

Drools Runtime:

Configure Workspace Settings...

The dialog that pops up shows the workspace settings for JBoss Rules runtimes. The first time you do this, the list of installed JBoss Rules runtimes is probably empty, as shown below. To create a new runtime on your file system, click the "Add..." button. This shows a dialog where you should give the new runtime a name (e.g. "JBoss Rules 5.0.0 runtime"), and a path to your JBoss Rules runtime on your file system. In this tutorial, we will simply create a new JBoss Rules 5 runtime from the jars embedded in the JBoss Rules IDE plugin. Click the "Create a new JBoss Rules 5 runtime ..." button and select the folder where you want this runtime to be stored and click the "OK" button. You will see the selected path showing up in the previous dialog. As we're all done here, click the "OK" button. You will see the newly created runtime shown in your list of JBoss Rules runtimes. Select this runtime as the new default runtime by clicking on the check box in front of your runtime name and click "OK". After successfully setting up your runtime, you can now finish the project creation wizard by clicking on the "Finish" button.



Select a default Drools Runtime

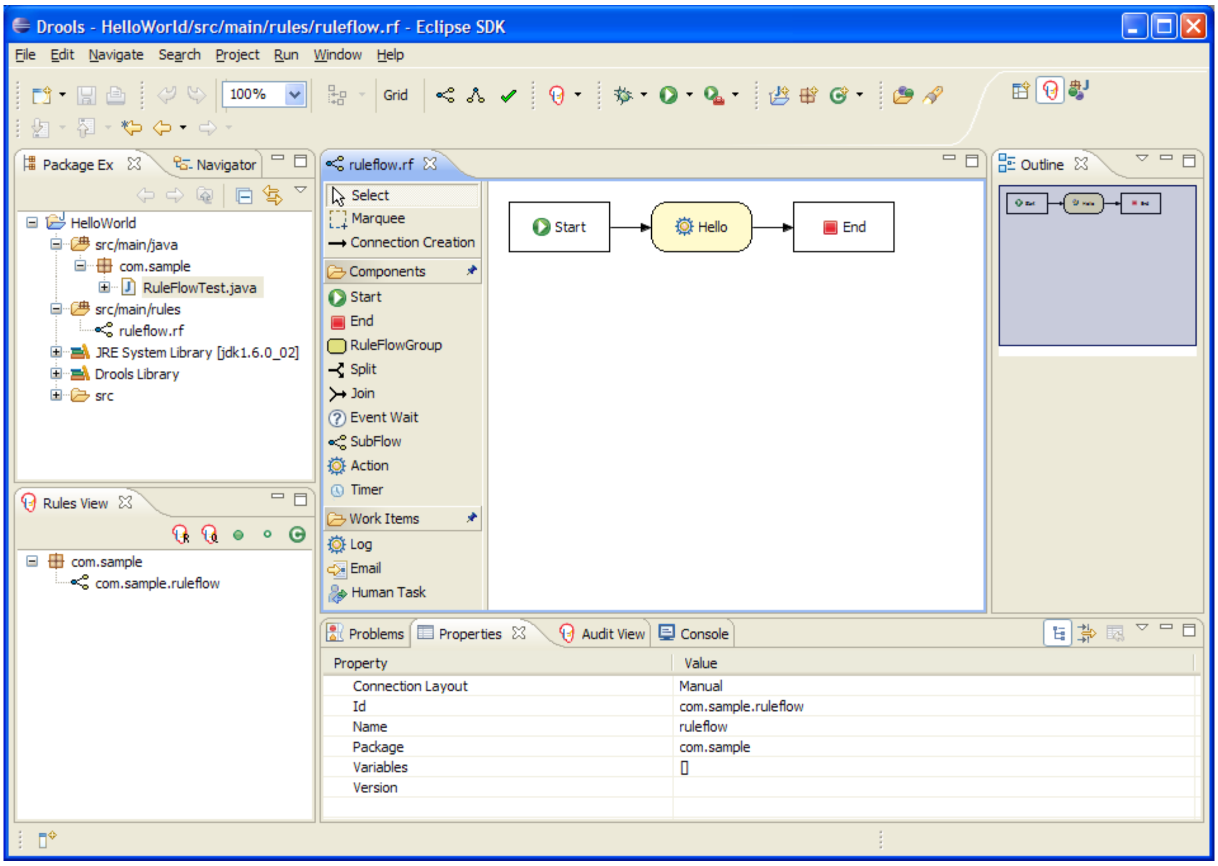
Add, remove or edit Drools Runtime definitions. By default, the checked Drools Runtime is added to the build path of newly created Drools projects.

Installed Drools Runtimes

Name	Location

The end result should look like this and contains:

1. **`ruleflow.rf`**: the process definition, which is a very simple process containing a Start node (the entry point), an Action node (that prints out "Hello World") and an End node (the end of the process).
2. **`RuleFlowTest.java`**: a Java class that executes the process.
3. The necessary libraries are automatically added to the project classpath as a JBoss Rules library.



Property	Value
Connection Layout	Manual
Id	com.sample.ruleflow
Name	ruleflow
Package	com.sample
Variables	[]
Version	

By double-clicking the **`ruleflow.rf`** file, the process will be opened in the RuleFlow editor. The RuleFlow editor contains a graphical representation of your process definition. It consists of nodes that are connected to each other. The editor shows the overall control flow, while the details of each of the elements can be viewed (and edited) in the Properties View at the bottom. The editor contains a palette at the left that can be used to drag-and-drop new nodes, and an outline view at the right.

This process is a simple sequence of three nodes. The Start node defines the start of the process. It is connected to an Action node (called "Hello" that simply prints out "Hello World" to the standard output. You can see this by clicking on the "Hello" node and checking the action property in the Properties View below. This node is then connected to an End node, signaling the end of the process.

While it is probably easier to edit processes using the graphical editor, users can also modify the underlying XML directly. The XML for our sample process is shown below (note that we did not include the graphical information here for simplicity). The process element contains parameters like the name and id of the process, and consists of three main subsections: a header (where information like variables, globals and imports can be defined), the nodes and the connections.

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://drools.org/drools-5.0/process"
         xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
         xs:schemaLocation="http://drools.org/drools-5.0/process drools-processes-5.0.xsd"
         type="RuleFlow"
         name="ruleflow"
         id="com.sample.ruleflow"
         package-name="com.sample" >

  <header>
  </header>

  <nodes>
    <start id="1" name="Start" x="16" y="16" />
    <actionNode id="2" name="Hello" x="128" y="16" >
      <action type="expression"
                 dialect="mvel">System.out.println("Hello World");</action>
    </actionNode>
    <end id="3" name="End" x="240" y="16" />
  </nodes>

  <connections>
    <connection from="1" to="2" />
    <connection from="2" to="3" />
  </connections>

</process>
```

3.2. Executing your first process

To execute this process, right-click on **`RuleFlowTest.java`** and select "Run As..." and "Java Application". When the process in executed, the following output should appear in the Console window:

```
Hello World
```

If you look at the code of class **`RuleFlowTest`** (see below), you will see that executing a process requires a few steps:

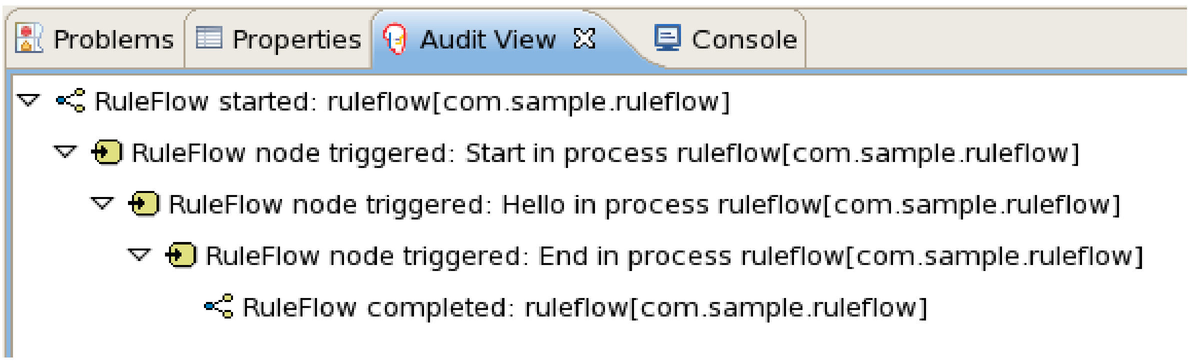
1. You should first create a Knowledge Base. A Knowledge Base contains all the knowledge (i.e., processes, rules, etc.) that are relevant in your application. This Knowledge Base is usually created once, and then reused. In this case, the Knowledge Base only consists of our sample process.

2. Next, you should create a session to interact with the engine. Note that we also add a logger to the session to log execution events and make it easier to visualize what is going on.

3. Finally, you can start a new instance of the process by invoking the **`startProcess(String processId)`** method on the session. This starts the execution of your process instance, resulting in the executions of the Start node, the Action node, and the End node, in this order, after which the process instance will be completed.

```
package com.sample;

import org.drools.KnowledgeBase;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.ResourceFactory;
import org.drools.logger.KnowledgeRuntimeLogger;
import org.drools.logger.KnowledgeRuntimeLoggerFactory;
import org.drools.runtime.StatefulKnowledgeSession;

/**
 * This is a sample file to launch a process.
 */
public class ProcessTest {

  public static final void main(String[] args) {
    try {
      // load up the knowledge base
      KnowledgeBase kbase = readKnowledgeBase();
      StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
      KnowledgeRuntimeLogger logger = KnowledgeRuntimeLoggerFactory.newFileLogger(ksession,
 "test");
      // start a new process instance
      ksession.startProcess("com.sample.ruleflow");
      logger.close();
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }

  private static KnowledgeBase readKnowledgeBase() throws Exception {
    KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
    kbuilder.add(ResourceFactory.newClassPathResource("ruleflow.rf"), ResourceType.DRF);
    return kbuilder.newKnowledgeBase();
  }

}
```

Congratulations, you have successfully executed your first process! Because we added a logger to the session, you can easily check what happened internally by looking at the audit log. Select the "Audit View" tab on the bottom right, next to the Console tab. Click on the "Open Log" button (the first one one the right of the view) and navigate to the newly created **`test.log`** file in your project folder. (If you are not sure where this project folder is located, right-click on the project folder and you will find the location in the "Resource" section). An image like the one below should be shown. It is a tree view of the events that occurred at runtime. Events that were executed as the direct result of another event are shown as the children of that event. This log shows that after starting the process, the Start node, the Action node and the End node were triggered, in that order, after which the process instance was completed.



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

You can now start experimenting and designing your own process by modifying our example. Note that you can validate your process by clicking on the "Check the ruleflow model" button, i.e., the green check box action in the upper toolbar that shows up if you are editing a process. Processes will also be validated upon save, and errors will be shown in the Error View.

Continue reading our documentation to learn about our more advanced features.

Rule Flow

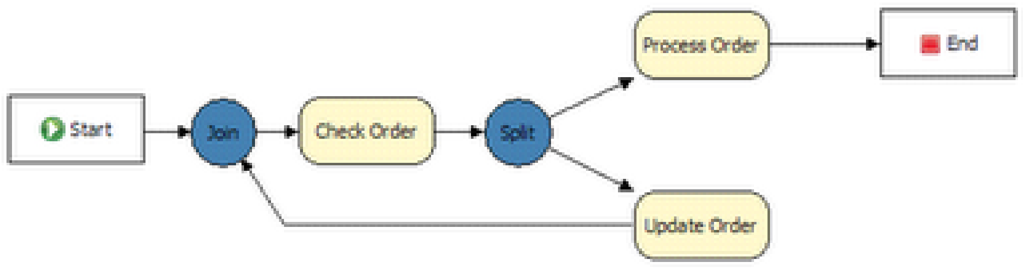


Figure 4.1. Ruleflow

A RuleFlow is a process 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.

4.1. Creating a RuleFlow Process

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

1. Using the graphical RuleFlow editor in the JBoss Rules plug-in for Eclipse
2. As an XML file, according to the XML process format as defined in the XML Schema definition for JBoss Rules processes.
3. By directly creating a process using the Process API.

4.1.1. Using the Graphical RuleFlow Editor

The graphical RuleFlow 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 RuleFlow editor is part of the JBoss Rules plug-in for Eclipse. Once you have set up a JBoss Rules project (check the IDE chapter if you do not know how to do this), 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 ruleflow in and select "New", then "Other...". Choose the section on "JBoss Rules" and then pick "RuleFlow file". This will create a new .rf file.

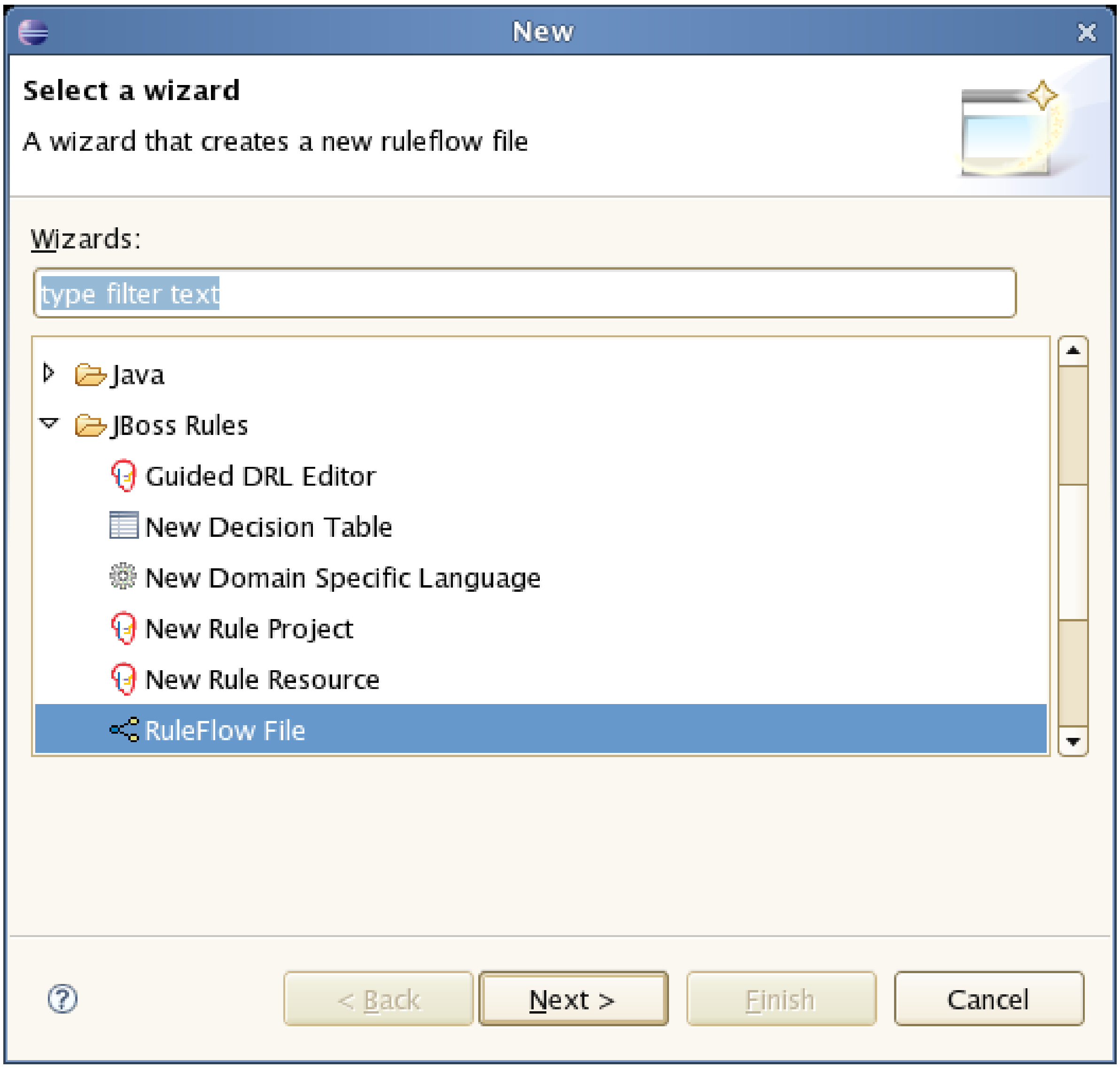


Figure 4.2. Creating a new RuleFlow file

Next you will see the graphical RuleFlow editor. Now would be a good time to switch to the JBoss Rules Perspective (if you haven't done so already). This will tweak the user interface so that it is optimal for rules. Then, 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.

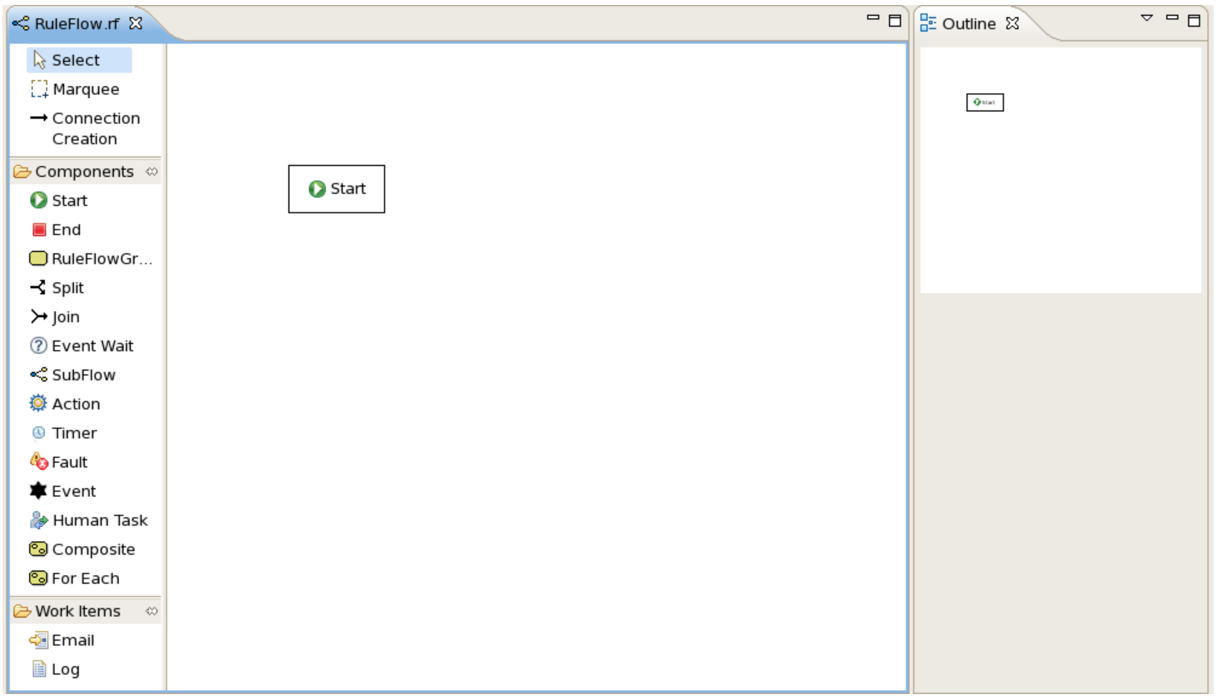


Figure 4.3. New RuleFlow process

The RuleFlow 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 "RuleFlowGroup" icon in the "Components" palette of the GUI: you can then draw a few rule flow groups. Clicking on an element in your rule flow 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 "Connection Creation" 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. You'll probably need to check the process for any missing information (by pressing the green "Check" icon in the IDE menu bar) before trying to use it in your application.

4.1.2. 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 node, an Action node that prints "Hello World" to the console, and an End node.

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://drools.org/drools-5.0/process"
         xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
         xs:schemaLocation="http://drools.org/drools-5.0/process drools-processes-5.0.xsd"
         type="RuleFlow" name="ruleflow" id="com.sample.ruleflow" package-name="com.sample" >

  <header>
  </header>

  <nodes>
    <start id="1" name="Start" x="16" y="16" />
    <actionNode id="2" name="Hello" x="128" y="16" >
      <action type="expression" dialect="mvel" >System.out.println("Hello World");</action>
    </actionNode>
    <end id="3" name="End" x="240" y="16" />
  </nodes>
  <connections>
    <connection from="1" to="2" />
    <connection from="2" to="3" />
  </connections>

</process>
```

The process XML file should 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> (where process-level information like variables, globals, imports and swimlanes 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.

4.1.3. 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.drools.workflow.core`** and **`org.drools.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 are added below.

4.1.3.1. Example 1

This is a simple example of a basic process with a ruleset node only:

```

RuleFlowProcessFactory factory =
    RuleFlowProcessFactory.createProcess("org.drools.HelloWorldRuleSet");
factory
    // Header
    .name("HelloWorldRuleSet")
    .version("1.0")
    .packageName("org.drools")
    // Nodes
    .startNode(1).name("Start").done()
    .ruleSetNode(2)
        .name("RuleSet")
        .ruleFlowGroup("someGroup").done()
    .endNode(3).name("End").done()
    // Connections
    .connection(1, 2)
    .connection(2, 3);
RuleFlowProcess process = factory.validate().getProcess();
```

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

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

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

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

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

4.1.3.2. Example 2

This example is using Split and Join nodes:

```
RuleFlowProcessFactory factory =
    RuleFlowProcessFactory.createProcess("org.drools.HelloWorldJoinSplit");
factory
    // Header
    .name("HelloWorldJoinSplit")
    .version("1.0")
    .packageName("org.drools")
    // Nodes
    .startNode(1).name("Start").done()
    .splitNode(2).name("Split").type(Split.TYPE_AND).done()
    .actionNode(3).name("Action 1")
        .action("mvel", "System.out.println(\"Inside Action 1\")").done()
    .actionNode(4).name("Action 2")
        .action("mvel", "System.out.println(\"Inside Action 2\")").done()
    .joinNode(5).type(Join.TYPE_AND).done()
    .endNode(6).name("End").done()
    // Connections
    .connection(1, 2)
    .connection(2, 3)
    .connection(2, 4)
    .connection(3, 5)
    .connection(4, 5)
    .connection(5, 6);
RuleFlowProcess process = factory.validate().getProcess();
```

This shows a simple example using Split and Join nodes. As you can see, a Split node can have multiple outgoing connections, and a Join node multiple incoming connections. To understand the behavior of the different types of Split and Join nodes, take a look at the documentation for each of these nodes.

4.1.3.3. Example 3

Now we show a more complex example with a ForEach node, where we have nested nodes:

```
RuleFlowProcessFactory factory =
    RuleFlowProcessFactory.createProcess("org.drools.HelloWorldForeach");
factory
    // Header
    .name("HelloWorldForeach")
    .version("1.0")
    .packageName("org.drools")
    // Nodes
    .startNode(1).name("Start").done()
    .forEachNode(2)
```

```
        // Properties
        .linkIncomingConnections(3)
        .linkOutgoingConnections(4)
        .collectionExpression("persons")
        .variable("child", new ObjectDataType("org.drools.Person"))
        // Nodes
        .actionNode(3)
            .action("mvel", "System.out.println(\"inside action1\")").done()
        .actionNode(4)
            .action("mvel", "System.out.println(\"inside action2\")").done()
        // Connections
        .connection(3, 4)
        .done()
    .endNode(5).name("End").done()
    // Connections
    .connection(1, 2)
    .connection(2, 5);
RuleFlowProcess process = factory.validate().getProcess();
```

Here you can see how we can include a ForEach node with nested action nodes. Note the **`linkIncomingConnections()`** and **`linkOutgoingConnections()`** methods that are called to link the ForEach node with the internal action node. These methods are used to specify the first and last nodes inside the ForEach composite node.

4.2. 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. Note that this is almost identical to adding rules to the Knowledge Base, except for the type of knowledge added:

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

   After adding all your knowledge to the builder (you can add more than one process, and even rules), 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 executed if you explicitly state that they should be executed. 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.MyProcess");
   ```

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. If your process also requires the execution of rules during the execution of the process, you also need to call the **`ksession.fireAllRules()`** method to make sure the rules are executed as well. That's it!

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.

You can also start a process from within a rule consequence, or from inside a process action, using the predefined kcontext parameter:

```
kcontext.getKnowledgeRuntime().startProcess("com.sample.MyProcess");
```

4.3. Detailed Explanation of the Different Node Types

A ruleflow 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.
- *Exception Handlers*: Specify the behaviour when a fault occurs in the process. See section “*Exceptions*” 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 RuleFlow process supports different types of nodes:

- *Name*: The display name of the node.
- *StartNodeId*: The id of the node (within this node container) that should be triggered for each of the elements in a collection.
- *EndNodeId*: The id of the node (within this node container) that represents the end of the flow contained in this node. When this node is completed, the execution of the ForEach node will also be completed for the current collection element. The outgoing connection is triggered if the collection is exhausted. All other executing nodes within this composite node will be cancelled.
- *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.

15. **WorkItem (or Service Task)**: Represents an (abstract) unit of work that should be executed in this process. All work that is executed outside the process engine should be represented (in a declarative way) using a WorkItem node. Different types of work items are predefined, e.g., sending an email, logging a message, etc. Users can define domain-specific 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 Domain_Specific_Processes for a detailed explanation and illustrative examples of how to define and use work items in your processes. When a WorkItem node is reached in the process, the associated work item is executed. A WorkItem node 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.
    - *Wait for completion*: If the property "Wait for completion" is true, the WorkItem node will only continue if the created work item has terminated (completed or aborted) its execution; otherwise it will continue immediately after starting the work item.
    - *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. Note that you can use result mappings only when "Wait for completion" is set to true.
    - *On-entry and on-exit actions*: Actions that are executed upon entry or exit of this node, respectively.
    - *Timers*: Timers that are linked to this node. See the section “*Timers*” for details.
    - *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}`**.
    - *MetaData*: Metadata related to this node.

4.4. Data

While the flow graph 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 Composite node. 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);
  ```

- WorkItem and SubFlow nodes 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 interpolating 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, exception handlers can read error data from a specific 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 that are considered immutable with regard to rule evaluation, and data in the Knowledge Session. The Knowledge Session can be accessed in actions using the Knowledge Context:

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

4.5. Constraints

Constraints can be used in various locations in your processes, for example in a Split node using OR or XOR decisions, or as a constraint for a State node. JBoss Rules Flow 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 JBoss Rules rule conditions. They use the JBoss Rules 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 ...
```

4.6. Actions

Actions can be used in different ways:

- Within an Action node,
- As entry or exit actions, with a number of nodes,
- Actions specifying the behavior of exception handlers.

Actions have access to globals and the variables that are defined for the process and the predefined variable **`context`**. 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 = context.getNodeInstance();
String name = node.getNodeName();
```

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

```
WorkflowProcessInstance proc = context.getProcessInstance();
proc.signalEvent( type, eventObject );
```

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

JBoss Rules 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 );
```

4.7. Events

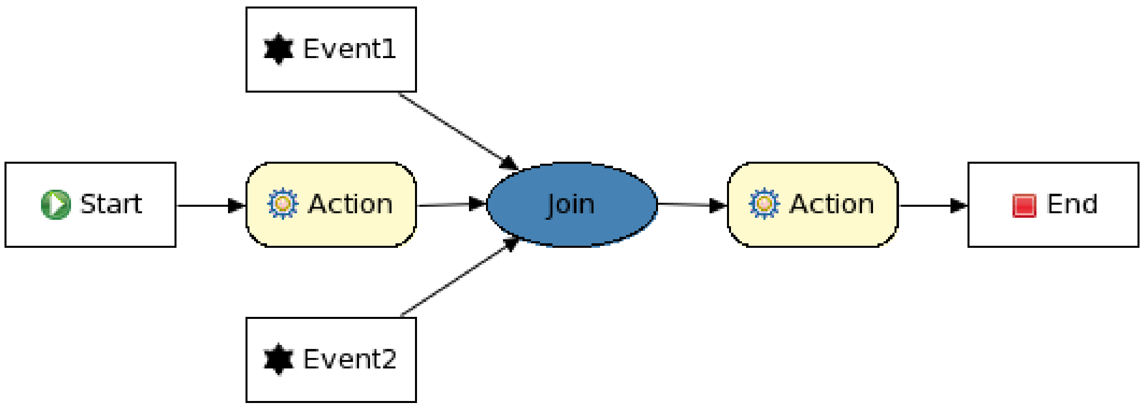


Figure 4.5. A sample process using events

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 Event nodes. 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 signalled 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:

  ```
  context.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:

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

Events could also be used to start a process. Whenever a Start node 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.

