JBoss Enterprise SOA Platform 5.2

Rule Flow Component Guide

for Business Rules Developers

JBoss Enterprise SOA Platform 5.2 Rule Flow Component Guide for Business Rules Developers
Edition 1

Read this guide to learn how to use JBoss Enterprise SOA Platform's Rule Flow component to develop business rules and undertake service orchestration tasks.

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. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at *http://access.redhat.com*. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at *https://www.redhat.com/mailman/listinfo*. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. Give us Feedback

If you find a typographical error, or know how this guide can be improved, we would love to hear from you. Submit a report in Bugzilla against the product **`JBoss Enterprise SOA Platform`** and the component **`doc-Rule_Flow_Component_Guide`**. The following link will take you to a pre-filled bug report for this product: *http://bugzilla.redhat.com/*[2].

Fill out the following template in Bugzilla's **`Description`** field. Be as specific as possible when describing the issue; this will help ensure that we can fix it quickly.

```
Document URL:


Section Number and Name:


Describe the issue:


Suggestions for improvement:


Additional information:

```

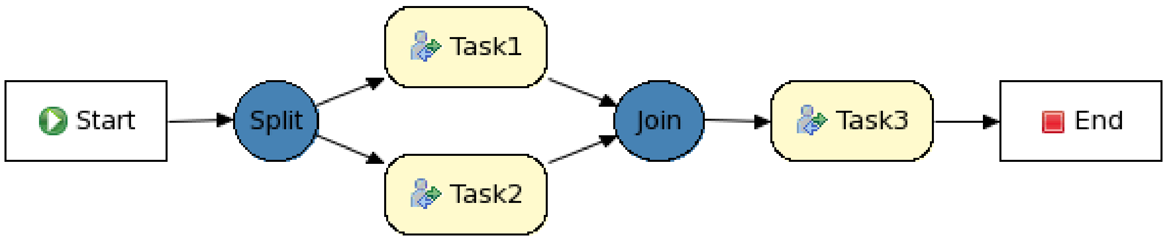
Be sure to give us your name so that you can receive full credit for reporting the issue.

[2] https://bugzilla.redhat.com/enter_bug.cgi?product=JJBoss%20Enterprise%20SOA%20Platform%205&component=doc-Rule_Flow_Component_Guide&version=52

Introduction

The **JBoss Enterprise SOA Platform**'s *Rule Flow* module is a work-flow or business process engine. It allows you to integrate your corporation's processes and rules in a sophisticated way.

A *business process* describes the order in which a series of steps must be executed. They are traditionally depicted using flow charts. For example, the following figure shows a process where, first of all, Task1 and Task2 must be executed in parallel. After they are completed, Task3 must be executed:



The following chapters will teach you everything you need to know about using the **JBoss Enterprise SOA Platform** Rule Flow Engine. These are some of its distinguishing features:

1. Advanced integration of processes and rules: processes and rules are usually considered to be 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, Rule Flow provides a high level of integration "out-of-the-box." This allows you to use rules to define parts of your company's business logic when you are documenting business processes and vice versa.

2. Unification of processes and rules: conceptually, rules, processes and event processing are all different types of knowledge. A unified API and tooling are provided so that you can combine these three types from the one tool. Unified knowledge repositories, audit logs, debugging and debugging features are also provided.

3. Declarative modelling: the Rule Flow Engine tries to keep processes as *declarative* as possible. In other words, it allows you to focus on what you want to happen without worrying about how.

   Because of this, you rarely have to hard-code details into your process. Instead, Ruleflow offers you the ability to describe your work in an abstract way (by, for instance, letting you use *pluggable work items* or a business scripting language).

   You can create domain-specific extensions which will make it simpler to read, update or create these processes as they are using domain-specific concepts that are closely related to the problem at hand.

4. Generic process engine supporting multiple process languages: Red Hat does not believe that there is one process language that fits all purposes. Therefore, the Rule Flow engine allows you to define and execute different types of process languages, including the native Rule Flow language, WS-BPEL (a standard targeted towards web service orchestration), OSWorkflow (another existing work-flow language), JPDL (the JBoss Business Process Definition Language) and so on.

Note

All of these languages are based on the same set of core "building blocks." This makes it easier for you to implement your own process language by reusing and recombining them should you wish to do so.

Using Rule Flow for the First Time

Read this section to learn how to create and execute your first Ruleflow process.

2.1. Creating Your First Process

Use the **JBoss Business Developer Studio** (JBDS) to create an executable project that contains the files necessary to start defining and executing processes.

Step through the **wizard** to generate a basic project structure, a class-path, a sample process and execution code. (To create a new JBoss Rules project, left-click on the **JBoss Rules action** button (with the JBoss Rules heading) in the IDE toolbar and select **New JBoss Rules Project**.

Note

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 select **File**, then **New** followed by **Project...**, and in the JBoss Rules directory, select **JBoss Rules Project**.

Give your project a name and click **Next**.

In the following dialogue box, you can select which elements you wish to add to your project by default. Since you are creating a new process, "untick" the first two check-boxes and select the last two. This will generate a sample process and a Java class to execute this process.

If you have not yet set up a *JBoss Rules run-time*, do so now. A JBoss Rules run-time is a collection of *Java Archive* files (JARs) that represent one specific release of the JBoss Rules project JARs.

To create a runtime, either point the IDE to the release of your choice, or create a new runtime on your file system from the JARs included in the JBoss Rules IDE plug-in. (Since you want to use the JBoss Rules version included in this plug-in, you will do the latter this time.)

Note

You will only have to do this once; the next time you create a JBoss Rules project, it will automatically use the default runtime (unless you specify otherwise).

Unless you have already set up a JBoss Rules run-time, click the **Next** button.

A dialogue box will appear, 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.

The dialogue box that will appear shows you the workspace settings for the JBoss Rules run-times. (The first time you do this, the list of installed JBoss Rules run-times will be empty.)

To create a new run-time on your file system, click the **Add...** button.

Use the dialogue box that appears to give the new run-time a name (such as "JBoss Rules 5.2 runtime"), and put a path to your JBoss Rules run-time on your file system.

Click the **Create a new JBoss Rules 5 runtime...** button and select the directory in which you want this run-time to be stored.

Click the **OK** button. You will see the path you selected showing up in the dialogue box.

Click the **OK** button. You will see the newly created run-time shown in your list of all the JBoss Rules run-times.

Select this runtime and make it the new default by clicking on the check box next to its name and clicking **OK**.

After successfully setting up your run-time, you can now dismiss the **Project Creation Wizard** by clicking on the **Finish** button.

The end result will contain the following:

1. **`ruleflow.rf`**: this is the process definition file. In this case, you have 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`**: this is the Java class that executes the process.

3. the libraries you require. These are automatically added to the project class-path in the form of a single JBoss Rules library.

Double-click on the **`ruleflow.rf`** file. The process will open in the Rule Flow Editor. (The Rule Flow 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.

On the left-hand side of the Editor window, you will see a **palette**. Use this to drag-and-drop new nodes. You will also find an outline view on the right-hand side.

Note

While most readers will find it easier to use the Editor, you can also modify the underlying XML directly if you wish. The XML for your sample process is shown below (note that the graphical information is omitted here for the sake of 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>
```

2.2. Executing Your First Process

To execute your process, right-click on **`RuleFlowTest.java`** and select **Run As...**, followed by **Java Application**.

When the process executes, the following output will appear in the Console window:

```
Hello World
```

Look at the code of **`RuleFlowTest`** class:

```
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();
  }

}
```

As you can see, the execution process is made up of a number of steps:

1. Firstly, a *knowledge base* is created. A knowledge base contains all the *knowledge* (such as words, processes, rules, and so forth) that are needed by your application. This knowledge base is usually created once, and then reused multiple times. In this case, the knowledge base consists of the sample process only.

2. Next, a session for interaction with the engine is generated.

   A *logger* is then added to the session. This records all execution events and make it easier for you to visualise what is happening.

3. Finally, you can start a new instance of the process by invoking the `startProcess(String processId)` method on the session. When you do this, your process instance begins to run, resulting in the executions of the Start node, the Action node, and the End node in order. When they finish the process instance will conclude.

Because you added a logger to the session, you can review what happened by looking at the *audit log*:

Select the **Audit View** tab on the bottom right of the window, (next to the **Console** tab.)

Click on the **Open Log** button (the first one on the right) and navigate to the newly created **`test.log`** file (in your **`project`** directory.)

Note

If you are not sure where this **`project`** directory is located, right-click on it and you will find the location listed in the **Resource** section

A tree view will appear. This shows the events that occurred at run-time. 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.

You can now start to experiment by designing your own process by modifying the example. To validate your processes, click on the **Check the rule-flow model** button (this is the green check box action in the upper tool-bar that appears when you are editing a process.) Processes are also automatically validated when you save them. You can see the debugging information in the **Error View**.

Rule Flows

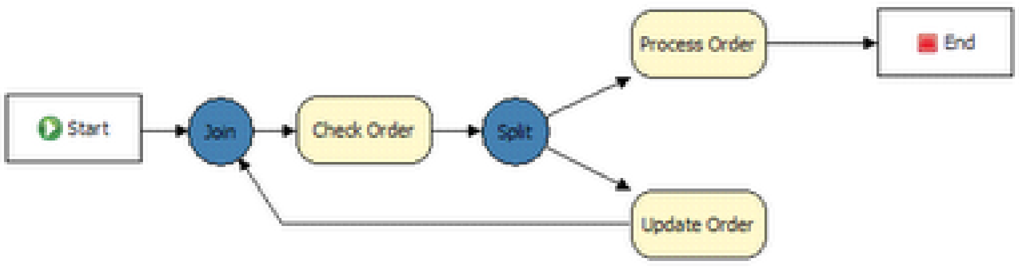


Figure 3.1. A Rule Flow

A *rule flow* is a flow chart that describes the order in which a series of steps need to be undertaken. It consists of a collection of *nodes* that are linked to each other by 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 supplied. Read the rest of this chapter to learn how to define and use rule flows in your application.

3.1. Creating a Rule Flow Process

Create rule flows in one of these three ways:

1. By using the graphical Rule Flow Editor (part of the JBDS' JBoss Rules plug-in.)
2. By writing 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.

3.1.1. Using the Graphical Rule Flow Editor

The Rule Flow Editor is a graphical tool that allows you to create a process by dragging and dropping different nodes onto a canvas. It then allows you to edit the properties of these nodes.

Once you have set up a JBoss Rules project in the JBDS, you can start adding processes: When in a project, launch the **New** wizard by using the Ctrl+N shortcut or by right-clicking on the directory in which you would like to put your rule flow and selecting **New**, then **Other...**.

Choose the section on **JBoss Rules** and then pick **Rule Flow file**. A new **`.rf`** file is created.

The Rule Flow Editor now appears.

Switch to the **JBoss Rules Perspective**. This will tweak the user interface so that it is optimal for rules. Then,

Next, ensure that you can see the **Properties View** (at the bottom of the JBDS window). If you cannot see the properties view, open it by going to the **Window** menu, clicking **Show View** and then **Other...**.

Next, under the **`General`** directory, select the **Properties View**.

The Rule Flow 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 and add it by clicking on your preferred location. For example, click on the **RuleFlowGroup** icon in the **Components** palette of the GUI and then draw a few rule flow groups.

Clicking on an element in your rule flow allows you to set its properties. 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.

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

Finally, check the process for any missing information (by pressing the green **Check** icon in the IDE menu bar) before using it in your application.

3.1.2. Defining Processes Using XML

You can also specify processes by writing the underlying XML by hand. The syntax of these XML processes is defined by a schema definition. For example, the following XML fragment shows a simple process made up 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 must contain only 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 are 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. Use these to define the various parameters and sub-elements for that node type.

3.1.3. Defining Processes Using the Process API

Warning

Red Hat does not recommend using the APIs directly. You should always use the Graphical Editor or hand-code XML. This section is only included for the sake of completeness.

It is possible to define a rule flow directly via the Process API. The most important process elements are defined in the **`org.drools.workflow.core`** and **`org.drools.workflow.core.node`** packages.

The `fluent` API allows you to construct processes in a readable manner using factories. At the end, you can validate the process that you were constructing manually.

3.1.3.1. Example One

This is a simple example of a basic process that has a rule set 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();
```

Note from the above that you start by calling the static `createProcess()` method from the **`RuleFlowProcessFactory`** class. This method creates a new process with the given id.

A typical process consists of three parts:

The header part is made up of global elements like the name of the process, imports, variables and so on.

The nodes section contains all the different nodes that make up the process.

The connections section finally links these nodes to each other to create a flow chart.

In the example above, the header contains the name and the version of the process. It also contains the package name. Following on from that, you can start adding nodes to the current process. If you are using auto-completion you can see that you different methods are available to you to create each of the supported node types at your disposal.

To start adding nodes to the process in this example, call the `startNode()`, `ruleSetNode()` and `endNode()` methods.

You will see that these methods return a specific **`NodeFactory`**, that allows you to set their properties.

Once you have finished configuring a specific node, call the `done()` method to return to the current **`RuleFlowProcessFactory`** so you can add more nodes if necessary.

When you have finished adding all the nodes, connect them by calling the `connection` method.

Finally, call the `validate()` method to check your work. This will also retrieve the **`RuleFlowProcess`** object you created.

3.1.3.2. Example Two

This example shows you how to use 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();
```

Note from the above that a Split node can have multiple outgoing connections, and a Join node multiple incoming connections.

3.1.3.3. Example Three

This more complex example demonstrates the use of a ForEach node and nested action 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();
```

Note how 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.

3.2. Using a Process in Your Application

There are two things you need to do to be able to execute processes from within your application:

Firstly, you need to create a knowledge base that contains the definition of the process,

Secondly you need to start the process by creating a session to communicate with the process engine.

