Red Hat JBoss BPM Suite 6.0

Development Guide

For Red Hat JBoss Developers

Last Updated: 2017-11-17

Red Hat JBoss BPM Suite 6.0 Development Guide

For Red Hat JBoss Developers

Kanchan Desai
kadesai@redhat.com

Doug Hoffman

Eva Kopalova

David Le Sage
Red Hat Engineering Content Services
dlesage@redhat.com

Red Hat Content Services

Legal Notice

Abstract

A guide to using API's in BPM Suite for developers

Table of Contents

CHAPTER 1. INTRODUCTION

1.1. ABOUT RED HAT JBOSS BPM SUITE

Red Hat JBoss BPM Suite is an open source business process management suite that combines Business Process Management and Business Rules Management and enables business and IT users to create, manage, validate, and deploy Business Processes and Rules.

Red Hat JBoss BRMS and Red Hat JBoss BPM Suite use a centralized repository where all resources are stored. This ensures consistency, transparency, and the ability to audit across the business. Business users can modify business logic and business processes without requiring assistance from IT personnel.

To accommodate Business Rules component, Red Hat JBoss BPM Suite includes integrated Red Hat JBoss BRMS.

Business Resource Planner is included as a technical preview with this release.

Report a bug

1.2. SUPPORTED PLATFORMS

Red Hat JBoss BPM Suite and Red Hat JBoss BRMS are supported on the following containers:

- Red Hat JBoss Enterprise Application Platform 6.1.1
- Red Hat JBoss Web Server 2.0 (Tomcat 7)
- IBM WebSphere Application Server 8

Report a bug

1.3. USE CASE: PROCESS-BASED SOLUTIONS IN THE LOAN INDUSTRY

Red Hat JBoss BPM Suite (BPMS) can be deployed to automate business processes, such as automating the loan approval process at a retail bank. This is a typical 'Specific Process-Based' deployment that might be the first step in a wider adoption of BPM throughout an enterprise. It leverages both the BPM and business rules features of BPMS.

A retail bank offers several types of loan products each with varying terms and eligibility requirements. Customers requiring a loan must file a loan application with the bank, which then processes the application in several steps, verifying eligibility, determining terms, checking for fraudulent activity, and determining the most appropriate loan product. Once approved, the bank creates and funds a loan account for the applicant, who can then access funds. The bank must be sure to comply with all relevant banking regulations at each step of the process, and needs to manage its loan portfolio to maximize profitability. Policies are in place to aid in decision making at each step, and those policies are actively managed to optimize outcomes for the bank.

Business analysts at the bank model the loan application processes using the BPMN2 authoring tools (Process Designer) in BPM Suite:

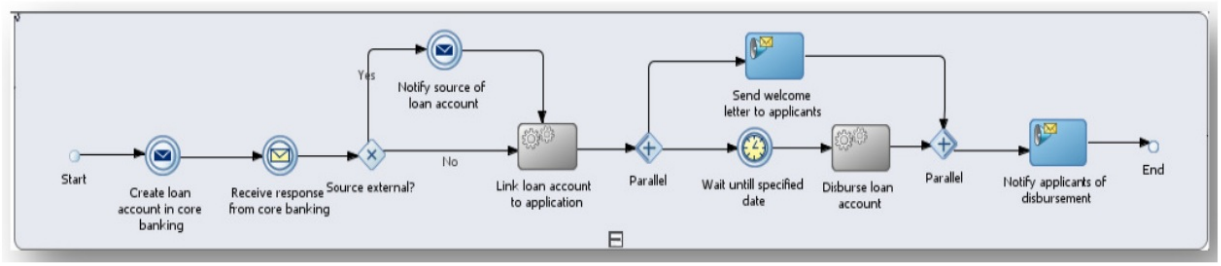


Figure 1.1. High-level loan application process flow

Business rules are developed with the rule authoring tools in BPM Suite to enforce policies and make decisions. Rules are linked with the process models to enforce the correct policies at each process step.

The bank's IT organization deploys the BPM Suite so that the entire loan application process can be automated.

