JBoss Enterprise BRMS Platform 5

BRMS Rule Flow Component Guide

for Business Rules Developers

Edition 5.3.0

Last Updated: 2017-11-17

JBoss Enterprise BRMS Platform 5 BRMS Rule Flow Component Guide

for Business Rules Developers
Edition 5.3.0

Edited by

David Le Sage (Product Version)
Red Hat Engineering Content Services

Abstract

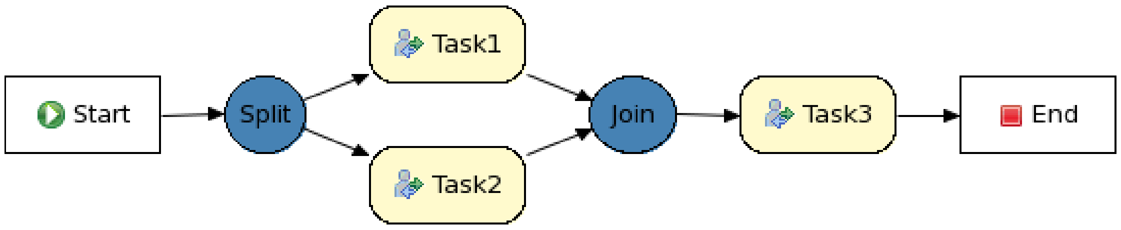
Read this guide to learn how to use JBoss Enterprise BRMS Platform's Rule Flow component to develop business rules and undertake service orchestration tasks.

Table of Contents

CHAPTER 1. INTRODUCTION

The **JBoss Enterprise BRMS Platform**'s *Rule Flow* module is a work-flow process engine.

A *Rule Flow* 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 BRMS Platform** Rule Flow Engine.

CHAPTER 2. 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, clear 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 visualize 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`**.