1. *Creating a knowledge base*: once you have a valid process, you can add it to your knowledge base. Note that this process is almost identical to that for adding rules to the knowledge base: only the type of knowledge that is added is changed:

   ```
   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), create a new knowledge base:

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

   Warning

   This will throw an exception if the knowledge base contains errors (because it will not be able to parse your processes correctly).

2. *Starting a process*: processes are only executed if you explicitly state that they should be. 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 them. To activate a particular process, call the `startProcess` method:

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

The `startProcess` method's parameter represents the id. of the process 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**.

Important

If your process also needs to execute rules, you must also call the `ksession.fireAllRules()` method.

Note

You can specify additional parameters to pass input data to the process. To do so, use the `startProcess(String processId, Map parameters)` method. This method takes an additional set of parameters as name-value pairs and copies to the newly-created process instance as top-level variables.

Note

To start a process from within a rule consequence, or from inside a process action, use the predefined **`kcontext`** parameter:

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

3.3. Detailed Explanation of the Different Node Types

A rule-flow process is a flow chart that depicts different types of nodes which are linked by connections. The process itself exposes the following properties:

- *Id*: this is the process' unique id.
- *Name*: this is the process' unique display name.
- *Version*: this is the process' version number.
- *Package*: this is the paclage (or name-space) in which the process is stored.
- *Variables*: you can define variables to store data during the execution of your process.
- *Swimlanes*: these specify the *actor* responsible for the execution of *human tasks.*
- *Exception Handlers*: use these specify what is expected to happen when a fault occurs in the process.
- *Connection Layouts*: use these to specify what your connections are to look like on the canvas:
  - Manual always draws your connections as lines going straight from their start points to their end points (with the option 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.
- The Manhattan option draws connections using horizontal and vertical lines only.

You can use these types of nodes when creating a rule flow:

1. **Start Event**: this is the start of the rule flow. (A rule flow must have only one start node. It cannot have incoming connections but must have one outgoing connection.) Whenever a rule flow process is started, execution will commence at this node and automatically continue to the first node linked from it, and so on.

   The Start Event node possesses the following properties:

   - *Id*: this is the id. of the node (and is unique within one node container).
   - *Name*: this is the node's display name.
   - *Triggers*: you can specify triggers that, when activated, will automatically start the process. Examples are a constraint trigger that automatically launches the process if a given rule or constraint is satisfied, and an event trigger that automatically starts the process if a specific event is signalled.

     Note

     You cannot yet specify these triggers in the Graphical Editor. Edit the XML file instead to add them.
   - *MetaData*: this is meta-data related to this node.

2. **End Event**: this is the end of the rule flow. A rule flow must have at least one end node. The End node must have one incoming connection and cannot have any outgoing connections.

   This node possesses the following properties:

   - *Id*: this is the id. of the node (and is unique within one node container).
   - *Name*: this is the node's display name.
   - *Terminate*: an End node can be terminate the entire process (this is the default) or just one path. If the process is terminated, every active node (even those on parallel paths) in this rule flow is cancelled.

     Non-terminating End nodes end terminate the current path, while other parallel paths remain.
   - *MetaData*: this is meta-data related to this node.

3. **Rule Task (or RuleFlowGroup)**: use this node to represent a set of rules you wish to have evaluated. A RuleFlowGroup node should have one incoming connection and one outgoing connection.

   To make rules part of a specific rule flow group, use the ruleflow-group header attribute. When a RuleFlowGroup node is reached, the engine will start executing any rules that are part of the corresponding `ruleflow-group`. Execution will automatically continue to the next node once there are no more active rules in that group.

This means that you can add new activations (belonging to the currently active rule flow group) to the Agenda even if the facts have been modified by other rules.

Note

The rule flow will immediately process the next node if it encounters a rule flow group containing no active rules. If the rule flow group was already active, it will remain so and execution will only continue if every active rule has been run.

This contains the following properties:

- *Id*: this is the id. of the node (and is unique within one node container).
- *Name*: this is the node's display name.
- *RuleFlowGroup*: this is the name of the rule flow group that represents the set of rules for this node.
- *Timers*: these are any timers that are linked to this node.
- *MetaData*: this is meta-data related to this node.

4. **Diverging Gateway (or Split)**: use this to create branches in your rule flow. A Split node must have one incoming connection and two or more outgoing connections.

   There are three types of Split node:

   - **`AND`** means that the control flow will continue in all outgoing connections simultaneously.
   - **`XOR`** means that no more or less than one of the outgoing connections will be chosen. The decision is made by evaluating the constraints that are linked to each of the outgoing connections. Constraints are specified using the same syntax as the left-hand side of a rule. The constraint with the lowest priority number that evaluates to true is selected.

     Warning

     Make sure that at least one of the outgoing connections will evaluate to **`true`** at run time (the rule flow will throw an exception if there are none). For example, you could use a connection which is always true (default) with a high priority number to specify what should happen if none of the other connections can be taken.
   - **`OR`** means that all outgoing connections whose condition evaluates to **`true`** are selected. Conditions are similar to the **`XOR`** split, except that no priorities are taken into account.

Warning

Make sure that at least one of the outgoing connections will evaluate to **`true`** at run time (the rule flow will throw an exception if there are none). For example, you could use a connection which is always true (default) with a high priority number to specify what should happen if none of the other connections can be taken.

This node contains the following properties:

- *Id*: this is the id. of the node (and is unique within one node container).
- *Name*: this is the node's display name.
- *Type*: this is the node type (**`AND`**, **`XOR`** or **`OR`**.)
- *Constraints*: these are the constraints linked to each of the outgoing connections (in case of an **`(X)OR`** split).
- *MetaData*: this is meta-data related to this node.

5. **Converging Gateway (or Join)**: use this to synchronise multiple branches. A join node must have two or more incoming connections and one outgoing connection. Four types of split are available to you:

   - **`AND`** means that it will wait until all incoming branches are completed before continuing.
   - **`XOR`** means that it continues as soon as one of its incoming branches has been completed. (If it is triggered from more than one incoming connection, it will activate the next node for each of those triggers.)
   - **`Discriminator`** means that it will continue if one of its incoming branches has been completed. Other incoming branches are registered as they complete until all connections have finished At that point, the node will be reset, so that it can be triggered again when one of its incoming branches has been completed once more.
   - **`n-of-m`** means that it continues if **`n`** of its **`m`** incoming branches have been completed. The variable **`n`** could either be hard-coded to a fixed value, or refer to a process variable that will contain the number of incoming branches for which it must wait.

   This node contains the following properties:

   - *Id*: this is the id. of the node (and is unique within one node container).
   - *Name*: this is the node's display name.
   - *Type*: this is the node type (**`AND`**, **`XOR`** or **`OR`**.)
   - *n*: this is the number of incoming connections for which it must wait (in case of a **`n-of-m`** join).
   - *MetaData*: this is meta-data related to this node.

6. **State**: this node represents a *wait state*. A state must have one incoming connection and one or more outgoing connections.

For each of the outgoing connections, you can specify a rule constraint to define how long the process should wait before continuing. For example, a constraint in an order entry application might specify that the process should wait until no more errors are found in the given order.

To specify a constraint, use the same syntax as you would for the left-hand side of a rule.

When it reaches this node, the engine will check the associated constraints. If one of the constraint evaluates to **`true`** directly, the flow will continue immediately. Otherwise, the flow will continue if one of the constraints is satisfied later on, for example when a fact is inserted, updated or removed from the working memory.

Note

You can also signal a state manually to make it progress to the next state, using `ksession.signalEvent("signal", "name")` where name should either be the name of the constraint for the connection that should be selected, or the name of the node to which you wish to move.

A state contains the following properties:

- *Id*: this is the id. of the node (and is unique within one node container).
- *Name*: this is the node's display name.
- *Constraints*: use these to define when the process can leave this state and continue for each of the outgoing connections.
- *Timers*: these are any timers that are linked to this node.
- *On-entry and on-exit actions*: these are actions that are executed upon entry or exit of this node, respectively.
- *MetaData*: this is meta-data related to this node.

7. **Reusable Sub-Process (or SubFlow)**: this represents the invocation of another process from within the parent process. A sub-process node must have one incoming connection and one outgoing connection.

   When a SubFlow node is reached, the engine will start the process with the given id.

   This node contains the following properties:

   - *Id*: this is the id. of the node (and is unique within one node container).
   - *Name*: this is the node's display name.
   - *ProcessId*: this is the id. of the process that is to be executed.
   - *Wait for completion*: if you set this property to **`true`**, the SubFlow node will only continue if it has terminated its execution (by other completing or aborting it); otherwise it will continue immediately after having started the sub-process.
   - *Independent*: if you set this property to **`true`**, the sub-process will start as an independent process. This means that the SubFlow process will not terminate if this it reaches an end node; otherwise the active sub-process will be cancelled on termination (or abortion) of the process.

- *On-entry and on-exit actions*: these are actions that are executed upon entry or exit of this node, respectively.
- *Parameter in/out mapping*: you can also define sub-flow nodes by using *in-* and *out-mappings* for variables. The value of variables in this process will be used as parameters when starting the process. The value of the variables in the sub-process will be copied to the variables of this process when the sub-process has been completed.

  Note

  You can only use out mappings when Wait for completion is set to **`true`**.

- *Timers*: these are any timers that are linked to this node.
- *MetaData*: this is meta-data related to this node.

8. **Action (or Script Task)**: this node represents an action that should be executed in this rule flow. An action node should have one incoming connection and one outgoing connection. The associated action specifies what should be executed, the dialect used for coding the action (such as Java or MVEL), and the actual action code.

   This code can access any global, the predefined variable called **`drools`** referring to a **`KnowledgeHelper`** object (which can, for example, be used to retrieve the Working Memory by calling `drools.getWorkingMemory()`), and the variable **`kcontext`** that references the **`ProcessContext`** object. (This latter object can, for example, be used to access the current **`ProcessInstance`** or **`NodeInstance`**, and to obtain and set variables).

   When the rule flow reaches an Action node, it will execute the action and then continue to the next node.

   The Action node possesses the following properties:

   - *Id*: this is the id. of the node (and is unique within one node container).
   - *Name*: this is the node's display name.
   - *Action*: this is the action associated with the node.
   - *MetaData*: this is meta-data related to this node.

9. **Timer Event**: this node represents a timer that can trigger one or multiple times after a given period. A Timer node must have one incoming connection and one outgoing connection.

   The timer delay specifies how long (in milliseconds) the timer should wait before triggering the first time. The timer period specifies the time between two subsequent triggers. A period of **`0`** means that the timer should only be triggered once. When the rule flow reaches a Timer node, it starts the associated timer.

   The timer is cancelled if the timer node is cancelled (by, for instance, completing or aborting the process).

   The Timer node contains the following properties:

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

- *Name*: this is the node's display name.
- *Timer delay*: this is the delay (in milliseconds) that the node should wait before triggering the first time.
- *Timer period*: this is the period (in milliseconds) between two subsequent triggers. If the period is **`0`**, the timer should only be triggered once.
- *MetaData*: this is meta-data related to this node.

10. **Error Event (or Fault)**: use a Fault node to signal an exceptional condition in the process. It must have one incoming connection and no outgoing connections.

    When the rule flow reaches a fault node, it will throw a fault with the given name. The process will search for an appropriate exception handler that is capable of handling this kind of fault. If no fault handler is found, the process instance is aborted.

    A Fault node contains the following properties:

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

11. **(Message) Event**: use this Event node to respond to internal or external events during the execution of the process. An Event node must have no incoming connections and one outgoing connection. It specifies the type of event that is expected. Whenever that type of event is detected, the node connected to this Event node is triggered.

    It contains the following properties:

    - *Id*: this is the id. of the node (and is unique within one node container).
    - *Name*: this is the node's display name.
    - *EventType*: this is the type of event that is expected.
    - *VariableName*: this is the name of the variable that will contain the data (if any) associated with this event.
    - *Scope*: you can use this node to listen to internal events only (that is, events that are signalled to this process instance directly, by using `processInstance.signalEvent(String type, Object data)`.)

      You can define it as external, by using `workingMemory.signalEvent(String type, Object event)`. In this case, it will also be listening to external events that are signalled to the process engine directly .
    - *MetaData*: this is meta-data related to this node.

12. **User Task (or Human Task)**: Processes can also involve tasks that need to be executed by humans. A Human Task node represents an atomic task to be executed by a human actor. It must have one incoming connection and one outgoing connection. Human Task nodes can be used in combination with *Swimlanes* to assign multiple human tasks to similar actors.

Note

A Human Task node is actually nothing more than a specific type of work item node (of type "Human Task").

A Human Task node contains the following properties:

- *Id*: this is the id. of the node (and is unique within one node container).
- *Name*: this is the node's display name.
- *TaskName*: this is the name of the human task.
- *Priority*: this is an integer indicating the priority of the human task.
- *Comment*: this is a comment associated with the human task.
- *ActorId*: this is the id. of the actor responsible for executing the human task. You can specify a list of them, using a comma to separate each one.
- *Skippable*: this specifies whether the human task can be skipped, (that is, whether the actor is allowed to decide if he or she should do the task or not).
- *Content*: this is the data associated with the task.
- *Swimlane*: this is the swimlane to which the node belongs. Swimlanes make it easy to assign multiple human tasks to the same actor.
- *Wait for completion*: If you set this property to **`true`**, the human task node will only continue if the human task has been terminated; otherwise it will continue immediately after creating the human task.
- *On.entry and on-exit actions*: these are the actions that are executed upon entry and exit of this node, respectively.
- *Parameter mapping*: use this to copy the value of process variables to human task parameters. Upon creation of the human tasks, the values are copied.
- *Result mapping*: this allows copying of the human task's result parameters value to a process variable. Upon completion of the human task, the values are copied.

Note

You can use result mappings only when Wait for completion is set to **`true`**. A human task has a result variable called `Result` that contains the data returned by the human actor.

The `ActorId` variable contains the id. of the actor who actually executed the task.

- *Timers*: these are the Timers that are linked to this node.
- *MetaData*: this is meta-data related to this node.

13. **Sub-Process (or Composite)**: A Composite node is a node that can contain other nodes so that it acts as a node container. This allows not only the embedding of a part of the flow within such a Composite node, but also the definition of additional variables and exception handlers that are accessible for all nodes inside this container. A Composite node should have one incoming connection and one outgoing connection. It contains the following properties:

    - *Id*: this is the id. of the node (and is unique within one node container).
    - *Name*: this is the node's display name.
    - *StartNodeId*: this is the id. of the node container node that should be triggered.