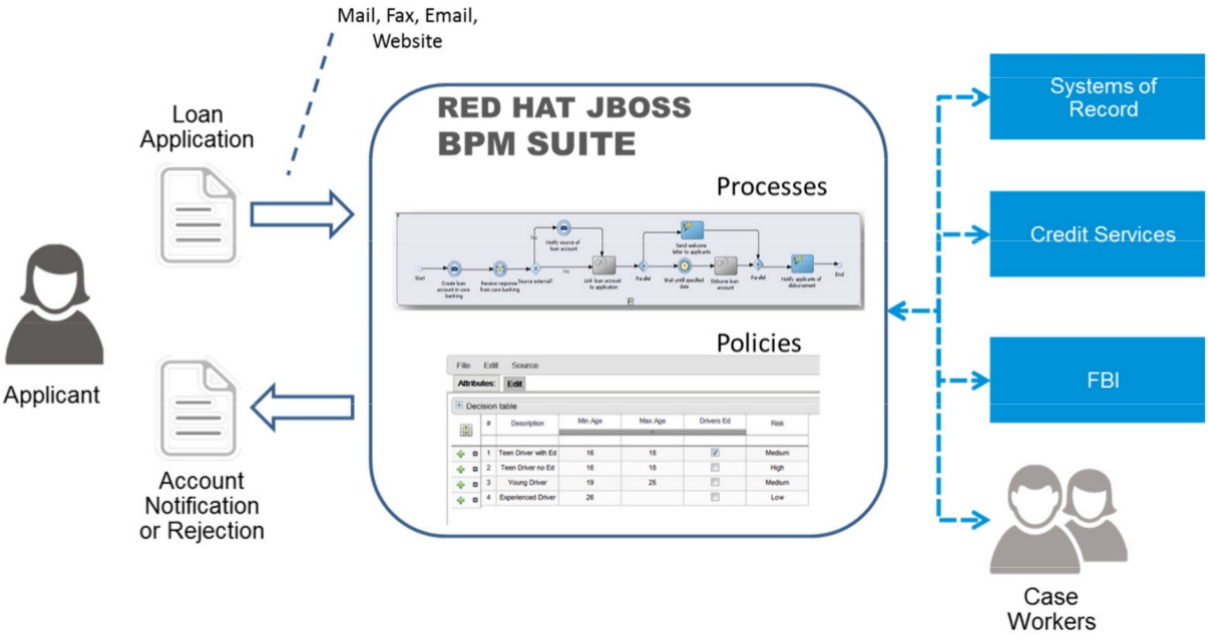


Figure 1.2. Loan Application Process Automation

The entire loan process and rules can be modified at any time by the bank's business analysts. The bank is able to maintain constant compliance with changing regulations, and is able to quickly introduce new loan products and improve loan policies in order to compete effectively and drive profitability.

Report a bug

1.4. INTEGRATED MAVEN DEPENDENCIES

Throughout the Red Hat JBoss BRMS and BPM Suite documentation, various code samples are presented with KIE API for the 6.1.x releases. These code samples will require Maven dependencies in the various **`pom.xml`** file and should be included like the following example:

```
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
```

```
    <version>1.1.1-redhat-2</version>
    <scope>compile</scope>
</dependency>
```

All the Red Hat JBoss related product dependencies can be found at the following location: Red Hat Maven Repository

NOTE

The set of online remote repositories is a technology preview source of components. As such, it is not in scope of patching and is supported only for use in development environment. Using the set of online repositories in production environment is a potential source of security vulnerabilities and is therefore not a supported use case. For more information see https://access.redhat.com/site/maven-repository.

Report a bug

CHAPTER 2. INTRODUCTION TO JBOSS RULES

2.1. THE BASICS

2.1.1. Business Rules Engine

Business Rules Engine is the rules engine provided as part of the Red Hat JBoss BPM Suite product. It is based on the community Drools Expert product.

Report a bug

2.1.2. The JBoss Rules Engine

The JBoss Rules engine is the computer program that applies rules and delivers Knowledge Representation and Reasoning (KRR) functionality to the developer.

Report a bug

2.1.3. Expert Systems

Expert systems are often used to refer to *production rules systems* or *Prolog-like systems*. Although acceptable, this comparison is technically incorrect as these are frameworks to build expert systems with, rather than expert systems themselves. An expert system develops once there is a model demonstrating the nature of the expert system itself; that is, a domain encompassing the aspects of an expert system which includes facilities for knowledge acquisition and explanation. *Mycin* is the most famous expert system.

Report a bug

2.1.4. Production Rules

A *production rule* is a two-part structure that uses first order logic to represent knowledge. It takes the following form:

```
when
 <conditions>
then
 <actions>
```

Report a bug

2.1.5. The Inference Engine

The *inference engine* is the part of the JBoss Rules engine which matches production facts and data to rules. It will then perform actions based on what it infers from the information. A production rules system's inference engine is *stateful* and is responsible for *truth maintenance*.

Report a bug

2.1.6. Production Memory

The **`production memory`** is where rules are stored.

Report a bug

2.1.7. Working Memory

The **`working memory`** is the part of the JBoss Rules engine where facts are asserted. From here, the facts can be modified or retracted.

Report a bug

2.1.8. Conflict Resolution Strategy

Conflict resolution is required when there are multiple rules on the agenda. As firing a rule may have side effects on the working memory, the rule engine needs to know in what order the rules should fire (for instance, firing ruleA may cause ruleB to be removed from the agenda).

Report a bug

2.1.9. Hybrid Rule Systems

A hybrid rule system pertains to using both forward-chaining and backward-chaining rule systems to process rules.

Report a bug

2.1.10. Reasoning Capabilities

JBoss Rules uses backward-chaining *reasoning capabilities* to help infer which rules to apply from the data.

Report a bug

CHAPTER 3. EXPERT SYSTEMS

3.1. PHREAK ALGORITHM

3.1.1. PHREAK Algorithm

The PHREAK algorithm used in JBoss Rules incorporates all of the existing code from Rete00. It is an enhancement of the Rete algorithm, and PHREAK incorporates the characteristics of a lazy, goal oriented algorithm where partial matching is aggressively delayed, and it also handles a large number of rules and facts. PHREAK is inspired by a number of algorithms including the following: LEAPS, RETE/UL and Collection-Oriented Match.

PHREAK has all the enhancements listed in the Rete00 algorithm. In addition, it also adds the following set of enhancements:

- Three layers of contextual memory: Node, Segment and Rule memories.
- Rule, segment, and node based linking.
- Lazy (delayed) rule evaluation.
- Stack based evaluations with pause and resume.
- Isolated rule evaluation.
- Set oriented propagations.

Report a bug

3.1.2. Three Layers of Contextual Memory

Rule evaluations in the PHREAK engine only occur while rules are linked. Before entering the beta network, the insert, update, and delete actions are queued for deployment. The rules are fired based on a simple heuristic; that is, based on the rule most likely to result in the firing of other rules, the heuristic selects the next rule for evaluation which delays the firing of other rules. Once all the inputs are populated, the rule becomes linked in. Next, a goal is created that represents the rule, and it is placed into a priority queue, which is ordered by salience. The queues themselves are associated with an AgendaGroup, and only the active AgendaGroup will inspect its queue by submitting the rule with the highest salience for evaluation. The actions of insert, update, delete phase and fireAllRules phase are achieved by this point. Accordingly, only the rule for which the goal was created is evaluated, and other potential rule evaluations are delayed. Node sharing is achieved through the process of segmentation while individual rules are evaluated.

PHREAK has 3 levels of memory. This allows for much more contextual understanding during evaluation of a Rule.