CHAPTER 3. RULE FLOWS

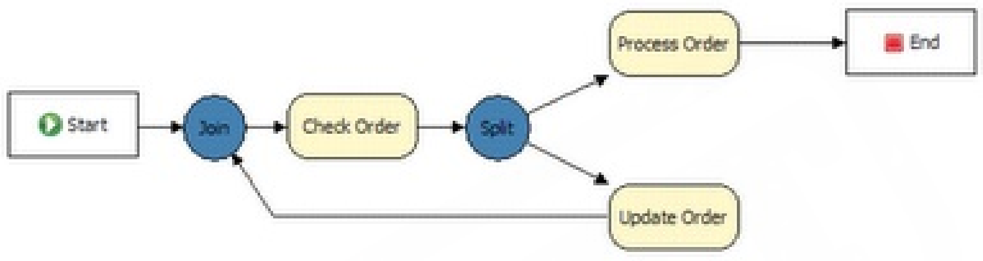


Figure 3.1. A Rule Flow

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.M
   yProcess");
   ```

3.3. AVAILABLE 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 package (or name-space) in which the process is stored.
- *Variables*: you can define variables to store data during the execution of your process.
- *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.

The following types of nodes are available when creating a rule flow:

- Section 3.3.1, “Start Event”
- Section 3.3.2, “End Event”
- Section 3.3.3, “Rule Task (or RuleFlowGroup)”
- Section 3.3.4, “Diverging Gateway (or Split)”
- Section 3.3.5, “Converging Gateway (or Join)”
- Section 3.3.6, “State”
- Section 3.3.7, “Reusable Sub-Process (or SubFlow)”
- Section 3.3.8, “Action (or Script Task)”
- Section 3.3.9, “Timer Event”
- Section 3.3.10, “Error Event (or Fault)”
- Section 3.3.11, “(Message) Event”
- Section 3.3.12, “Sub-Process (or Composite)”
- Section 3.3.13, “Multiple Instance (or ForEach)”

3.3.1. Start Event

This is the start of the rule flow. (A rule flow must have only one start node. It cannot have incoming connections and 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.

3.3.2. 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.

3.3.3. 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.

3.3.4. 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.

3.3.5. Converging Gateway (or Join)

Use this to synchronize 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.

3.3.6. 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.

3.3.7. 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.

3.3.8. 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.

3.3.9. 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.

3.3.10. 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.

3.3.11. (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.

3.3.12. 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 behavior to occur when a fault is encountered.

3.3.13. 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.

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. Sub-scopes 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 behavior 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