    - *EndNodeId*: this is the id. of the node container node that that represents the end of the flow. When this node is completed, the composite node will also complete and move to the outgoing connection. Every other node executing within this composite node will be cancelled.
    - *Variables*: you can add additional data storage variables.
    - *Exception Handlers*: use these to specify the behaviour to occur when a fault is encountered.

14. **Multiple Instance (or ForEach)**: a ForEach node is a special composite that allows you to execute the flow contained therein multiple times, once for each element in a collection. A ForEach node must have one incoming connection and one outgoing connection.

    A ForEach node awaits the completion of the embedded flow for each of the collection''s elements before continuing.

    This node contains the following properties:

    - *Id*: this is the id. of the node (and is unique within one node container).
    - *Name*: this is the node's display name.
    - *StartNodeId*: this is the id. of the node container node that should be triggered.
    - *EndNodeId*: this is the id. of the node container node that that represents the end of the flow. When this node is completed, the composite node will also complete and move to the outgoing connection. Every other node executing within this composite node will be cancelled.
    - *CollectionExpression*: this is the name of a variable that represents the collection of elements over which you will *iterate*. Set the collection variable to **`java.util.Collection`**.
    - *VariableName*: this is the name of the variable which contains the current element from the collection. This gives sub-nodes contained in the composite node access to the selected element.

15. **WorkItem (or Service Task)**: this node represents an abstract unit of work that is to be executed during this rule flow. You must represent all work that is executed outside the process engine (in a declarative way) using this type of node.

    Some different types of work are predefined. These include sending an e.-mail and logging a message.

You can define domain-specific work items. To do so, use a unique name and defining the input and output that you expect.

When the rule flow reaches WorkItem node, the associated work item is executed. A WorkItem node must have one incoming connection and one outgoing connection.

- *Id*: this is the id. of the node (and is unique within one node container).
- *Name*: this is the node's display name.
- *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*: use this to copy the process variables' values to the work item's parameters.
- *Result mapping*: use this to copy the result parameters to a process variable. Each type of work can have result parameters. These can be returned after the work item has been completed. Use a result mapping to copy the given result parameter's value to the process variable. For example, the "FileFinder" work item returns a list of files that match the given search criteria within the result parameter Files. You can then bind this list of files to a process variable for use within the rule flow. Upon completion of the work item, the values will be copied.

  Note

  You can only use result mappings when Wait for completion is set to **`true`**.

- *On-entry and on-exit actions*: these are actions that are executed upon entry or exit of this node, respectively.
- *Timers*: these are the timers that are linked to this node.
- *Additional parameters*: you can define additional parameters for each type of work. For example, the Email work item has thse additional parameters: From, To, Subject and Body. You can either provide values for these parameters directly, or define a parameter mapping that will copy the given variable's values to the appropriate parameter; if both are specified, the mapping will have precedence.

  To embed a value, use parameters of type String with ***`#{expression}`*** . The value will be retrieved when the work item is created and the substitution expression will be replaced by the result of calling `toString()` on the variable. The expression can simply be the name of a variable (in which case it resolves to the value of the variable), but you can use more advanced MVEL expressions as well, such as **`#{person.name.firstname}`**.
- *MetaData*: this is meta-data related to this node.

3.4. Data

While the rule flow is designed specifically to allow you to create process control flows, you also have to plan it from a data perspective. Throughout the execution of a process, data is retrieved, stored, passed on and used.

To store run-time data while a process is executing, use *variables*. A variable is defined by a name and a data type. This could be something very basic, such as Boolean, int, or String, or it could be any kind of Object sub-class.

Define variables inside a *variable scope*. The top-level scope is that for the process itself. Subscopes can be defined via a composite node. Variables that are defined in sub-scopes can only be accessed by nodes within that scope.

Whenever a variable is accessed, the process will search for the appropriate definitive variable scope.

You are allowed to nest variable scopes. 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.

You can use variables in these ways:

- you can set process-level variables when starting a process by providing a map of parameters to the invocation of the `startProcess` method. These parameters are then set as variables on the process scope.
- actions can access variables directly. They do so by using the name of the variable as a parameter name:

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

  You can change the value of a variable via the *knowledge context*:

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

- you can make WorkItem and SubFlow nodes pass the value of parameters to the "outside world" by mapping the variable to one of the work item parameters. To do so, either use a parameter mapping or interpolate it into a String parameter, using ***`#{expression}`*** . You can also copy a WorkItem's output to a variable via a result mapping.
- various other nodes can also access data. Event nodes, for example, can store the data associated with an event in a variable. Exception handlers can read error data from a specific variable. Check the properties of the different node types for more information.

Finally, every process and rule can access *globals*. These are globally-defined variables that are considered immutable with regard to rule evaluation and data in the knowledge session.

You can access the knowledge session via the actions in the knowledge context:

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

3.5. Constraints

You can use constraints in a multitude of locations in your rule flow. You can, for example use them in a Split node using **`OR`** or **`XOR`** decisions, or as a constraint for a State node. The Rules Flow Engine supports two types of constraints:

- *Code constraints* are Boolean expressions, evaluated directly immediately upon arrival. You can write them in either of these two dialects: Java and MVEL. Both have direct access to the globals and variables defined in the process.

  Here is an example of a constraint written in Java, **`person`** being a variable in the process:

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

  Here is the same constraint written in MVEL:

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

- *Rule constraints* are the same as normal **JBoss Rules** conditions. They use the **JBoss Rules** Rule Language's syntax to express what are potentially very complex constraints. These rules can, (like any other rule), refer to data in the working memory. You can also refer to globals directly.

  Here is an example of a valid rule constraint:

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

  This searches the working memory for people older than twenty.

Rule constraints do not have direct access to variables that have been defined inside the rule flow. You can, however, possible to refer to the current process instance inside a rule constraint, by adding the process instance to the working memory and matching it to the process instance in your rule constraint.

Red Hat has added special logic to make sure that a processInstance variable of the type WorkflowProcessInstance will only match the current process instance and not to other process instances in the working memory. You, however, are responsible for inserting the process instance into the session and, updating it, using, for example, either Java code or an on-entry or on-exit or explicit process action.

The following example of a rule constraint will search for a person with the same name as the value stored in the process variable name:

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

3.6. Actions

You can use actions in these ways:

- within an Action node,
- as entries or exits, (with a number of nodes),
- to specify the the behaviour of exception handlers.

Actions have access to globals and those variables that are defined for the process and the predefined **`context`** variable. This latter is of the type **`org.drools.runtime.process.ProcessContext`** and can be used for the following tasks:

- obtaining the current node instance. The node instance can be queried for such information as its name and type. You can also cancel it:

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

- obtaining the current process instance. A process instance can be queried for such information as its name and processId. It can also be aborted or signalled via an internal event:

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

- obtaining or setting the value of variables.
- accessing the knowledge run-time, in order to do things like start a process, signal external events or insert data.

Java actions should be valid Java code.

MVEL actions can use this business scripting language to express the action. MVEL accepts any valid Java code but also provides support for nested accesses of parameters (such as, **`person.name`** instead of **`person.getName()`**), and various other advantages. Thus, MVEL expressions are normally more convenient for the business user. For example, an action that prints out the name of the person in the rule flow's requester variable will: look like this:

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

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

3.7. Events

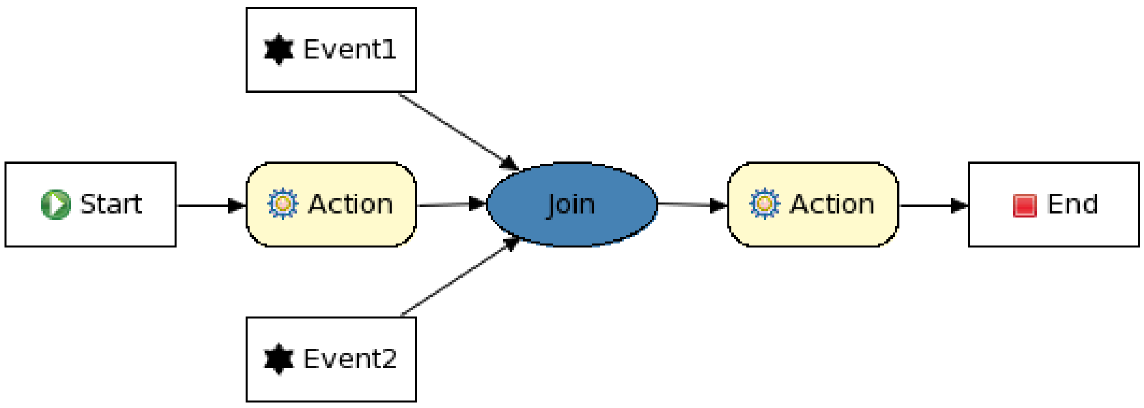


Figure 3.2. A sample process using events

When you execute a process, the Rule Flow Engine makes sure that all of the relevant tasks are executed according to the process plan. It does so by requesting the execution of work items and waiting for the results. However, you can also make the rule flow respond to events that were not directly requested by the Engine. By explicitly representing these events in a rule flow, you allow yourself to specify how the process should react to them.

Each events has an associated type. It may also have associated data. You can define your own event types and their associated data.

To specify how a rule flow is to respond to events, use 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.

You can signal an event to a running instance of a process in these ways:

- via internal events: to make an action inside a rule flow signal the occurrence of an internal event, using code like this:

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

- via external event: to notify a process instance of an external event use code like this:

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

- via external event using event correlation: instead of notifying a process instance directly, you can make the Rule Flow Engine automatically determine which process instances might be interested in an event using *event correlation*. This is based on the event type. Use this code to make a process instance that contains an event node listening for a particular external event will be notified whenever such an event occurs:

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

You can also use events to start a rule flow. Whenever a Start node defines an event trigger of a specific type, a new rule flow instance will launch.

3.8. Exceptions

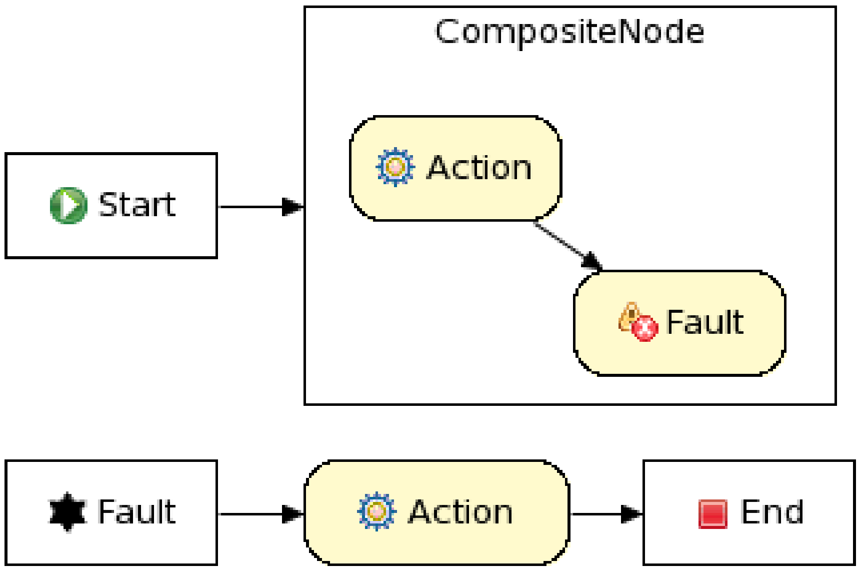


Figure 3.3. A sample process using exception handlers

If an exceptional condition occurs during the execution of a rule flow, a fault will be raised. The rule flow will then search for an appropriate exception handler that is capable of handling this type of fault.

As with events, each fault has an associated type. They may also have associated data. You can define both your own types and your own data.

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 the exception handler to match.

Rule flows and Composite nodes can both define exception handlers.

You can nest exception handlers; 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 abort, resulting in the cancellation of all nodes inside the process.

You can also use exception handlers to specify a fault variable. In this case, any data associated with the fault will be copied to this variable. This allows subsequent Action nodes in the rule flow to access the fault data and take appropriate action based on it.

Exception handlers need to be told how to respond to a given fault. In most cases, the behaviour required of them cannot be expressed in a single action. Red Hat therefore recommends that you have the exception handler signal an event of a specific type (in this case "Fault") by using this code:

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

3.9. Timers

Use timers to set a time delay for a trigger. You can use them to specify supervision periods, to trigger certain logic after a certain period, or to repeat some action at regular intervals.

You must configure a timer node so that it has both a *delay* and a *period*. The delay specifies the how long (in milliseconds) to wait after node activation before triggering the timer for the first time. The period defines the duration of time between subsequent trigger activations. If you set the period to **`0`**, the timer will only run once.

The timer service is responsible for making sure that timers are triggered at the correct moment. You can also cancel timers. This means that they will no longer be triggered.

You can use timers in these ways:

- you can add a Timer node to the rule flow. When the node is activated, it starts the timer, and its triggers (once or repeatedly) activate the Timer node's successor. This means that the timer's outgoing connection is triggered multiple times if you set the period. Cancelling a Timer node also cancels the associated timer, after which nothing will be triggered anymore.
- you can associate timers with event-based nodes like WorkItem, SubFlow and so forth. 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 regular notifications to alert that the execution of tasks is taking too long to perform, or to signal a fault if a supervision period expires.

  When the node owning the timer completes, the timer is automatically cancelled.

3.10. Updating Rule Flows

Over time, your business processes are likely to evolve as you refine them or due to changing requirements. You cannot actually update a rule flow to mirror this but you can deploy a new version of it. The old process will still exist because existing process instances might still need the old one's definition. Because of this, you have to give the new process different id., but you can use the same name and version parameter.

Whenever a rule flow is updated, it is important that you determine what is to happen to the already process instances that are already running. Here are your options:

- *Proceed*: you allow the running process instance to proceed as normal, using the definition as it was defined when the instance was started. In other words, the already-running instance will proceed as if the rule flow has not been updated. Only when you start new instances, will the updated version be used.
- *Abort (and restart)*: you abort the running instance. If necessary, restart it so that it will use the new version of the rule flow.
- *Transfer*: you migrate the process instance to the new process definition, meaning that it will continue executing based on the updated rule flow logic.

By default, the Rule Flow Engine uses the "proceed" approach.

3.10.1. Rule Flow Instance Migration

A rule flow instance contains all the run-time information needed to continue execution at some later point in time. This includes all of the data linked to this process instance (stored in variables), and also the current state of the process diagram. For each active node, a node instance represents this.

A node instances also contain an 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 rule flow instance only contains the run-time state. It is only indirectly linked to a particular rule flow (via an id. reference) that represents the logic that it needs to follow when running. As a result, to update a running process instance to a newer version of the new rule flow, you simply have to update the linked process id.

However, this does not take into account fact that you might need to migrate the state of the rule flow instance as well. In cases where the process is only extended and all existing wait states are kept, this is relatively straightforward, as the run-time state does not need to change at all. However, at other times a more sophisticated mapping may be needed. For example, when you remove an existing wait state, or split into multiple wait states, you cannot update the existing rule flow instance. Likewise, when a new process variable is introduced, you might need to initialise that variable correctly prior to using it in the remainder of the process.

To handle this, you can use the **`WorkflowProcessInstanceUpgrader`** to upgrade a rule flow process instance to a newer one. To use this tool, you will need to provide the process instance and the new process' id. By default, the Rules Flow Engine will automatically map old node instances to new ones 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, separated by a colon). These ids allow you to uniquely identify a node when composites are used (as a node id. is only unique within its node container.)

Here is an 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 generate your own custom mappers for specific situations. To do so, follow these instructions:

Firstly, disconnect the process instance.

Next, change the state accordingly.

Finally, reconnect the process instance.

3.11. Assigning Rules to a Rule Flow Group

When you are dealing with many large rule sets, managing the order in which rules are evaluated can become complex. Rule Flow allows you to specify the order in which rule sets are to be evaluated. It does so by providing you with a flow chart. Use this chart to define which rule sets should be evaluated in sequence and which in parallel, and to specify conditions under which rule sets should be evaluated. Read this section to learn more about this functionality and to see some examples.

A rule flow can handle conditional branching, parallelism, and synchronisation.

To use a rule flow to describe the order in which rules should be evaluated, follow these steps:

First sort your rules into groups using the ruleflow-group rule attribute (**options** in the GUI).

Next, create a rule flow graph (which is a flow chart) that graphically orders the sequence in which the ruleflow-group should be evaluated.Here is an example:

```
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 rule consequence's rule flow context. Through this context, you can access the rule flow or node instance that triggered the ruleflow-group. You can also set or retrieve variables:

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

3.12. Example Rule Flows

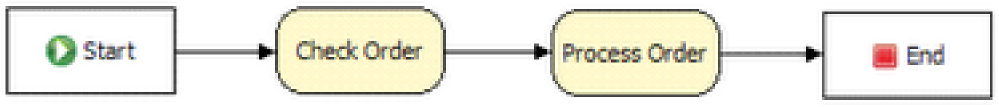
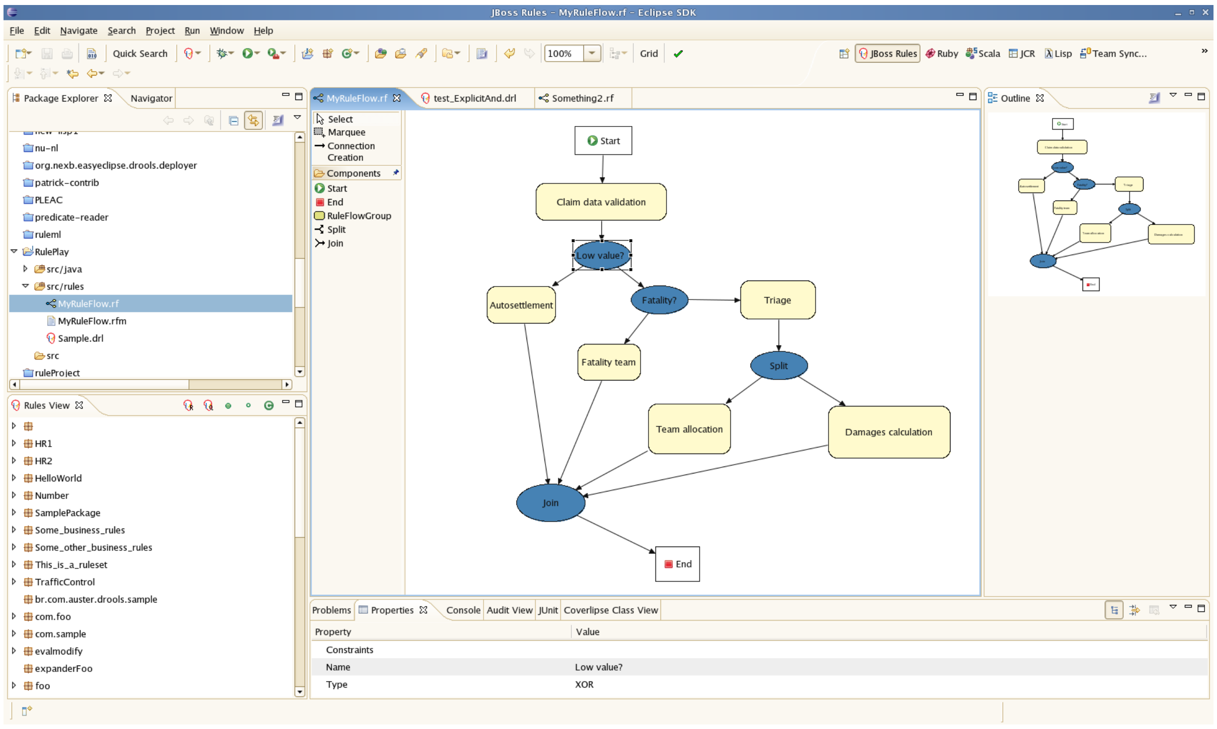


Figure 3.4. A Simple Rule Flow

The rule flow above specifies that the rules in the **`Check Order`** group must be executed before the rules in the **`Process Order`** group. You could achieve similar results using salience, but this is harder to maintain and makes a time relationship implicit in the rules (or Agenda groups.) By contrast, using a rule-flow makes the processing order explicit, in its own layer on top of the rule structure, allowing you to manage complex business processes more easily.

In practice, if you are using rule-flow, you are most likely doing more than just setting a linear sequence of groups to progress though. You will be using Split and Join nodes to model branches and define flows by connections, from the Start to ruleflow-groups, to Splits and then on to more groups, Joins, and so on. Do all of via a graphical editor:



Property	Value
Constraints	
Name	Low value?
Type	XOR

Figure 3.5. A Complex Rule Flow

The rule flow depicted above represents a more complex business process for finalising an insurance claim:

First of all, the claim data validation rules are processed. These perform data integrity checks for consistency and completeness.

Next, in a Split node, a conditional decision is made based on the value of the 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" set of fatality-specific rules should take effect, with more processing to follow.

Based on a few conditions, many different control flows are possible.

Note

All the rules can be in one package, with the control flow definition being stored separately.

To edit Split nodes, follow this process:

Firstly, click on the node.

From the **properties panel** that appears, choose the type: **`AND`**, **`OR`** or **`XOR`**. If you choose **`OR`**, then any of the split's potential outputs will be allowed to occur, meaning that processing can proceed in parallel along two or more different paths. If you chose **`XOR`**, then only one path will be taken.

If you choose **`OR`** or **`XOR`**, there will be a square button on the right-hand side of the **Constraints** row.

Click on this button to open the **Constraint Editor**. This is a text editor with which you add *constraints* (which are like the conditional part of a rule.)

Note

These constraints operate on facts in the working memory. In the example above, 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.

Set the conditions that will decide which outgoing path to follow.

The API

Use the API for these two tasks: to create a knowledge base containing your rule flow definitions and to create a session.

4.1. Knowledge Base

The knowledge-based API allows you to create a single knowledge base that contains all the knowledge your rule flows need. You can be reuse it across sessions.

The knowledge base includes all your rule flow definitions (and other "knowledge types" such as example rules).

This code shows you how to create a knowledge base consisting of only one process definition, using a knowledge builder to add the resource (which comes from the class-path in this case):

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

Note

The knowledge-based API allows you to add different types of resources, such as processes and rules, in almost identical ways, to the same knowledge base. This enables a user who knows how to use the Rule Flow engine to start using **JBoss Rules Fusion** almost immediately, and even to integrate these different types of knowledge.

4.2. Session

Next, you must create a session to interact with the Engine. The following code shows you how to do this, and how to start a process (via its id.):

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

The `ProcessRuntime` interface defines all of the session 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();
```

4.3. Events

Both the *stateful* and *stateless knowledge sessions* provide methods that allow you to register and remove listeners. Use **`ProcessEventListener`** objects to listen to process-related events (like starting or completing a process or entering or leaving a node.) Here are the different methods for it:

```
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 );

}
```

You can create an audit log based on the information provided by these process listeners. Red Hat provides you with the following ones out-of-the-box:

1. Console logger: this outputs every event to the console.
2. File logger: this outputs every event to an XML file. This log file might then be used in the IDE to generate a tree-based visualisation 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 pre-defined threshold, it cannot be used when debugging processes at run-time. The threaded file logger writes the events to a file after a specified time interval, making it possible to use the logger to visualise progress in real-time, making it useful for debugging.

Use the **`KnowledgeRuntimeLoggerFactory`** to add a logger to your session:

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

Note

When creating a console logger, pass the knowledge session for it as an argument.

The file logger must also be supplied requires the name of the log file to be created.

The threaded file logger requires the interval (in milliseconds) after which the events are to be saved.

You can open the log file in JBDS. To do so, go to the **Audit View**. Here you will see the events depicted in the form of a tree. (Anything that occurs between the **`before`** and **`after`** events is shown as a child of that event.)

Persistence

The Rule Flow Engine lets you store certain information (such as the rule flow run-time state and definitions) persistently.

5.1. Run-Time State

Whenever you start a rule flow, a *process instance* is created. This process instance represents the execution of the rule flow in that specific context. For example, when executing a rule flow that specifies how to process a sales order, one instance is created for each sales request.

Important

The instance contains the minimal run-time information that it needs to resume process instance at some later time. It does not include information about the history of that instance if that information is no longer needed.

You can make the run-time state of an executing process *persistent*. This allows you 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.

You can "plug" different persistence tools into the Rule Flow Engine to achieve this. By default, though, persistence functionality is switched off.

5.1.1. Binary Persistence

A *binary persistence mechanism* allows you to save the state of a process instance as a binary data set. Note that these binary data sets are usually relatively small, as they only contain the minimal amount of data needed to resume execution. For a simple process instance, this usually consists of a few node instances, (that is, any nodes that are currently executing and, possibly, some variable values.)

5.1.2. Safe Points

The state of a process instance is stored at so-called *safe points* during execution. When no more actions can be performed, 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.

5.1.3. Configuring Persistence

To activate the persistence functionality, you must firstly supply a configuration file and some dependencies.

Note

The persistence feature is based on the `Java Persistence` API (`JPA`) and can thus work with several persistence mechanisms.

Red Hat supplies you with **Hibernate**, but feel free to employ an alternative. An **H2** database is used to store the data, but you might want to choose your own alternative to this, as well.

First, add the necessary dependencies to your class-path. If you are using JBDS, add the **`JAR`** files to your JBoss Rules run-time directory or by manually adding these dependencies to your project. The first one you must add is **`drools-persistence-jpa.jar`**, as this file contains the code for saving the run-time state whenever necessary. The other other dependencies you supply will vary depending on the persistence solution and database you have chosen. For the default combination of **Hibernate** and **H2**, these are the dependencies you need:

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 Rule Flow Engine to save its state 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 use the **`JPAKnowledgeService`** to recreate a session based on a specific id:

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

By default, **`drools-persistence-jpa.jar`** contains a configuration file, called **`persistence.xml`**, that configures JPA to use **Hibernate** and the **H2** database. This file, found in the **`META-INF`** directory, is shown below:

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

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

You will need to override these defaults if you want to change them, by adding your own **`persistence.xml`** file to your class-path, preceding the default one in **`drools-persistence-jpa.jar`**.

This configuration file refers to a data source called **`jdbc/processInstanceDS`**. Use the following Java fragment to configure this source if you are using the **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();
```

5.1.4. Transaction Boundaries

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

You need to register a transaction manager before using user-defined transactions. (The following sample code uses the **Bitronix** transaction manager.)

Next, use the `Java Transaction` API to specify the boundaries:

```
// 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();
```

5.2. Process Definitions

Process definition files are written in an XML format. Stored them on a file system during development. However, whenever you want to make your knowledge accessible to one or more Rule Flow Engines in production, Red Hat recommends using a knowledge repository that (logically) centralises your knowledge.