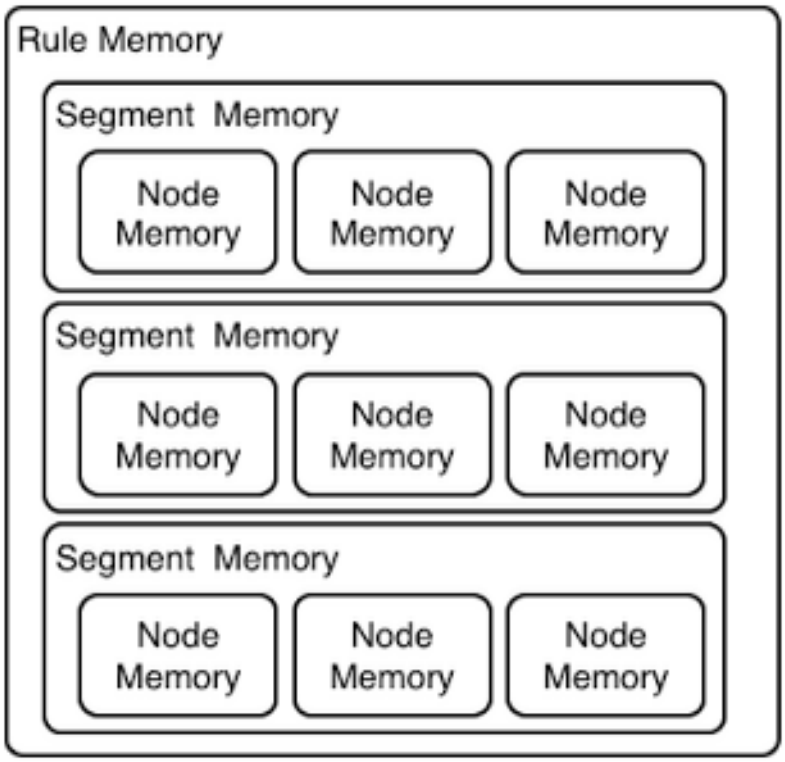


Figure 3.1. PHREAK 3 Layered memory system

Report a bug

3.1.3. Rule, Segment, and Node Based Linking

The Linking and Unlinking uses a layered bit mask system based on a network segmentation. When the rule network is built, segments are created for nodes that are shared by the same set of rules. A rule itself is made up from a path of segments; if there is no sharing, the rule will be a single segment. A bit-mask offset is assigned to each node in the segment. Another bit-mask (the layering) is assigned to each segment in the rule's path. When there is at least one input (data propagation), the node's bit is set to 'on'. When each node has its bit set to 'on,' the segment's bit is also set to 'on'. Conversely, if any node's bit is set to 'off', the segment is then also set to 'off'. If each segment in the rule's path is set to 'on', the rule is said to be linked in, and a goal is created to schedule the rule for evaluation.

The following examples illustrates the rule, segment, and node based linking in PHREAK algorithm.

Example 1: Single rule, no sharing
The example shows a single rule, with three patterns; A, B and C. It forms a single segment with bits 1, 2 and 4 for the nodes.

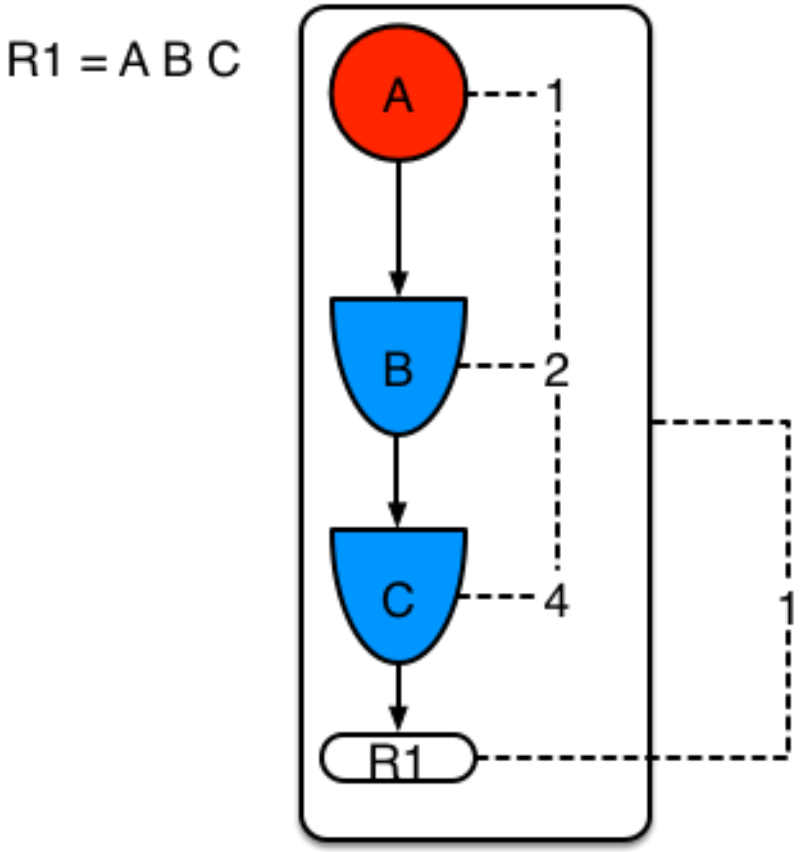


Figure 3.2. Example for a Single rule with no sharing

Example 2: Two rules with sharing

The following example demonstrates what happens when another rule is added that shares the pattern A. A is placed in its own segment, resulting in two segments per rule. These two segments form a path for their respective rules. The first segment is shared by both paths. When A is linked, the segment becomes linked; it then iterates each path the segment is shared by, setting the bit 1 to "on'. If B and C are later turned 'on', the second segment for path R1 is linked in; this causes bit 2 to be turned 'on' for R1. With bit 1 and bit 2 set to 'on' for R1, the rule is now linked and a goal is created to schedule the rule for later evaluation and firing.