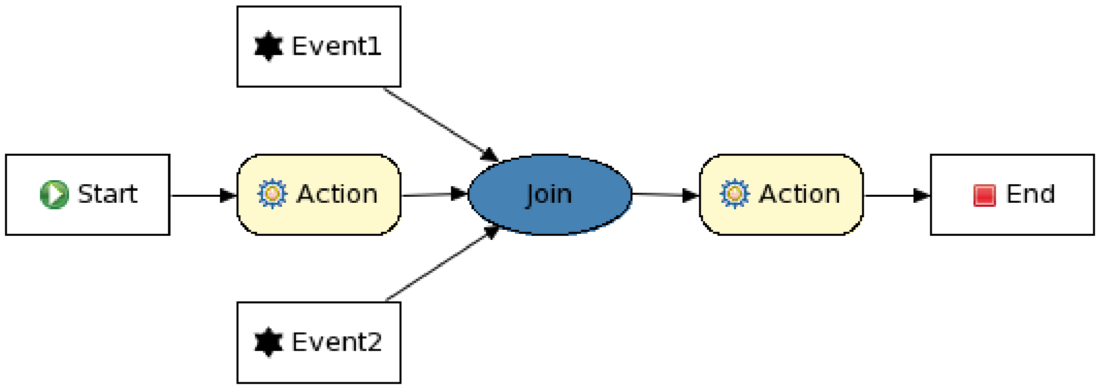


Figure 3.2. A sample process using events

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

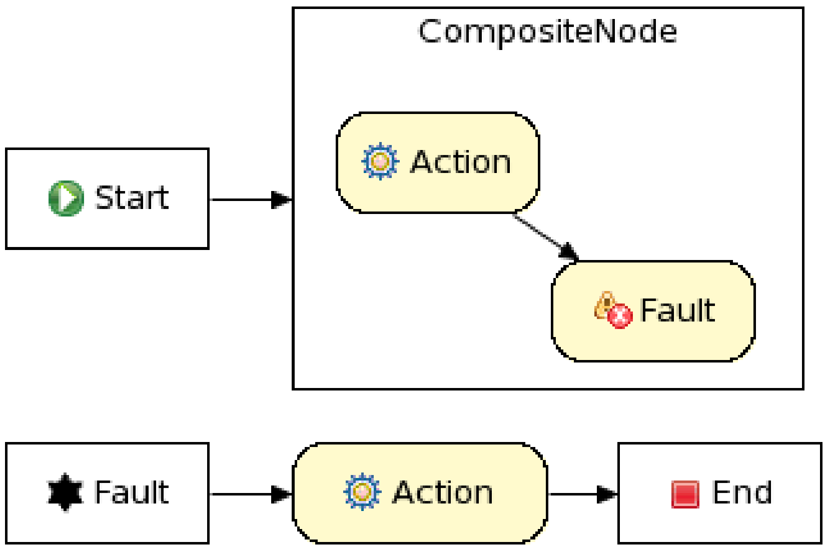


Figure 3.3. A sample process using exception handlers

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 behavior 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 initialize 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 synchronization.

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

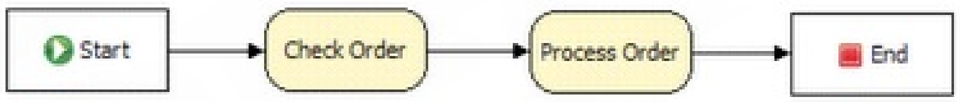
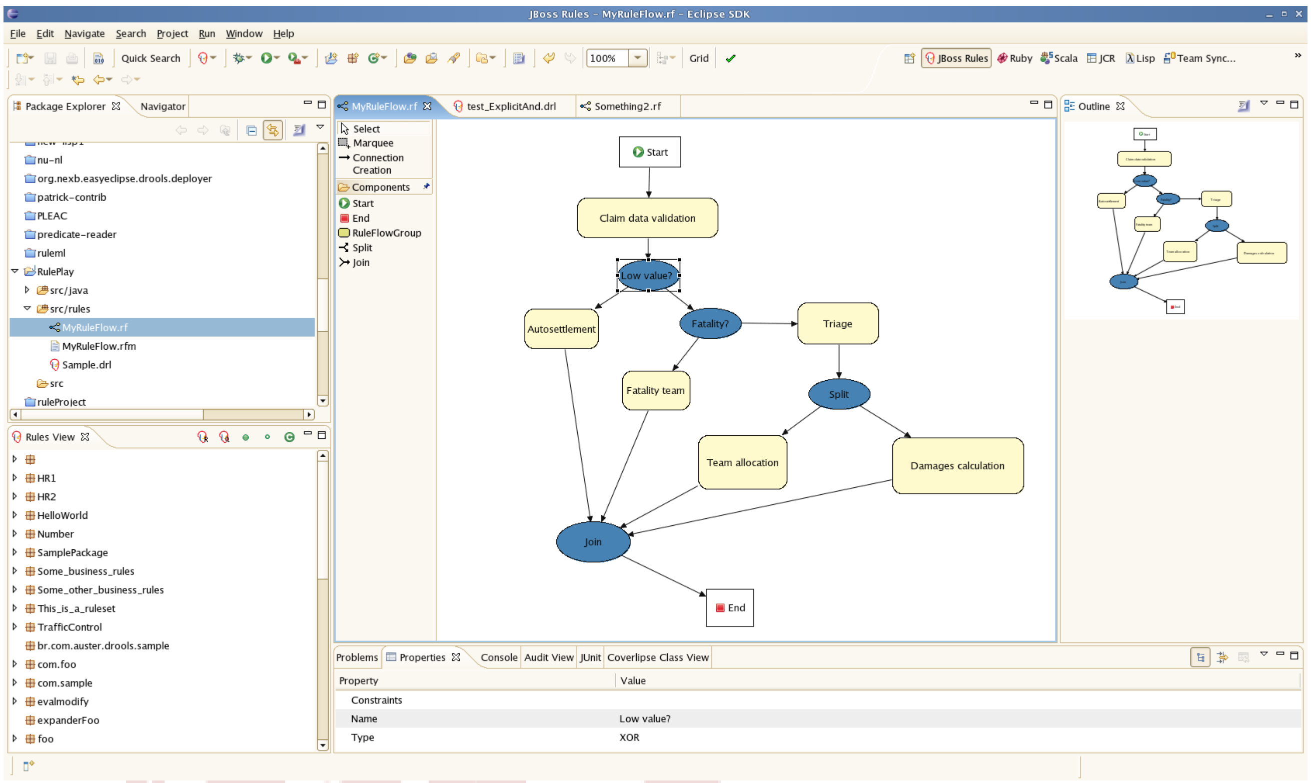


Figure 3.4. A Simple Rule Flow

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 finalizing 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.

CHAPTER 4. 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 visualization of the events that occurred during execution.

3. Threaded file logger: Because a file logger writes the events to disk only when closing the logger or when the number of events in the logger reaches a 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 visualize 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.)

CHAPTER 5. 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.

5.1. 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.

5.1.1. 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.

5.1.2. 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.

5.2. 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 visualization.

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.

APPENDIX A. © 2011

APPENDIX B. REVISION HISTORY

Revision 5.3.0-15.402	**Fri Oct 25 2013**	**Rüdiger Landmann**
Rebuild with Publican 4.0.0		
Revision 5.3.0-15.33	**2012-07-22**	**Anthony Towns**
Rebuild for Publican 3.0		
Revision 5.2.0-3	**Thur Jan 12 2012**	**L Carlon**
Removed the chapters dealing with Business Activity monitoring and the web console as these are not supported in Ruleflow.		
Revision 5.2.0-2	**Mon Nov 21 2011**	**L Carlon**
Removed the BPMN2.0 notation chapter as BPMN2.0 is not supported in Ruleflow.		
Revision 5.2.0-1	**Thu Sept 30 2011**	**L Carlon**
Updated for the JBoss Enterprise BRMS Platform		
Revision 5.2.0-0	**Thu Jul 14 2011**	**David Le Sage**
First edition of the book		

INDEX