The BRMS provides exactly that. It consists of a repository for storing different kinds of knowledge (including rules and object models). It allows you to retrieve this knowledge via WebDAV or by through a *knowledge agent*. and provides a web application that allows your corporate users to view and possibly update the information in the knowledge repository.

5.3. History Log

You may find it convenient to store information about rule flow instance executions, so that you can later verify that they have worked, or analyse their efficiency.

It is not a good idea to store history information in the run-time database, as this will rapidly grow, and monitoring and analysis queries might influence the performance of your Rule Flow Engine. That is why history information is stored in a separate location.

This history log records the events generated by the run-time engine during execution. The engine provides a generic mechanism that is able to listen to different kinds of events. You can use filters to ensure so that only the information in which you are interested is stored.

5.3.1. Storing Process Events in a Database

The **`drools-bam`** module contains an event listener that stores process-related information in a database. The database contains two tables, one for storing process instance information and one for node instance information.

1. *ProcessInstanceLog:* this lists the process instance id., the process id., the start date and, if applicable, the end date for every 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 via one of its incoming connections or is exited through one of its outgoing connections, that information is stored in this table.

To log process history information in a database like this, you need to register the logger:

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

// invoke methods one your session here

logger.dispose();
```

Note

This logger is like any other so you can add one or more filters by calling the `addFilter` method to ensure that only relevant information is stored in the database.

Important

You should dispose the logger when it is no longer needed.

To change the database in which the information is to be stored, modify the **`hibernate.cfg.xml`** file.

You can run various queries to analyse the history log. The **`ProcessInstanceDbLog`** (found in the **`org.drools.process.audit`** package) provides some examples but you can create your own as well.

Rules and Processes

Read this chapter to learn more about integrating rules and processes.

6.1. Approach

Use work flow languages to create flow charts that describe the order in which activities are to be performed.

These charts are very good at providing overall control flow. However, processes can become very complex if you need to use them to define complex business decisions, or to handle a lot of exceptional situations or need them to handle external events.

On the other hand, rules are very good at describing complex decisions and can "reason over" large amounts of data and events. It is, however, very difficult to create large flow charts using rules.

In the past, users were forced to choose between one or the other. Problems that required complex reasoning about large amounts of data required you to use 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 can combine both processes and rules to write their business logic in the format that best suits their needs.

A rules and process engine will determine the next step it needs to undertake by analysing the knowledge base and the current state of the application, thereby integrating rules and processes.

6.1.1. Teaching a Rules Engine About Processes

To "teach" a rules engine about processes, information about the current state of the processes is sent to the working memory.

6.2. Example

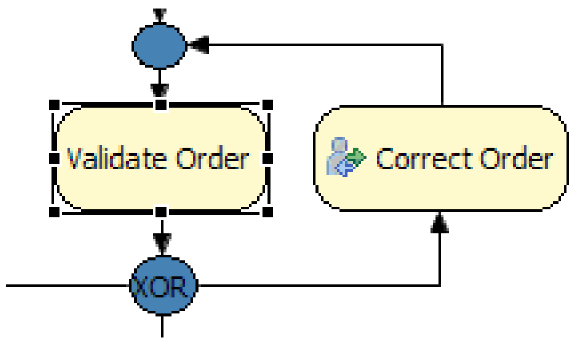
6.2.1. Evaluating a Set of Rules in Your Process

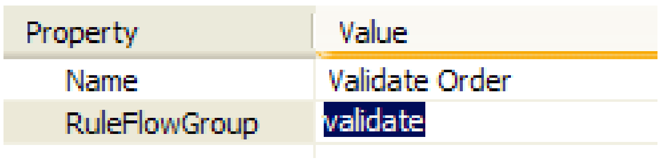
You can easily include a set of rules in the process. To do so, use the ruleflow-group attribute.

When you activate a RuleSet node for the group, the rules are evaluated. A rule 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:

```
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
```

The same ruleflow-group is shown in this figure:





Property	Value
Name	Validate Order
RuleFlowGroup	validate

Figure 6.1. RuleSet node and one of its rules

6.2.2. Using Rules to Evaluate Constraints

You can use rules to express and evaluate complex constraints in your rule flow. For example, you can use rules to define which the execution path to take at a Split node. Similarly, you can use rule to define a Wait duration.

For example, you can use rules to decide the next action after validating the order. Here are the conditions:

if the order contains errors, a sales representative should try to correct it;

orders with a **`value > 1000$`** are more important, so a senior sales representative should attend to these ones;

all other orders should just proceed normally.

Use decision node to select one of these alternatives, and write rules to describe the constraints for each of them.

6.2.3. Assignment Rules

Use *human tasks* to describe work that needs to be executed manually by an employee. You can base the selection of employee to do the work on the current state of the process and the history. Do so by creating *assignment rules*. These assignment rules will then be applied automatically whenever a new human task needs to be executed.

Note

The rules shown below are written in a Domain Specific Language (DSL), tailored to the specific requirements of an 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
```

6.2.4. Describing Exceptional Situations Using Rules

You can use rules to take exceptional situations into account. If you add all of this information to the regular process' control flow you will make things too complicated. Instead, use rules to handle each situation separately, leaving the core process in its simple form. This also makes it much easier to adapt existing processes to take previously unanticipated events into account.

6.2.5. Modularising Concerns Using Rules

Use rules add additional concerns to the process without making the overall control flow more complex. For example, you can define rules to log certain information during the execution of the process. The original process is not altered, as all logging functionality is modularised.

This greatly improves reusability as it allows you to apply the same strategy to different processes. It also aids readability and maintainability.

6.2.6. Rules for Altering Process Behaviour Dynamically

Use rules to fine-tune your processes dynamically. Imagine that you detect a problem with one of the processes. To troubleshoot, you can add new rules, at runtime, to log additional information or to handle specific process states. Once the problem is solved you can easily remove these rules again.

You could also ensure that different strategies were selected dynamically, depending on the current status. For example, based on the current load of all the services, rules that optimise the current load could be selected.

Here is an example that shows you how to dynamically add or remove logging for the "Check Order" task. When the **Debugging output** check-box in the main application window is ticked, the rule shown below will be loaded dynamically. It instructs the engine to write log output to the console. When you uncheck the box, the rule is removed 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
```

6.2.7. Integrated Tooling

Processes and rules are integrated in the JBDS. They are both treated as different types of business logic, to be managed in an almost identical manner. For example the processes for, loading a rule flow or a set of rules into the Engine are very similar to each other.

Also, different rule implementations, such DRL and DSL, are handled in an identical 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;
}
```

Note

The audit log is also integrated: it shows how rules and processes influence each other.

6.2.8. Domain-Specific Rules and Processes

You do not need to define rules using the core rule language syntax. You can also use domain-specific languages, decision tables or guided editors.

This example defines a domain-specific language that you can use to describe assignment rules, based on the type of task, its properties and the rule flow in which it is defined:

```

/********** 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
```

Note

This makes assignment rules much more understandable for non-experts.

Domain-Specific Processes

7.1. Introduction

The unified rules and processes framework allows you to extend the default programming constructs with domain-specific extensions in order to simplify development in a particular application domain. This tutorial teaches you the first steps towards creating *domain-specific process languages*.

Most process languages offer some generic action (node) construct in order to allow you to "plug in" custom user actions. However, these actions are usually low-level and you are required to write custom code to incorporate them 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 aimed at one particular application domain and therefore can offer constructs that are closely related to the problem the you are trying to solve. This makes the processes easier to understand and "self-documenting."

The first step is to create *work items*. These represent atomic units of work. They possess these characteristics:

1. domain-specific

2. declarative (what, not how)

3. high-level (no code)

4. can be customised for a particular context

7.2. Example: Notifications

Your first exercise is to build a simple work item for sending notifications.

You must give work item a unique name. You can also provide 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.

The sample notification work item can be defined using a work definition with four parameters and no results:

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

7.2.1. Creating the Work Definition

You must specify *work definitions* via one or more configuration files on the project class-path, where all of 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. Use MVEL to "read in" the configuration file, as it allows you to do more advanced work):

```
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"
  ]

]
```

When you save the file, name it **`MyWorkDefinitions.conf`**.

7.2.2. Registering the Work Definition

Use the `JBoss Rules Configuration` API's drools.workDefinitions property to register work definition files for your project. This property represents a list of files (separated by spaces) containing work definitions:

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

7.2.3. Using Your New Work Item in Your Processes

Once you have created and registered your work definition, you can start to use it in your rule flow.

The Process Editor's **palette** contains a separate section where the different work items that you have defined for your project appear.

Drag and drop a notification node onto your rule flow.

Next, go to the **properties view** and fill in the details. All work items also have these three properties (in addition to any defined by the work item):

1. Parameter Mapping: use this to map the value of a rule flow variable to a work item parameter. This allows you to customise the work item based on the current state of the actual process instance. (For example, the priority of the notification could be dependent on some process-specific information).

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

3. Wait for completion: by default, the rule flow waits until the requested work item has been completed before continuing. It is also possible to continue immediately after the work item has been requested (and not waiting for the results). To do so, set Wait for completion to **`false`**.

7.2.4. Executing Work Items

The Rule Flow engine contains a sub-component called the *WorkItemManager*. This is responsible for delegating the work items to *WorkItemHandlers* that execute them. Once they have done so, they notify the WorkItemManager that they have completed their work.

In order to execute notification work items, create a **`NotificationWorkItemHandler`**. (Use 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 e.-mail and then tells the WorkItemManager that the work item has been completed.

Note

Not all work items can be completed immediately. In cases where they take some time to process, execution can continue asynchronously and the WorkItemManager can be notified later.

In these situations, it is also possible that a work item can be aborted before it has been completed. Use the `abort` method to specify how you would like to do so.

To register WorkItemHandlers, use this API:

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

Human Tasks

As mentioned earlier, an important aspect of rule flow and business process management is human task management. While some of the work performed in a process can be executed automatically, other tasks need to be undertaken by your firm's staff members.

The Rules Flow Engine allows you to add human tasks to processes via a special human task node. This node allows process designers to define the type of task, the actor(s), the data associated with the task.

There is also a *task service* whose role it is to manage these human tasks.

Note

You can, however, integrate any other solution you wish, as it is fully pluggable.

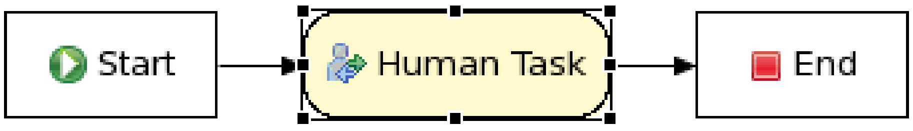
To utilise this functionality, you first need to undertake the following steps:

add human task nodes to your rule flow;

integrate a task management component of your choice (such as the WS-HT implementation provided by Red Hat out-of-the-box);

provide a user interface with which the employees undertaking the tasks can interact.

8.1. Adding Human Tasks to Rule Flows



The properties of a human task node were described in Chapter Three. You can edit these in the **properties view**.

Note

You can either directly specify the values of the different parameters (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 rule flow instance.

For example, parameters of type *`String`* can use **`#{expression}`** to embed a value. 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 creating an e.-mail, the body could contain something like "Dear **`#{customer.name}`**, ...".

For other types of variables, you can map the value to a parameter using the *`parameter mapping`*.

8.1.1. Swimlanes

You can use human task nodes in combination with swimlanes to assign multiple human tasks to similar actors. Tasks in the same swimlane will be assigned to the same actor. Whenever the first task in a swimlane is created, and you specify an actorId for that task, that actorId will be assigned to the swimlane as well. Every subsequent tasks created in that swimlane will use that actorId, even if a different actorId has been specified for one of them.

Whenever a human task belonging to a swimlane is completed, the actorId of that swimlane is changed to that of the actorId that executed that human task. This allows you to assign a human task to a group of users, and to assign all future tasks in that swimlane to the user that claimed the first task. It also means that the tasks will be automatically reassigned if, at some point, one of the tasks is undertaken by another employee.

To add a human task to a swimlane, simply specify the name of the swimlane as the value of that task node's *`Swimlane`* parameter.

Your rule flow must also define all of the swimlanes that it possesses. To make this so, follow these steps:

Open the **process properties** by clicking on the rule flow's background.

Next, click on the Swimlanes property.

Now, add the swimlanes.

8.2. Human Task Management Component

As far as the Rule Flow Engine is concerned, human tasks are similar to any other external service that needs to be invoked. They are, therefore, implemented as an extension of normal work items.

