jBPM Console NG User Guide

Version 6.0.0.CR1

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


jBPM

Chapter 1. Core Engine: API

This chapter introduces the API you need to load processes and execute them. For more detail on how to define the processes themselves, check out the chapter on BPMN 2.0.

To interact with the process engine (for example, to start a process), you need to set up a session. This session will be used to communicate with the process engine. A session needs to have a reference to a knowledge base, which contains a reference to all the relevant process definitions. This knowledge base is used to look up the process definitions whenever necessary. To create a session, you first need to create a knowledge base, load all the necessary process definitions (this can be from various sources, like from classpath, file system or process repository) and then instantiate a session.

Once you have set up a session, you can use it to start executing processes. Whenever a process is started, a new process instance is created (for that process definition) that maintains the state of that specific instance of the process.

For example, imagine you are writing an application to process sales orders. You could then define one or more process definitions that define how the order should be processed. When starting up your application, you first need to create a knowledge base that contains those process definitions. You can then create a session based on this knowledge base so that, whenever a new sales order comes in, a new process instance is started for that sales order. That process instance contains the state of the process for that specific sales request.

A knowledge base can be shared across sessions and usually is only created once, at the start of the application (as creating a knowledge base can be rather heavy-weight as it involves parsing and compiling the process definitions). Knowledge bases can be dynamically changed (so you can add or remove processes at runtime).

Sessions can be created based on a knowledge base and are used to execute processes and interact with the engine. You can create as many independent session as you need and creating a session is considered relatively lightweight. How many sessions you create is up to you. In general, most simple cases start out with creating one session that is then called from various places in your application. You could decide to create multiple sessions if for example you want to have multiple independent processing units (for example, if you want all processes from one customer to be completely independent from processes for another customer, you could create an independent session for each customer) or if you need multiple sessions for scalability reasons. If you don't know what to do, simply start by having one knowledge base that contains all your process definitions and create one session that you then use to execute all your processes.

1.1. The jBPM API

The jBPM project has a clear separation between the API the users should be interacting with and the actual implementation classes. The public API exposes most of the features we believe "normal" users can safely use and should remain rather stable across releases. Expert users can

still access internal classes but should be aware that they should know what they are doing and that the internal API might still change in the future.

As explained above, the jBPM API should thus be used to (1) create a knowledge base that contains your process definitions, and to (2) create a session to start new process instances, signal existing ones, register listeners, etc.

1.1.1. Knowledge Base

The jBPM API allows you to first create a knowledge base. This knowledge base should include all your process definitions that might need to be executed by that session. To create a knowledge base, use a knowledge builder to load processes from various resources (for example from the classpath or from the file system), and then create a new knowledge base from that builder. The following code snippet shows how to create a knowledge base consisting of only one process definition (using in this case a resource from the classpath).

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

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

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

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

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

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

```
      <resource source='file:/path_to_process/folder2/' type='BPMN2'/>
   </add>
</change-set>
```

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

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

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

```
ResourceChangeScannerConfiguration sconf = ResourceFactory.getResourceChangeScannerService().ne
sconf.setProperty( "drools.resource.scanner.interval", "10" ); // every 10s
ResourceFactory.getResourceChangeScannerService().configure( sconf );
ResourceFactory.getResourceChangeScannerService().start();
ResourceFactory.getResourceChangeNotifierService().start();
KnowledgeAgentConfiguration aconf = KnowledgeAgentFactory.newKnowledgeAgentConfiguration();
aconf.setProperty("drools.agent.newInstance", "false");
KnowledgeAgent    kagent    =    KnowledgeAgentFactory.newKnowledgeAgent("Folder
 changeset", aconf);
kagent.applyChangeSet(ResourceFactory.newClassPathResource("changesetFolder.xml"));
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

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

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

1.1.2. Session

Once you've loaded your knowledge base, you should create a session to interact with the engine. This session can then be used to start new processes, signal events, etc. The following code snippet shows how easy it is to create a session based on the previously created knowledge base, and to start a process (by id).

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

The `ProcessRuntime` interface defines all the session methods for interacting with processes, as shown below.

```
    /**
     * Start a new process instance.  The process (definition) that should
     * be used is referenced by the given process id.
     *
     * @param processId  The id of the process that should be started
     * @return the ProcessInstance that represents the instance of the process that was started
     */
    ProcessInstance startProcess(String processId);

    /**
     * Start a new process instance.  The process (definition) that should
     * be used is referenced by the given process id.  Parameters can be passed
     * to the process instance (as name-value pairs), and these will be set
     * as variables of the process instance.
     *
     * @param processId  the id of the process that should be started
     * @param parameters  the process variables that should be set when starting the process instance
     * @return the ProcessInstance that represents the instance of the process that was started
     */
    ProcessInstance startProcess(String processId,
                                 Map<String, Object> parameters);

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

```
     * @param event the data associated with this event
     */
    void signalEvent(String type,
                     Object event);

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

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

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

    /**
     * Aborts the process instance with the given id.  If the process instance has been complet
     * (or aborted), or the process instance cannot be found, this method will throw an
     * IllegalArgumentException.
     *
     * @param id the id of the process instance
```

```
     */
    void abortProcessInstance(long processInstanceId);

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

1.1.3. Events

The session provides methods for registering and removing listeners. A `ProcessEventListener` can be used to listen to process-related events, like starting or completing a process, entering and leaving a node, etc. Below, the different methods of the `ProcessEventListener` class are shown. An event object provides access to related information, like the process instance and node instance linked to the event. You can use this API to register your own event listeners.