4.8. Exceptions

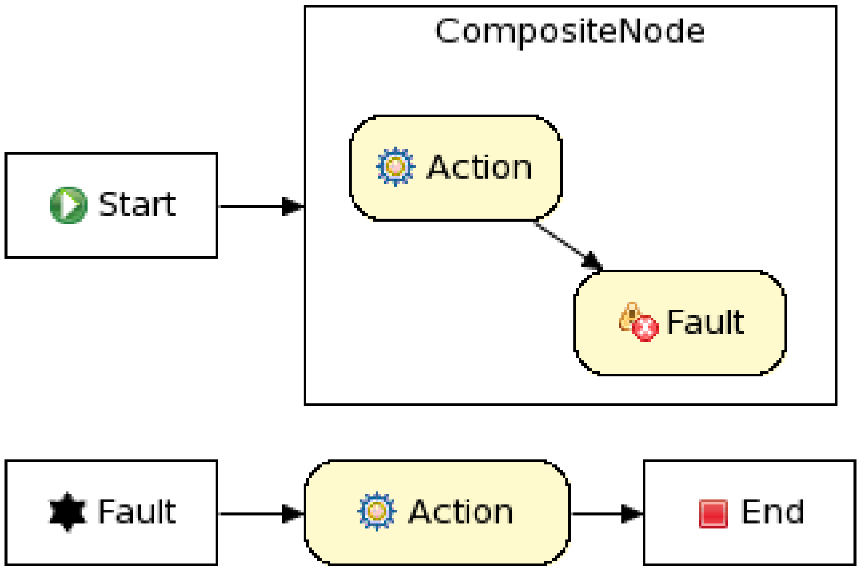


Figure 4.6. A sample process using exception handlers

Whenever an exceptional condition occurs during the execution of a process, a fault could be raised to signal the occurrence of this exception. The process will then search for an appropriate exception handler that is capable of handling such a fault.

Similar to events, faults also have a type and possibly data associated with the fault. Users are free to define their own types of faults, together with their data.

Faults are effected by a Fault node, generating a fault of the given type, indicated by the fault name. If the Fault node specifies a fault variable, the value of the given variable will be associated with the fault.

Whenever a fault is created, the process will search for an appropriate exception handler that is capable of handling the given type of fault. Processes and Composite nodes both can define exception handlers for handling faults. Nesting of exception handlers is allowed; a node will always search for an appropriate exception handler in its parent container. If none is found, it will look in that one's parent container, and so on, until the process instance itself is reached. If no exception handler can be found, the process instance will be aborted, resulting in the cancellation of all nodes inside the process.

Exception handlers can also specify a fault variable. The data associated with the fault (if any) will be copied to this variable whenever an exception handler is selected to handle a fault. This allows subsequent Action nodes in the process to access the fault data and take appropriate action based on this data.

Exception handlers need to define an action that specifies how to respond to the given fault. In most cases, the behavior that is needed to react to the given fault cannot be expressed in one action. It is therefore recommended to have the exception handler signal an event of a specific type (in this case "Fault") using

```
context.getProcessInstance().signalEvent("FaultType", context.getVariable("FaultVariable");
```

4.9. 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 node is set up with a delay and a period. The delay specifies the amount of time (in milliseconds) to wait after node activation before triggering the timer the first time. The period defines the time between subsequent trigger activations. A period of 0 results in a one-shot timer.

The 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 node 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 may be associated with event-based nodes like WorkItem, SubFlow, etc. A timer associated with a node is activated whenever the node becomes active. The associated action is executed whenever the timer triggers. You may use this, for instance, to send out notifications, at regular intervals, when the execution of a task takes too long, or to signal an event or a fault in case a time supervision expires. When the node owning the timer completes, the timer is automatically cancelled.

4.10. 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, JBoss Rules Flow 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.

4.10.1. 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, JBoss Rules Flow 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.ruleflow"
KnowledgeBuilder kbuilder = ...
StatefulKnowledgeSession ksession = ...
ProcessInstance processInstance = ksession.startProcess("com.sample.ruleflow");

// add a new version of the process "com.sample.ruleflow2"
kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(..., ResourceType.DRF);
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.ruleflow2", 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.

4.11. Assigning Rules to a Ruleflow Group

JBoss Rules already provides some functionality to define the order in which rules should be executed, like salience, activation groups, etc. When dealing with potentially many large rule-sets, managing the order in which rules are evaluated might become complex. Ruleflow allows you to specify the order in which rule sets should be evaluated by using a flow chart. This allows you to define which rule sets should be evaluated in sequence or in parallel, to specify conditions under which rule sets should be evaluated. This chapter contains a few ruleflow examples.

A ruleflow is a graphical description of a sequence of steps that the rule engine needs to take, where the order is important. The ruleflow can also deal with conditional branching, parallelism, and synchonization.

To use a ruleflow to describe the order in which rules should be evaluated, you should first group rules into ruleflow groups using the **`ruleflow-group`** rule attribute ("options" in the GUI). Then you should create a ruleflow graph (which is a flow chart) that graphically describe the order in which the rules should be considered, by specifying the order in which the ruleflow-groups should be evaluated.

```

rule 'YourRule'
    ruleflow-group 'group1'
when
    ...
then
    ...
end
```

This rule belongs to the ruleflow-group called "group1".

Rules that are executing as part of a ruleflow-group that is triggered by a process, can also access the process context in the rule consequence. Through the process context, the process instance or node instance that triggered the ruleflow-group can be accessed, or variables could be set or retrieved, e.g.

```
drools.getContext(ProcessContext.class).getProcessInstance()
```

4.12. A Simple Ruleflow

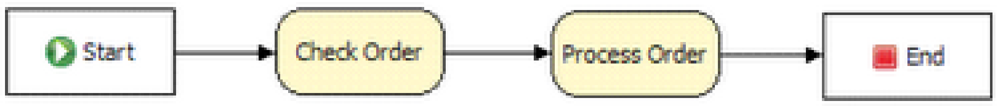
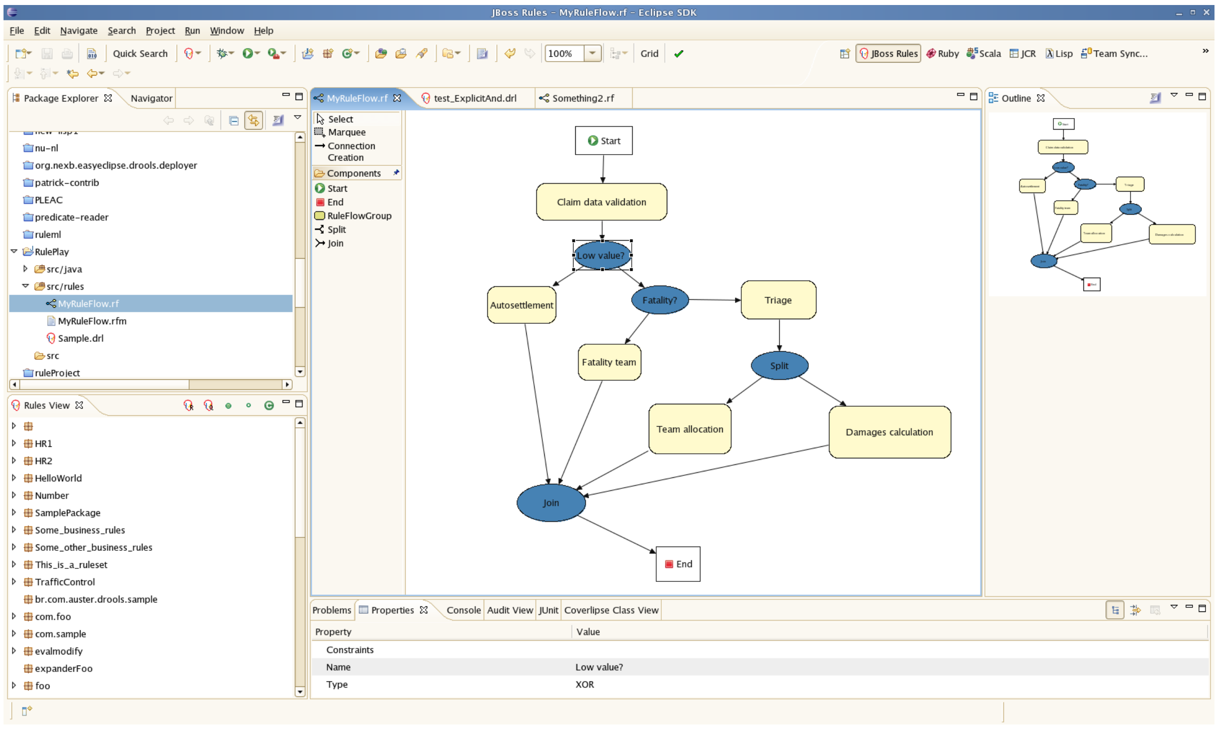


Figure 4.7. Ruleflow

The above rule flow specifies that the rules in the group "Check Order" must be executed before the rules in the group "Process Order". This means that first only rules which are marked as having a ruleflow-group of "Check Order" will be considered, and then, only if there aren't any more of those, the rules of "Process Order". That's about it. You could achieve similar results with either using salience, but this is harder to maintain and makes the time-relationship implicit in the rules, or Agenda groups. However, using a ruleflow makes the order of processing explicit, in a layer on top of the rule structure, so that managing more complex situations becomes much easier.

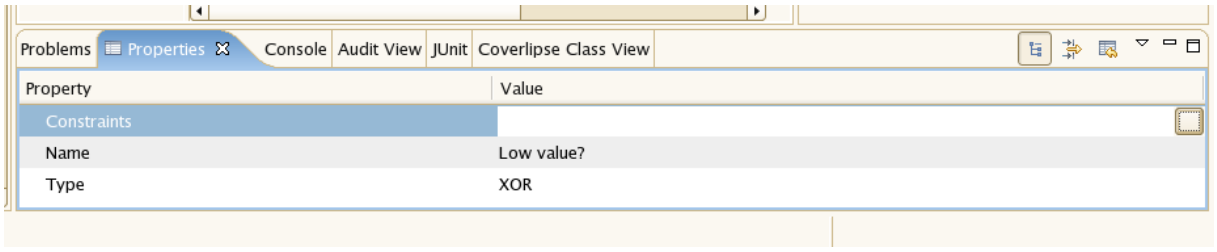
In practice, if you are using ruleflow, you will most likely be doing more than setting a simple sequence of groups to progress though. You'll use Split and Join nodes for modeling branches of processing, and define the flows of control by connections, from the Start to ruleflow groups, to Splits and then on to more groups, Joins, and so on. All this is done in a grphic editor.



Property	Value
Constraints	
Name	Low value?
Type	XOR

Figure 4.8. Complex ruleflow

The above flow is a more complex example, representing the rule flow for processing an insurance claim. Initially the claim data validation rules are processed, checking for data integrity, consistency and completeness. Next, in a Split node, there is a decision based on a condition based on the value ofthe claim. Processing will either move on to an "auto-settlement" group, or to another Split node, which checks whether there was a fatality in the incident. If so, it determines whether the "regular" of fatality specific rules should take effect, with more processing to follow. Based on a few conditions, many different control flows are possible. Note that all the rules can be in one package, with the control flow definition being separated from the actual rules.



Property	Value
Constraints	
Name	Low value?
Type	XOR

Figure 4.9. Split types

To edit Split nodes you click on the node, which will show you a properties panel as shown above. You then have to choose the type: AND, OR, and XOR. If you choose OR, then any of the "outputs" of the split can happen, so that processing can proceed, in parallel, along two or more paths. If you chose XOR, then only one path is chosen.

If you choose OR or XOR, the "Constraints" row will have a square button on the right hand side. Clickin on this button opens the Constraint editor, where you set the conditions deciding which outgoing path to follow.

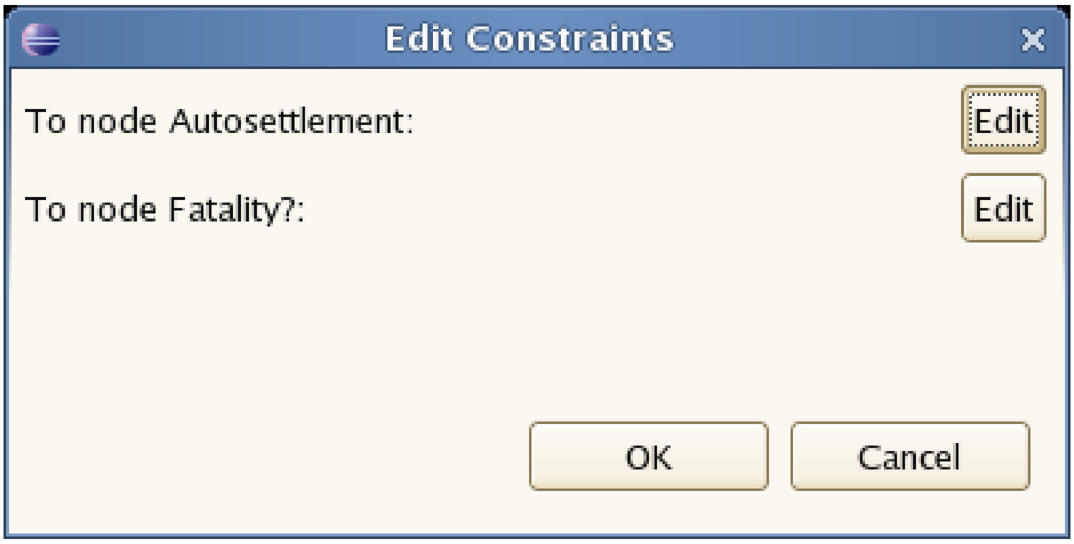
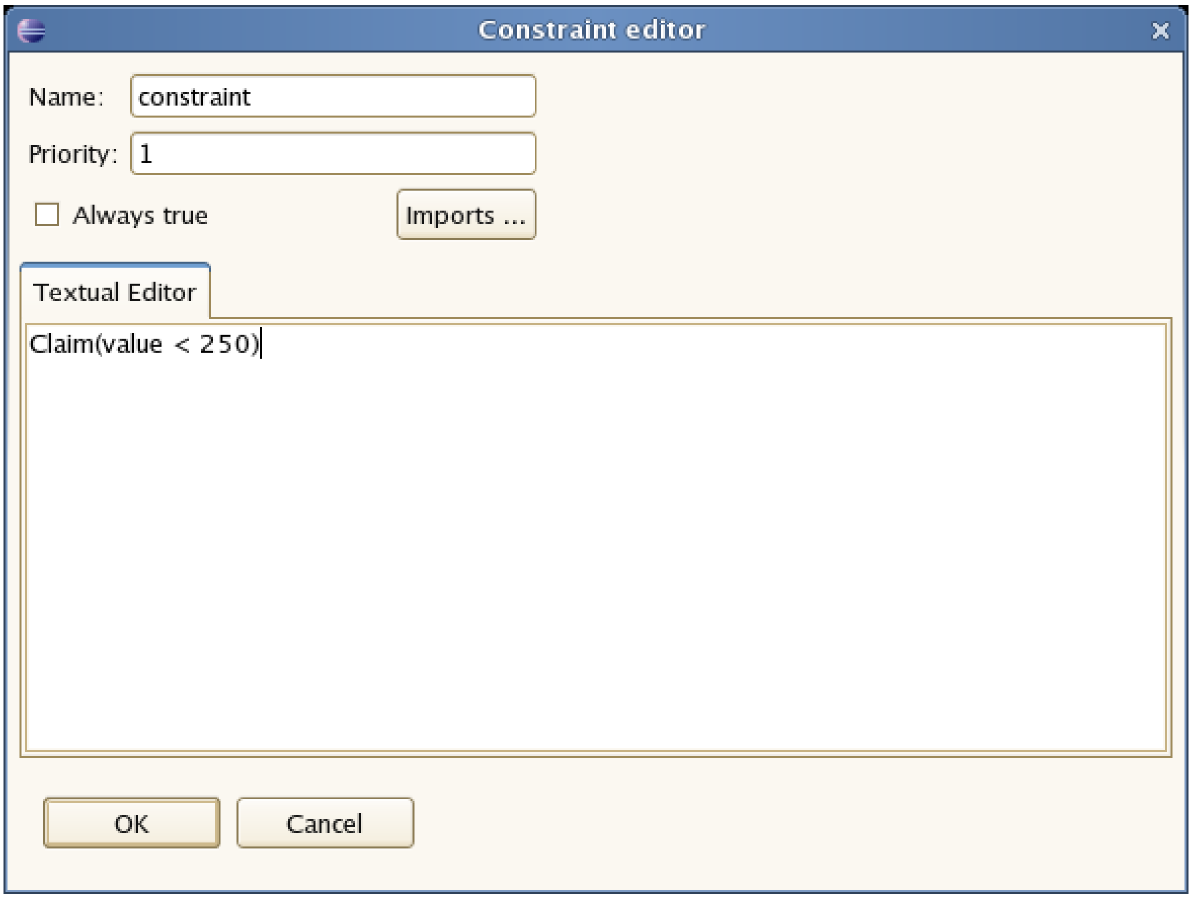


Figure 4.10. Edit constraints

Choose the output path you want to set the constraints for (e.g. Autosettlement), and then you should see the following constraint editor:



Name: constraint
Priority: 1
☐ Always true

```
Claim(value < 250)
```

Figure 4.11. Constraint editor

This is a text editor where the constraints - which are like the condition part of a rule - are entered. These constraints operate on facts in the working memory. In the above example, there is a check for claims with a value of less than 250. Should this condition be true, then the associated path will be followed.

4.13. Using JBoss Rules 4.x RuleFlow Processes

The XML format that was used in JBoss Rules4 to store RuleFlow processes was generated automatically, using XStream. As a result, it was hard to read by human readers and difficult to maintain and extend. The new JBoss Rules Flow XML format has been created to simplify this. This however means that, by default, old RuleFlow processes cannot simply be executed on the JBoss Rules5 engine.

We do however provide a Rule Flow Migrator that allows you to transform your old .rf file to the new format. It uses an XSLT transformation to generate the new XML based on the old content. You can use this class to manually transform your old processes to the new format once when upgrading from JBoss Rules4.x to JBoss Rules5.x. You can however also let the KnowledgeBuilder automatically upgrade your processes to the new format when they are loaded into the Knowledge Base. While this requires a conversion every time the process is loaded into the Knowledge Base, it does support a more seamless upgrade. To enact this automatic upgrade you need to set the "drools.ruleflow.port" system property to "true", for example by adding **`-Ddrools.ruleflow.port=true`** when starting your application, or by calling **`System.setProperty("drools.ruleflow.port", "true")`**.

The JBoss Rules Eclipse plugin also automatically detects if an old RuleFlow file is opened. At that point, it will automatically perform the conversion and show the result in the graphical editor. You then need to save this result, either in a new file or overwriting the old one, to retain the old process in the new format. Note that the plugin does not support editing and saving processes in the old JBoss Rules4.x format.

Drools Flow API

The Drools Flow API should 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. Knowledge Base

Our knowledge-based API allows you to first create one Knowledge Base that contains all the necessary knowledge and can be reused across sessions. This knowledge base includes all your process definitions (and other knowledge types like for example rules). The following code snippet shows how to create a Knowledge Base consisting of only one process definition, using a Knowledge Builder to add the resource (from the classpath in this case).

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

Note that 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 Drools Flow to start using Drools Fusion almost instantaneously, and even to integrate these different types of Knowledge.

5.2. 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. Consult the JavaDocs to get a detailed explanation for each of the methods.

```
ProcessInstance startProcess(String processId);
ProcessInstance startProcess(String processId, Map<String, Object> parameters);
void signalEvent(String type, Object event);
void signalEvent(String type, Object event, long processInstanceId);
Collection<ProcessInstance> getProcessInstances();
ProcessInstance getProcessInstance(long id);
void abortProcessInstance(long id);
WorkItemManager getWorkItemManager();
```

5.3. Events

Both the stateful and stateless knowledge sessions provide methods for registering and removing listeners. **`ProcessEventListener`** objects 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 a **`ProcessEventListener`** 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.

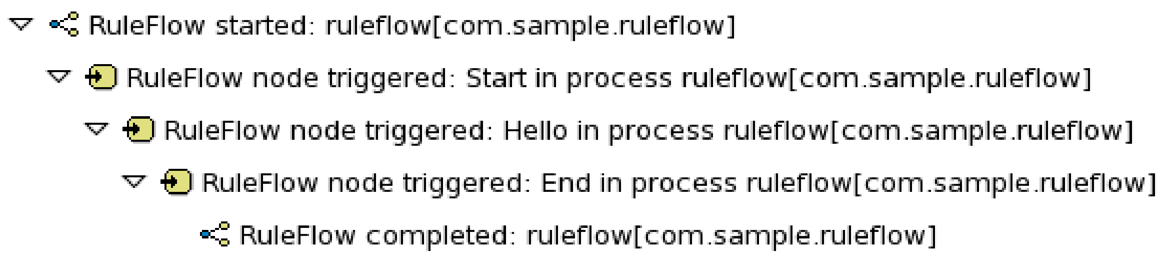
2. File logger: This logger writes out all the events to a file using an XML representation. This log file might then be used in the IDE to generate a tree-based visualization of the events that occurred during execution.

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.



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

Persistence

Drools Flow allows the persistent storage of certain information, i.e., the process runtime state, the process definitions and the history information.

6.1. 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. Drools Flow 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.

6.1.1. Binary Persistence

Drools Flow 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.

6.1.2. 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.

6.1.3. 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 mighto 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 (cf. chapter IDE Features), or by manually adding these dependencies to your project. First of all, you need the jar file **`drools-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 dependencies is needed:

1. drools-persistence-jpa (org.drools)
2. persistence-api-1.0.jar (javax.persistence)
3. hibernate-entitymanager-3.4.0.GA.jar (org.hibernate)
4. hibernate-annotations-3.4.0.GA.jar (org.hibernate)
5. hibernate-commons-annotations-3.1.0.GA.jar (org.hibernate)
6. hibernate-core-3.3.0.SP1.jar (org.hibernate)
7. dom4j-1.6.1.jar (dom4j)
8. jta-1.0.1B.jar (javax.transaction)
9. btm-1.3.2.jar (org.codehaus.btm)
10. javassist-3.4.GA.jar (javassist)
11. slf4j-api-1.5.2.jar (org.slf4j)
12. slf4j-jdk14-1.5.2.jar (org.slf4j)
13. h2-1.0.77.jar (com.h2database)
14. commons-collections-3.2.jar (commons-collections)

Next, you need to configure the Drools 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.

```
// create the entity manager factory and register it in the environment
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory( "org.drools.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.

By default, **`drools-persistence-jpa.jar`** contains a configuration file that configures JPA to use Hibernate and the H2 database, called **`persistence.xml`** in the META-INF directory, as shown below. You will need to override these defaults if you want to change them, by adding your own **`persistence.xml`** in your classpath, preceding the default one in **`drools-persistence-jpa.jar`**. Refer to the JPA and Hibernate documentation for more information on how to do this.

```
<?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.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.drools.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.processinstance.ProcessInstanceEventInfo</class>
    <class>org.drools.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/processInstanceDS");
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();
```

6.1.4. 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.drools.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" );
ksession.fireAllRules();

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

6.2. 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 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.

6.3. 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.

6.3.1. Storing Process Events in a Database

The drools-bam module contains an event listener that stores process-related information in a database using Hibernate. 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.

PROCESSINSTANCEID	PROCESSID	START_DATE	END_DATE

ID	TYPE	NODEINSTANCEID	NODEID	PROCESSINSTANCEID	PROCESSID	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 = ...;
WorkingMemoryDbLogger logger = new WorkingMemoryDbLogger(ksession);

// invoke methods one your session here

logger.dispose();
```

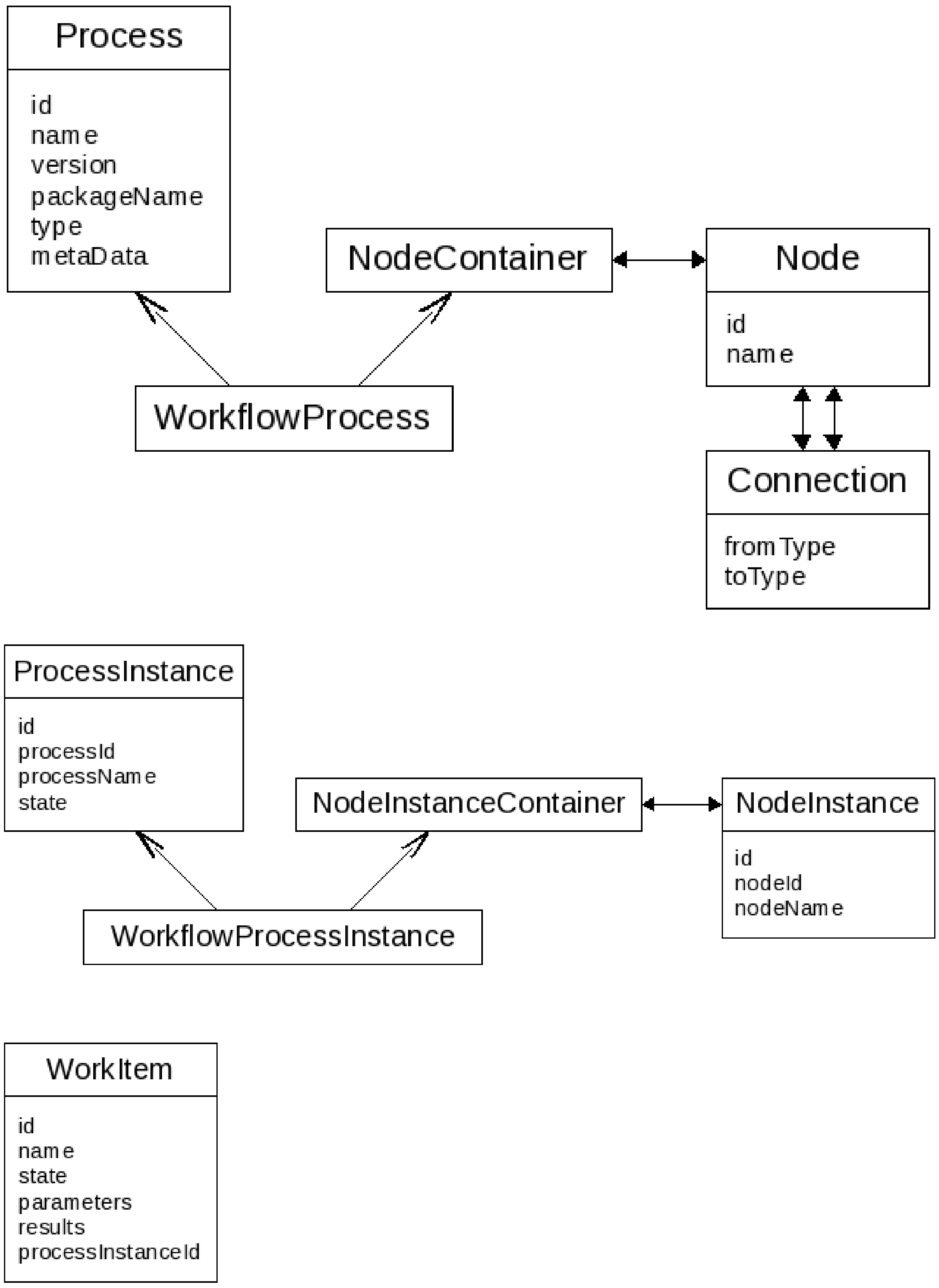
Note that this logger is like any other audit logger, which means that you can add one or more filters by calling the method **`addFilter`**`d` to ensure that only relevant information is stored in the database. Only information accepted by all your filters will appear in the database. You should dispose the logger when it is no longer needed.