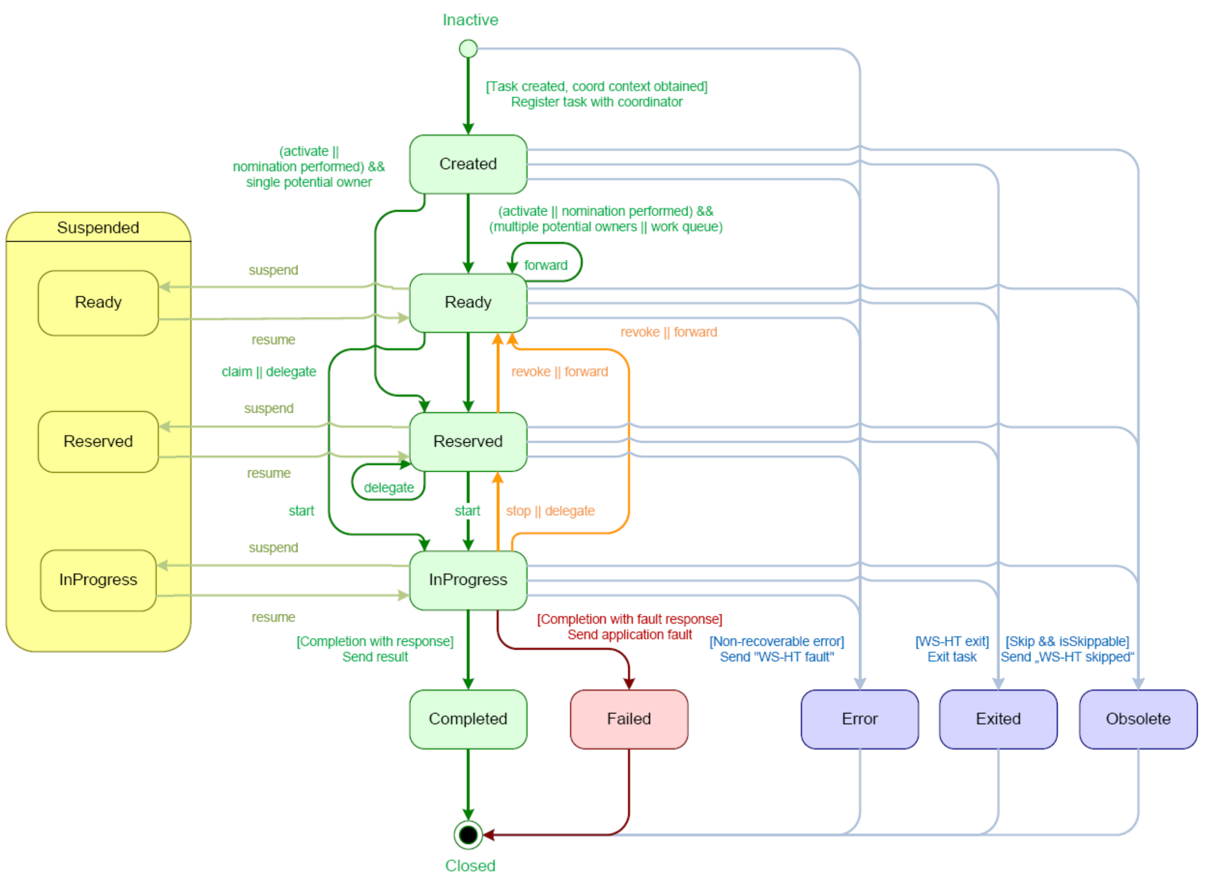
As a result, the process itself only contains an abstract description of the human tasks that need to be executed, and a work item handler is responsible for binding these abstract tasks to a specific implementation.

To accomplish this, Red Hat provides an implementation based on the WS-HumanTask specification. It manages the task life cycle (creation, claiming and completion) and stores the state of the task persistently. It also supports features like internationalisation, calendar integration, assignation, delegation and deadline functionality.

8.2.1. Task Life-Cycle

From the perspective of the rule flow, whenever a human task node is triggered, a human task is created. The rule flow will then only proceed on from that point when the task has been completed or aborted (unless you of course specify that the process does not need to wait for completion. To do so, set the Wait for completion property to **`true`**).

However, each human task also has its own separate life-cycle, as depicted in this diagram:



Each new task commences life in the **Created** state.

It then automatically switches to the **Ready** state, at which point the task will show up on the task list of every employee who has the permission to take it.

Once a staff member has claimed a task, the status is changed to **Reserved**.

Note

Note that if only one actor has permission to claim a task, that task will be immediately assigned to that employee.

After claiming the task, the employee can then, at some point, decide to start working on it, at which point the status will be changed to **InProgress**.

Finally, once the task has been performed, the employee must indicate that he or she has completed it (and can input any associated result data). When they do so, the status is changed to **Completed**.

If the task could not be completed, the employee can also indicate this by using a fault response. (They can also potentially input associated fault data.) When they do so, the status is changed to **Failed**.

The above is the normal life-cycle. The service also allows you to specify many other custome life-cycle methods. Here are some examples:

- task delegation, in which case it is assigned to another actor
- task revocation, in which case it is returned to all the potential actors' task lists

- task suspension and resumption
- stopping a task mid-way to completion
- skipping a task (if the task has been marked as skippable), in which case it will not be executed.

8.2.2. Linking the Task Component to the Rule Flow Engine

The task management component is integrated with the Rules Flow Engine just like any other external service, via the registration of a work item handler that is responsible for translating the abstract work item (in this case a human task) to a specific invocation.

Red Hat has chosen to implement this specific work handler: **`org.drools.process.workitem.wsht.WSHumanTaskHandler`**. You can easily link to it using code 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's port 9123. To change this, invoke the WSHumanTaskHandler's `setConnection(ipAddress, port)` method.

The WSHumanTaskHandler uses **Mina** *(http://mina.apache.org/)*[1] to test client/server architecture behaviour. **Mina** uses messaging to enable the client to communicate with the server. That is why the WSHumanTaskHandler has a MinaTaskClient that create different messages to give the user different actions. These actions are subsequently executed by the server.

In the **Mina** client, you will see these methods have been implemented to enable you to interact 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 these methods, you can create a GUI for your employees to do the tasks they have been assigned.

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

[1]

- taskId: this is the id. of the task on which the employee is working. You will most likely obtain this id. from the user task list in the GUI.
- userId: this is the id of the employee who is undertaking the work. You will most likely take the user's application log-in id.
- responseHandler: this is the handler used to catch the response. Users can input results here or just inform the system that they have finished the task.

All of these methods create messages that are sent back to the server, and the server will execute the logic that implements the correct action. Here is a typical message:

```
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 );
}
```

As you can see, a command is created. The method's arguments are inserted inside this command, along with information about the type of operation that we are trying to execute. This command is then sent to the server via the `session.write( cmd )` method.

When the server receives the command, the relevant logic is executed.

Look at the `messageReceived` method's **`TaskServerHandler`** class (found in taskOperation.) This is executed using the taskServiceSession that is responsible for all of the operations that occur when the tasks are first created and when the user is not interacting with them.

8.2.3. Starting the Task Management Component

The task management component is a completely independent service. Red Hat therefore recommends that you start it as a separate service as well. To do so, use this code:

```
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 to store all task information in a persistent manner. To configure this functionality, you need to modify some settings in the **`persistence.xml`** configuration file.

The following extract shows an example of how to use the task management component in conjunction with **Hibernate** and the 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, make sure that you add all the necessary users and groups to the database. They must all be predefined before you try to assign a task to that user or group.

To add the users and group to the database, use the `taskSession.addUser(user)` and `taskSession.addGroup(group)` methods.

Important

You need at least one "Administrator" account as all administrative tasks are automatically assigned to this user .

Note

In the **`src/test/java source`** directory, you will find the `drools-process-task` module. This module contains the **`org.drools.task.RunTaskService`** class, which you can use to start a task server. It automatically adds the users and groups defined in **`LoadUsers.mvel`** and **`LoadGroups.mvel`** configuration files.

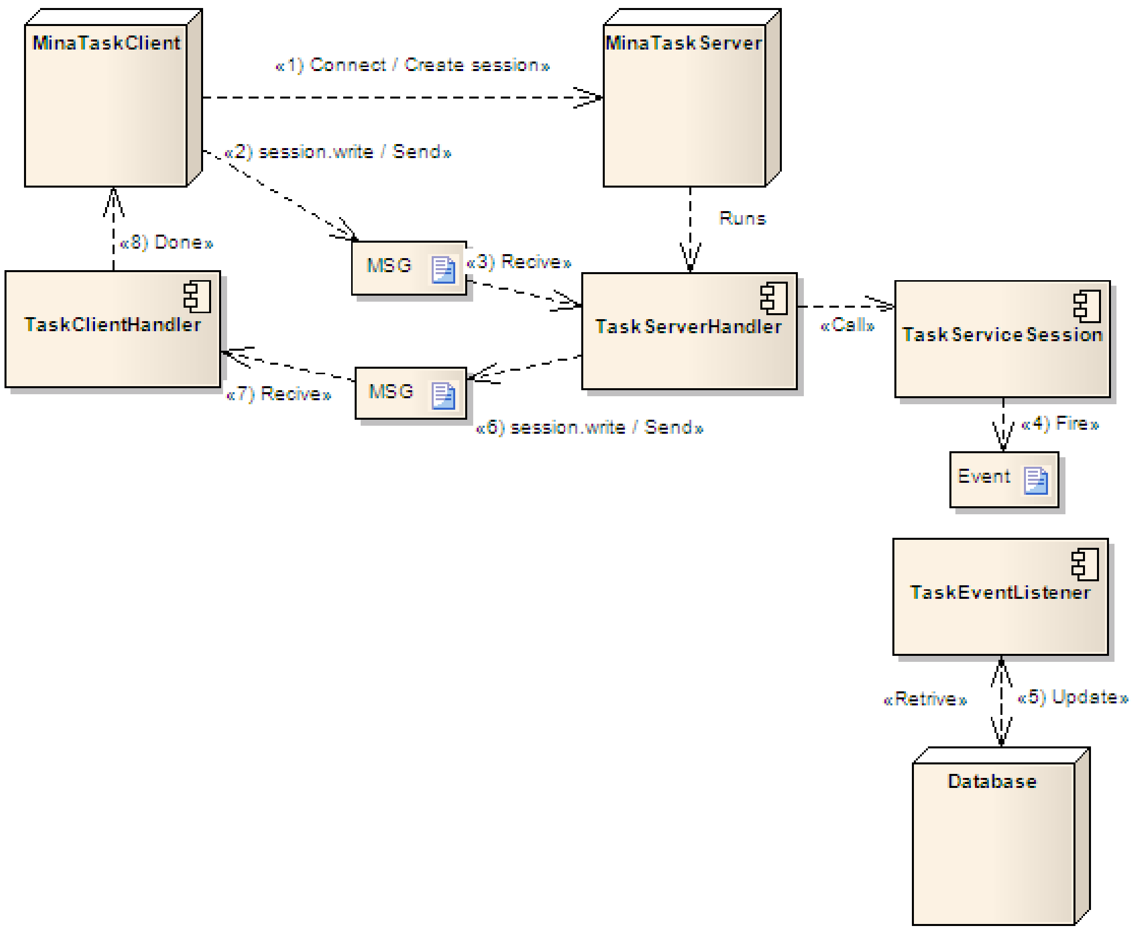
8.2.4. Interacting With the Task Management Component

The task management component exposes various methods that you can use to manage the life-cycle of the tasks through a Java API.

Warning

Ensure that end users do not interact with this low-level API directly. Rather, they should use one of the task-list clients. (These clients interact with the task management component through this API.)

This chart explains the interation between the client and the API:



Look at MinaTaskClient and MinaTaskServer in the diagram above. They communicate with each other in order to query and manipulate human tasks. This is how they interact:

- Imagine a client needs to complete a task. The user creates an instance of MinaTaskClient and connects it to the MinaTaskServer so that they can communicate. (This is Step One of the diagram.)
- The client calls the MinaTaskClient's `complete()` method, and supplies any corresponding arguments. This will generate a new message (or command) that will be inserted in the session that the client opens when it connects to the server.

  The message must specify a type that the server recognises so that it knows what to do when it receives it. (This is the second step in the diagram.)
- The TaskServerHandler now detects that there is a new message in the session so it analyses it. In this case, the message is of the type **`Operation.Complete`**, because the client has finished the task successfully.

  To complete the task that the user has indicated is finished, use the `TaskServiceSession`method. This will run a specific type of event that is processed by one of the TaskEventListener's sub-classes. (This process is represented by Steps Three and Four in the image.)
- When the event is received by TaskEventListener, the latter will know how to modify the task's status. This is because the EntityManager has retrieved and modified the status of a specific task via the database.

In this case, because you are finishing a task, the status will be updated to **`Completed`**. (This is the fifth step depicted in the image.)

- Now, that the changes have been made, you must notify the client that the task has been completed succesfuly. To achieve this, create a response message that TaskClientHandler will receive and use to inform the MinaTaskClient of what has happened. (This process is represented by Steps Six, Seven and Eight in the image.)

8.3. Human Task Management Interface

8.3.1. JBDS Integration

The JBoss Rules IDE comes with the **`org.drools.eclipse.task`** plug-in. Use it to test and debug rule flows that include human tasks.

The JBDS is equipped with a **Human Task View**. Use it to connect to a running task management component and request those tasks relevant to a particular user (that is the tasks for whom the user is either a potential or actual owner).

You can then run through the life-cycles of these tasks, by claiming or releasing, starting or stopping or completing them.

You can choose which of **JBoss Rules**' task management components you wish to connect to via the **JBoss Rules Task Preference Page**. To access this, follow these instructions:

click **Window**,

click **Preferences**,

select **JBoss Rules Task**,

specify the URL and port to which you wish to connect. (The default is 127.0.0.1:9123).

Debugging Processes

Read this chapter to learn how to debug processes. You can inspect the current state of your running processes. The tools also aid you to visualised how they will look during execution.

Note

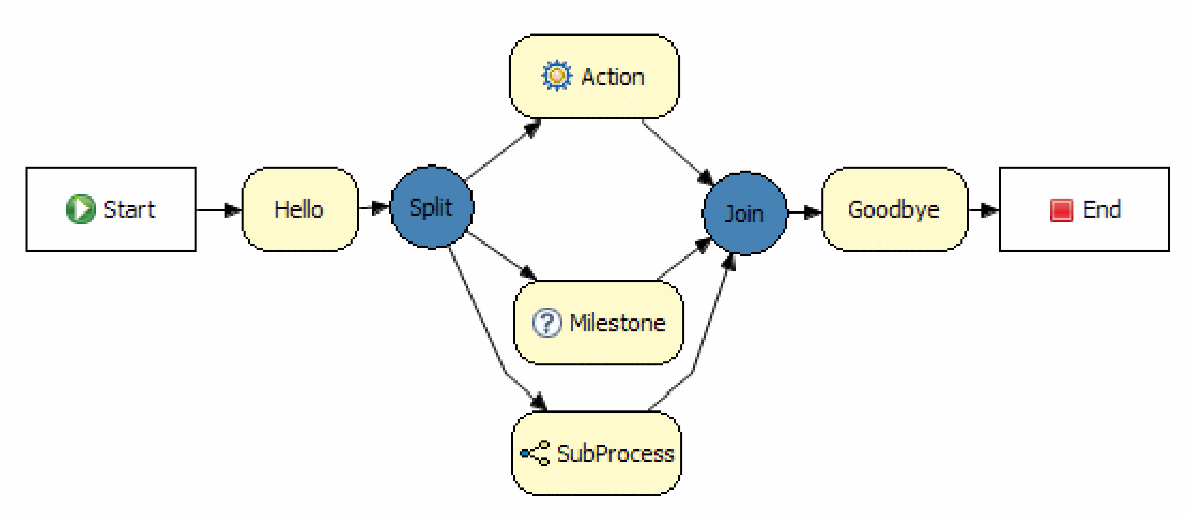
You cannot put *break-points* directly in a rule flow's nodes. You can, however, put them inside rules and on any Java code you might have (such as your application code). Using these break-points, you can inspect the internal state of each of your processes.

The following example illustrates the software's debugging capabilities.

9.1. A Simple Example

Our example contains two rule flows and some rules (used inside the ruleflow-groups):

1. the main rule flow contains some of the most commonly-used nodes: a start and end node (obviously), two ruleflow-groups, an action (that simply prints a string to the default output), a *milestone* (this is a wait state that is triggered when a specific Event is inserted in the working memory) and a sub-process:



2. the sub-process contains another milestone that also waits for (another) specific Event in the working memory.

3. there are only two rules (one for each ruleflow-group).They simply print out a **`hello world`** and **`goodbye world`** message to the default output.

To simulate an execution of this rule flow, start the process, fire every rule (that is, the hello rule), then add the specific milestone events for each of the and finally fire all of the rules again (resulting in the executing of the goodbye rule). The console output will look like this:

```
Hello World
Executing action
Goodbye World
```

9.2. Debugging the Process

Now you are going to add four break-points to the rule-flow. These are, in the order in which they will be encountered:

1. at the start of the hello rule's consequence;
2. just prior to the triggering event for the milestone in the main process
3. just after the triggering event for the milestone in the sub-process
4. at the start of the goodbye rule's consequence

When debugging the application, you can use the following debug views to track the rule flow's progress:

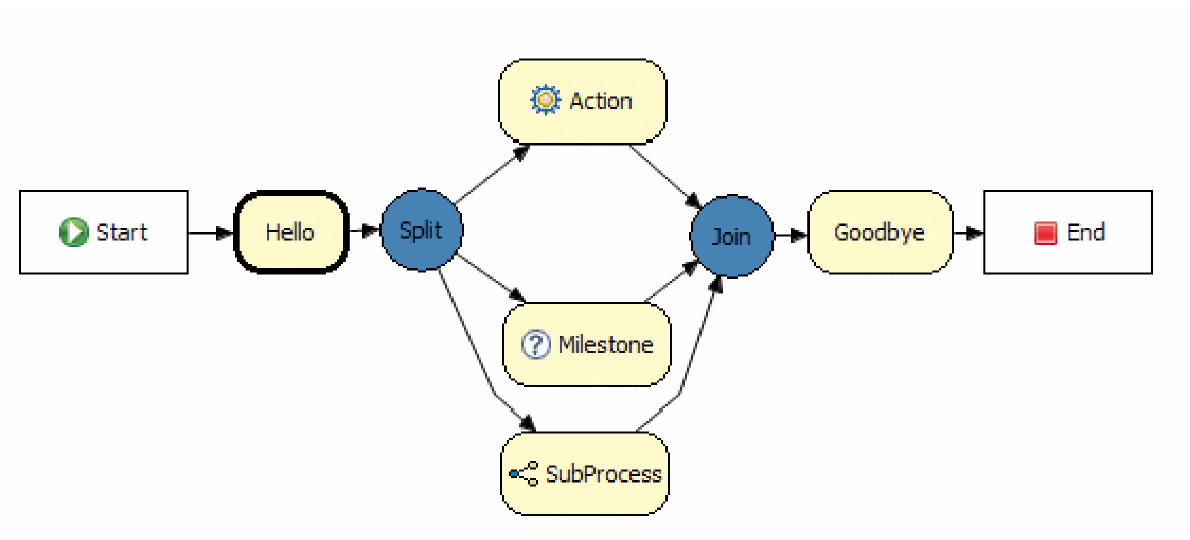
1. the **working memory view**. This shows all of the data in the working memory.
2. the **agenda view**. This shows all of the activations in the agenda.
3. the **global data view**. This displays all of the globals.
4. the default **Java Debug views**. These display the current line and the value of the known variables, and this for both normal Java code and for rules.
5. the **process instances view**. This displays all running rule-flows and their states.
6. the **audit view**. This shows the audit log.

9.2.1. The Process Instances View

The **process instances view** shows the currently-running rule flow instances. When you double-click on a process instance, the viewer will display its progress.

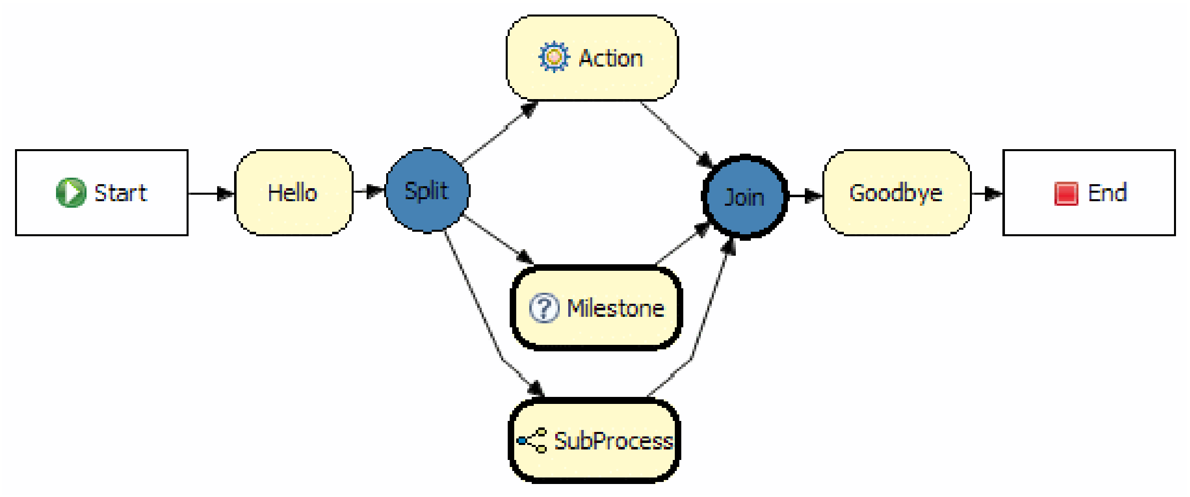
This is what you will see at each of the break-points in turn:

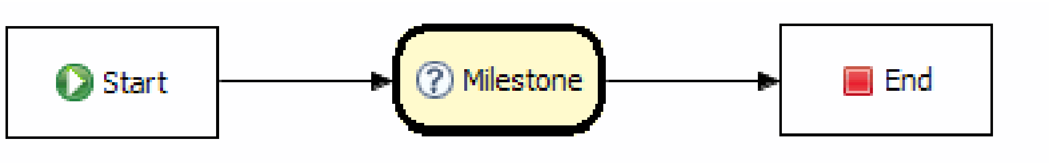
1. at the start of the hello rule's consequence, only the hello ruleflow-group is active. It is awaiting the execution of the hello rule:



2. once that rule has executed, the action, the milestone and the sub-process are all triggered. The action is executed immediately, triggering the join (which will simply wait until all incoming connections are triggered). The sub-process waits at the milestone.

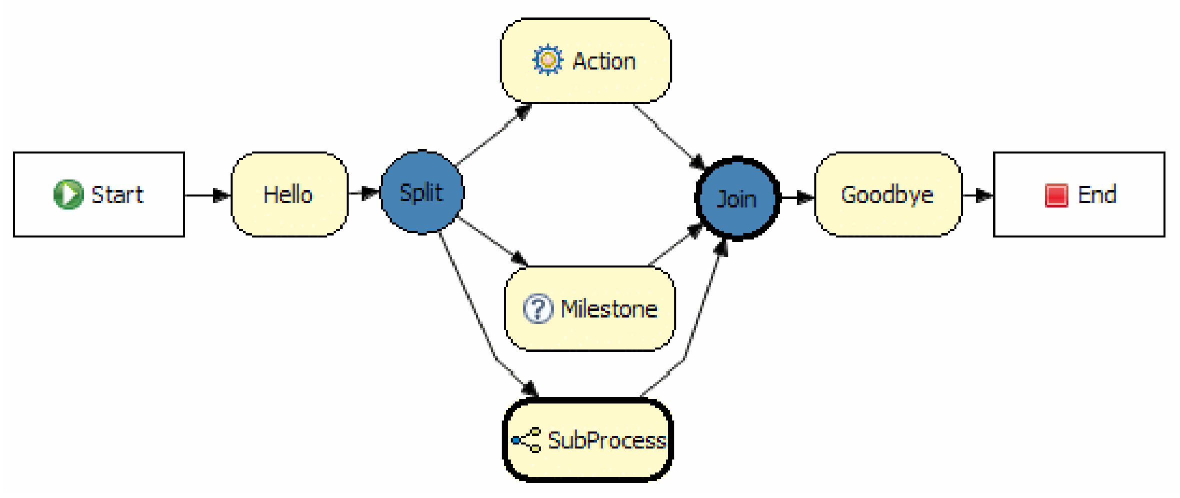
   Before the triggering event for the milestone is inserted in the main process, there are now two process instances. They look like this:

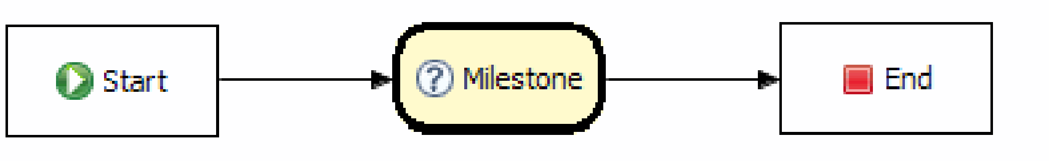




3. when you activate the event for the milestone in the main process, you also trigger the join (which will simply wait until every incoming connections has been activated).

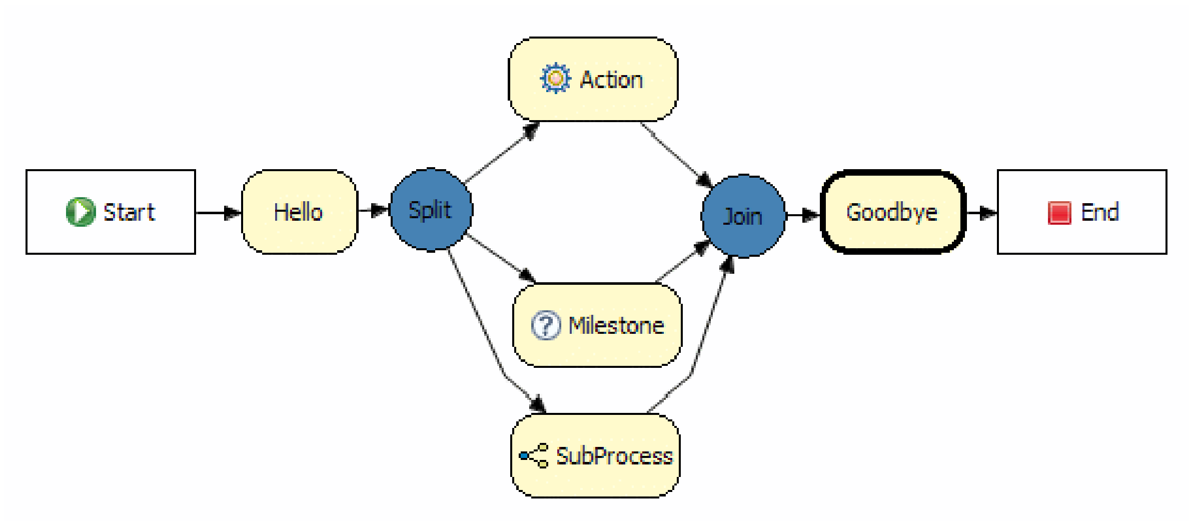
   At that point (which is before you insert the triggering event for the milestone in the sub-process), the rule flow will look like this:





4. when you trigger the event for the milestone in the sub-process, the rule flow instance will be completed and this will also activate the join, which will then continue and trigger the goodbye ruleflow-group (as all its incoming connections have been triggered.)

   If you run all of the rules, you will trigger the breakpoint in the goodbye rule:



5. after you execute the goodbye rule, the main rule flow will also be completed and the execution will have reached its conclusion.

9.2.2. The Audit View

You can look at the result in the **audit view**.

Note

The object insertion events may seem a little out of place. This is caused by the fact that they are only logged after (and never before) they are inserted, making it difficult to pinpoint their exact locations.

JBoss Rules IDE Features

The JBDS' JBoss Rules plug-in provides a few additional features that some business developers may find interesting. Read this chapter to learn about them.

10.1. JBoss Rules Run-times

A *JBoss Rules run-time* is a collection of **JAR** files that represent one specific release of the JBoss Rules project **JAR**s. To create a run-time, you must point the IDE to the release of your choice.

Note

You can create a new run-time based on the latest **JBoss Rules** project JARs which come included with the plug-in itself.

Note

You are required to specify a default **JBoss Rules** run-time for your workspace, but each individual project can override the default and select the run-time most appropriate for it.

10.1.1. Defining a JBoss Rules Run-time

Follow these instructions to define one or more **JBoss Rules** run-times:

go to the **Window** menu

selecting the **Preferences** menu item

a **Preferences** dialogue box, containing all of your settings, appears

on the left-hand side of this dialogue box, under the **JBoss Rules** category, select **Installed JBoss Rules run-times**. The panel on the right will then update to display all of your currently-defined run-times.

to define a new run-time, click on the **add** button. A dialogue box will appear.

input the name of your runtime and the path to its location on your file system.

In general, you have two options:

1. if you simply want to use the default JAR files as included with the **JBoss Rules** plug-in, just click the **Create a new JBoss Rules 5 run-time...** button.

   A file browser will appear, asking you to select the directory in which you want this run-time to be created. The plug-in will then automatically copy every required dependency into this directory.

2. if you want to use one specific release of the **JBoss Rules** project, you should create a directory on your file system that contains all of the required libraries and dependencies. Instead of creating a new **JBoss Rules** run-time as explained above, give your run-time a name and then select the directory that you just created, containing all of the required JARs.

after clicking the **OK** button, your newly-created run-time will appear in the right-hand panel alongside all the others.

click on check box in front of the newly-created run-time to make it the default . It will now be used as the run-time for all of your future **JBoss Rules** project (unless you select a project-specific one.)

Note

You can add as many **JBoss Rules** run-times as you need.

Important

You will need to restart the IDE if you changed the default run-time. This will ensure that all of your projects will use it. Their class-paths will update automatically.

10.1.2. Selecting a Run-time for Your JBoss Rules project

Whenever you create a **JBoss Rules** project (by using the **New JBoss Rules Project** wizard or by converting an existing Java project into a JBoss Rules project using the **Convert to JBoss Rules Project** command), the plug-in will automatically add all of the **`JAR`**s it needs to your project's class-path.

The default run-time will be used unless you specify otherwise when you are creating it. However, you can change the run-time at any time. To do so, follow these steps:

open the project's **properties**,

select the **JBoss Rules category**,

tick the **Enable project specific settings** check box

select the run-time you desire from the drop-down list.

Note

If you click the **Configure workspace settings...** link, the preferences will display, showing you the currently installed **JBoss Rules** run-times. You can add new run-times from this screen.

Note

If you de-select the **Enable project specific settings** check box, the default run-time will be used.

10.2. Process Skins

Use *process skins* to control the appearance of a rule flow's nodes.

Note

You can also change the appearance of the various node types by implementing your own `SkinProvider`.

BPMN is a popular language used employed by corporate developers to model business processes. Red Hat has created a BPMN skin that maps Rule Flow concepts to the equivalent BPMN visualisation.

You can change the process skin via the **JBoss Rules Preferences** dialogue box.

After reopening the editor, the rule flow will reappear, displaying the new BPMN skin.

Business Activity Monitoring

Monitor the performance of your rule flows diligently so that you can detect and react to anomalies early. Red Hat supplies real-time analysis tools that allow you to intervene directly, and sometimes even automatically, when problems arise.

You can create reports based on process engine-generated events.

11.1. Reporting

If you add a history logger to the process engine, it will record every relevant event in a database. You can then use this information to analyse the performance of your rule flows. Use JBDS' *Business Intelligence Reporting Tool* (BIRT) to create reports based on key performance indicators. To create custom reports, use the predefined data sets (as these contain all rule flow history information), and any other data sources you may wish to add.

The BIRT allows you to define include charts in your reports, preview the output, and export it as a web page.

A simple report might consist of the number of requests per hour and the average completion time for these requests. You can use this information to check for an unexpected drop or rise in the number of requests or an increase in the average processing time.

11.2. Direct Intervention

Most of the time, you will need to manually intervene if you spot a problem. However, in certain circumstances you can define automatic responses.

To monitor the Rule Flow Engine, add a listener that forward 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 one.

You can then use *complex event processing* (CEP) rules to specify how to process these events. For example, you might write rules that generate higher-level business events based on a specific pattern in the low-level events. The rules can also specify how to respond to specific situations.

Below is a sample rule that accumulates all start process events over the last hour, for one specific order rule flow. It uses the **Fusion CEP Engine**'s *sliding window* functionality to achieve this. This rule prints out an error message if more than 1000 process instances were started in the last hour (the aim being to detect potential overloading of the server).

Note

In real life, it would be better to send out an e.-mail or other form of notification to the person responsible instead of simply producing an error message.