```
public interface ProcessEventListener {

  void beforeProcessStarted( ProcessStartedEvent event );
  void afterProcessStarted( ProcessStartedEvent event );
  void beforeProcessCompleted( ProcessCompletedEvent event );
  void afterProcessCompleted( ProcessCompletedEvent event );
  void beforeNodeTriggered( ProcessNodeTriggeredEvent event );
  void afterNodeTriggered( ProcessNodeTriggeredEvent event );
  void beforeNodeLeft( ProcessNodeLeftEvent event );
  void afterNodeLeft( ProcessNodeLeftEvent event );
  void beforeVariableChanged(ProcessVariableChangedEvent event);
  void afterVariableChanged(ProcessVariableChangedEvent event);

}
```

A note about before and after events: these events typically act like a stack, which means that any events that occur as a direct result of the previous event, will occur between the before and the after of that event. For example, if a subsequent node is triggered as result of leaving a node, the node triggered events will occur inbetween the beforeNodeLeftEvent and the afterNodeLeftEvent of the node that is left (as the triggering of the second node is a direct result of leaving the first node). Doing that allows us to derive cause relationships between events more easily. Similarly, all node triggered and node left events that are the direct result of starting a process will occur between the beforeProcessStarted and afterProcessStarted events. In general, if you just want to be notified when a particular event occurs, you should be looking at the before events only (as they occur immediately before the event actually occurs). When only looking at the after events, one might get the impression that the events are fired in the wrong order, but because the after

events are triggered as a stack (after events will only fire when all events that were triggered as a result of this event have already fired). After events should only be used if you want to make sure that all processing related to this has ended (for example, when you want to be notified when starting of a particular process instance has ended.

Also note that not all nodes always generate node triggered and/or node left events. Depending on the type of node, some nodes might only generate node left events, others might only generate node triggered events. Catching intermediate events for example are not generating triggered events (they are only generating left events, as they are not really triggered by another node, rather activated from outside). Similarly, throwing intermediate events are not generating left events (they are only generating triggered events, as they are not really left, as they have no outgoing connection).

jBPM out-of-the-box provides a listener that can be used to create an audit log (either to the console or the a file on the file system). This audit log contains all the different events that occurred at runtime so it's easy to figure out what happened. Note that these loggers should only be used for debugging purposes. The following logger implementations are supported by default:

1. Console logger: This logger writes out all the events to the console.
2. File logger: This logger writes out all the events to a file using an XML representation. This log file might then be used in the IDE to generate a tree-based visualization of the events that occurred during execution.
3. Threaded file logger: Because a file logger writes the events to disk only when closing the logger or when the number of events in the logger reaches a predefined level, it cannot be used when debugging processes at runtime. A threaded file logger writes the events to a file after a specified time interval, making it possible to use the logger to visualize the progress in realtime, while debugging processes.

The `KnowledgeRuntimeLoggerFactory` lets you add a logger to your session, as shown below. When creating a console logger, the knowledge session for which the logger needs to be created must be passed as an argument. The file logger also requires the name of the log file to be created, and the threaded file logger requires the interval (in milliseconds) after which the events should be saved. You should always close the logger at the end of your application.

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

The log file that is created by the file-based loggers contains an XML-based overview of all the events that occurred at runtime. It can be opened in Eclipse, using the Audit View in the Drools

Eclipse plugin, where the events are visualized as a tree. Events that occur between the before and after event are shown as children of that event. The following screenshot shows a simple example, where a process is started, resulting in the activation of the Start node, an Action node and an End node, after which the process was completed.

1.2. Knowledge-based API

As you might have noticed, the API as exposed by the jBPM project is a knowledge API. That means that it doesn't just focus on processes, but potentially also allows other types of knowledge to be loaded. The impact for users that are only interested in processes however is very small. It just means that, instead of having a ProcessBase or a ProcessSession, you are using a KnowledgeBase and a KnowledgeSession.

However, if you ever plan to use business rules or complex event processing as part of your application, the knowledge-based API allows users to add different types of resources, such as processes and rules, in almost identical ways into the same knowledge base. This enables a user who knows how to use jBPM to start using Drools Expert (for business rules) or Drools Fusion (for event processing) almost instantaneously (and even to integrate these different types of Knowledge) as the API and tooling for these different types of knowledge is unified.