To specify the database where the information should be stored, modify the file **`hibernate.cfg.xml`** file. By default, it uses a memory-resident database (H2). Consult the Hibernate documentation if you do not know how to do this.

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. Class **`ProcessInstanceDbLog`** (in package **`org.drools.process.audit`**) shows some examples on how to retrieve all process instances, one specific process instance (by id), all process instances for one specific process, all node instances of a specific process instance, etc. You can of course easily create your own Hibernate queries, or access the information in the database directly.

By default, the audit logger uses the H2 memory-resident database that is recreated on startup. You can change this default by including your own configuration file **`hibernate.cfg.xml`**. This allows you, for example, to change the underlying database, etc. Refer to the Hibernate documentation for more information on how to do this.

JBoss Rules Flow Process Model



Rules and Processes

JBoss Rules Flow is a workflow and process engine that allows advanced integration of processes and rules. This chapter discusses the integration of rules and processes, ranging from simple to advanced scenarios.

8.1. Why Use Rules in Processes?

Workflow languages that depend purely on process constructs (like nodes and connections) to describe the business logic of applications tend to be quite complex. While these workflow constructs are very well suited to describe the overall control flow of an application, it can be very difficult to describe complex logic and exceptional situations. Therefore, executable processes tend to become very complex. We believe that, by extending a process engine with support for declarative rules in combination with these regular process constructs, this complexity can be kept under control.

1. Simplicity: Complex decisions are usually easier to specify using a set of rules. Rules can pinpoint complex business logic more easily, using their advanced constraint language. Multiple rules can be combined, each describing a part of the business logic.

2. Agility: Rules and processes can have a separate life cycle. This means that we can change the rules describing some crucial decision points without having to change the process itself. Rules can be added, removed or modified to fine-tune the behavior of the process to the constantly evolving requirements and environment.

3. Different scope: Rules can be reused across processes or outside processes. Therefore, your business logic is not locked inside your processes.

4. Declarativeness: Focus on describing "what" instead of "how".

5. Granularity: It is easy to write simple rules that handle specific circumstances. Processes are more suited to describe the overall control flow but tend to become very complex if they also need to describe a lot of exceptional situations.

6. Data-centric: Rules can easily handle large data sets.

7. Performance: Rule evaluation is optimized.

8. Advanced condition and action language: Rule languages support advanced features like custom functions, collections, conditional elements, including quantifiers, etc.

9. High-level: By using DSLs, business editors, decision tables, and decision trees, your business logic could be described in a way that can be understood (and possibly even modified) by business users.

8.2. Why Integrate Rules and Processes in a Single Engine?

JBoss Rules Flow combines a process and a rules engine in one software product. This offers several advantages, compared to trying to loosely couple an existing process and rules product.

1. Simplicity: Easier for end user to combine both rules and processes.

2. Encapsulation: Sometimes close integration between processes and rules is beneficial.

3. Performance: No unnecessary passing, transformation or synchronization of data

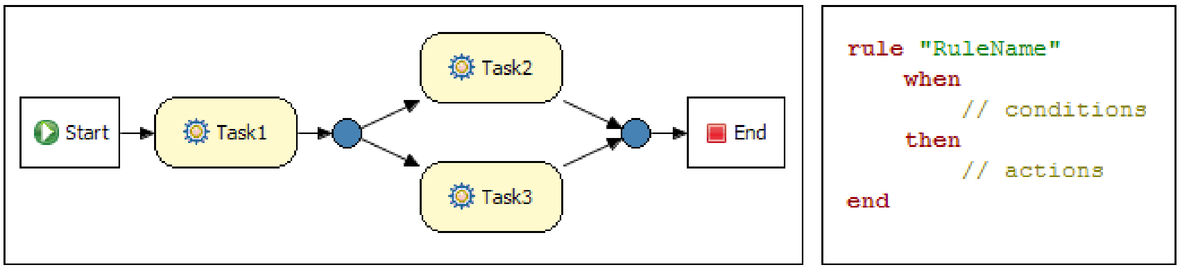
4. Learning curve: Easier to learn one product.

5. Manageability: Easier to manage one product, rules and processes can be similar artefacts in a larger knowledge repository.

6. Integration of features: We provide an integrated IDE, audit log, web-based management platform, repository, debugging, etc.

8.3. Approach

Workflow languages describe the order in which activities should be performed using a flow chart. A process engine is responsible for selecting which activities should be executed based on the current state of the executing processes. On the other hand, rules are composed of a set of conditions that describe when a rule is applicable and an action that is executed when the conditions are met. The rules engine is then responsible for evaluating and executing the rules. It decides which rules need to be executed based on the current state of the application.



```
rule "RuleName"
    when
        // conditions
    then
        // actions
end
```

Workflow processes are very good at describing the overall control flow of (possibly long-running) applications. However, processes that are used to define complex business decisions, to handle a lot of exceptional situations, and need to respond to various external events tend to become very complex indeed. Rules are very good at describing complex decisions and reasoning about large amounts of data or events. It is, however, not trivial to define long-running processes using rules.

In the past, users were forced to choose between defining their business logic using either a process or a rules engine. Problems that required complex reasoning about large amounts of data used a rules engine, while users that wanted to focus on describing the control flow of their processes were forced to use a process engine. However, businesses nowadays might want to combine both processes and rules in order to be able to define all their business logic in the format that best suits their needs.

Basically, both a rules and a process engine will derive the next steps that need to be executed by looking at its Knowledge Base (a set of rules or processes, respectively) and the current known state of the application (the data in the Working Memory or the state of the executing process instances, respectively). If we want to integrate rules and processes, we need an engine that can decide the next steps taking into account the logic that is defined inside both the processes and the rules.

8.3.1. Teaching a Rules Engine About Processes

It is very difficult (and probably very inefficient as well) to extend a process engine to also take rules into account. The process engine would need to check for rules that might need to be executed at every step and would have to keep the data that is used by the rules engine up to date. However, it is not that difficult to "teach" a rules engine about processes. If the current state of the processes is also inserted as part of the Working Memory data the rules engine reasons about, and we instruct the rules engine how to derive the next steps of an executing process, the rules engine will then be able to derive the next steps taking rules and processes into account jointly.

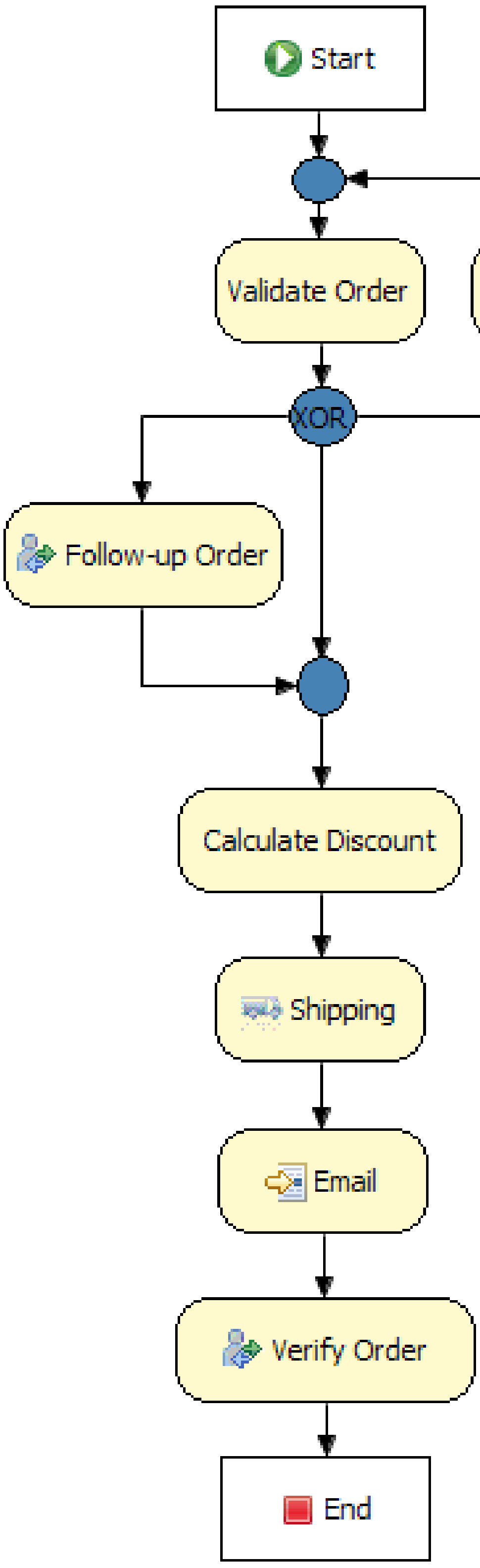
8.3.2. Inversion of Control

From the process perspective, this means that there is an inversion of control. A normal process engine exercises full control, deriving the next steps based on the current state of the process instance. If needed, it can contact external services to retrieve additional information, but it solely decides which steps to take, and is alone responsible for executing these steps.

However, only our extended rules engine (that can reason jointly about rules and processes) is capable of deriving the next steps taking both rules and processes into account. If a part of the process needs to be executed, the rules engine will request the process engine to execute this step. Once this step has been performed, the process engine returns control to the rules engine to again derive the next steps. This means that the control on what to do next has been inverted: the process engine itself no longer decides the next step to take but our enhanced rules engine will be in control, notifying the process engine what to execute next, and when.

8.4. Example

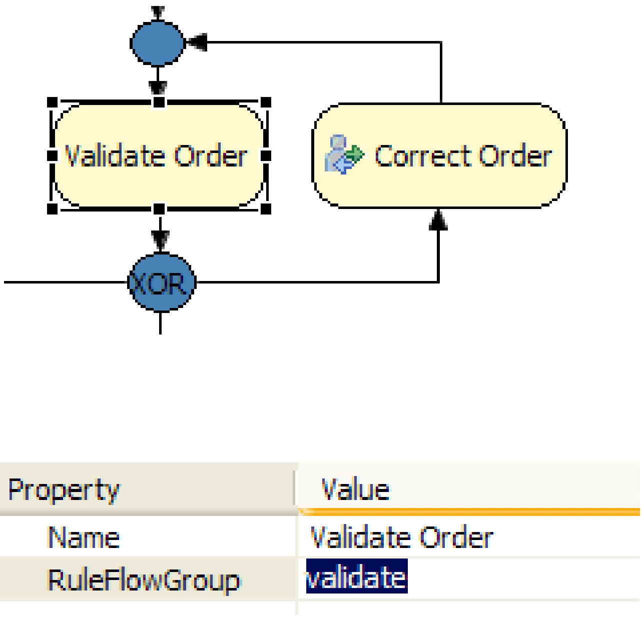
The drools-examples project contains a sample process (**`org.drools.examples.process.order`**) that illustrates some of the advantages of being able to combine processes and rules. This process describes an order application where incoming orders are validated, discounts are calculated and shipping of the goods is requested.



8.4.1. Evaluating a Set of Rules in Your Process

JBoss Rules Flow can easily include a set of rules as part of the process. The rules that need to be evaluated should be grouped in a ruleflow group, using the **`ruleflow-group`** rule attribute. Activating a RuleSet node for the group triggers the evaluation of these rules in your process. This example uses two RuleSet nodes in the process: one for the validation of the order and one for calculating the discount. For example, one of the rules for validiting an order is shown below. Note the **`ruleflow-group`** attribute, which ensures that this rule is evaluated as part of the RuleSet node with the same ruleflow group shown in the figure.

```
rule "Invalid item id"
    ruleflow-group "validate"
    lock-on-active true
when
    o: Order()
    i: Order.OrderItem() from o.getOrderItems()
    not (Item() from itemCatalog.getItem(i.getItemId()))
then
    System.err.println("Invalid item id found!");
    o.addError("Invalid item id " + i.getItemId());
end
```

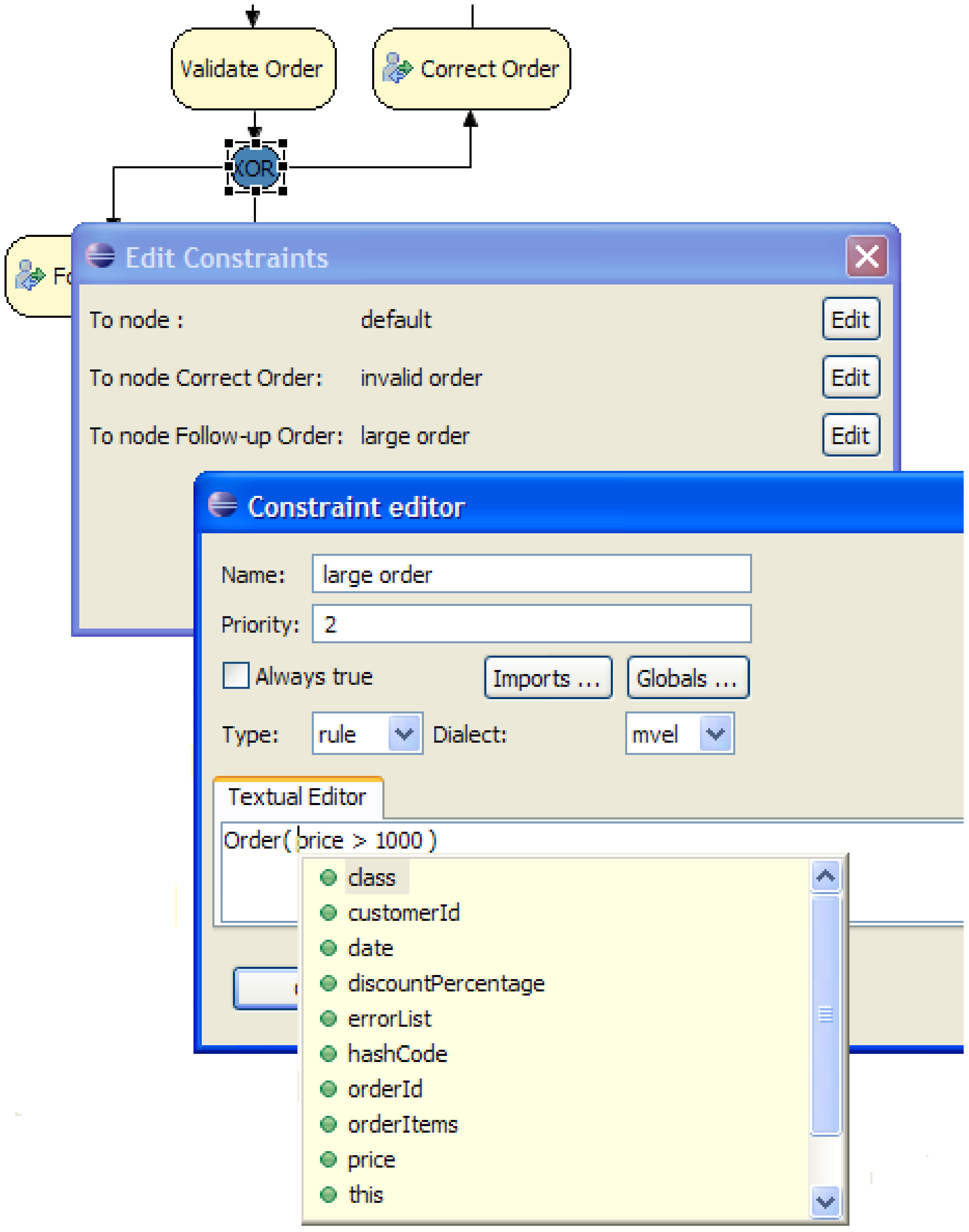


Property	Value
Name	Validate Order
RuleFlowGroup	validate

Figure 8.1. RuleSet node and one of its rules

8.4.2. Using Rules for Evaluating Constraints

Rules can be used for expressing and evaluating complex constraints in your process. For example, when to decide about the choice of the execution path at a Split node, rules could be used to define these conditions. Similarly, a Wait state could use a rule to define the wait duration. This example uses rules for deciding the next action after validating the order. If the order contains errors, a sales representative should try to correct the order. Orders with a value > 1000$ are more important, so that a senior sales representative should attend to the order. All other orders should just proceed normally. A decision node is used to select one of these alternatives, and rules are used to describe the constraints for each of them.



To node : default Edit
To node Correct Order: invalid order Edit
To node Follow-up Order: large order Edit

Name: large order
Priority: 2
Always true

Type: rule
Dialect: mvel

```
Order( price > 1000 )
```

8.4.3. Assignment Rules

Human tasks can be used in a process to describe work that needs to be executed by a human actor. The selection of the actor could be based on the current state of the process and the history. Assignment rules describe how to determine the actor, based on this information. These assignment rules will then be applied automatically whenever a new human task needs to be executed.

Note that the rules shown below are written in a Domain Specific Language (DSL), tailored to the specific requirements for formulating conditions in the order processing environment.

```
/********** Generic assignment rules **********/

rule "Assign 'Correct Order' to any sales representative"
    salience 30
    when
        There is a human task
        - with task name "Correct Order"
        - without actor id
    then
        Set actor id "Sales Representative"
end

/********** Assignment rules for the RuleSetExample process **********/

rule "Assign 'Follow-up Order' to a senior sales representative"
    salience 40
    when
        Process "org.drools.examples.process.ruleset.RuleSetExample" contains a human task
        - with task name "Follow-up Order"
        - without actor id
    then
        Set actor id "Senior Sales Representative"
end
```

8.4.4. Describing Exceptional Situations Using Rules

Rules can be used for describing exceptional situations and how to respond to these situations. Adding all this information in the control flow of the regular process makes the basic process much more complex. Rules can be used to handle each of these situations separately, leaving the core process in its simple form. It also makes it much easier to adapt existing processes to take previously unanticipated events into account.

8.4.5. Modularizing Concerns Using Rules

The process defines the overall control flow. Rules could be used to add additional concerns to this process without making the overall control flow more complex. For example, rules could be defined to log certain information during the execution of the process. The original process is not altered, whereas all logging functionality is cleanly modularized as a set of rules. This greatly improves reusability, allowing users to easily apply the same strategy to different processes, readability (by not altering the control flow of the original process) and maintainability, due to the separation of the logging strategy rules from those of the process itself.

8.4.6. Rules for Altering Process Behavior Dynamically

Rules let you dynamically fine-tune the behavior of your processes. Imagine that a problem is encountered, at runtime, with one of the processes. Now, new rules could be added, at runtime, to log additional information or for handling specific process states. Once the problem is solved or the circumstances have changed, these rules can easily be removed again. Based on the current status, different strategies could be selected dynamically. For example, based on the current load of all the services, rules could be used to optimize the process to the current load. This process contains a simple example that allows you to dynamically add or remove logging for the "Check Order" task. When the "Debugging output" checkbox in the main application window is checked, the rule shown below is loaded dynamically, to write log output to the console whenever the "Check Order" task is requested. Unchecking the box will dynamically remove the rule again.

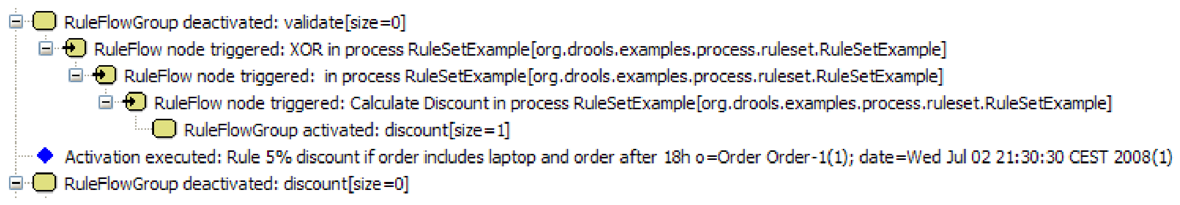
```
rule "Log the execution of 'Correct Order'"
    salience 25
when
    workItemNodeInstance: WorkItemNodeInstance( workItemId <= 0, node.name == "Correct
 Order" )
    workItem: WorkItemImpl( state == WorkItemImpl.PENDING ) from
 workItemNodeInstance.getWorkItem()
then
    ProcessInstance proc = workItemNodeInstance.getProcessInstance();
    VariableScopeInstance variableScopeInstance =
      (VariableScopeInstance)proc.getContextInstance( VariableScope.VARIABLE_SCOPE );
    System.out.println( "LOGGING: Requesting the correction of " +
                        variableScopeInstance.getVariable("order"));
end
```

8.4.7. Integrated Tooling

Processes and rules are integrated in the JBoss Rules Eclipse IDE. Both processes and rules are simply considered as different types of business logic, to be managed almost identically. For example, loading a process or a set of rules into the engine is very similar. Also, different rule implementations, such DRL or DSL, are handled in a uniform way.

```
private static KnowledgeBase createKnowledgeBase() throws Exception {
    KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
    kbuilder.add( ResourceFactory.newClassPathResource(
                  "RuleSetExample.rf", OrderExample.class), ResourceType.DRF );
    kbuilder.add( ResourceFactory.newClassPathResource(
                  "workflow_rules.drl", OrderExample.class), ResourceType.DRL );
    kbuilder.add( ResourceFactory.newClassPathResource(
                  "assignment.dsl", OrderExample.class), ResourceType.DSL );
    kbuilder.add( ResourceFactory.newClassPathResource(
                 "assignment.dslr", OrderExample.class), ResourceType.DSLR );

    KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
    kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
    return kbase;
}
```

Our audit log also contains an integrated view, showing how rules and processes are influencing each other. For example, a part of the log shows how rule "5% discount" is executed as part of the node "Calculate Discount".



```
RuleFlowGroup deactivated: validate[size=0]
  RuleFlow node triggered: XOR in process RuleSetExample[org.drools.examples.process.ruleset.RuleSetExample]
    RuleFlow node triggered:  in process RuleSetExample[org.drools.examples.process.ruleset.RuleSetExample]
      RuleFlow node triggered: Calculate Discount in process RuleSetExample[org.drools.examples.process.ruleset.RuleSetExample]
        RuleFlowGroup activated: discount[size=1]
Activation executed: Rule 5% discount if order includes laptop and order after 18h o=Order Order-1(1); date=Wed Jul 02 21:30:30 CEST 2008(1)
RuleFlowGroup deactivated: discount[size=0]
```

8.4.8. Domain-specific Rules and Processes

Rules do not need to be defined using the core rule language syntax, but they also can be defined using our more advanced rule editors, using domain-specific languages, decision tables, guided editors, etc. Our example defines a domain-specific language for describing assignment rules, based on the type of task, its properties, the process it is defined in, etc. This makes the assignment rules much more understandable for non-experts.

```
/********** Generic assignment rules **********/

rule "Assign 'Correct Order' to any sales representative"
    salience 30
    when
        There is a human task
        - with task name "Correct Order"
        - without actor id
    then
        Set actor id "Sales Representative"
end

/********** Assignment rules for the RuleSetExample process **********/

rule "Assign 'Follow-up Order' to a senior sales representative"
    salience 40
    when
        Process "org.drools.examples.process.ruleset.RuleSetExample" contains a human task
        - with task name "Follow-up Order"
        - without actor id
    then
        Set actor id "Senior Sales Representative"
end
```

Domain-specific processes

9.1. Introduction

One of the goals of our unified rules and processes framework is to allow users to extend the default programming constructs with domain-specific extensions that simplify development in a particular application domain. While JBoss Rules has been offering constructs to create domain-specific rule languages for some time now, this tutorial describes our first steps towards domain-specific process languages.

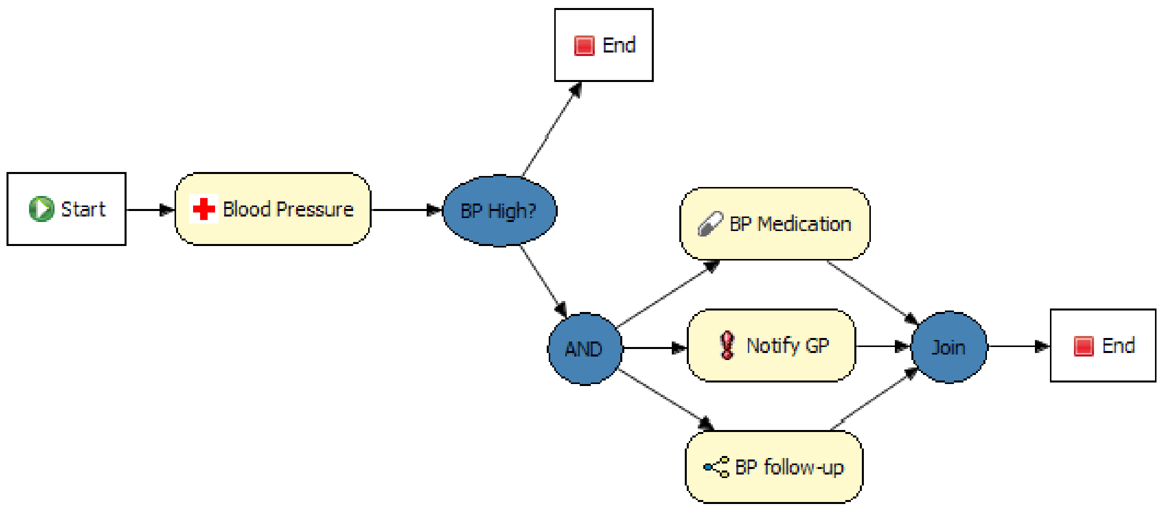
Most process languages offer some generic action (node) construct that allows plugging in custum user actions. However, these actions are usually low-level, where the user is required to write custom code to implement the work that should be incorporated in the process. The code is also closely linked 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, which represent atomic units of work that need to be executed. These work items 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 work items 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 work items and integrate them in our process language(s). For example, the next figure shows an example of a process in a healthcare context. The process includes domain-specific work items for ordering nursing tasks (e.g. measuring blood pressure), prescribing medication and notifying care providers.