When a rule is evaluated, its segments allow the results of matching to be shared. Each segment has a staging memory to queue all insert, update, and deletes for that segment. If R1 is evaluated, it will process A and result in a set of tuples. The algorithm detects that there is a segmentation split, and it will create peered tuples for each insert, update, and delete in the set and add them to R2's staging memory. These tuples will be merged with any existing staged tuples and wait for R2 to eventually be evaluated.

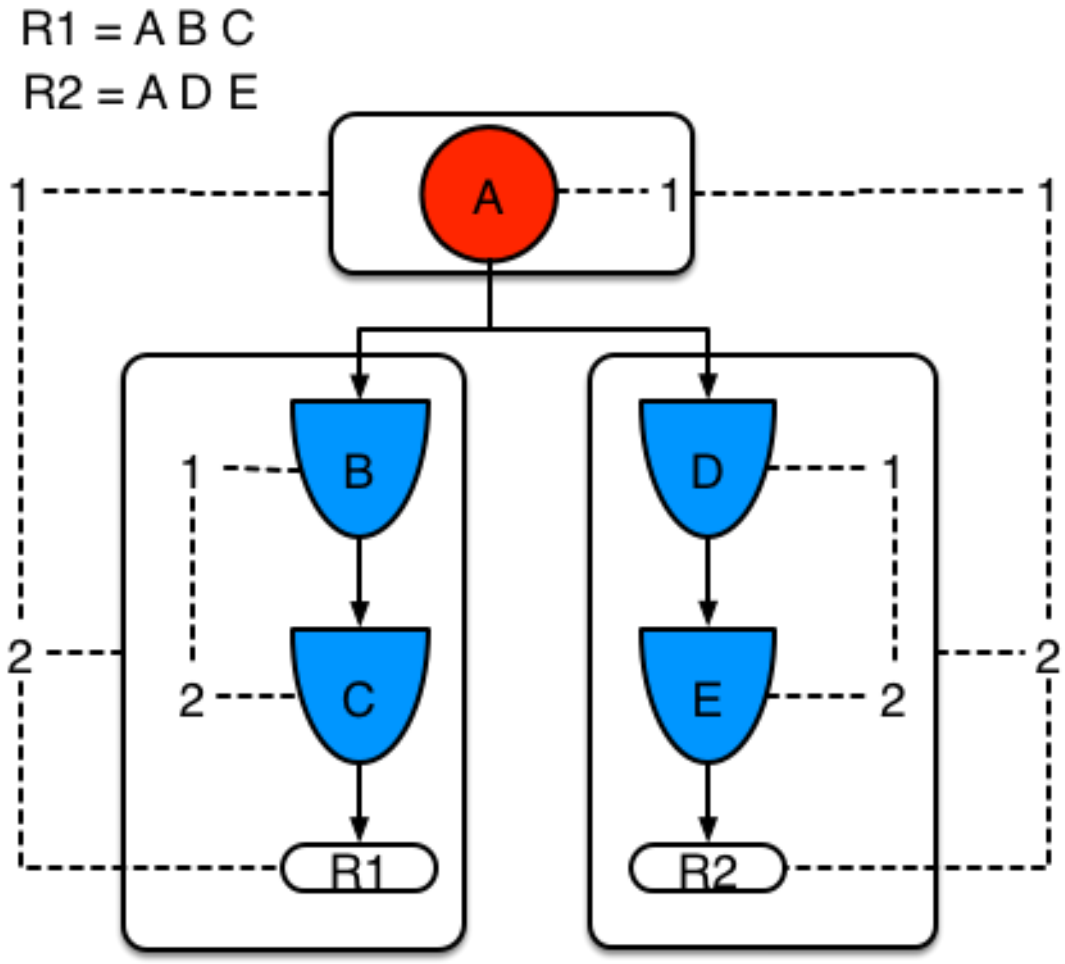


Figure 3.3. Example for two rules with sharing

Example 3: Three rules with sharing

The following example adds a third rule and demonstrates what happens when A and B are shared. Only the bits for the segments are shown this time. It demonstrating that R4 has 3 segments, R3 has 3 segments, and R1 has 2 segments. A and B are shared by R1, R3, and R4. Accordingly, D is shared by R3 and R4.