```
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
```

You can use the rules to alter the behaviour of a rule-flow automatically at run-time. In other words, whenever a specific situation is detected, additional rules could be added to the knowledge base that will modify the rule flow.

For example, whenever a large amount of user requests made within a specific time frame are detected, an additional validation could be added to the process, enforcing some sort of flow control to reduce the frequency of incoming requests.

There is also the possibility of deploying additional logging rules as the consequence of detecting problems. As soon as the situation reverts 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 was adopted by Red Hat for modeling in the Rule Flow Engine. BPMN 2.0 not only defines a standard for how to graphically represent a business process also includes execution semantics for the defined elements, and an XML format that dictates how to store (and share) rule flow definitions.

The Rule Flow Engine allows you to execute processes defined using the BPMN 2.0 XML format, just as it allows you to execute processes using the custom RuleFlow format. This means that you can use the same API, engine and components to execute and manage your BPMN 2.0 processes.

Red Hat has not yet implemented all node types and attributes as defined in the BPMN 2.0 specification, but does support a very significant subset, which includes all common node types. Here is a list 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

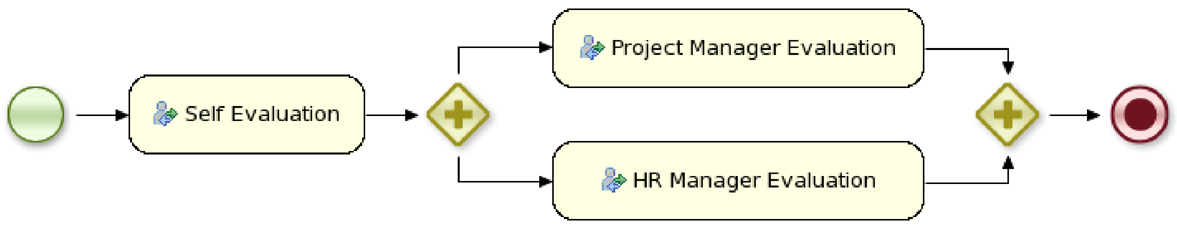
Data

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

Connecting Objects

- Sequence flow

Here is an example involving a corporate human resource department's process for evaluating employee performance. Whenever an evaluation rule flow for a specific employee is launched, that employee must first perform a self-evaluation, after which the project manager and human resource manager must also fill in their evaluation forms, as shown in the figure below:



An executable version of this process expressed coded in the BPMN 2.0 XML format will look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"
             targetNamespace="http://www.jboss.org/drools"
             typeLanguage="http://www.java.com/javaTypes"
             expressionLanguage="http://www.mvel.org/2.0"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
             xmlns: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>
```

Note

The process will need to contain all the details to make it executable, including all of the parameters for each of the tasks present, hence the length of the rule flow definition.

To create your own process using BPMN 2.0 format, choose one of these alternatives:

- make a new Flow file using the JBDS' JBoss Rules plug-in wizard. Then, on 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 Rule Flow format.

  Important

  This, however, is not a real BPMN 2.0 editor, as it still uses different attributes. Fortunately, though, it does however save the rule flow using valid BPMN 2.0 syntax.

  Also note that the editor does not yet support all of the node types and attributes that are supported by the execution engine.

- **Oryx** is an open-source web-based editor that supports the BPMN 2.0 format. Red Hat has embedded it into the JBoss Rules engine in order to supply BPMN 2.0 rule flow visualisation and editing functionality. You can use the **Oryx** editor (as either a standalone or integrated tool) to create and edit BPMN 2.0 processes.

  Once you have done so, export them to BPMN 2.0 format so they can be executed.

  Warning

  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 manually create BPMN 2.0 process files by hand-coding the XML.

Use this code 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();
}
```

Use this code to execute the rule flow:

```
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);
```

12.1. Current Limitations

Warning

Since the BPMN 2.0 specification is still being finalised, BPMN 2.0 execution is still an experimental feature.

It is possible that the XSD that defines the format might still change slightly as the specification matures, so keep this in mind if you decide to start using the format.

BPMN uses the same execution engine and constructs as the Rule Flow format however (as it is just another XML serialisation 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.

The use of a specification should give you many advantages, as it allows you to share your processes across tools and possibly even engines as it defines the exact format (and even execution semantics) for each of the elements.

Important

At this point in time, however, it is likely that different tools are using different intermediate versions of the specification. Red Hat believes that this issue will resolve itself naturally over time once the specification is finalised and everyone is using the same version, but until then, you can encounter compatibility issues related to this problem.

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. To resolve this, the specification is allows for additional node types, attributes and so forth. Red Hat tries to limit the use of custom extensions to a minimum but sometimes has to define additional attributes to express features that it belieces is important but cannot be expressed via the 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.

An orange check mark means that the functionality is available via a custom extension that Red Hat has implemented.

Note

Red Hat has decided to delay implementation of those features that cannot be expressed in BPMN 2.0 by default like, for example, the on-entry and on-exit actions and the state node.

When and how these will be supported is a decision for a later date.

Table 12.1. 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	✓	✓
- signal trigger	✓	✓
- parameter mapping	✓	✓
2. End Node	✓	✓
- terminate	✓	✓
3. Action Node	✓	✓
- Java dialect	✓	✓

Feature	JBoss Rules BPMN	JBoss Rules Flow
* 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		✓
7. State Node		✓
- timers		✓
- on entry actions		✓
- on exit actions		✓
- automatic transition constraints		✓
- manual transition signal		✓

Feature	JBoss Rules BPMN	JBoss Rules Flow
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)	✓	✓
- swimlane	✓	✓
12. Composite Node	✓	✓
- timers		✓
- on entry actions		✓
- on exit actions		✓
- variables	✓	✓

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

Console

You can manage your rule flows through a web console that comes supplied with the product.

The **JBoss Rules** build system generates two **WAR** archive files for you. Deploy them to your application server to run.

13.1. Running the Process Management Console

To access the console, launch a web browser and go to the following address: *http://localhost:8080/gwt-console*[1]

Note

If you have changed your application server's default settings, the address or port number may be different.

A log in screen will appear. Input your user name and password.

The **Process Management Workbench** screen will now open. On the right-hand side, you will see several tabs. These are for process instance management, human task lists and reporting respectively.

13.1.1. Managing Rule Flow Instances

Use the **Processes** screen to inspect those rule flow definitions that are currently part of the installed knowledge base. Here you can also start new instances and manage the ones.

13.1.1.1. Inspecting Rule Flow Definitions

When you open the **Process Definition List**, every rule flow stored in the **JBoss Rules**' default package is shown. You can either start a new rule flow instance or inspect those belonging to a specific process.

13.1.1.2. Starting New Rule Flow Instances

To start a new instance for one specific rule flow definition, follow these steps:

select the definition from the **process definition list;**

click on the **select the process instances** tab;

click on the **Start** button.

Note

When a form is associated with this particular process, it will be displayed. After you complete this form, the rule flow will launch and make use of the information you have provided.

[1]

13.1.1.3. Managing Rule Flow Instances

The **process instances** tab also contains a table that shows you all of running instances of that specific rule flow definition. Select an instance to see its details.

13.1.1.4. Inspecting a Rule Flow Instance State

To inspect specific rule flow instance's top-level variables, click on the **Instance Data** button. This will show you how each variable to a corresponding value.

13.1.1.5. Inspecting Rule Flow Instance Variables

To inspect the state of a specific rule flow instance, follow these steps:

click on the **Diagram** button;

you will then see a flow chart. On this chart, red triangles represent nodes that are currently active.

Note

Multiple instances of one node may be executing simultaneously. They will still be depicted by only one red triangle.

13.1.2. Human Task Lists

Employees can go to the **task management** section to see their **current task lists**.

The **group task list** shows a pool of every task that has not yet been claimed by one specific user.

The **personal task list** shows every tasks that is assigned to the user who is currently logged in.

To execute a task, follow these steps:

select it from your **personal task list**;

click on **View**.

If a form is associated with the selected task, it will be shown. After you complete the form, the task will also be completed.

13.1.3. Reporting

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

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

Rule Flow comes with some sample reports that may help you to visualise generic execution characteristics (like the number of active process instances per process.)

To create customised reports, replace the report template files in the `report` directory with your own versions.

13.2. Adding New Task Forms

You can use forms to start new rule flows or complete human tasks.

To create a form for a specific rule flow definition, you need to make a **Freemarker** template. Save it with a filename of this format: **`{processId}.ftl`**.

Use HTML code to model the template. Here is a sample form that starts the employee evaluation process mentioned above:

```
<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>
```

Note

You will find this template comes with this product as **`com.sample.evaluation.ftl`**.

To link task forms to human tasks, you similarly creating a Freemarker template and give it a name in this format: **`{taskName}.ftl`**.

The form has access to a *`task`* parameter that represents the current human task, thus allowing you to dynamically adjust the form based on the input.

Note

The task parameter is a Task model object as defined in the `drools-process-task` module.

This allows you to customise the task form based on, for instance, the description or input data related to that task. The evaluation form shown earlier uses the task parameter to access the description of the task. That description is then displayed when the task appears on screen:

```
<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>
```

The employee adds data to the form by filling it in. This data is stored in parameters, completing the task.

For instance, in the example above, when you complete the task, the Map of outcome parameters will include result variables called performance, initiative, change and communication. To access these result parameters from the related rule flow, map them to to the process variables.

Appendix A. © 2011

Appendix B. Revision History

Revision 1.0-0 Fri Jul 8 2011 **David Le Sage** *dlesage@redhat.com*

Initial conversion of community documentation.

Index