9.2. 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]
```

9.2.1. Creating the work definition

All work 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"
  ]

]
```

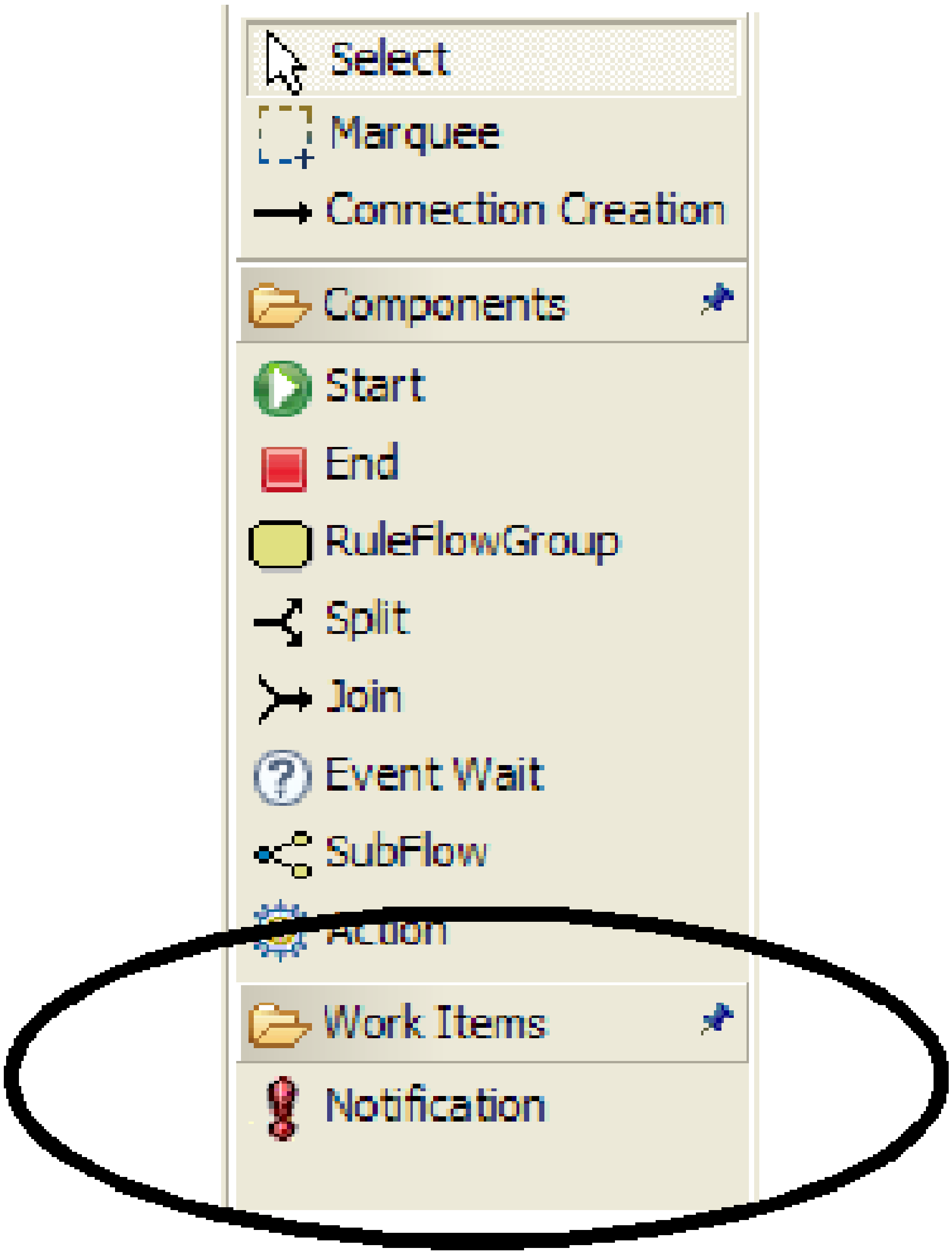
9.2.2. Registering the work definition

The JBoss Rules 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
```

9.2.3. 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 work items 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.

9.2.4. Executing work items

The JBoss Rules 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.process.instance.WorkItem;
import org.drools.process.instance.WorkItemHandler;
import org.drools.process.instance.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:

```
  workingMemory.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. The next section shows an example of how to use specialized work item handlers during testing.

9.3. Testing processes using work items

Customizable execution depending on context, easier to manage changes in environment (by changing handler), sharing processes accross contexts (using different handlers), testing, simulation (custom test handlers)

9.4. Future

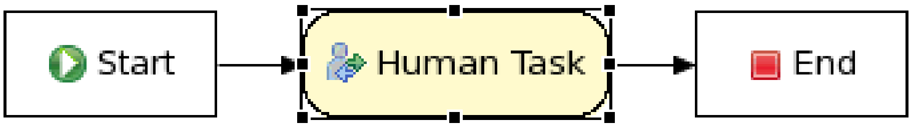
Our process framework is based on the (already well-known) idea of a Process Virtual Machine (PVM), where the process framework can be used as a basis for multiple process languages. This allows users to more easily create their own process languages, where common services provided by the process framework (e.g. persistence, audit) can be (re)used by the process language designer. Processes are represented as a graph of nodes, each node describing a part of the process logic. Different types of nodes are used for expressing different kinds of functionality, like creating or merging parallel flows (split and join), invoking a sub process, invoking external services, etc. One of our goals is creating a truly pluggable process language, where language designers can easily plug in their own node implementations.

Human Tasks

An important aspect of work flow and BPM (business process management)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. JBoss Rules Flow supports the use of human tasks inside processes using a special human 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 human 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 human 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.

10.1. Human tasks inside processes

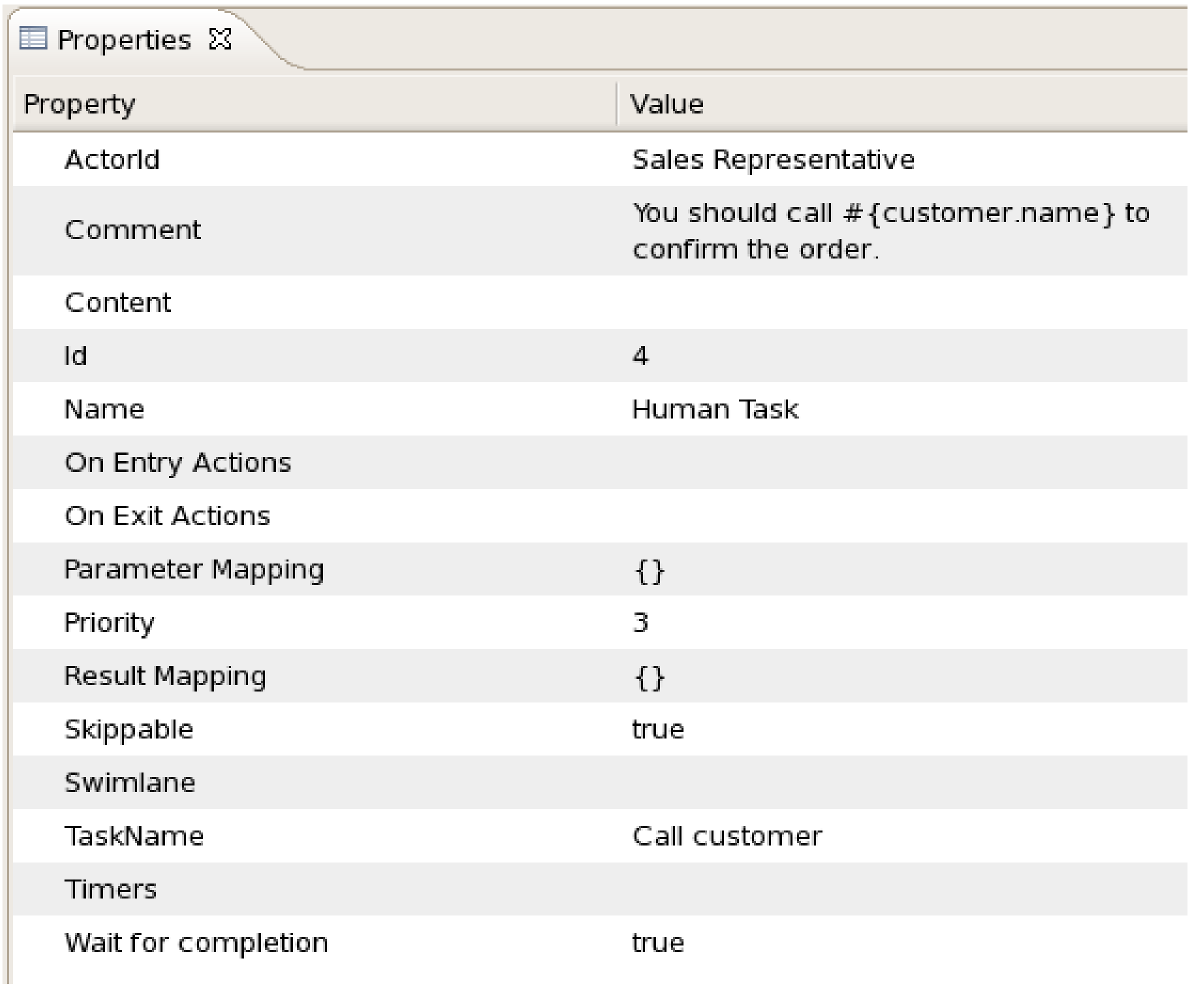


JBoss Rules Flow supports the use of human tasks inside processes using a special human task node (as shown in the figure above). A human task node represents an atomic task that needs to be executed by a human actor. Although JBoss Rules Flow has a special human task node for including human tasks inside 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 human task node is that we have added support for swimlanes, making it easier to assign tasks to users (see below). A human task node contains the following properties:

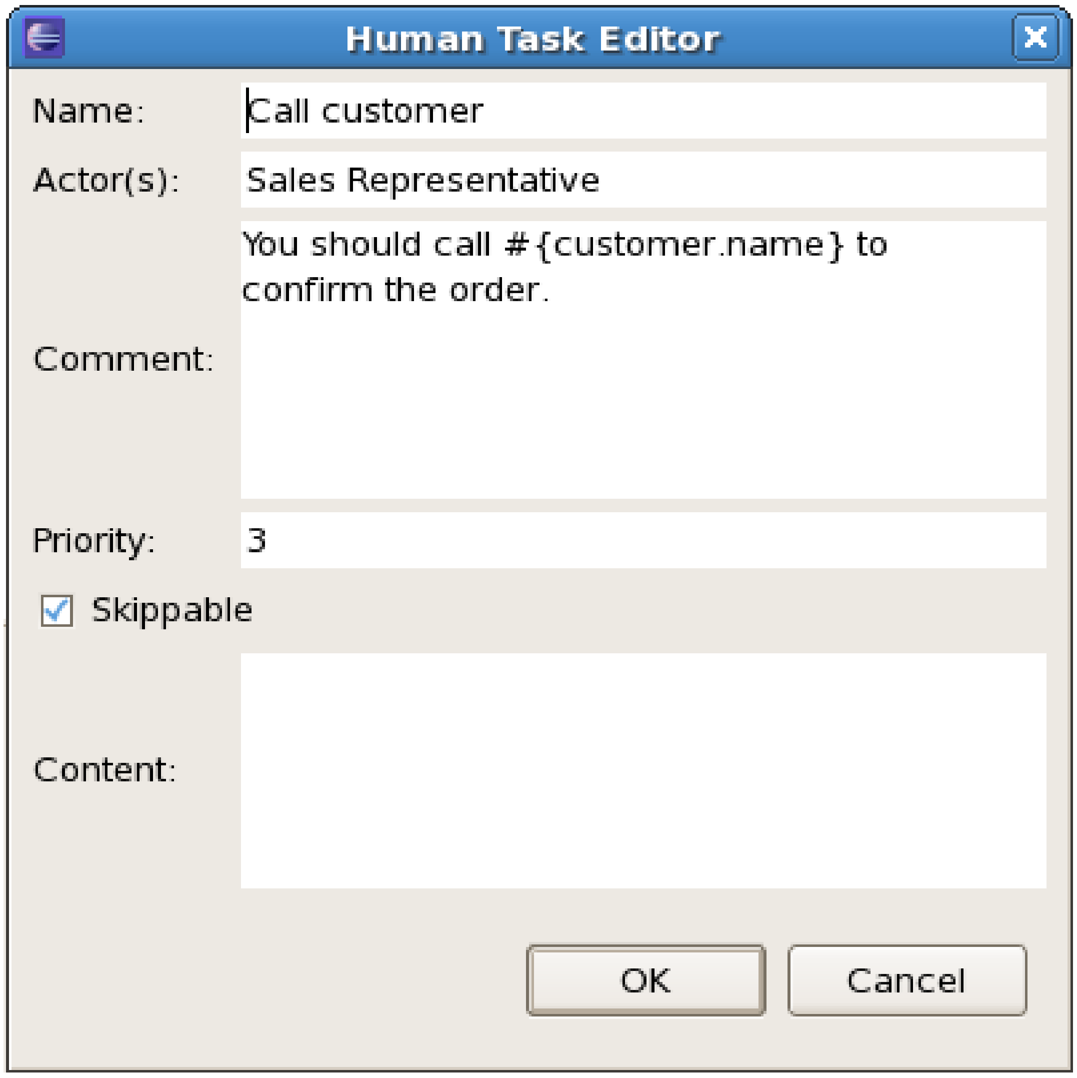
- *Id*: The id of the node (which is unique within one node container).
- *Name*: The display name of the node.
- *TaskName*: The name of the human task.
- *Priority*: An integer indicating the priority of the human task.
- *Comment*: A comment associated with the human task.
- *ActorId*: The actor id 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 human task node, or the most important properties can also be edited by double-clicking the human task node, after which a custom human task node editor is opened, as shown below as well.



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



Name: Call customer

Actor(s): Sales Representative

Comment: You should call #{customer.name} to confirm the order.

Priority: 3

[✓] Skippable

Content:

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.

10.1.1. Swimlanes

Human 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 human 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.

10.2. Human task management component

As far as the JBoss Rules Flow 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.

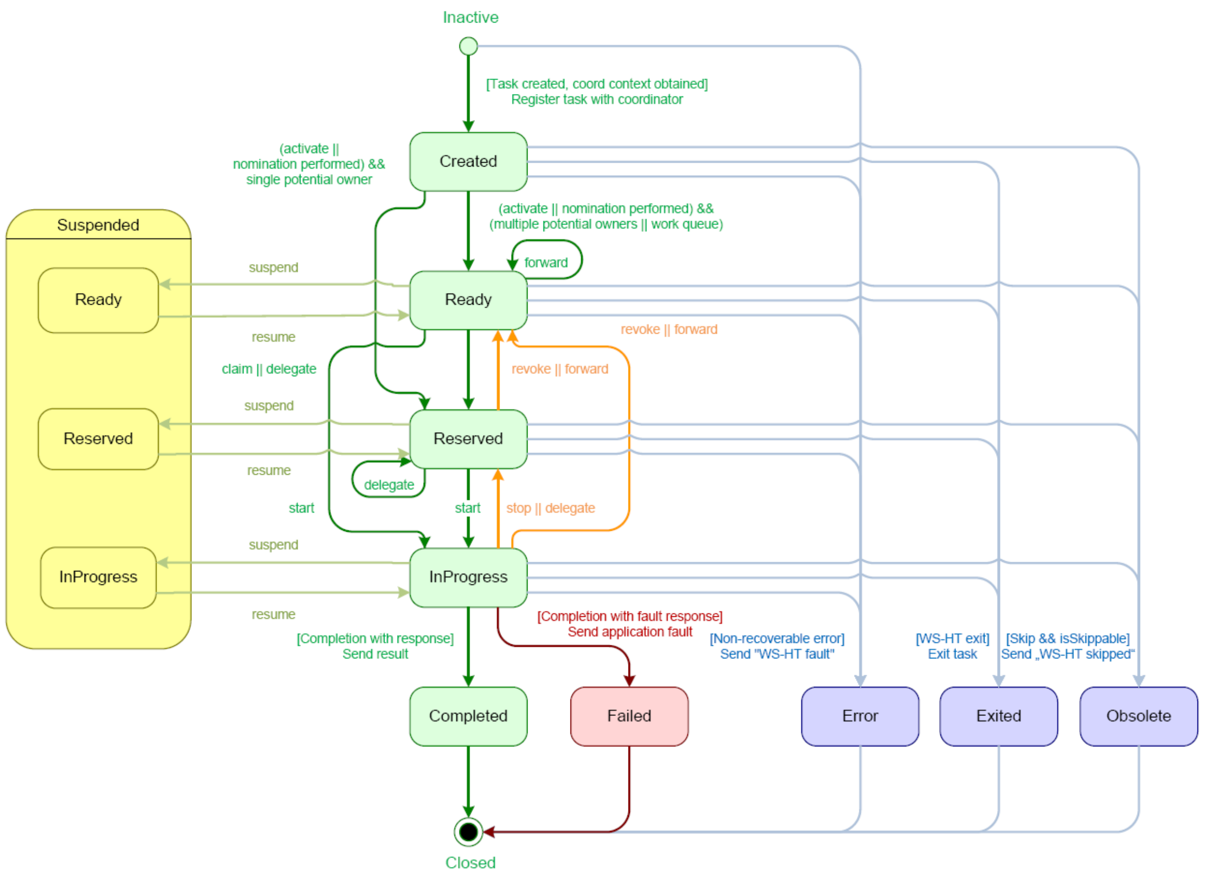
We do however provide an implementation of such a human task management component based on the WS-HumanTask specification. If you do not have the requirement to integrate a specific human task component yourself, you can use this service. It manages the task life cycle of the tasks (creation, claiming, 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*[1].

10.2.1. Task life cycle

Looking from the perspective of the process, whenever a human 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.

[1] http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf



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

10.2.2. Linking the task component to the JBoss Rules Flow engine

The task management component needs to be integrated with the JBoss Rules Flow 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.drools.process.workitem.wsht.WSHumanTaskHandler in the drools-process-task module) so you can easily link this work item handler like this:

```
 StatefulKnowledgeSession session = ...;
 session.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.

At this moment WSHumanTaskHandler is using Mina *(http://mina.apache.org/)*[2] for testing the behavior in a client/server architecture. Mina uses messages between client and server to enable the client comunicate with the server. That's why WSHumanTaskHandler have a MinaTaskClient that create different messages to give the user different actions that are executed for the server.

In the client (MinaTaskClient in this implementation) we should see the implementation of the following methods for interacting with 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 )
...
```

Using this methods we will implement any kind of GUI that the end user will use to do the task that they have assigned. If you take a look a this method signatures you will notice that almost all of this method takes the following arguments:

- **taskId**: the id of the task with we are working. Probably you will pick this Id from the user task list in the UI (User Interface).
- **userId**: the id of the user that is executing the action. Probably the Id of the user that is signed in the application.
- **responseHandler**: this is the handler have responsibility to catch the response and get the results or just let us know that the task is already finished.

[2] http://mina.apache.org/

As you can imagine all the methods create a message that will be send to the server, and the server will execute the logic that implement the correct action. A creation of one of this messages will be like this:

```
public void complete(long taskId,
                     String userId,
                     ContentData outputData,
                     TaskOperationResponseHandler responseHandler) {
  List<Object> args = new ArrayList<Object>( 5 );
  args.add( Operation.Complete );
  args.add( taskId );
  args.add( userId );
  args.add( null );
  args.add( outputData );
  Command cmd = new Command( counter.getAndIncrement(),
                             CommandName.OperationRequest,
                             args );

  handler.addResponseHandler( cmd.getId(),
                              responseHandler );
  session.write( cmd );
}
```

Here we can see that a Command is created and the arguments of the method are inserted inside the command with the type of operation that we are trying to execute and then this command is sended to the server with session.write(cmd) method.

If we see the server implementation, when the command is recived, we find that depends of the operation type (here Operation.Complete) will be the logic that will be executed. If we look at the class TaskServerHandler in the messageReceived method the taskOperation is executed using the taskServiceSession that is the responsible for get, persist and manipulate all the Human Task Information when the tasks are created and the user is not interacting with them.

10.2.3. 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. To start the task server, you can use the following code fragment:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("org.drools.task");
taskService = new TaskService(emf);
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.drools.task.Attachment</class>
    <class>org.drools.task.Content</class>
    <class>org.drools.task.BooleanExpression</class>
    <class>org.drools.task.Comment</class>
    <class>org.drools.task.Deadline</class>
    <class>org.drools.task.Comment</class>
    <class>org.drools.task.Deadline</class>
    <class>org.drools.task.Delegation</class>
    <class>org.drools.task.Escalation</class>
    <class>org.drools.task.Group</class>
    <class>org.drools.task.I18NText</class>
    <class>org.drools.task.Notification</class>
    <class>org.drools.task.EmailNotification</class>
    <class>org.drools.task.EmailNotificationHeader</class>
    <class>org.drools.task.PeopleAssignments</class>
    <class>org.drools.task.Reassignment</class>
    <class>org.drools.task.Status</class>
    <class>org.drools.task.Task</class>
    <class>org.drools.task.TaskData</class>
    <class>org.drools.task.SubTasksStrategy</class>
    <class>org.drools.task.OnParentAbortAllSubTasksEndStrategy</class>
    <class>org.drools.task.OnAllSubTasksEndParentEndStrategy</class>
    <class>org.drools.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>
```

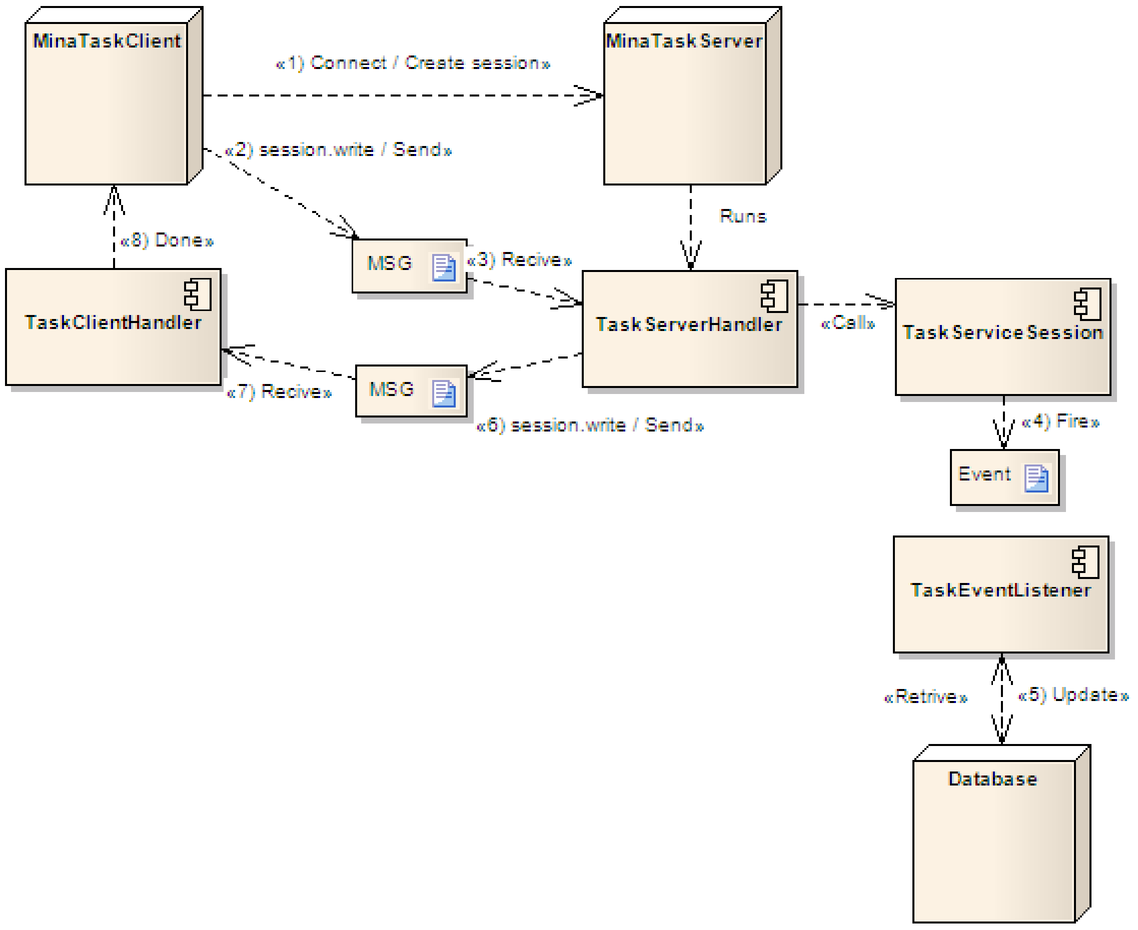
The first time you start the task management component, you need to make sure that all the necessary users and groups are added to the database. Our implementation requires all users and groups to be predefined before trying to assign a task to that user or group. So you need to make sure you add the 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 drools-process-task module contains a org.drools.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.

10.2.4. 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. These clients interact with the task management component using this API.

This interaction will be described with the following image:



As we can see in the image we have MinaTaskClient and MinaTaskServer. They communicate to each other sending messages to query and manipulate human tasks. Step by step the interactio n will be something like this:

- Some client need to complete some task. So he/she needs to create an instace of MinaTaskClient and connect it to the MinaTaskServer to have a session to talk to each other. This is the step one in the image.
- Then the client can call the method complete() in MinaTaskClient with the corresponding arguments. This will generate a new Message (or Command) that will be inserted in the session that the client open when it connects to the server. This message must specify a type that the server recognize and know what to do when the message is recieved. This is the step two in the image.
- At this moment TaskServerHandler noticed that there is a new message in the session so an analysis about what kind of message is will take place. In this case is the type of Operation.Complete, because the client is finishing succesfully some task. So we need to complete the task that the user want to finish. This is achieved using the TaskServiceSession that will fire an specific type of event that will be procesed by an specific subclass of TaskEventListener. This are step three and four in the image.
- When the event is recived by TaskEventListener it will know how to modify the status of the task. This is achieved using the EntityManager to retrieve and modify the status of an specific task from

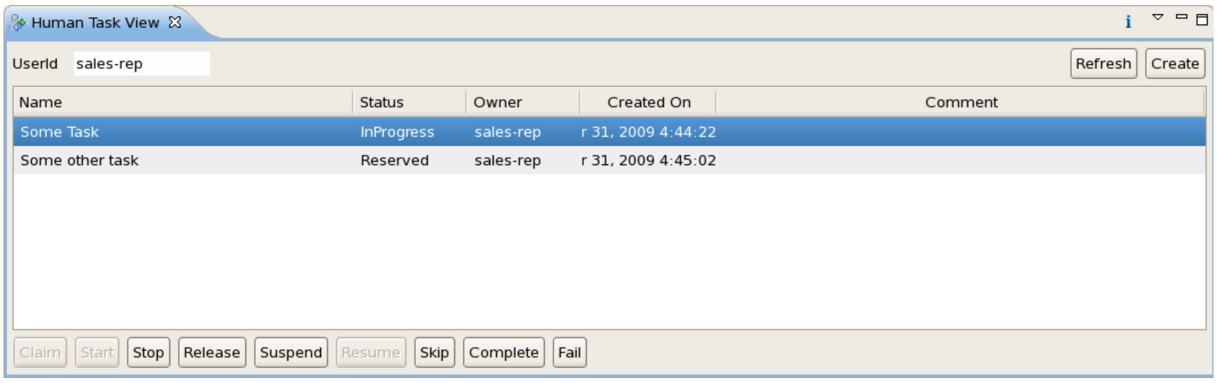
the database. In this case, because we are finishing a task, the status will be updated to Completed. This is step five in the image.

- Now, when the changes are made we need to notify the client about that the task was succesfully ended and this is achieved creating a response message that TaskClientHandler will receive and inform MinaTaskClient. This are steps six, seven and eight in the image.

10.3. Human Task Management Interface

10.3.1. Eclipse integration

The JBoss Rules 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 JBoss Rules Task preference page (select Window -> Preferences and select JBoss Rules Task). Here you can specify the url and port (default = 127.0.0.1:9123).



UserId sales-rep

Name	Status	Owner	Created On	Comment
Some Task	InProgress	sales-rep	r 31, 2009 4:44:22	
Some other task	Reserved	sales-rep	r 31, 2009 4:45:02	

10.3.2. Web-based Task View

We are targeting to add a web-based view that end users can use for managing their tasks for JBoss Rules 5.1.

Debugging processes

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

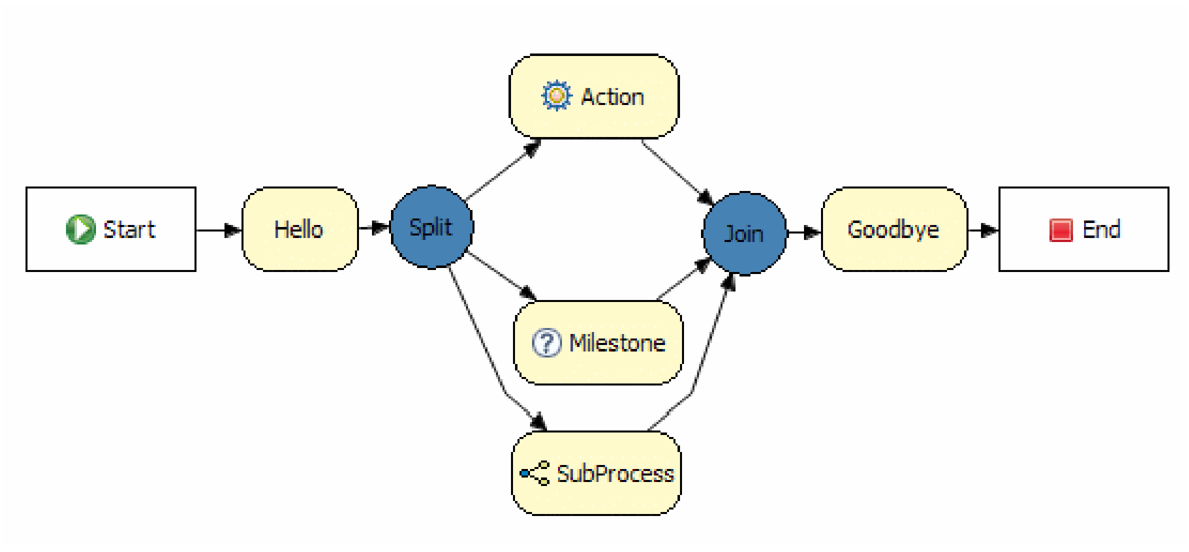
A screencast that shows processing debugging in action can be found at *http://downloads.jboss.com/drools/videos/OrderExample.swf*

We use a simple example throughout this section to illustrate the debugging capabilities. The example will be introduced first, followed by an illustration on how to use the debugging capabilities.

11.1. A simple example

Our example contains two processes and some rules (used inside the ruleflow groups):

1. The main process contains some of the most common nodes: a start and end node (obviously), two ruleflow groups, an action (that simply prints a string to the default output), a milestone (a wait state that is trigger when a specific Event is inserted in the working memory) and a subprocess.



2. The SubProcess simply contains a milestone that also waits for (another) specific Event in the working memory.

3. There are only two rules (one for each ruleflow group) that simply print out either a hello world or goodbye world to default output.

We will simulate the execution of this process by starting the process, firing all rules (resulting in the executing of the hello rule), then adding the specific milestone events for both the milestones (in the main process and in the subprocess) and finally by firing all rules again (resulting in the executing of the goodbye rule). The console will look something like this:

```
Hello World
Executing action
Goodbye cruel world
```

11.2. Debugging the process

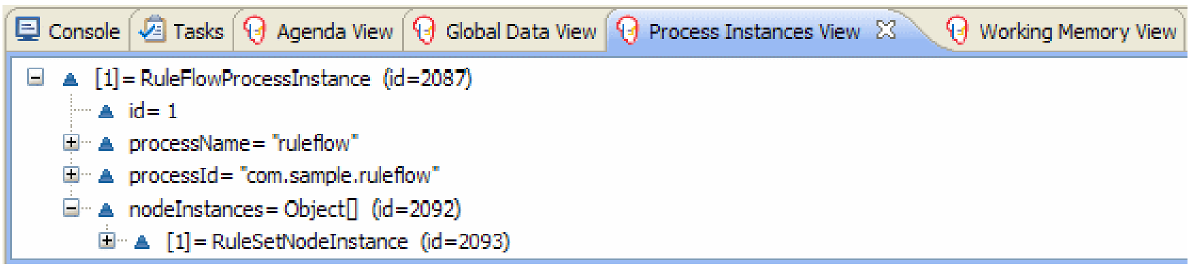
We now add four breakpoints during the execution of the process (in the order in which they will be encountered):

1. At the start of the consequence of the hello rule
2. Before inserting the triggering event for the milestone in the main process
3. Before inserting the triggering event for the milestone in the subprocess
4. At the start of the consequence of the goodbye rule

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

1. The working memory view, showing the contents (data) in the working memory.
2. The agenda view, showing all activations in the agenda.
3. The global data view, showing the globals.
4. The default Java Debug views, showing the current line and the value of the known variables, and this both for normal Java code as for rules.
5. The process instances view, showing all running processes (and their state).
6. The audit view, showing the audit log.

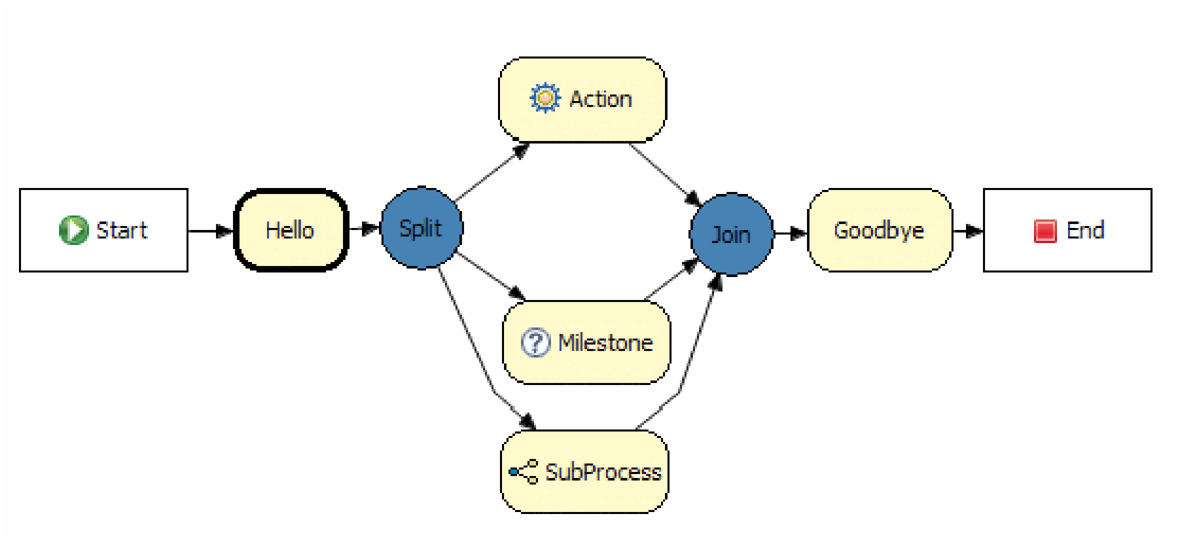
11.2.1. The Process Instances View



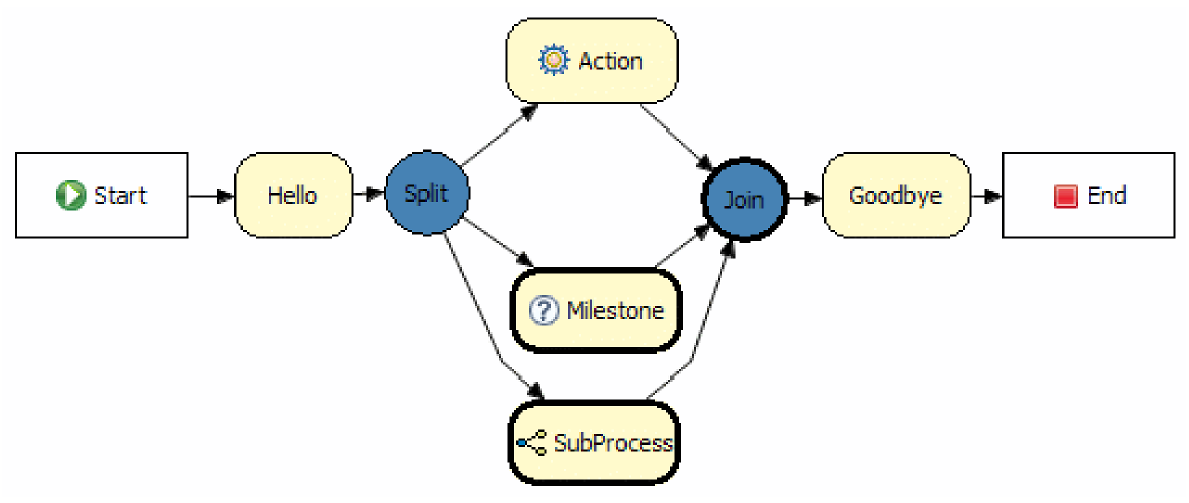
```
[1]= RuleFlowProcessInstance  (id=2087)
    id= 1
    processName= "ruleflow"
    processId= "com.sample.ruleflow"
    nodeInstances= Object[]  (id=2092)
        [1]= RuleSetNodeInstance  (id=2093)
```

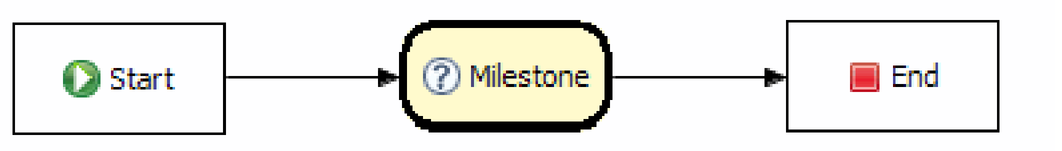
The process instances view shows the currently running process instances. The example shows that there is currently one running process (instance), currently executing one node (instance), i.e. RuleSet node. When double-clicking a process instance, the process instance viewer will graphically show the progress of the process instance. At each of the breakpoints, this will look like:

1. At the start of the consequence of the hello rule, only the hello ruleflow group is active, waiting on the execution of the hello rule:

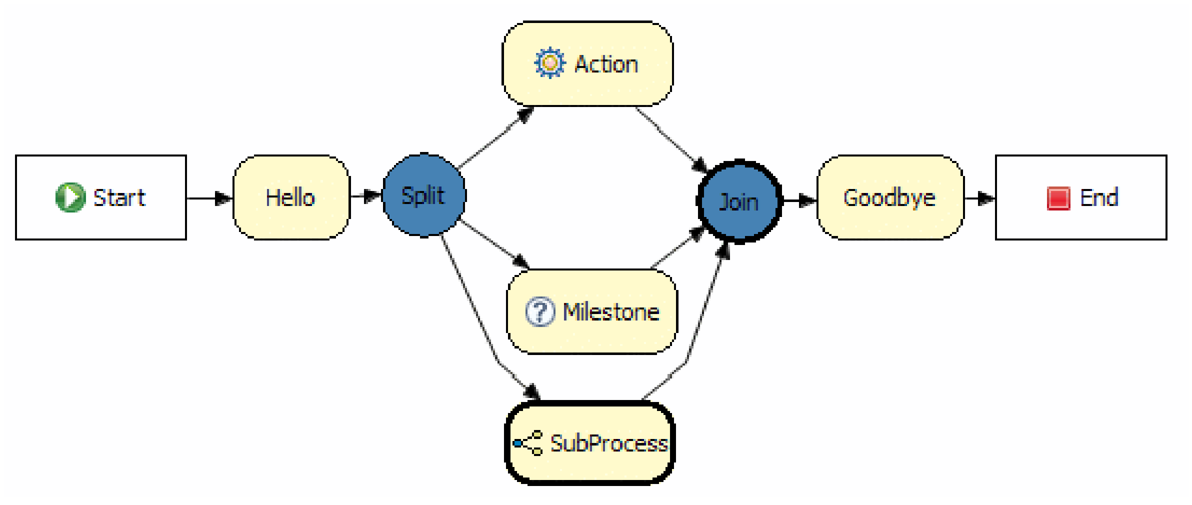


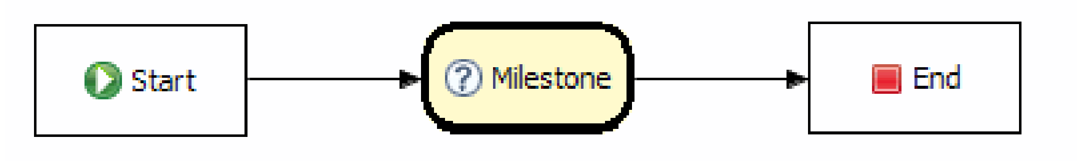
2. Once that rule has been executed, the action, the milestone and the subprocess will be triggered. The action will be executed immediately, triggering the join (which will simply wait until all incomming connections have been triggered). The subprocess will wait at the milestone. So, before inserting the triggering event for the milestone in the main process, there now are two process instances, looking like this:



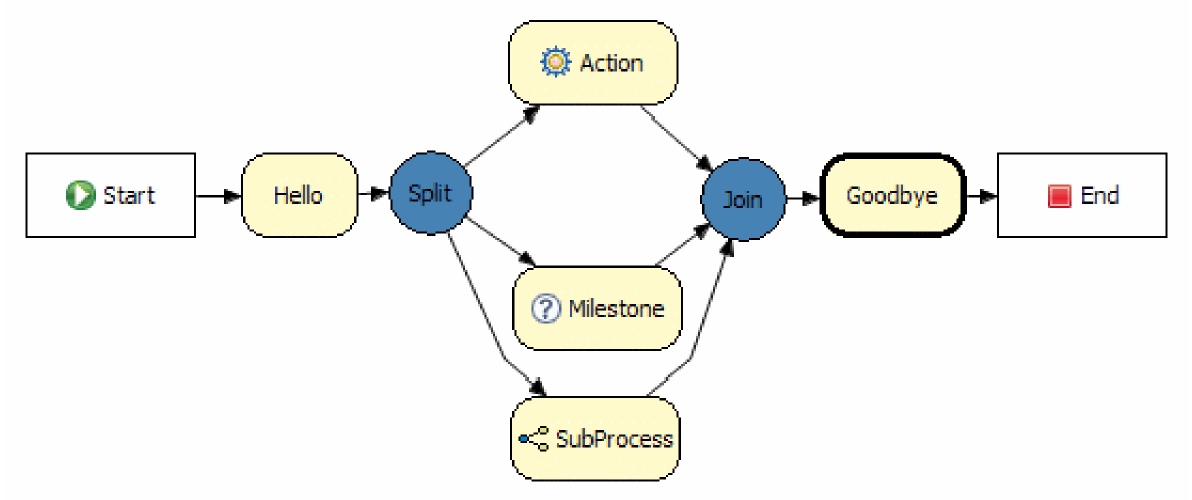


3. When triggering the event for the milestone in the main process, this will also trigger the join (which will simply wait until all incomming connections have been triggered). So at that point (before inserting the triggering event for the milestone in the subprocess), the processes will look like this:





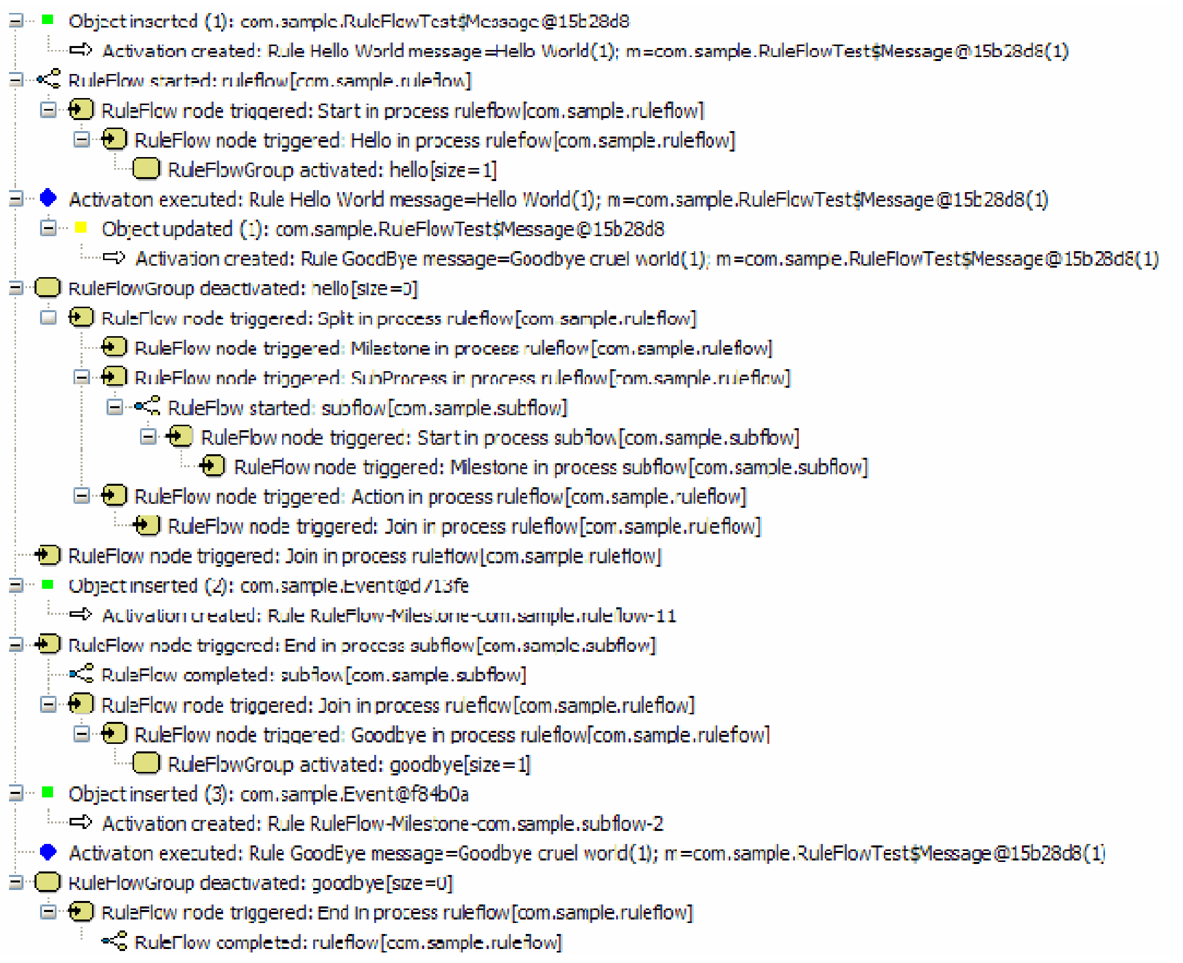
4. When triggering the event for the milestone in the subprocess, this process instance will be completed and this will also trigger the join, which will then continue and trigger the goodbye ruleflow group, as all its incomming connections have been triggered. Firing all the rules will trigger the breakpoint in the goodbye rule. At that point, the situation looks like this:



5. After executing the goodbye rule, the main process will also be completed and the execution will have reached the end.

11.2.2. The Audit View

For those who want to look at the result in the audit view, this will look something like this [Note: the object insertion events might seem a little out of place, which is caused by the fact that they are only logged after (and never before) they are inserted, making it difficult to exactly pinpoint their location.]