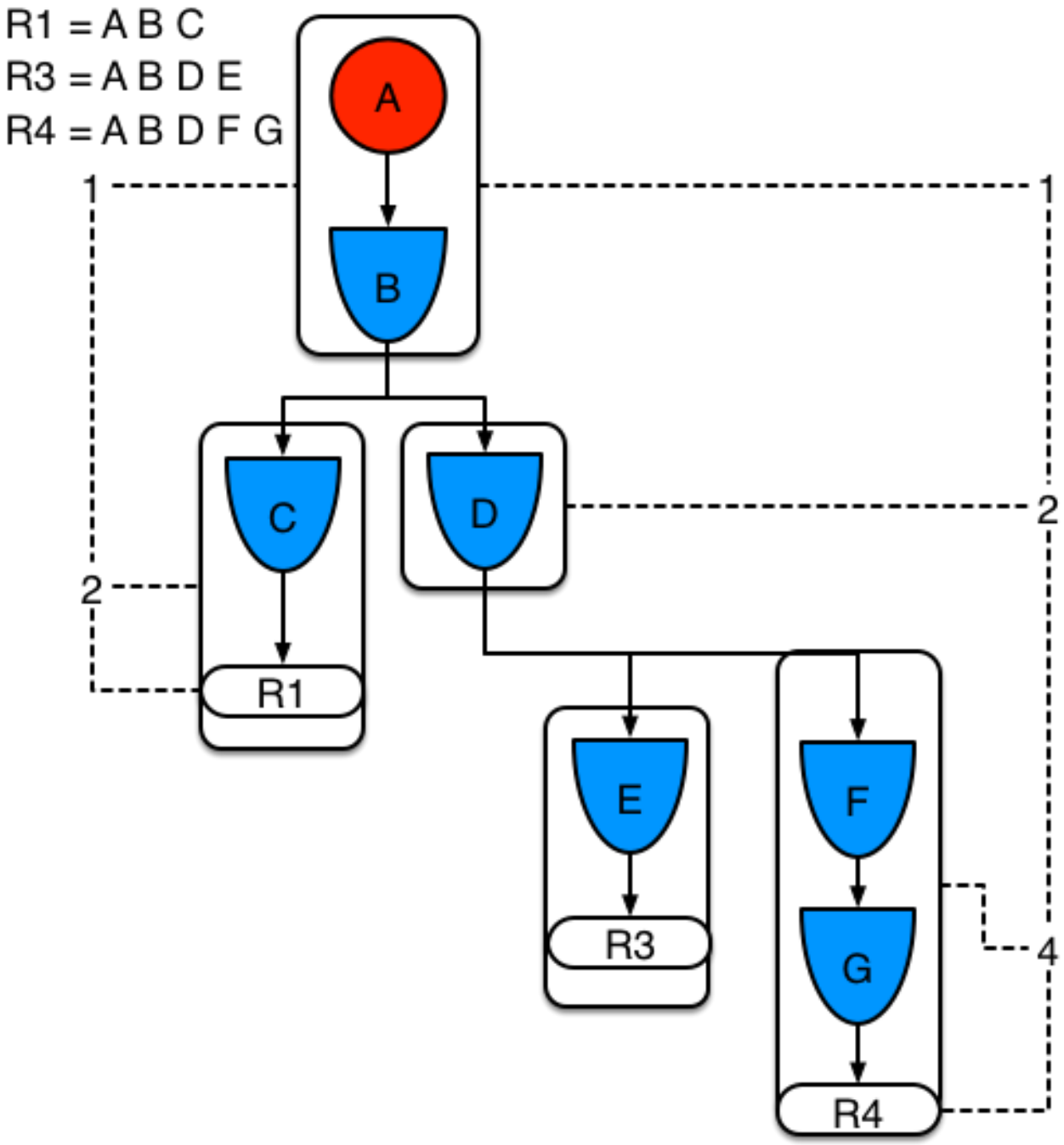


Figure 3.4. Example for Three rules with sharing

Example 4: Single rule, with sub-network and no sharing

Sub-networks are formed when a Not, Exists, or Accumulate node contain more than one element. In the following example, "B not(C)" forms the sub-network; note that "not(C)" is a single element and does not require a sub network, and it is merged inside of the Not node.

The sub-network gets its own segment. R1 still has a path of two segments. The sub-network forms another "inner" path. When the sub-network is linked in, it will link in the outer segment.

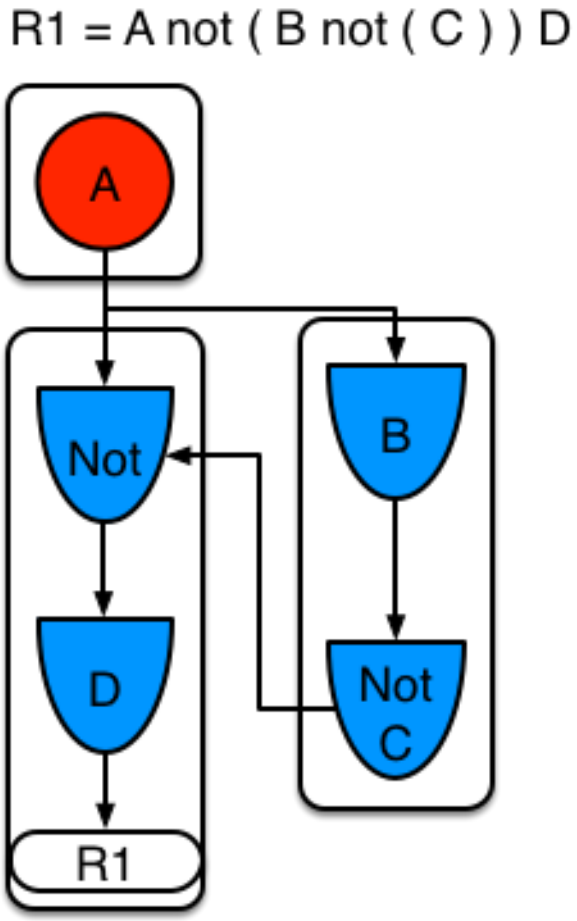


Figure 3.5. Example for a Single rule with sub-network and no sharing

Example 5: Two rules: one with a sub-network and sharing

The example shows that the sub-network nodes can be shared by a rule that does not have a sub-network. This results in the sub-network segment being split into two.

Note that nodes with constraints and accumulate nodes have special behaviour and can never unlink a segment; they are always considered to have their bits on.

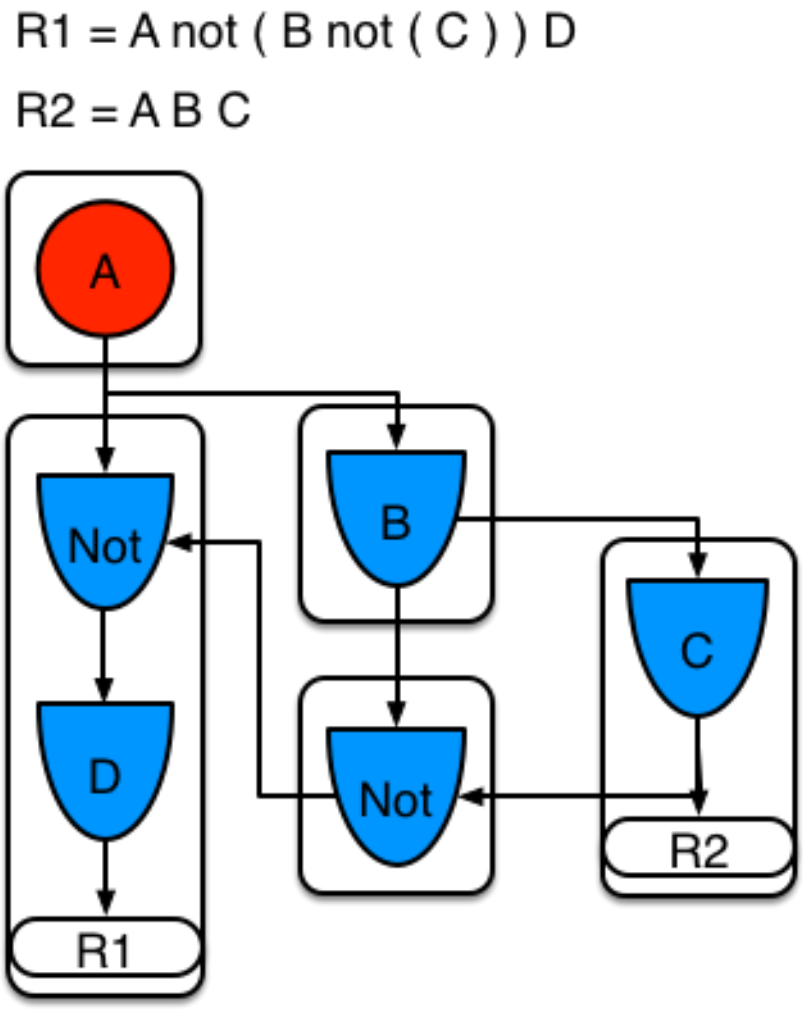


Figure 3.6. Example for Two rules, one with a sub-network and sharing

Report a bug

3.1.4. Delayed and Stack Based Evaluations

As discussed with the layering of segments in a rules path, a goal is created for rule evaluation based on the rule path segment's status. The same bit-mask technique is used to also track dirty nodes, segments, and rules. This process allows for a rule already linked to be scheduled for another evaluation if the previous evaluation labeled it dirty. This reevaluation ensures that no rule will ever evaluate partial matches. That is, it will not link rules with no data because they will not result in rule instances. Rete, however, will produce martial match attempt for all nodes, even empty ones.

While the incremental rule evaluation always starts from the root node, the dirty bit masks are used to allow nodes and segments that are not dirty to be skipped. This heuristic is fairy basic, and it uses at least one item of data per node. Future work attempts to delay linking even further, such as *arc consistency*, will determine if rule instance will fire regardless of matching.

During the phase of segmentation, other rules are delayed and evaluated. Note that all rule evaluations are incremental, and they will not waste work recomputing matches that have already been produced. The evaluations algorithm is stack based instead of method recursive. The evaluations can be paused and resumed at any time by using a StackEntry to represent the current node being evaluated. A StackEntry is created for the outer path segment and sub-network segment of a rule evaluation. When the evaluation reaches a sub-network, the sub-network segment is evaluated first. When the set reaches the end of the sub-networkd path, it is merged into a staging list for the outer node it reacts to. The

previous StackEntry is then resumed, it can then process the results of the sub-network. This process benefits from all the work being processed in batch before it is propagated to the child node, which is more efficient for accumulate nodes.

Report a bug

3.1.5. Propagations and Isolated Rules

PHREAK propagation is set oriented (or collection-oriented) instead of tuple oriented. For the evaluated rule, PHREAK will visit the first node and process all queued insert, update, and delete actions. The results are added to a set and the set is propagated to the child node. In the child node, all queued insert, update, and deletes are processed, which adds the results to the same set. Once the processing has finished, the set is propagated to the next child node, and this process repeats until the terminal node is reached. These actions create a single pass, pipeline type effect, that is isolated to the current rule being evaluated. Accordingly, this leads to the creation of a batch process effect which provides performance advantages for certain rule constructs such as sub-networks with accumulates.

As mentioned prior, the process of StackEntry adds the benefit of efficient batch-based work before propagating to the child node. This same stack system is efficient for backward chaining. When a rule evaluation reaches a query node, it pauses the current evaluation by placing it on the stack. The query is then evaluated and produces a result set, which is saved in a memory location for the resumed StackEntry to pick up and propagate to the child node. If the query itself called other queries, the process would repeat; accordingly, the current query would be paused, and a new evaluation would be created for the current query node.

Report a bug

3.1.6. RETE to PHREAK

In general, a single rule will not evaluate any faster with PHREAK than it does with RETE. Both a given rule and the same data set, which uses a root context object to enable and disable matching, attempt the same amount of matches, produce the same number of rule instances, and take roughly the same time. However, variations occur for the use case with subnetworks and accumulates.

PHREAK is considered more forgiving than RETE for poorly written rule bases; it also displays a more graceful degradation of performance as the number of rules and complexity increases.

RETE produces partial matches for rules that do not have data in all the joints; PHREAK avoids any partial matching. Accordingly, PHREAK's performance will not slow down as your system grows.

AgendaGroups did not help RETE performance, that is, all rules were evaluated at all times, regardless of the group. This also occurred with salience, which relied on context objects to limit matching attempts. PHREAK only evaluates rules for the active AgendaGroup. Within that active group, PHREAK will avoid evaluation of rules (via salience) that do not result in rule instance firings. With PHREAK, AgendaGroups and salience now become useful performance tools.

NOTE

With PHREAK, root context objects are no longer needed as they may be counter productive to performance. That is, they may force the flushing and recreation of matches for rules.