```
Object inserted (1): com.sample.RuleFlowTest$Message@15b28d8
    Activation created: Rule Hello World message=Hello World(1); m=com.sample.RuleFlowTest$Message@15b28d8(1)
RuleFlow started: ruleflow[com.sample.ruleflow]
    RuleFlow node triggered: Start in process ruleflow[com.sample.ruleflow]
        RuleFlow node triggered: Hello in process ruleflow[com.sample.ruleflow]
            RuleFlowGroup activated: hello[size=1]
Activation executed: Rule Hello World message=Hello World(1); m=com.sample.RuleFlowTest$Message@15b28d8(1)
    Object updated (1): com.sample.RuleFlowTest$Message@15b28d8
        Activation created: Rule GoodBye message=Goodbye cruel world(1); m=com.sample.RuleFlowTest$Message@15b28d8(1)
RuleFlowGroup deactivated: hello[size=0]
    RuleFlow node triggered: Split in process ruleflow[com.sample.ruleflow]
        RuleFlow node triggered: Milestone in process ruleflow[com.sample.ruleflow]
        RuleFlow node triggered: SubProcess in process ruleflow[com.sample.ruleflow]
            RuleFlow started: subflow[com.sample.subflow]
                RuleFlow node triggered: Start in process subflow[com.sample.subflow]
                    RuleFlow node triggered: Milestone in process subflow[com.sample.subflow]
        RuleFlow node triggered: Action in process ruleflow[com.sample.ruleflow]
            RuleFlow node triggered: Join in process ruleflow[com.sample.ruleflow]
RuleFlow node triggered: Join in process ruleflow[com.sample.ruleflow]
Object inserted (2): com.sample.Event@d713fe
    Activation created: Rule RuleFlow-Milestone-com.sample.ruleflow-11
RuleFlow node triggered: End in process subflow[com.sample.subflow]
    RuleFlow completed: subflow[com.sample.subflow]
    RuleFlow node triggered: Join in process ruleflow[com.sample.ruleflow]
        RuleFlow node triggered: Goodbye in process ruleflow[com.sample.ruleflow]
            RuleFlowGroup activated: goodbye[size=1]
Object inserted (3): com.sample.Event@f84b0a
    Activation created: Rule RuleFlow-Milestone-com.sample.subflow-2
Activation executed: Rule GoodBye message=Goodbye cruel world(1); m=com.sample.RuleFlowTest$Message@15b28d8(1)
RuleFlowGroup deactivated: goodbye[size=0]
    RuleFlow node triggered: End in process ruleflow[com.sample.ruleflow]
        RuleFlow completed: ruleflow[com.sample.ruleflow]
```

JBoss Rules IDE Features

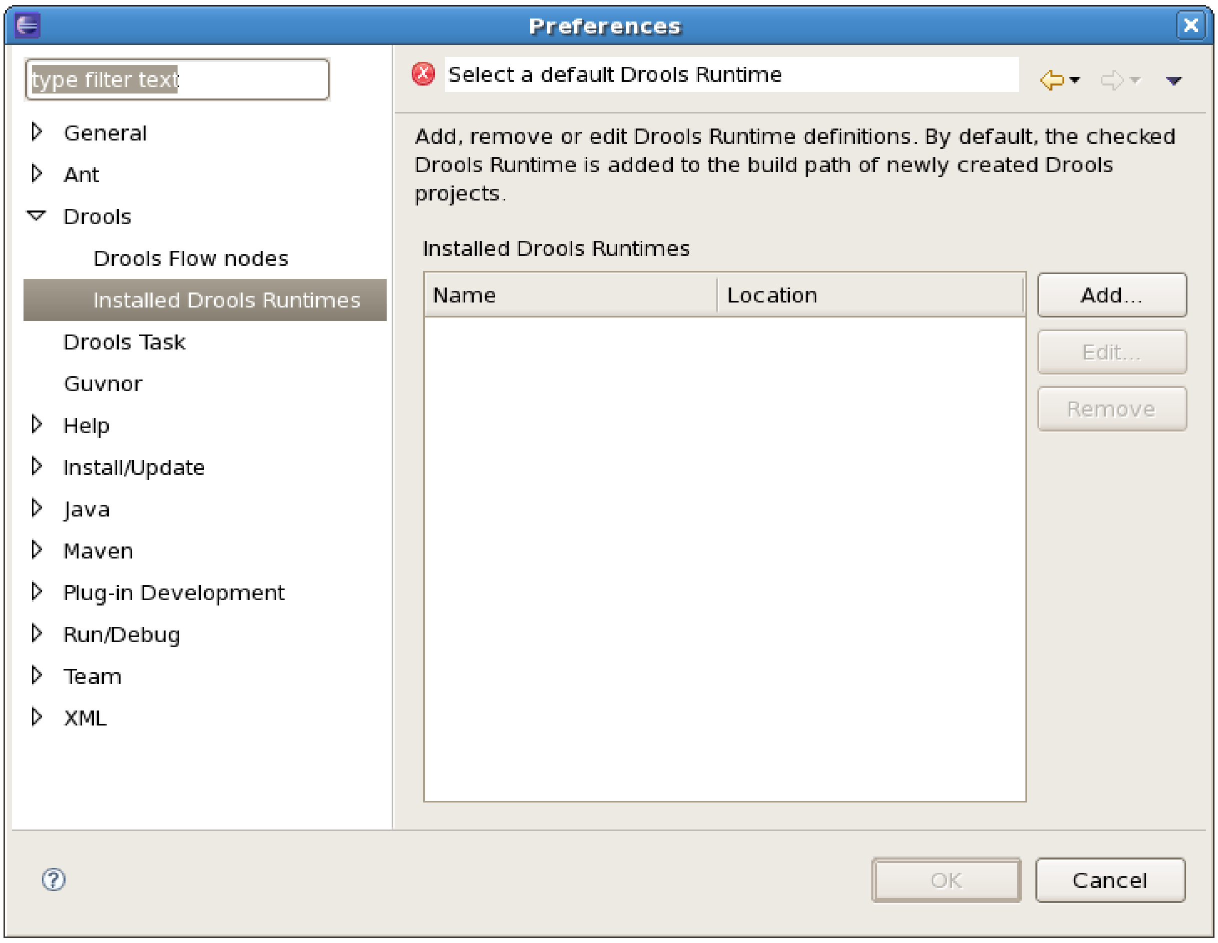
The JBoss Rules plugin for the IDE provides a few additional features that might be interesting for developers.

12.1. JBoss Rules Runtimes

A JBoss Rules runtime is a collection of jar files that represent one specific release of the JBoss Rules project jars. To create a runtime, you must point the IDE to the release of your choice. If you want to create a new runtime based on the latest JBoss Rules project jars included in the plugin itself, you can also easily do that. You are required to specify a default JBoss Rules runtime for your workspace, but each individual project can override the default and select the appropriate runtime for that project specifically.

12.1.1. Defining a JBoss Rules Runtime

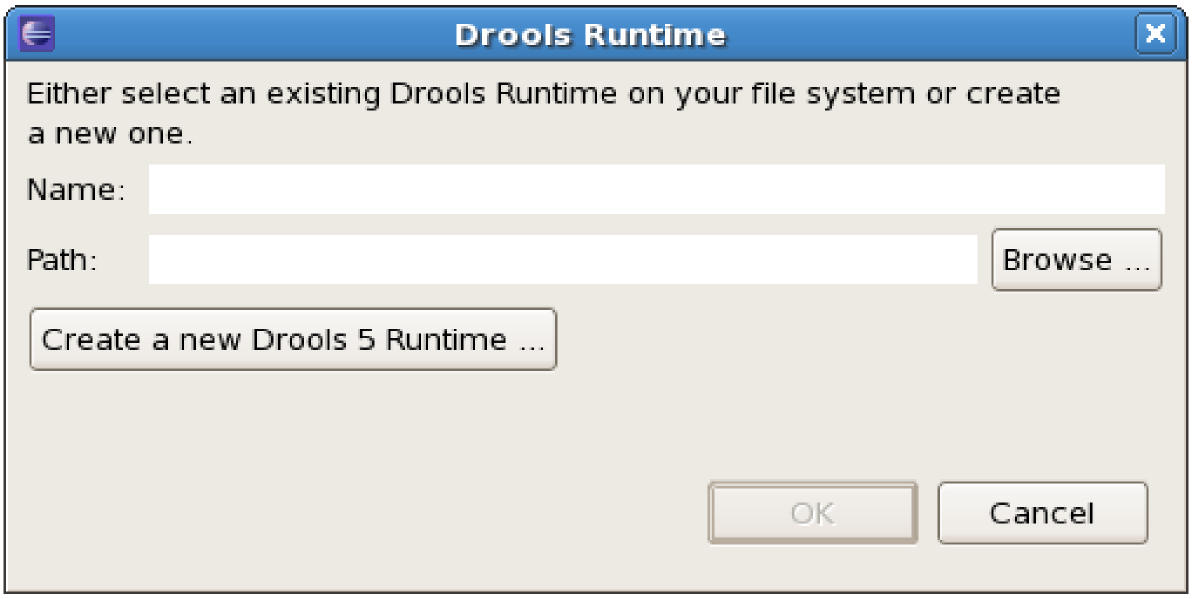
To define one or more JBoss Rules runtimes using the preferences view you open up your Preferences, by selecting the "Preferences" menu item in the menu "Window". A "Preferences" dialog should show all your settings. On the left side of this dialog, under the JBoss Rules category, select "Installed JBoss Rules runtimes". The panel on the right should then show the currently defined JBoss Rules runtimes. If you have not yet defined any runtimes, it should look like the figure below.



Select a default Drools Runtime

Add, remove or edit Drools Runtime definitions. By default, the checked Drools Runtime is added to the build path of newly created Drools projects.

To define a new JBoss Rules runtime, click on the add button. A dialog such as the one shown below should pop up, asking for the name of your runtime and the location on your file system where it can be found.


Drools Runtime

Either select an existing Drools Runtime on your file system or create a new one.

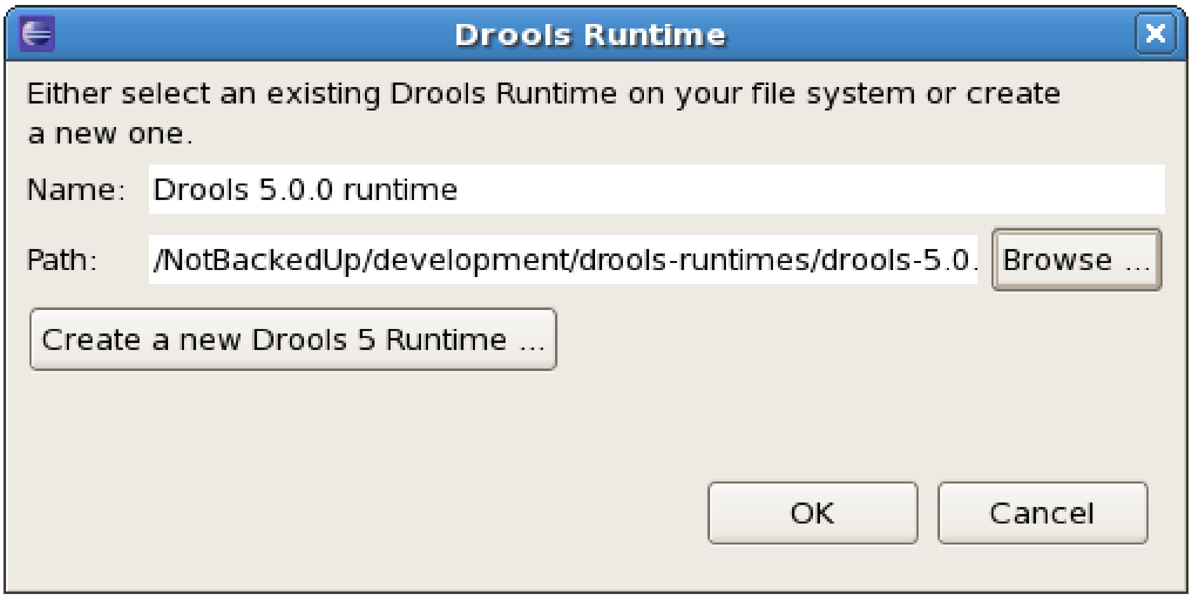
Name:

Path:

In general, you have two options:

1. If you simply want to use the default jar files as included in the JBoss Rules plugin, you can create a new JBoss Rules runtime automatically by clicking the "Create a new JBoss Rules 5 runtime ..." button. A file browser will show up, asking you to select the folder on your file system where you want this runtime to be created. The plugin will then automatically copy all required dependencies to the specified folder. After selecting this folder, the dialog should look like the figure shown below.

2. If you want to use one specific release of the JBoss Rules project, you should create a folder on your file system that contains all the necessary JBoss Rules libraries and dependencies. Instead of creating a new JBoss Rules runtime as explained above, give your runtime a name and select the location of this folder containing all the required jars.

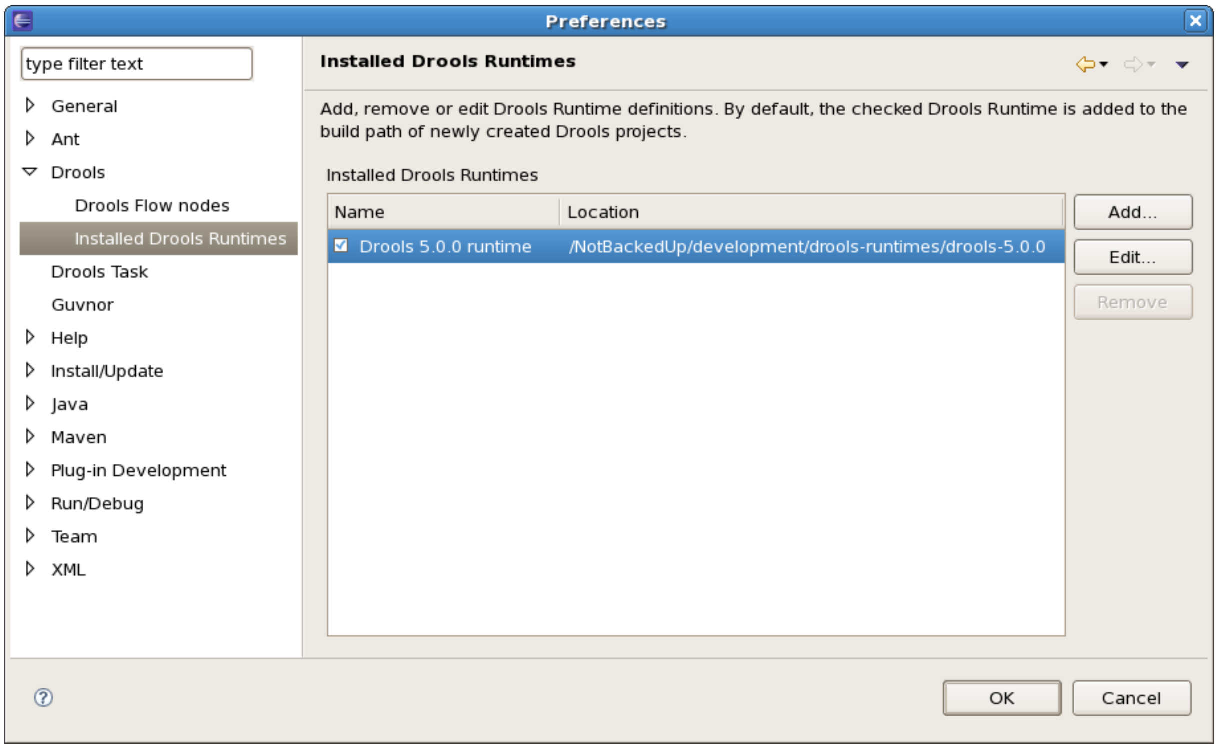

Drools Runtime

Either select an existing Drools Runtime on your file system or create a new one.

Name: Drools 5.0.0 runtime

Path: /NotBackedUp/development/drools-runtimes/drools-5.0.

After clicking the OK button, the runtime should show up in your table of installed JBoss Rules runtimes, as shown below. Click on checkbox in front of the newly created runtime to make it the default JBoss Rules runtime. The default JBoss Rules runtime will be used as the runtime of all your JBoss Rules project that have not selected a project-specific runtime.

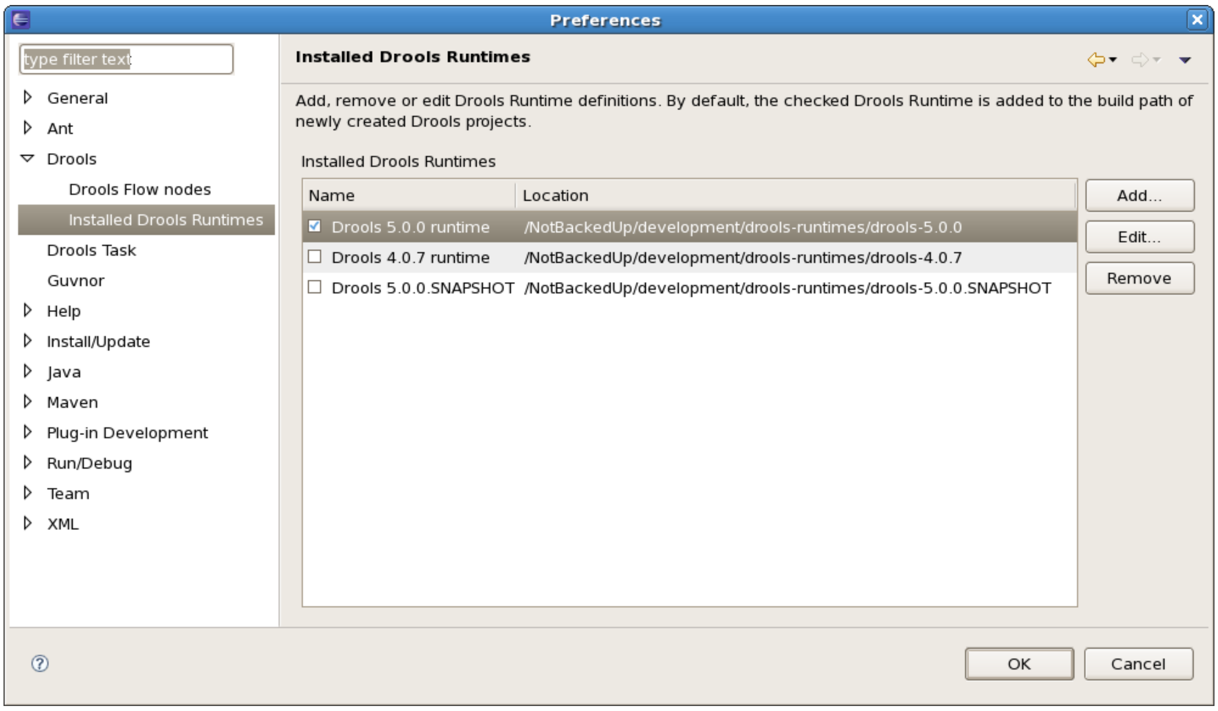

Installed Drools Runtimes

Add, remove or edit Drools Runtime definitions. By default, the checked Drools Runtime is added to the build path of newly created Drools projects.

Installed Drools Runtimes

Name	Location
☑ Drools 5.0.0 runtime	/NotBackedUp/development/drools-runtimes/drools-5.0.0

You can add as many JBoss Rules runtimes as you need. For example, the screenshot below shows a configuration where three runtimes have been defined: a JBoss Rules 4.0.7 runtime, a JBoss Rules 5.0.0 runtime and a JBoss Rules 5.0.0.SNAPSHOT runtime. The JBoss Rules 5.0.0 runtime is selected as the default one.


Installed Drools Runtimes

Add, remove or edit Drools Runtime definitions. By default, the checked Drools Runtime is added to the build path of newly created Drools projects.

Installed Drools Runtimes

Name	Location
☑ Drools 5.0.0 runtime	/NotBackedUp/development/drools-runtimes/drools-5.0.0
☐ Drools 4.0.7 runtime	/NotBackedUp/development/drools-runtimes/drools-4.0.7
☐ Drools 5.0.0.SNAPSHOT	/NotBackedUp/development/drools-runtimes/drools-5.0.0.SNAPSHOT

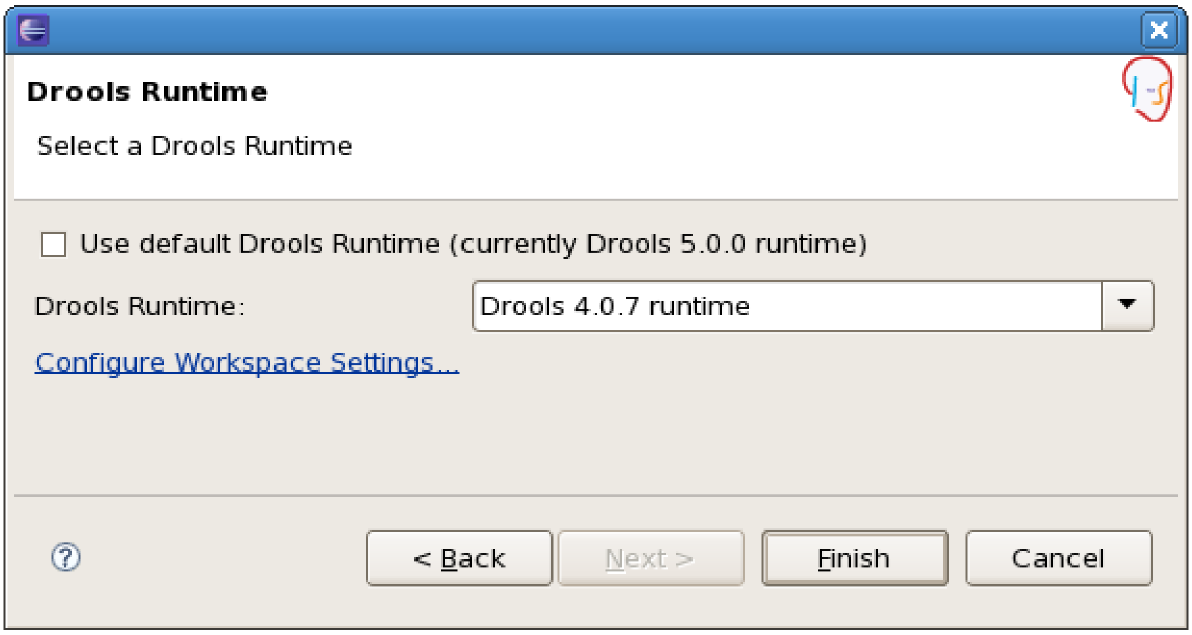
Note that you will need to restart the IDE if you changed the default runtime and you want to make sure that all the projects that are using the default runtime update their classpath accordingly.

12.1.2. Selecting a runtime for your JBoss Rules project

Whenever you create a JBoss Rules project (using the New JBoss Rules Project wizard or by converting an existing Java project to a JBoss Rules project using the action "Convert to JBoss Rules

Project" that is shown when you are in the JBoss Rules perspective and you right-click an existing Java project), the plugin will automatically add all the required jars to the classpath of your project.

When creating a new JBoss Rules project, the plugin will automatically use the default JBoss Rules runtime for that project, unless you specify a project-specific one. You can do this in the final step of the New JBoss Rules Project wizard, as shown below, by deselecting the "Use default JBoss Rules runtime" checkbox and selecting the appropriate runtime in the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed JBoss Rules runtimes will be opened, so you can add new runtimes there.

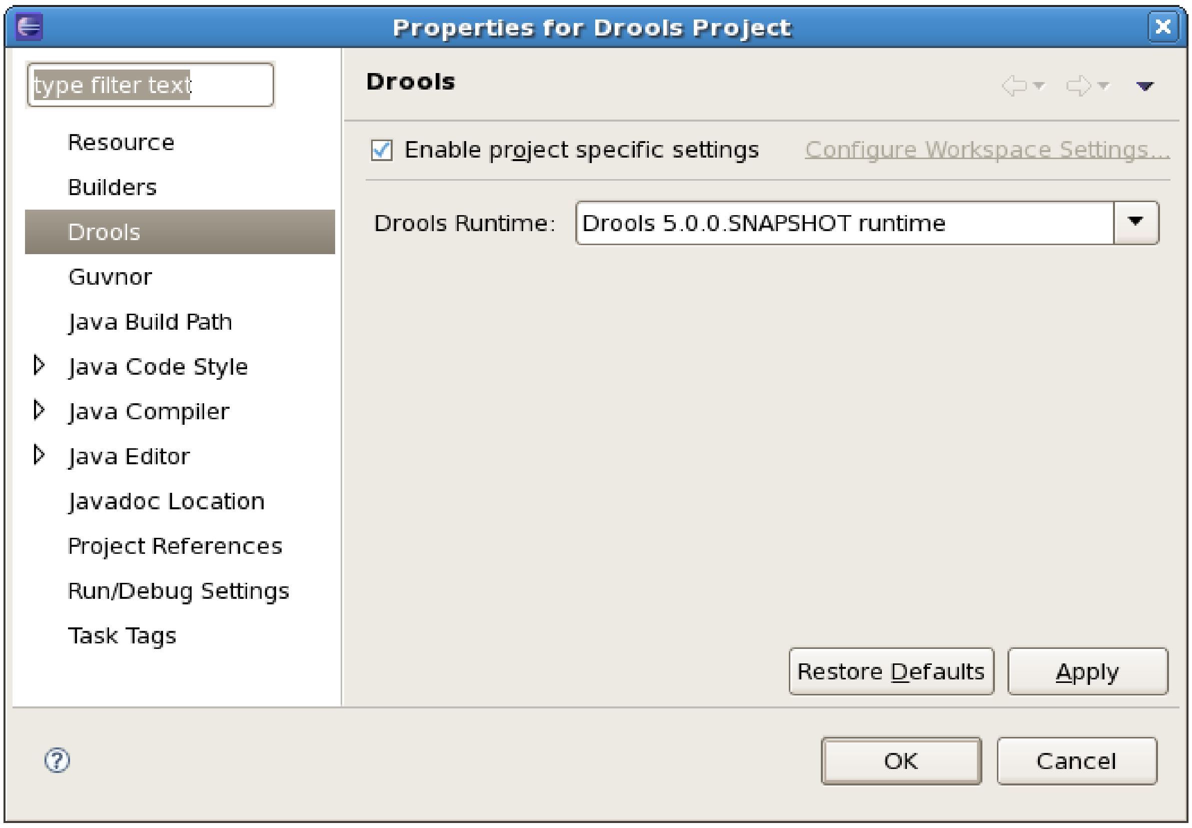

Drools Runtime

Select a Drools Runtime

☐ Use default Drools Runtime (currently Drools 5.0.0 runtime)

Drools Runtime: Drools 4.0.7 runtime

You can change the runtime of a JBoss Rules project at any time by opening the project properties and selecting the JBoss Rules category, as shown below. Mark the "Enable project specific settings" checkbox and select the appropriate runtime from the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed JBoss Rules runtimes will be opened, so you can add new runtimes there. If you deselect the "Enable project specific settings" checkbox, it will use the default runtime as defined in your global preferences.



✓ Enable project specific settings

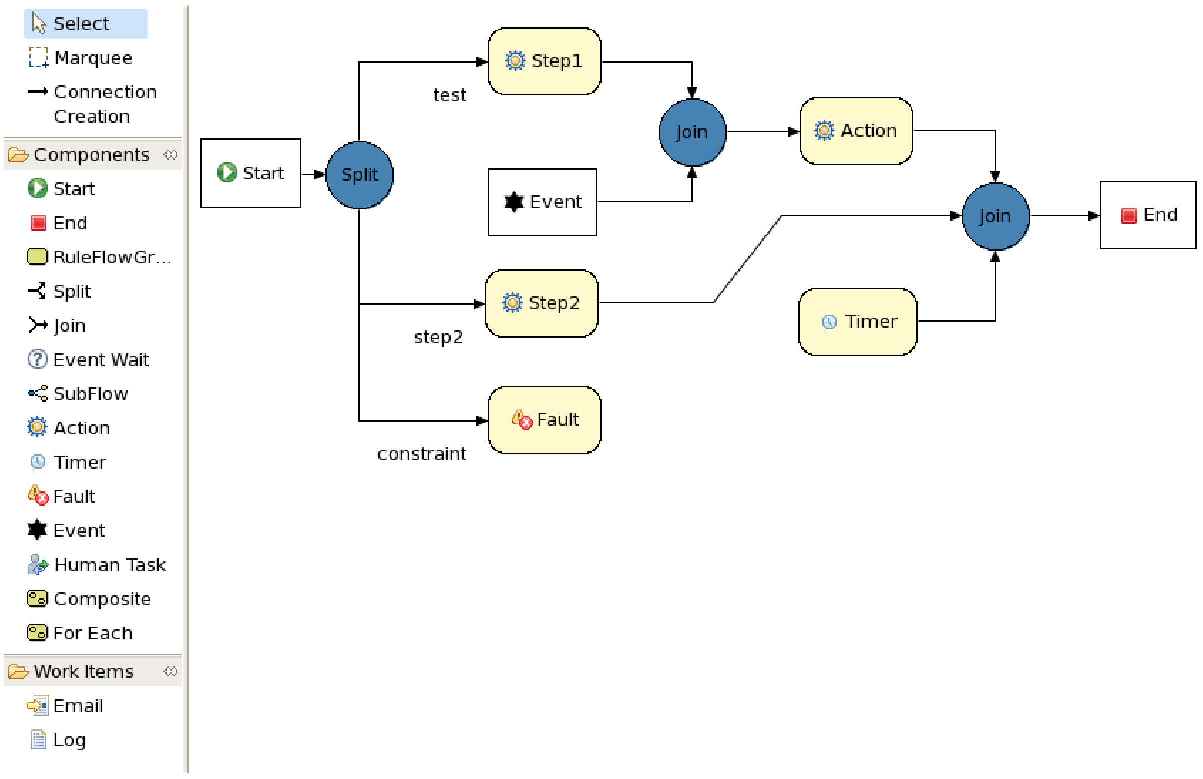
Drools Runtime: Drools 5.0.0.SNAPSHOT runtime

12.2. Process Skins

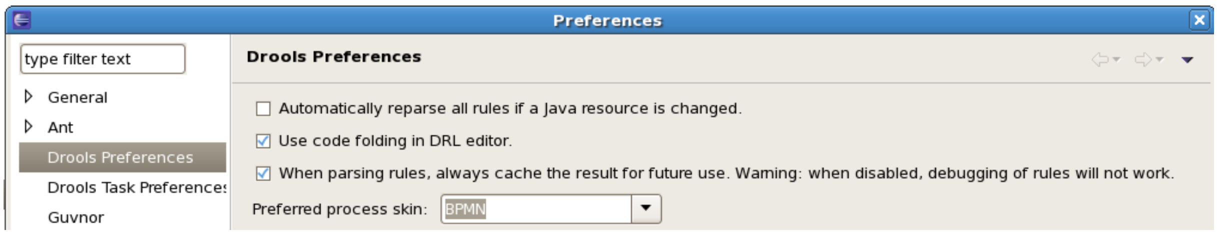
The concept of *process skins* provides a way of control the visualization of the different nodes of a processd. You may change the visualization of the various node types to the way you prefer by implementing your own **`SkinProvider`**.

BPMN is a popular language used by business users for modeling business processes. BPMN defines terminology, different types of nodes, how these should be visualized, etc. People who are familiar with BPMN might find it easier to implement an executable process (possibly based on a BPMN process diagram) using a similar visualization. We have therefore created a BPMN skin that maps the JBoss Rules Flow concepts to the equivalent BPMN visualization.

As an example, the following figure shows a process using some of the different types of nodes in the RuleFlow language using the default skin.



You may now change the preferred process skin in the JBoss Rules Preferences dialog:


Preferences

type filter text

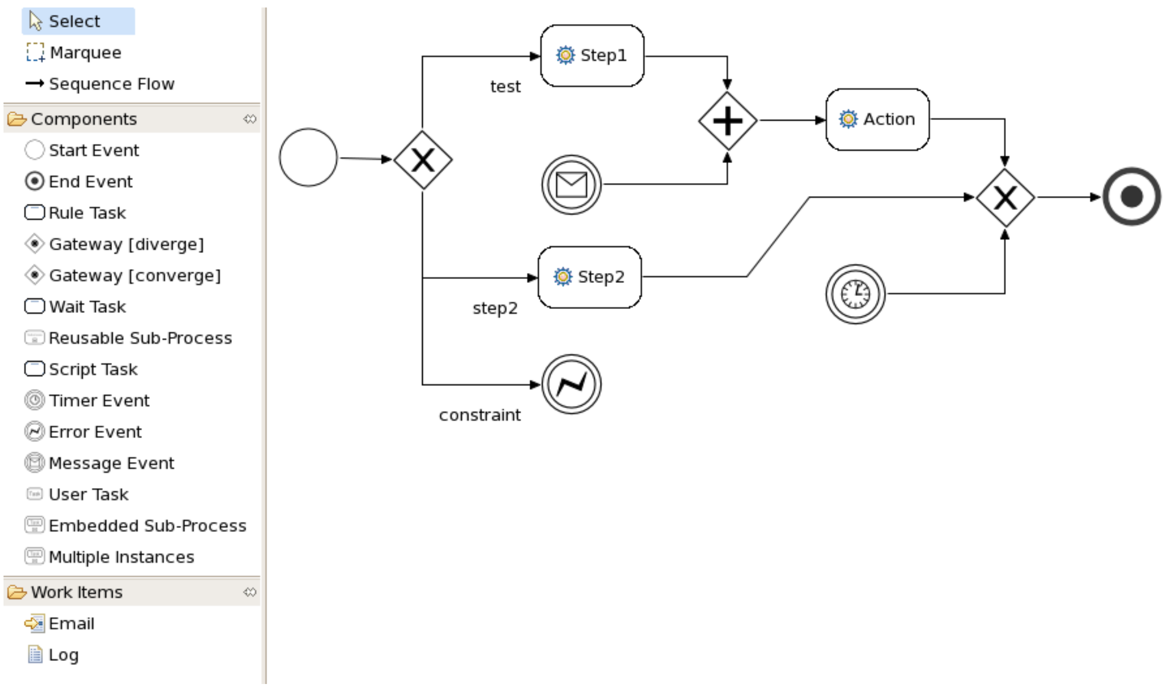
- General
- Ant
- Drools Preferences
- Drools Task Preferences
- Guvnor

Drools Preferences

- [ ] Automatically reparse all rules if a Java resource is changed.
- [x] Use code folding in DRL editor.
- [x] When parsing rules, always cache the result for future use. Warning: when disabled, debugging of rules will not work.

Preferred process skin: BPMN

After reopening the editor, the same process is displayed using the BPMN skin.



Business Activity Monitoring

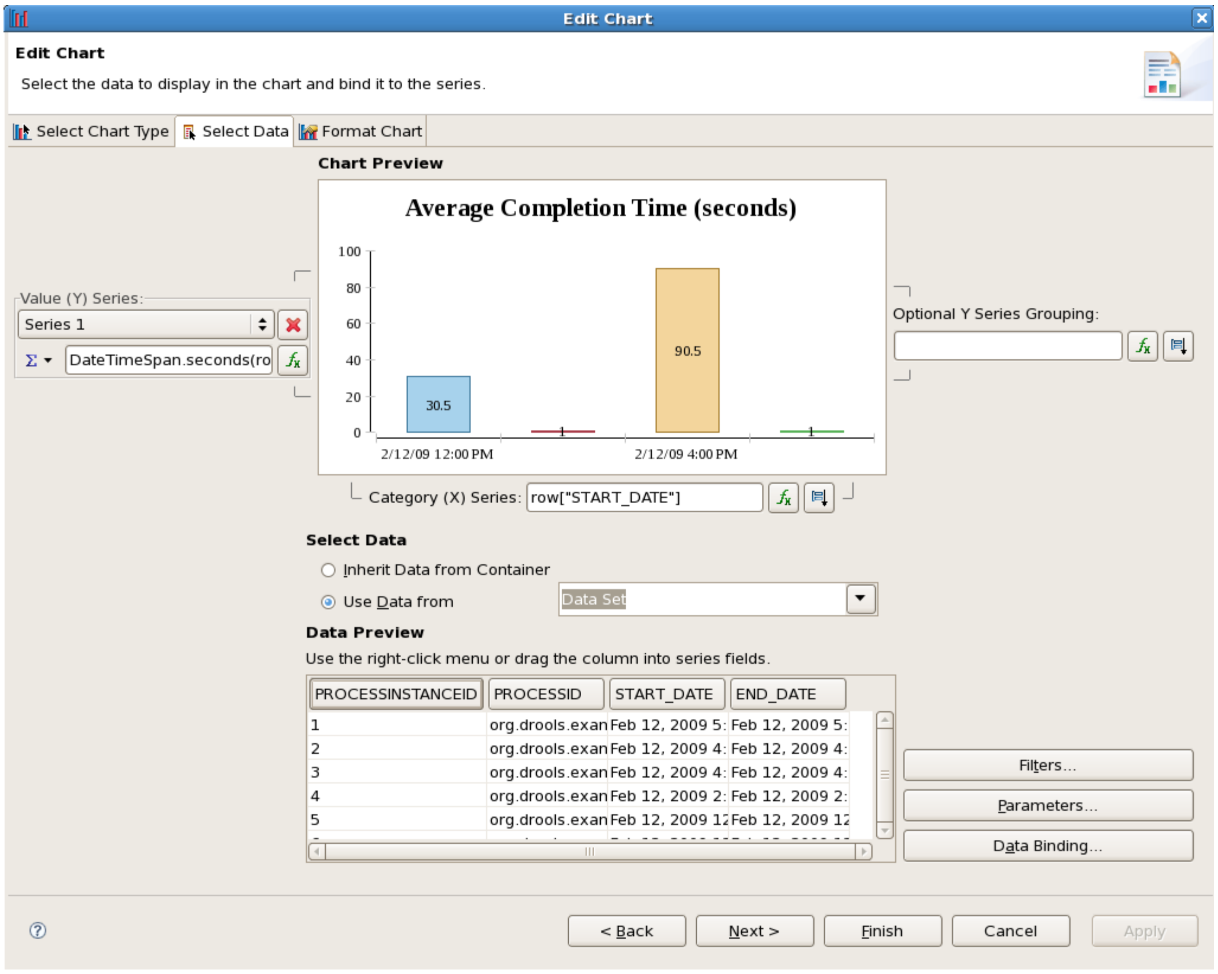
You need to actively monitor your processes to make sure you can detect any anomalies and react to unexpected events as soon as possible. Business Activity Monitoring is concerned with real-time monitoring of your processes and the option of intervening directly, possibly even automatically, based on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

JBoss Rules Flow 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 (JBoss Rules Fusion), as described in the next two sections. Future releases of the JBoss Rules 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.

13.1. 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.



Edit Chart

Select the data to display in the chart and bind it to the series.

Value (Y) Series: Series 1

DateTimeSpan.seconds(ro

Optional Y Series Grouping:

Category (X) Series: row["START_DATE"]

Select Data

( ) Inherit Data from Container

(●) Use Data from Data Set

Data Preview

Use the right-click menu or drag the column into series fields.

PROCESSINSTANCEID	PROCESSID	START_DATE	END_DATE
1	org.drools.exan	Feb 12, 2009 5:	Feb 12, 2009 5:
2	org.drools.exan	Feb 12, 2009 4:	Feb 12, 2009 4:
3	org.drools.exan	Feb 12, 2009 4:	Feb 12, 2009 4:
4	org.drools.exan	Feb 12, 2009 2:	Feb 12, 2009 2:
5	org.drools.exan	Feb 12, 2009 12	Feb 12, 2009 12

Figure 13.1. 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.

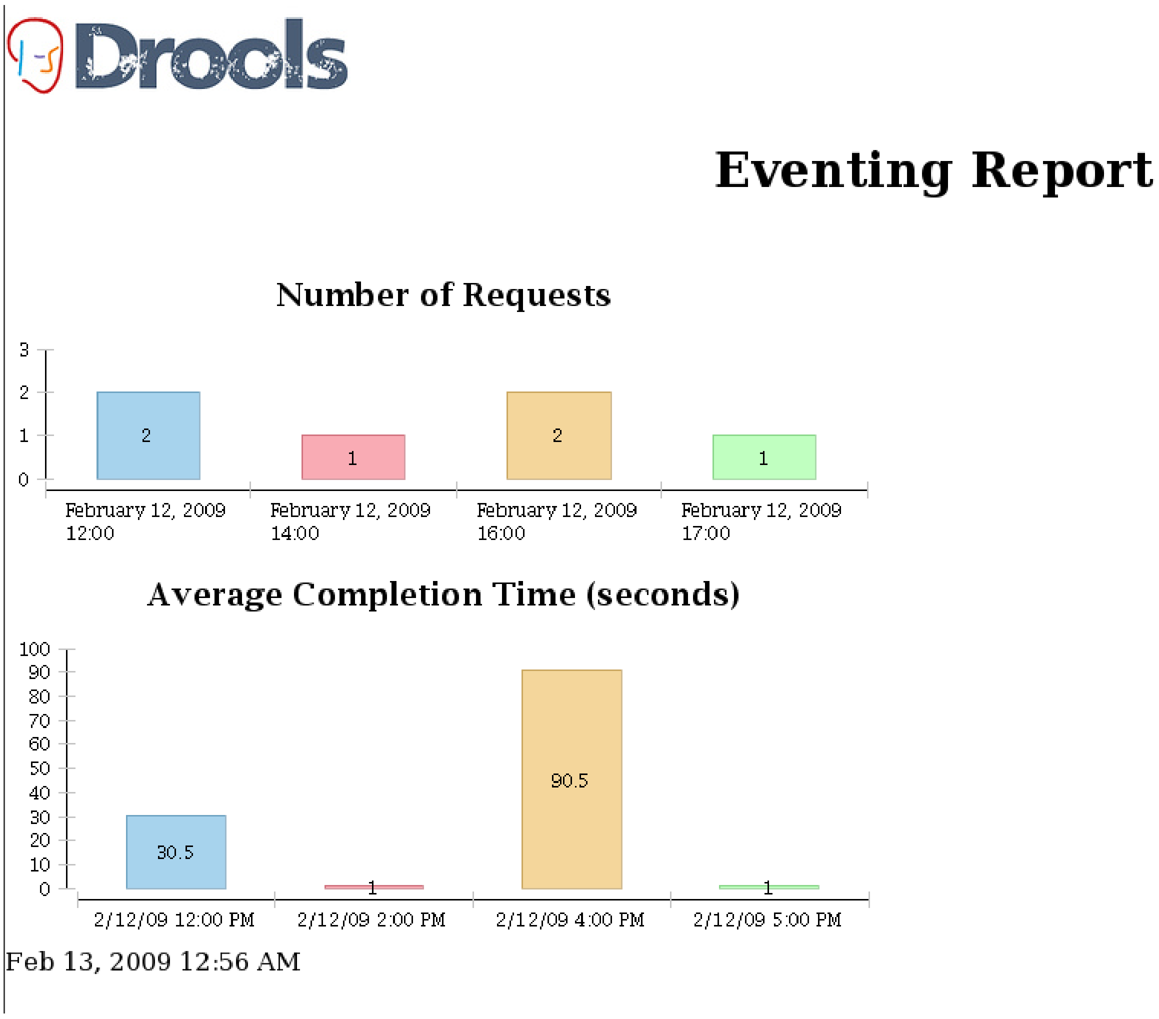


Figure 13.2. The eventing report

13.2. 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.

JBoss Rules 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.

Business Process Model and Notation (BPMN 2.0)

The Business Process Model and Notation (BPMN) 2.0 specification is steadily moving forward on its way to become a great standard, and we are adopting it for our process modeling in JBoss Rules Flow. BPMN 2.0 not only defines a standard on how to graphically represent a business process (like BPMN 1.1), but now also includes execution semantics for the elements defined, and an XML format on how to store (and share) process definitions.

JBoss Rules Flow allows you to execute processes defined using the BPMN 2.0 XML format, just the same way as it allows you to execute processes using the custom RuleFlow format. That means that you can use the same API, engine and components like Guvnor and the gwt-console, to execute and manage your BPMN 2.0 processes.

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

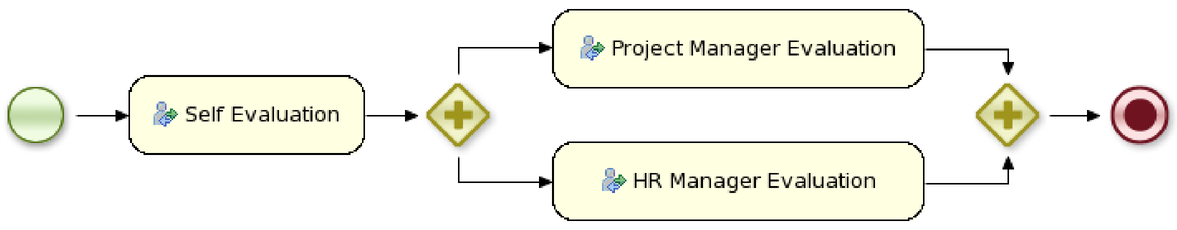
Data

- Java type language
- Process properties
- Embedded Sub-Process properties
- Activity properties

Connecting Objects

- Sequence flow

For example, consider the following BPMN process for performing evaluations. Whenever an evaluation process is started for a specific employee, that employee must first perform a self-evaluation, after which the project manager and human resource manager must also fill in their evaluation, as shown in the figure below.



An executable version of this process expressed using BPMN 2.0 XML would look something like this (note that the process needs to contain all the details to make it execuble, including all the parameters for each of the tasks present, hence the large process definition):

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