Report a bug

3.1.7. Switching Between PHREAK and ReteOO

Switching Using System Properties

For users to switch between the PHREAK algorithm and the ReteOO algorithm, the **`drools.ruleEngine`** system properties need to be edited with the following values:

```
drools.ruleEngine=phreak
```

or

```
drools.ruleEngine=reteoo
```

The previous value of "phreak" is the default value for 6.0.

The Maven GAV (Group, Artifact, Version) value for ReteOO is depicted below:

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-reteoo</artifactId>
  <version>${drools.version}</version>
</dependency>
```

Switching in KieBaseConfiguration

When creating a particular KieBase, it is possible to specify the rule engine algorithm in the KieBaseConfiguration:

```
import org.kie.api.KieBase;
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
...
```

```
KieServices kservices = KieServices.Factory.get();
KieBaseConfiguration kconfig =
kieServices.Factory.get().newKieBaseConfiguration();

// you can either specify phreak (default)
kconfig.setOption(RuleEngineOption.PHREAK);

// or legacy ReteOO
kconfig.setOption(RuleEngineOption.RETEOO);

// and then create a KieBase for the selected algorithm
(getKieClasspathContainer() is just an example)
KieContainer container = kservices.getKieClasspathContainer();
KieBase kbase = container.newKieBase(kieBaseName, kconfig);
```

NOTE

Take note that switching to ReteOO requires **`drools-reteoo-(version).jar`** to exist on the classpath. If not, the JBoss Rules Engine reverts back to PHREAK and issues a warning. This applies for switching with KieBaseConfiguration and System Properties.

Report a bug

3.2. RETE ALGORITHM

3.2.1. ReteOO

The Rete implementation used in JBoss Rules is called *ReteOO*. It is an enhanced and optimized implementation of the Rete algorithm specifically for object-oriented systems. The Rete Algorithm has now been deprecated, and PHREAK is an enhancement of Rete. However, Rete can still be used by developers. This section describes how the Rete Algorithm functions.

Report a bug

3.2.2. The Rete Root Node

Figure 3.7. ReteNode

When using ReteOO, the root node is where all objects enter the network. From there, it immediately goes to the ObjectTypeNode.

Report a bug

3.2.3. The ObjectTypeNode

The *ObjectTypeNode* helps to reduce the workload of the rules engine. If there are several objects and the rules engine tried to evaluate every single node against every object, it would waste a lot of cycles. To make things efficient, the ObjectTypeNode is used so the engine only passes objects to the nodes that match the object's type. This way, if an application asserts a new Account, it won't propagate to the nodes for the Order object.

In JBoss Rules, an object which has been asserted will retrieve a list of valid ObjectTypesNodes via a lookup in a HashMap from the object's Class. If this list doesn't exist it scans all the ObjectTypeNodes finding valid matches which it caches in the list. This enables JBoss Rules to match against any Class type that matches with an `instanceof` check.

Report a bug

3.2.4. AlphaNodes

AlphaNodes are used to evaluate literal conditions. When a rule has multiple literal conditions for a single object type, they are linked together. This means that if an application asserts an Account object, it must first satisfy the first literal condition before it can proceed to the next AlphaNode.

AlphaNodes are propagated using ObjectTypeNodes.

Report a bug

3.2.5. Hashing

JBoss Rules uses *hashing* to extend Rete by optimizing the propagation from ObjectTypeNode to AlphaNode. Each time an AlphaNode is added to an ObjectTypeNode, it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectType node, rather than propagating to each AlphaNode, it can instead retrieve the correct AlphaNode from the HashMap, thereby avoiding unnecessary literal checks.

When facts enter from one side, you can do a hash lookup returning potentially valid candidates (referred to as indexing). At any point a valid join is found, the Tuple will join with the Object (referred to as a partial match) and then propagate to the next node.

Report a bug

3.2.6. BetaNodes

BetaNodes are used to compare two objects and their fields. The objects may be the same or different types.

Report a bug

3.2.7. Alpha Memory

Alpha memory refers to the left input on a BetaNode. In JBoss Rules, this input remembers all incoming objects.

Report a bug

3.2.8. Beta Memory

Beta memory is the term used to refer to the right input of a BetaNode. It remembers all incoming tuples.

Report a bug

3.2.9. Lookups with BetaNodes

When facts enter from one side, you can do a hash lookup returning potentially valid candidates (referred to as indexing). At any point a valid join is found, the Tuple will join with the Object (referred to as a partial match) and then propagate to the next node.

Report a bug

3.2.10. LeftInputNodeAdapters

A *LeftInputNodeAdapter* takes an Object as an input and propagates a single Object Tuple.

Report a bug

3.2.11. Terminal Nodes

Terminal nodes are used to indicate when a single rule has matched all its conditions (that is, the rule has a full match). A rule with an 'or' conditional disjunctive connective will result in a sub-rule generation for each possible logically branch. Because of this, one rule can have multiple terminal nodes.

Report a bug

3.2.12. Node Sharing

Node sharing is used to prevent unnecessary redundancy. Because many rules repeat the same patterns, node sharing allows users to collapse those patterns so they do not have to be reevaluated for every single instance.

The following two rules share the first pattern but not the last:

```
rule
when
    Cheese( $cheddar : name == "cheddar" )
    $person: Person( favouriteCheese == $cheddar )
then
    System.out.println( $person.getName() + " likes cheddar" );
end
```