```
 <process processType="Private" isExecutable="true" id="com.sample.evaluation"
name="Evaluation Process" >


   <property id="employee" itemSubjectRef="_employeeItem"/>


   <startEvent id="_1" name="StartProcess" g:x="16" g:y="56" g:width="48" g:height="48" />
   <userTask id="_2" name="Self Evaluation" g:x="96" g:y="56" g:width="143" g:height="48" >
     <ioSpecification>
       <dataInput id="_2_CommentInput" name="Comment" />
       <dataInput id="_2_SkippableInput" name="Skippable" />
       <dataInput id="_2_TaskNameInput" name="TaskName" />
       <dataInput id="_2_ContentInput" name="Content" />
       <dataInput id="_2_PriorityInput" name="Priority" />
       <inputSet>
         <dataInputRefs>_2_CommentInput</dataInputRefs>
         <dataInputRefs>_2_SkippableInput</dataInputRefs>
         <dataInputRefs>_2_TaskNameInput</dataInputRefs>
         <dataInputRefs>_2_ContentInput</dataInputRefs>
         <dataInputRefs>_2_PriorityInput</dataInputRefs>
       </inputSet>
       <outputSet>
       </outputSet>
     </ioSpecification>
     <dataInputAssociation>
       <targetRef>_2_CommentInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression">You need to perform a self-evaluation</from>
         <to xs:type="tFormalExpression">_2_CommentInput</to>
       </assignment>
     </dataInputAssociation>
     <dataInputAssociation>
       <targetRef>_2_SkippableInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression">false</from>
         <to xs:type="tFormalExpression">_2_SkippableInput</to>
       </assignment>
     </dataInputAssociation>
     <dataInputAssociation>
       <targetRef>_2_TaskNameInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression">Performance Evaluation</from>
         <to xs:type="tFormalExpression">_2_TaskNameInput</to>
       </assignment>
     </dataInputAssociation>
     <dataInputAssociation>
       <targetRef>_2_ContentInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression"></from>
         <to xs:type="tFormalExpression">_2_ContentInput</to>
       </assignment>
     </dataInputAssociation>
     <dataInputAssociation>
       <targetRef>_2_PriorityInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression">1</from>
         <to xs:type="tFormalExpression">_2_PriorityInput</to>
       </assignment>
     </dataInputAssociation>
     <potentialOwner>
       <resourceAssignmentExpression>
         <formalExpression>#{employee}</formalExpression>
       </resourceAssignmentExpression>
     </potentialOwner>
   </userTask>
   <parallelGateway id="_3" name="Diverge" g:x="271" g:y="56" g:width="49" g:height="49"
gatewayDirection="Diverging" />
   <userTask id="_4" name="HR Manager Evaluation" g:x="352" g:y="96" g:width="225"
g:height="48" >
     <ioSpecification>
       <dataInput id="_4_CommentInput" name="Comment" />
       <dataInput id="_4_SkippableInput" name="Skippable" />
       <dataInput id="_4_TaskNameInput" name="TaskName" />
       <dataInput id="_4_ContentInput" name="Content" />
       <dataInput id="_4_PriorityInput" name="Priority" />
       <inputSet>
         <dataInputRefs>_4_CommentInput</dataInputRefs>
         <dataInputRefs>_4_SkippableInput</dataInputRefs>
         <dataInputRefs>_4_TaskNameInput</dataInputRefs>
         <dataInputRefs>_4_ContentInput</dataInputRefs>
         <dataInputRefs>_4_PriorityInput</dataInputRefs>
       </inputSet>
       <outputSet>
       </outputSet>
     </ioSpecification>
     <dataInputAssociation>
       <targetRef>_4_CommentInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression">You need to perform an evaluation for
#{employee}</from>
         <to xs:type="tFormalExpression">_4_CommentInput</to>
       </assignment>
     </dataInputAssociation>
     <dataInputAssociation>
       <targetRef>_4_SkippableInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression">false</from>
         <to xs:type="tFormalExpression">_4_SkippableInput</to>
       </assignment>
     </dataInputAssociation>
     <dataInputAssociation>
       <targetRef>_4_TaskNameInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression">Performance Evaluation</from>
         <to xs:type="tFormalExpression">_4_TaskNameInput</to>
       </assignment>
     </dataInputAssociation>
     <dataInputAssociation>
       <targetRef>_4_ContentInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression"></from>
         <to xs:type="tFormalExpression">_4_ContentInput</to>
       </assignment>
     </dataInputAssociation>
     <dataInputAssociation>
       <targetRef>_4_PriorityInput</targetRef>
       <assignment>
         <from xs:type="tFormalExpression">1</from>
         <to xs:type="tFormalExpression">_4_PriorityInput</to>
       </assignment>
     </dataInputAssociation>
     <potentialOwner>
       <resourceAssignmentExpression>
         <formalExpression>mary</formalExpression>
       </resourceAssignmentExpression>
     </potentialOwner>
   </userTask>
   <userTask id="_5" name="Project Manager Evaluation" g:x="352" g:y="16" g:width="225"
g:height="48" >
     <ioSpecification>
       <dataInput id="_5_CommentInput" name="Comment" />
       <dataInput id="_5_SkippableInput" name="Skippable" />
```

```
        <dataInput id="_5_TaskNameInput" name="TaskName" />
        <dataInput id="_5_ContentInput" name="Content" />
        <dataInput id="_5_PriorityInput" name="Priority" />
        <inputSet>
          <dataInputRefs>_5_CommentInput</dataInputRefs>
          <dataInputRefs>_5_SkippableInput</dataInputRefs>
          <dataInputRefs>_5_TaskNameInput</dataInputRefs>
          <dataInputRefs>_5_ContentInput</dataInputRefs>
          <dataInputRefs>_5_PriorityInput</dataInputRefs>
        </inputSet>
        <outputSet>
        </outputSet>
      </ioSpecification>
      <dataInputAssociation>
        <targetRef>_5_CommentInput</targetRef>
        <assignment>
          <from xs:type="tFormalExpression">You need to perform an evaluation for
#{employee}</from>
          <to xs:type="tFormalExpression">_5_CommentInput</to>
        </assignment>
      </dataInputAssociation>
      <dataInputAssociation>
        <targetRef>_5_SkippableInput</targetRef>
        <assignment>
          <from xs:type="tFormalExpression">false</from>
          <to xs:type="tFormalExpression">_5_SkippableInput</to>
        </assignment>
      </dataInputAssociation>
      <dataInputAssociation>
        <targetRef>_5_TaskNameInput</targetRef>
        <assignment>
          <from xs:type="tFormalExpression">Performance Evaluation</from>
          <to xs:type="tFormalExpression">_5_TaskNameInput</to>
        </assignment>
      </dataInputAssociation>
      <dataInputAssociation>
        <targetRef>_5_ContentInput</targetRef>
        <assignment>
          <from xs:type="tFormalExpression"></from>
          <to xs:type="tFormalExpression">_5_ContentInput</to>
        </assignment>
      </dataInputAssociation>
      <dataInputAssociation>
        <targetRef>_5_PriorityInput</targetRef>
        <assignment>
          <from xs:type="tFormalExpression">1</from>
          <to xs:type="tFormalExpression">_5_PriorityInput</to>
        </assignment>
      </dataInputAssociation>
      <potentialOwner>
        <resourceAssignmentExpression>
          <formalExpression>john</formalExpression>
        </resourceAssignmentExpression>
      </potentialOwner>
    </userTask>
    <parallelGateway id="_6" name="Converge" g:x="603" g:y="55" g:width="49" g:height="49"
gatewayDirection="Converging" />
    <endEvent id="_7" name="EndProcess" g:x="690" g:y="56" g:width="48" g:height="48" >
      <terminateEventDefinition/>
    </endEvent>


    <sequenceFlow id="_1-_2" sourceRef="_1" targetRef="_2" />
    <sequenceFlow id="_2-_3" sourceRef="_2" targetRef="_3" />
    <sequenceFlow id="_3-_4" sourceRef="_3" targetRef="_4" g:bendpoints="[295,120]" />
    <sequenceFlow id="_3-_5" sourceRef="_3" targetRef="_5" g:bendpoints="[295,39]" />
    <sequenceFlow id="_5-_6" sourceRef="_5" targetRef="_6" g:bendpoints="[627,40]" />
    <sequenceFlow id="_4-_6" sourceRef="_4" targetRef="_6" g:bendpoints="[627,121]" />
    <sequenceFlow id="_6-_7" sourceRef="_6" targetRef="_7" />

  </process>

</definitions>
```

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

- Create a new Flow file using the JBoss Rules Eclipse plugin wizard and in the last page of the wizard, make sure you select JBoss Rules 5.1 code compatibility. This will create a new process using the BPMN XML format instead of the old RuleFlow format. Note however that this is not a real BPMN 2.0 editor, as it still uses different attributes. It does however save the process using valid BPMN 2.0 syntax. Also note that the editor does not yet 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 so they can be executed. Be aware however that Oryx is still using the BPMN 2.0 beta 1 format and that their implementation is currently incomplete (especially the import / export functionality).
- 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 BPMN process into your knowledge base ...

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

... and how to execute this process.

```
KnowledgeBase kbase = readKnowledgeBase();
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
KnowledgeRuntimeLogger logger = KnowledgeRuntimeLoggerFactory.newFileLogger(ksession,
 "test");
ksession.getWorkItemManager().registerWorkItemHandler("Human Task", new
 WSHumanTaskHandler());
// start a new process instance
Map<String, Object> params = new HashMap<String, Object>();
params.put("employee", "krisv");
ksession.startProcess("com.sample.evaluation", params);
```

14.1. Current limitations

Since the BPMN 2.0 specification is still being finalized, the BPMN 2.0 execution is still an experimental feature. It uses the same execution engine and constructs as the RuleFlow format however (it's just another XML serialization format). Therefore, all features and components that are available using the RuleFlow format also work for BPMN 2.0 processes. You simply have to use the right ResourceType when adding BPMN 2.0 processes to your knowledge base. Since the specification hasn't been finalized yet, it is possible that the XSD that defines the format might still change (slightly) due to updates of the specification, so keep this in mind if you decide to start using the BPMN 2.0 format.

The use of a specification should give you a lot of advantages, as it allows you to share your processes across tools and possibly even engines as the specification defines the exact format (and even execution semantics) for each of the elements. At this point however, it is likely that different tools are using different intermediate versions of the specification. We believe that this issue will automatically resolve itself over time once the specification is finalized and everyone is using the same version of the specification, but until then, you can encounter compatibility issues related to this problem. Please be a little patient with this.

Finally, the BPMN 2.0 specification defines a lot of node types and attributes, but nevertheless it is not possible to express everything using the constructs offered by the BPMN 2.0 specification only. However, the specification is designed to allow additional node types, attributes, etc. While we try to limit the use of custom extensions to a minimum, we sometimes have to define custom attributes to express features that we believe are important but cannot be expressed as core BPMN 2.0 syntax. The following table gives an overview of which features of the RuleFlow language have already been ported to the BPMN 2.0 XML format. A green check mark means that the functionality can be expressed using the features defined in the BPMN 2.0 specification. In those cases where we extend the BPMN 2.0 specification with additional attributes and/or elements, we show these using an orange check mark. As shown in the table below, most of the basic BPMN 2.0 nodes are already supported. We decided to not yet implement some of the features that cannot be expressed in BPMN 2.0 by default, like for example the on-entry / on-exit actions or the state node. We will decide later whether we want to support these features for BPMN 2.0 processes in the future, and how.

Table 14.1. Keywords

Feature	JBoss Rules BPMN	JBoss Rules Flow
A. Process-level		
Imports		✓
Function Imports		✓
Variable	✓	✓
- primitive Java types	✓	✓
- Java object types	✓	✓
- default value		✓
Swimlanes	✓	✓
Exception handlers		✓
- fault name		✓
- bind to variable		✓
- action		✓
B. Nodes		
1. Start Node	✓	✓
- rule trigger	✓	✓

Feature	JBoss Rules BPMN	JBoss Rules Flow
- signal trigger	✓	✓
- parameter mapping	✓	✓
2. End Node	✓	✓
- terminate	✓	✓
3. Action Node	✓	✓
- Java dialect	✓	✓
* access to variables, global, context	✓	✓
- MVEL dialect	✓	✓
* access to variables, global, context	✓	✓
4. RuleSet Node	✓	✓
- timers		✓
5. Split Node	✓	✓
- AND	✓	✓
- XOR	✓	✓
- OR	✓	✓
- Java code constraints	✓	✓
- MVEL code constraints	✓	✓
- rule constraints	✓	✓
- constraint names	✓	✓
- constraint priorities		✓
6. Join Node	✓	✓
AND	✓	✓
XOR	✓	✓
Discriminator		✓
n-of-m		✓

Feature	JBoss Rules BPMN	JBoss Rules Flow
7. State Node		✓
- timers		✓
- on entry actions		✓
- on exit actions		✓
- automatic transition constraints		✓
- manual transition signal		✓
8. SubProcess Node	✓	✓
- timers		✓
- on entry actions		✓
- on exit actions		✓
- wait for completion	✓	✓
- independant	✓	✓
- parameter mapping (in/out)	✓	✓
- dynamic process id	✓	✓
9. WorkItem Node	✓	✓
- parameters	✓	✓
- parameter mapping (in/out)	✓	✓
- timers		✓
- on entry actions		✓
- on exit actions		✓
- wait for completion		✓
10. Timer Node	✓	✓
- delay	✓	✓
- period		✓
11. Human Task Node (also see WorkItem Node)	✓	✓

Feature	JBoss Rules BPMN	JBoss Rules Flow
- swimlane	✓	✓
12. Composite Node	✓	✓
- timers		✓
- on entry actions		✓
- on exit actions		✓
- variables	✓	✓
- exception handlers		✓
- multiple entry points		✓
- multiple exit points		✓
13. ForEach Node	✓	✓
- bind to variable	✓	✓
- wait for completion		✓
- multiple entry points		✓
- multiple exit points		✓
14. Event Node	✓	✓
- bind to variable	✓	✓
- internal / external		✓
- event filters		✓
15. Fault Node	✓	✓
- fault name	✓	✓
- fault data	✓	✓
Graphical information (x, y, width, height)	✓	✓
C. Connections		
From, To	✓	✓
From type		✓
To type		✓

Feature	JBoss Rules BPMN	JBoss Rules Flow
Graphical information (bendpoints)	✓	✓

Console

JBoss Rules 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 JBoss Rules build system generates two wars for you that can 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. Download gwt-console-server-drools-{version}.war and gwt-console-drools-{version}.war and deploy them to your application server, {AS_HOME}/server/{configuration}/deploy (so for example, we are using jboss-4.2.3.GA/server/default/deploy).

15.1. Installation

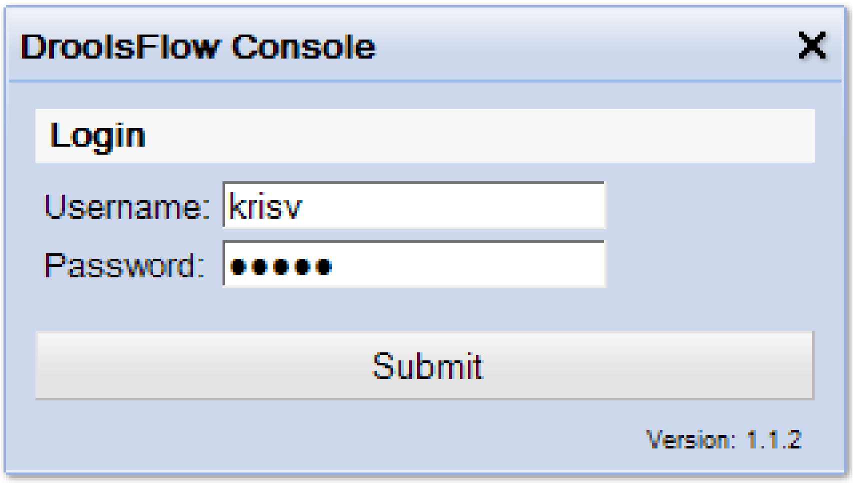
The easiest way to get started with the console is probably to use the installer. This will download, install and configure all the necessary components to get the console running, including Guvnor, a human task service, etc. Check out the chapter on the installer for more information. If you want to manually install the console, you can continue reading here.

You need to have an application server installed. This chapter assumes you are using the JBoss AS version 4.2.3.GA, but other versions or other application servers should be possible as well.

15.2. Running the process management console

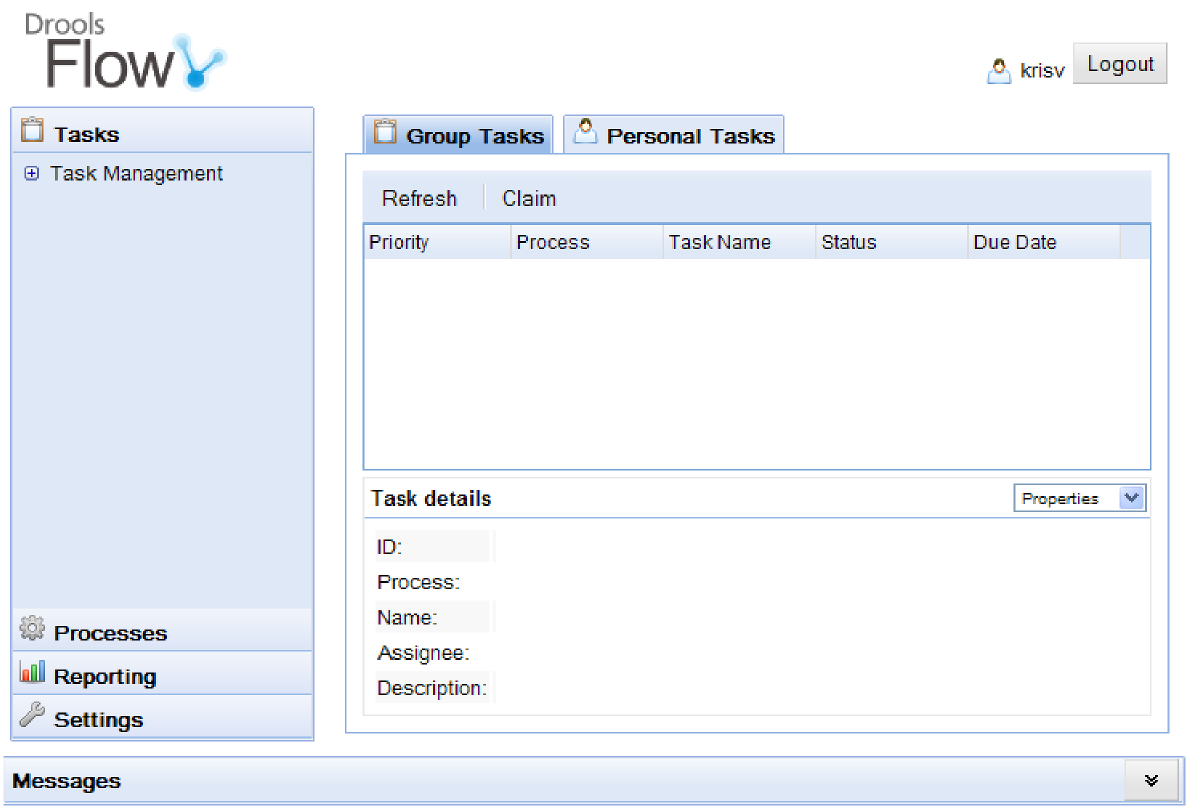
Now navigate to the following URL (replace the host and/or port depending on how the application server is configured): *http://localhost:8080/gwt-console*

A login screen should pop up, asking for your user name and password.



Username: krisv
Password: ●●●●●

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.



Priority	Process	Task Name	Status	Due Date

Task details

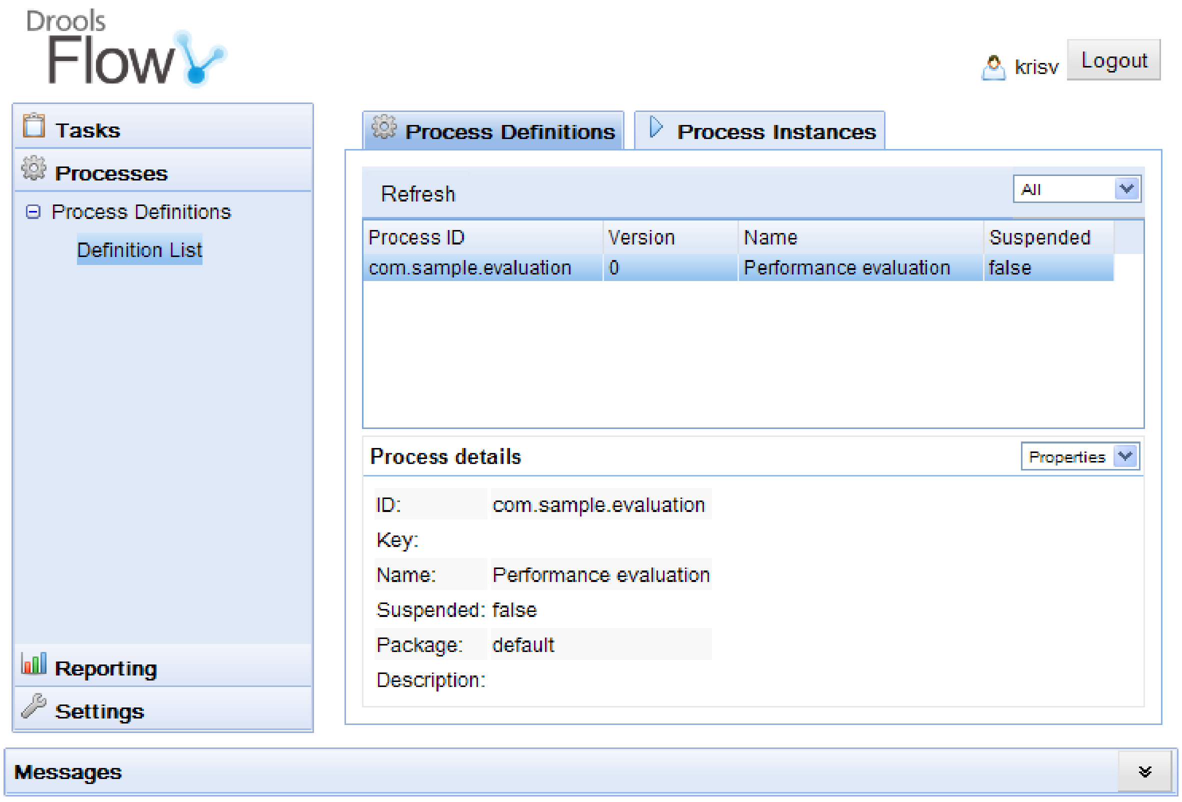
ID:
Process:
Name:
Assignee:
Description:

15.2.1. 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).

15.2.1.1. Inspecting process definitions

When you open the process definition list, all processes that are stored in the "default" package on Guvnor are shown. You can then either inspect process instances for one specific process or start a new process instance.

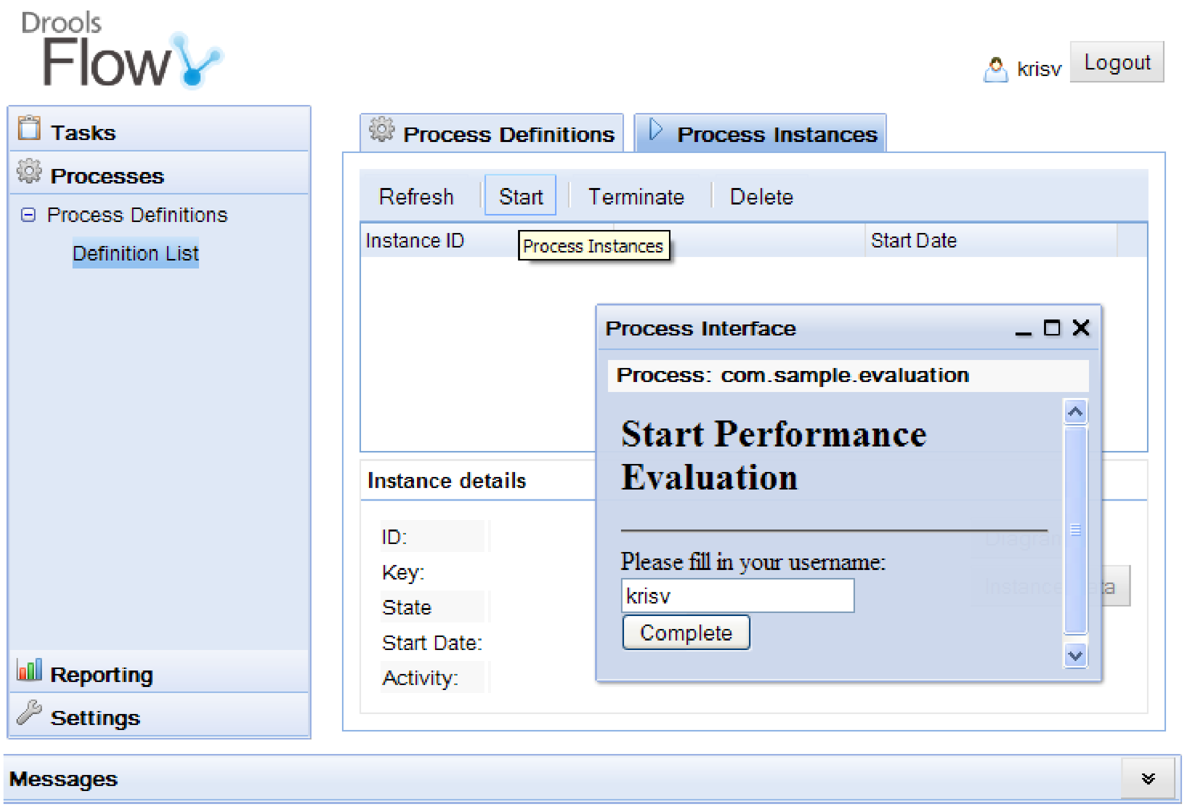


Process ID	Version	Name	Suspended
com.sample.evaluation	0	Performance evaluation	false

ID:	com.sample.evaluation
Key:	
Name:	Performance evaluation
Suspended:	false
Package:	default
Description:	

15.2.1.2. Starting new process instances

To start a new process instance for one specific process definition, select the process definition in the process definition list and select the process instances tab. Click on the "Start" button 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.



Process Interface

Process: com.sample.evaluation

Start Performance Evaluation

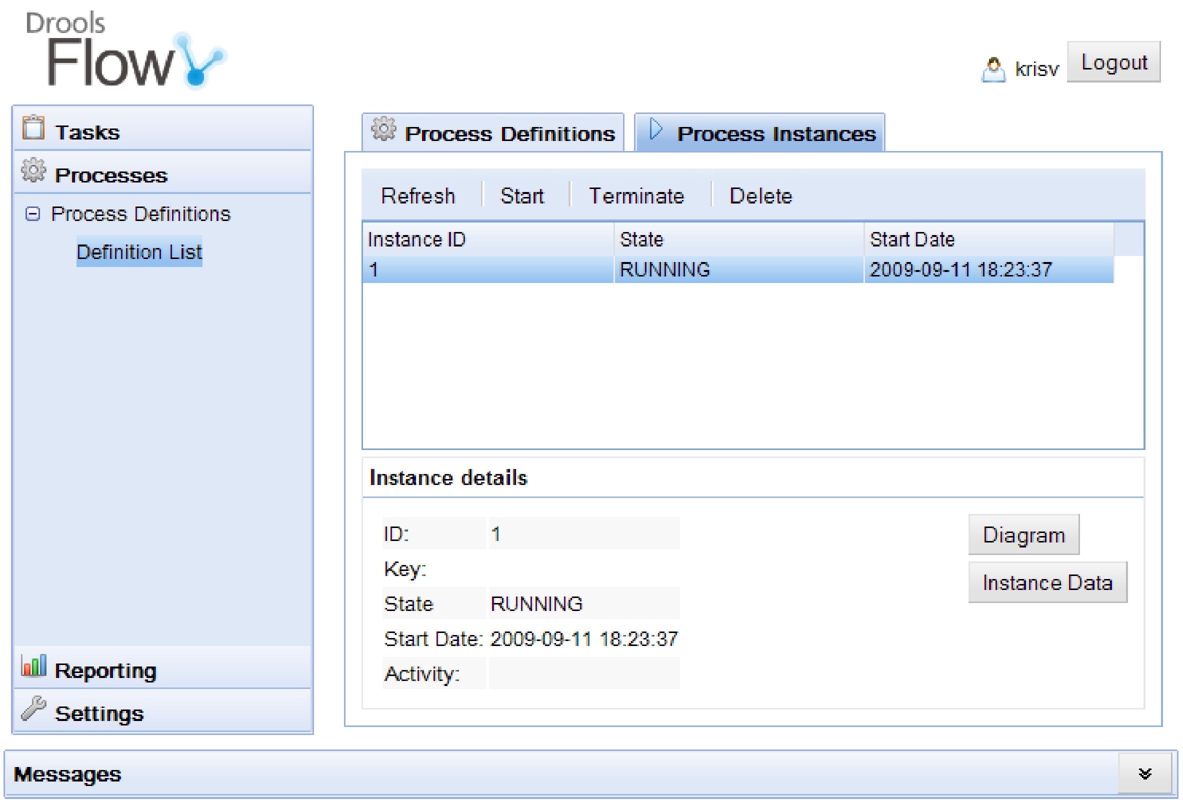
Please fill in your username:

krisv

Complete

15.2.1.3. Managing process instances

The process instances tab also contains a table showing all running instances of that specific process definition. Select a process instance to show the details of that specific process instance.

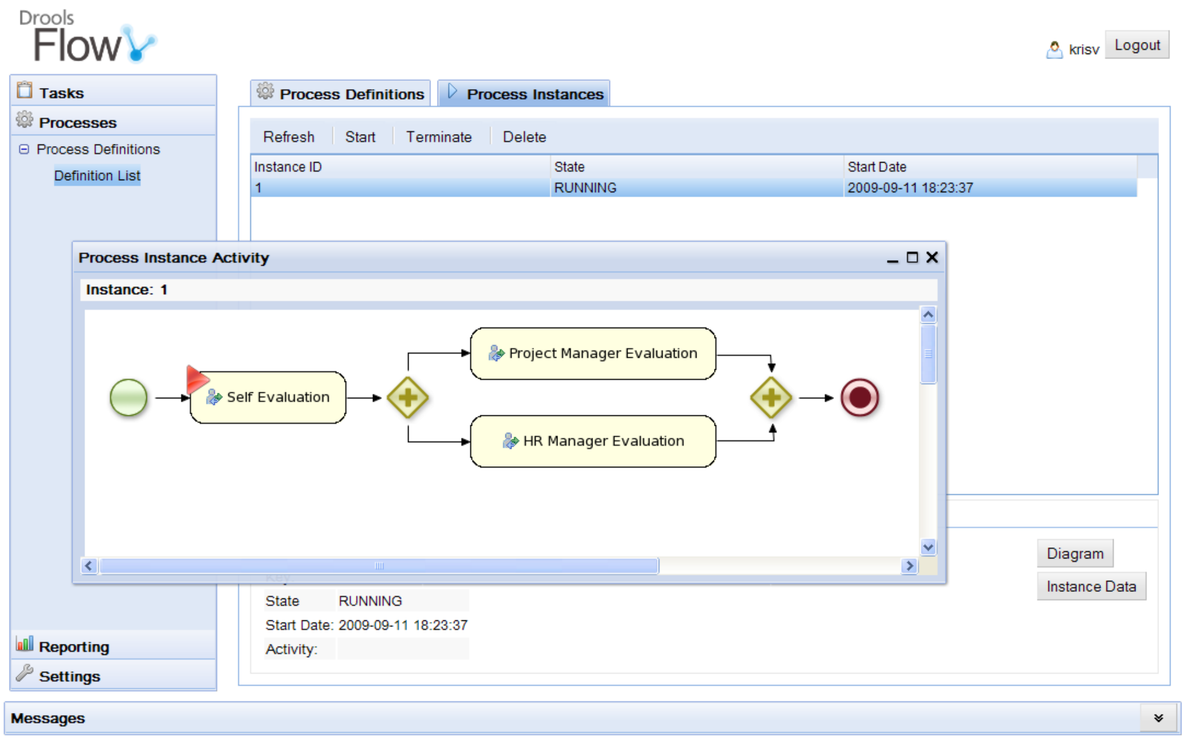


Instance ID	State	Start Date
1	RUNNING	2009-09-11 18:23:37

ID:	1
Key:	
State	RUNNING
Start Date:	2009-09-11 18:23:37
Activity:	

15.2.1.4. Inspecting process instance state

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.

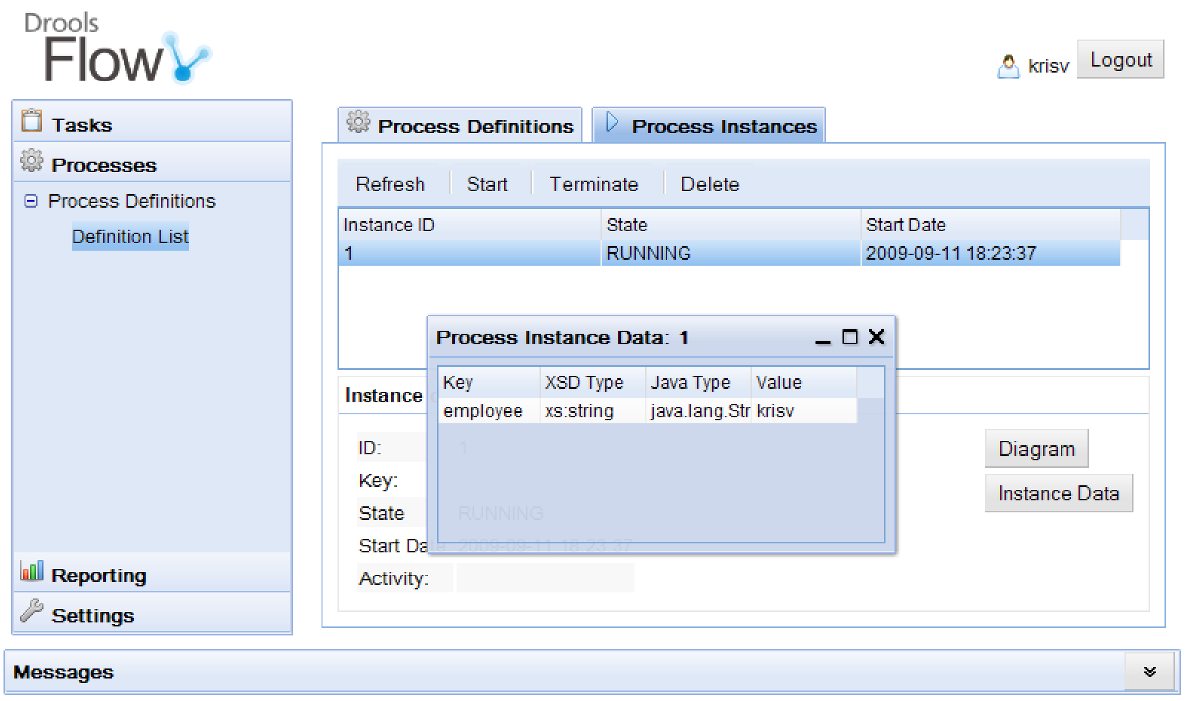


Instance ID	State	Start Date
1	RUNNING	2009-09-11 18:23:37

State	RUNNING
Start Date:	2009-09-11 18:23:37
Activity:	

15.2.1.5. Inspecting process instance variables

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.]



Instance ID	State	Start Date
1	RUNNING	2009-09-11 18:23:37

Process Instance Data: 1

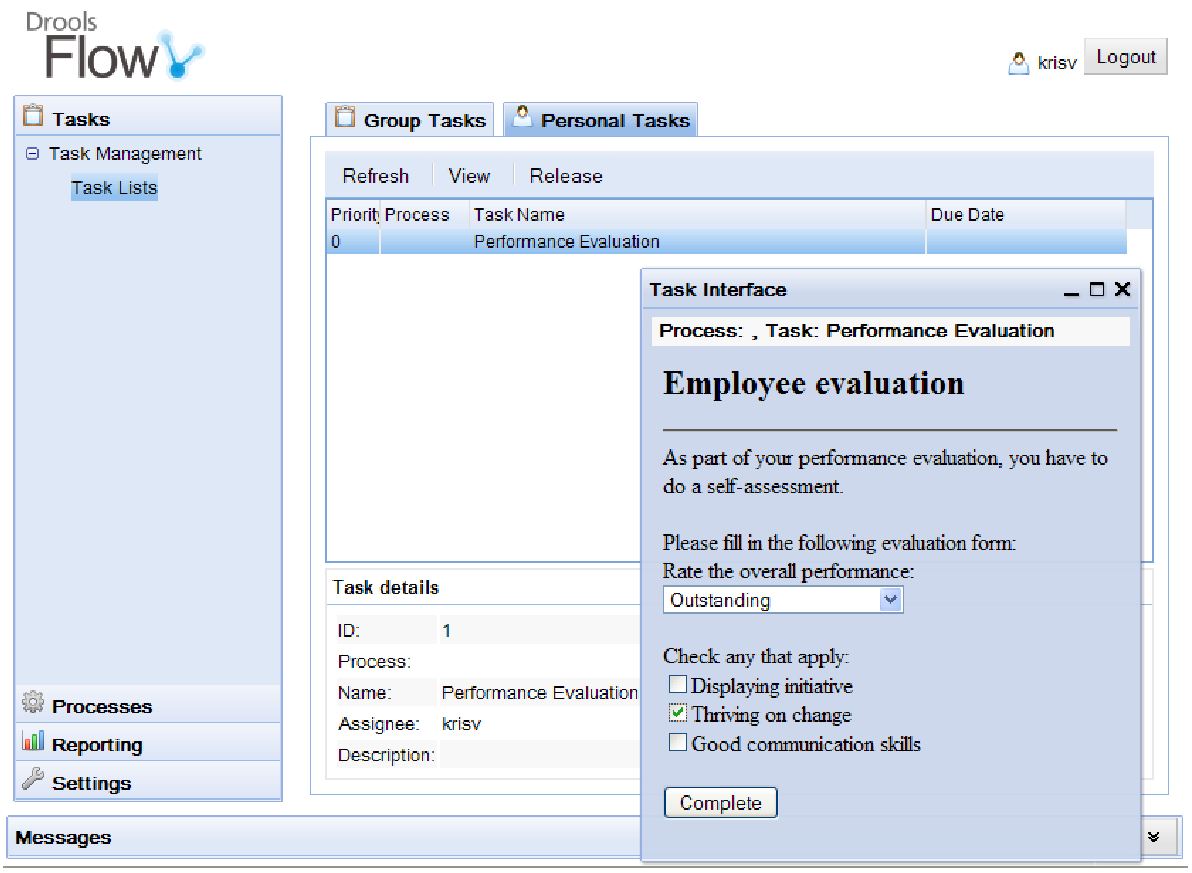
Key	XSD Type	Java Type	Value
employee	xs:string	java.lang.Str	krisv

Instance

ID: 1
Key:
State RUNNING
Start Date: 2009-09-11 18:23:37
Activity:

15.2.2. 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.



Priorit	Process	Task Name	Due Date
0		Performance Evaluation	

Task details

ID:	1
Process:	
Name:	Performance Evaluation
Assignee:	krisv
Description:	

Task Interface

Process: , Task: Performance Evaluation

Employee evaluation

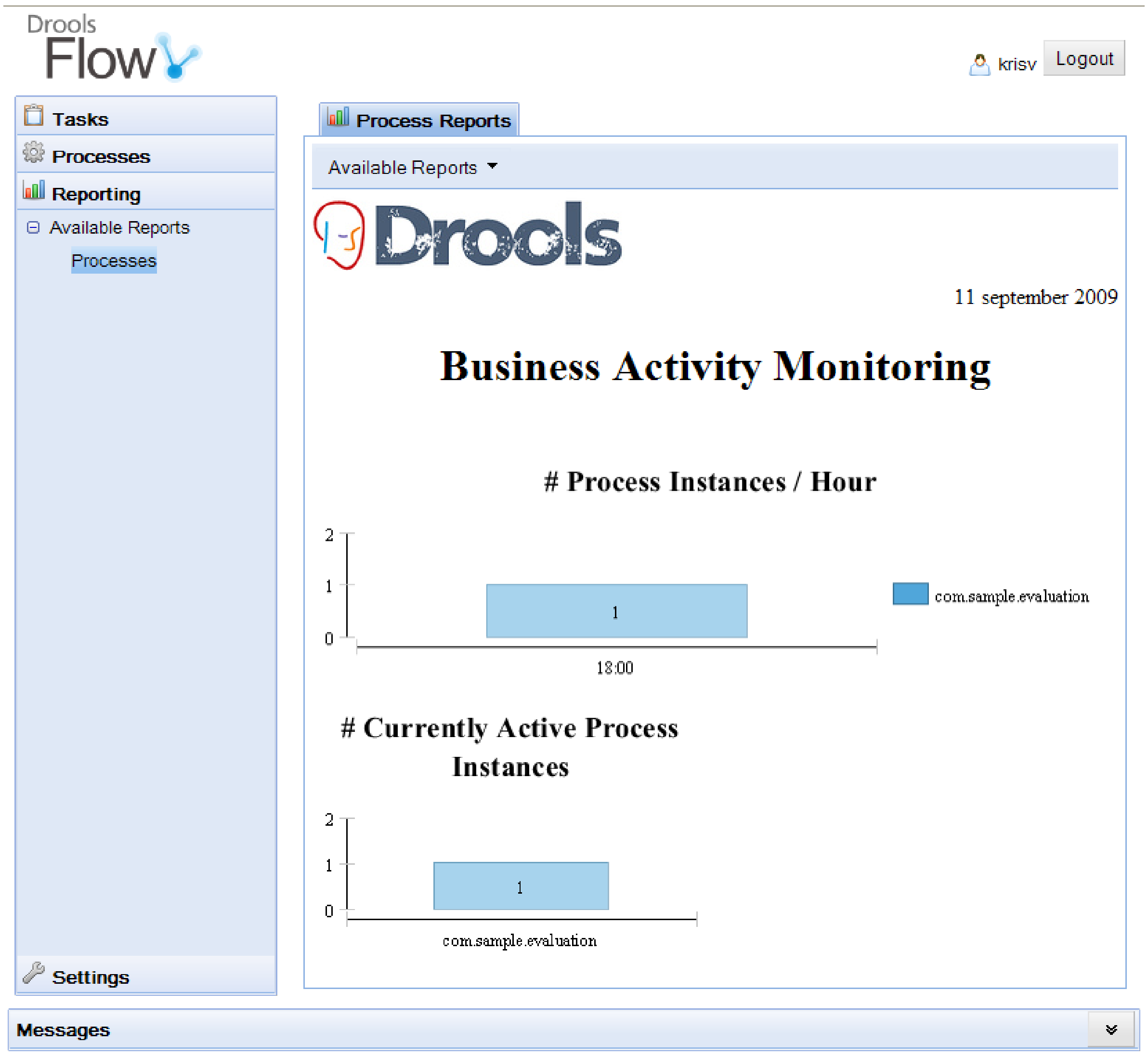
As part of your performance evaluation, you have to do a self-assessment.

Please fill in the following evaluation form:
Rate the overall performance: Outstanding

Check any that apply:
- [ ] Displaying initiative
- [x] Thriving on change
- [ ] Good communication skills

15.2.3. 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.

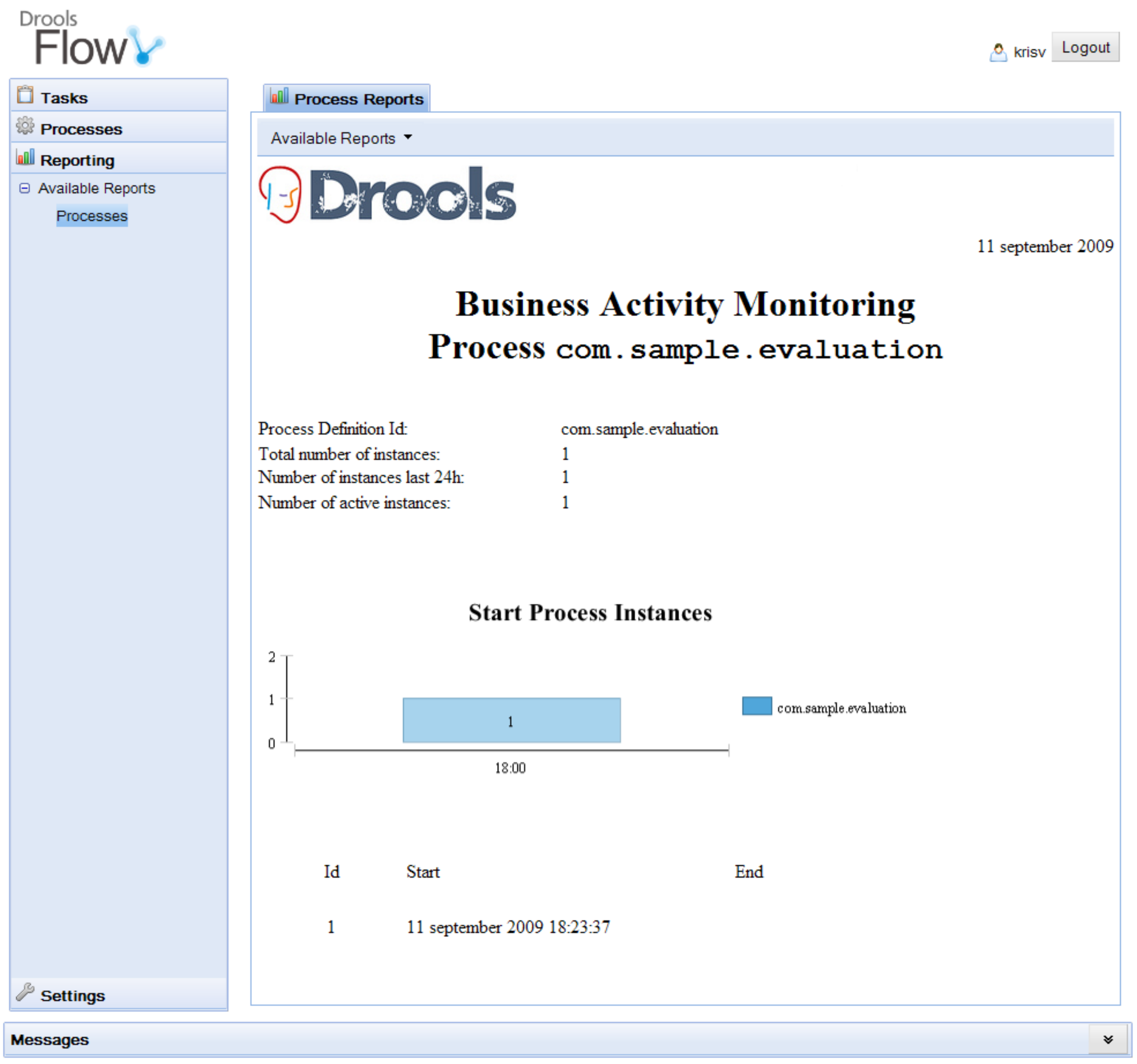


11 september 2009

Business Activity Monitoring

# Process Instances / Hour
2
1
0
1
18:00
com.sample.evaluation
# Currently Active Process Instances
2
1
0
1
com.sample.evaluation

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



11 september 2009

Business Activity Monitoring
Process com.sample.evaluation

Process Definition Id:	com.sample.evaluation
Total number of instances:	1
Number of instances last 24h:	1
Number of active instances:	1

Start Process Instances

Id	Start	End
1	11 september 2009 18:23:37	

JBoss Rules Flow 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.

15.3. 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 evalution 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 drools-gwt-form.jar in the server war.

Appendix A. Revision History

Revision 1.0 **Mon Nov 15 2010** **David Le Sage** *`dlesage@redhat.com`*

Initial creation of book by publican

Index