```
rule
when
    Cheese( $cheddar : name == "cheddar" )
    $person : Person( favouriteCheese != $cheddar )
then
    System.out.println( $person.getName() + " does not like cheddar" );
end
```

The complied Rete network displayed below shows that the alpha node is shared but the beta nodes are not. Each beta node has its own **`TerminalNode`**.

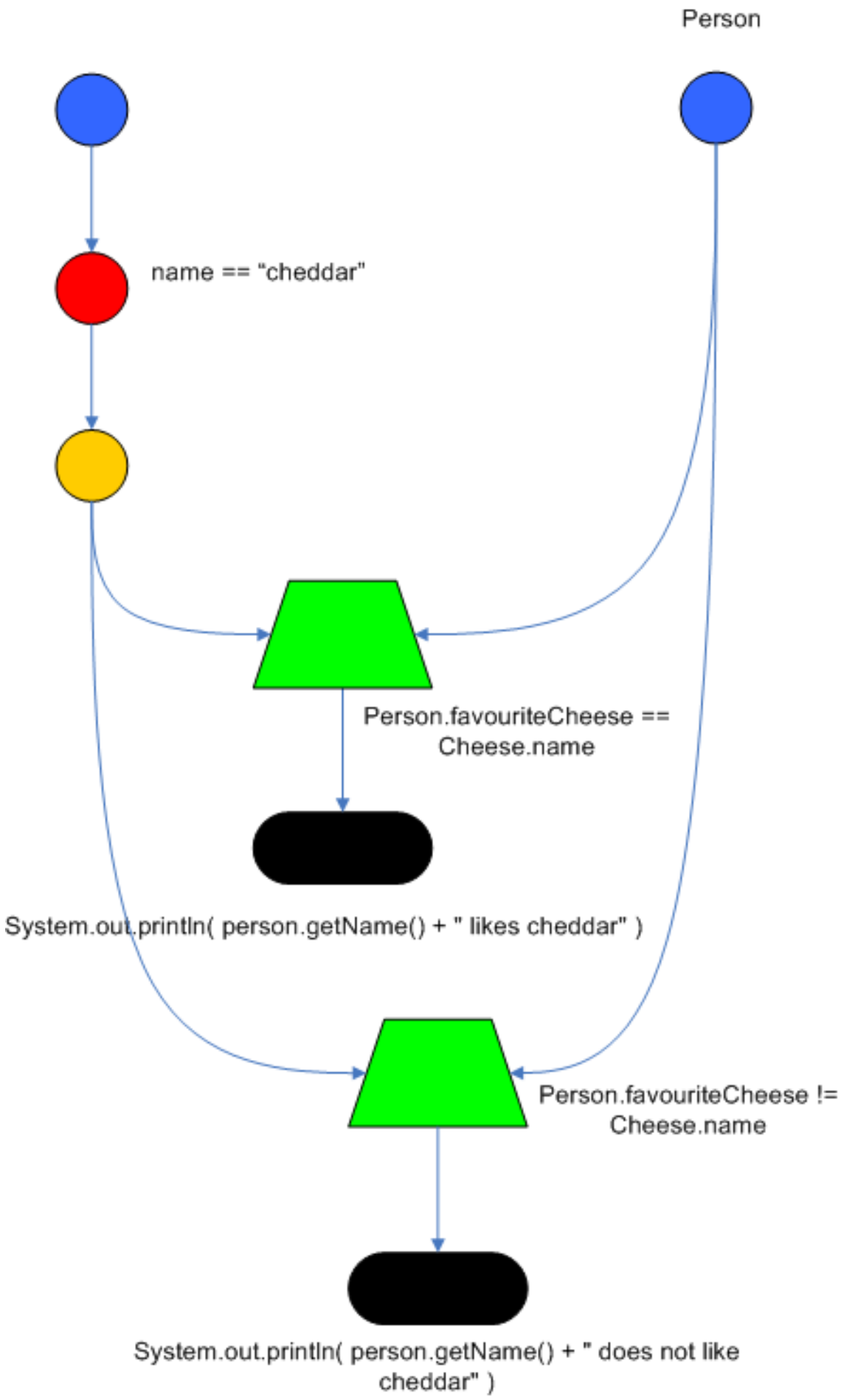


Figure 3.8. Node Sharing

Report a bug

3.2.13. Join Attempts

Each successful join attempt in RETE produces a tuple (or token, or partial match) that will be propagated to the child nodes. For this reason, it is characterized as a tuple oriented algorithm. For each child node that it reaches, it will attempt to join with the other side of the node; moreover, each successful join attempt will be propagated straight away. This creates a descent recursion effect that targets from the point of entry in the beta network and spreads to all the reachable leaf nodes.

Report a bug

3.3. STRONG AND LOOSE COUPLING

3.3.1. Loose Coupling

Loose coupling involves "loosely" linking rules so that the execution of one rule will not lead to the execution of another.

Generally, a design exhibiting loose coupling is preferable because it allows for more flexibility. If the rules are all strongly coupled, they are likely to be inflexible. More significantly, it indicates that deploying a rule engine is overkill for the situation.

Report a bug

3.3.2. Strong Coupling

Strong coupling is a way of linking rules. If rules are strongly-coupled, it means executing one rule will directly result in the execution of another. In other words, there is a clear chain of logic. (A clear chain can be hard-coded, or implemented using a decision tree.)

Report a bug

3.4. ADVANTAGES OF A RULE ENGINE

3.4.1. Declarative Programming

Declarative programming refers to the way the rule engine allows users to declare "what to do" as opposed to "how to do it". The key advantage of this point is that using rules can make it easy to express solutions to difficult problems and consequently have those solutions verified. Rules are much easier to read than code.

Report a bug

3.4.2. Logic and Data Separation

Logic and Data separation refers to the process of de-coupling logic and data components. Using this method, the logic can be spread across many domain objects or controllers and it can all be organized in one or more discrete rules files.

Report a bug

3.4.3. KIE Base

A kie base is a collection of rules which have been compiled by the **`KieBuilder`**. It is a repository of all the application's knowledge definitions. It may contain rules, processes, functions, and type models. The KieBase itself does not contain instance data (known as facts). Instead, sessions are created from the KieBase into which data can be inserted and where process instances may be started. It is recommended that KieBases be cached where possible to allow for repeated session creation.

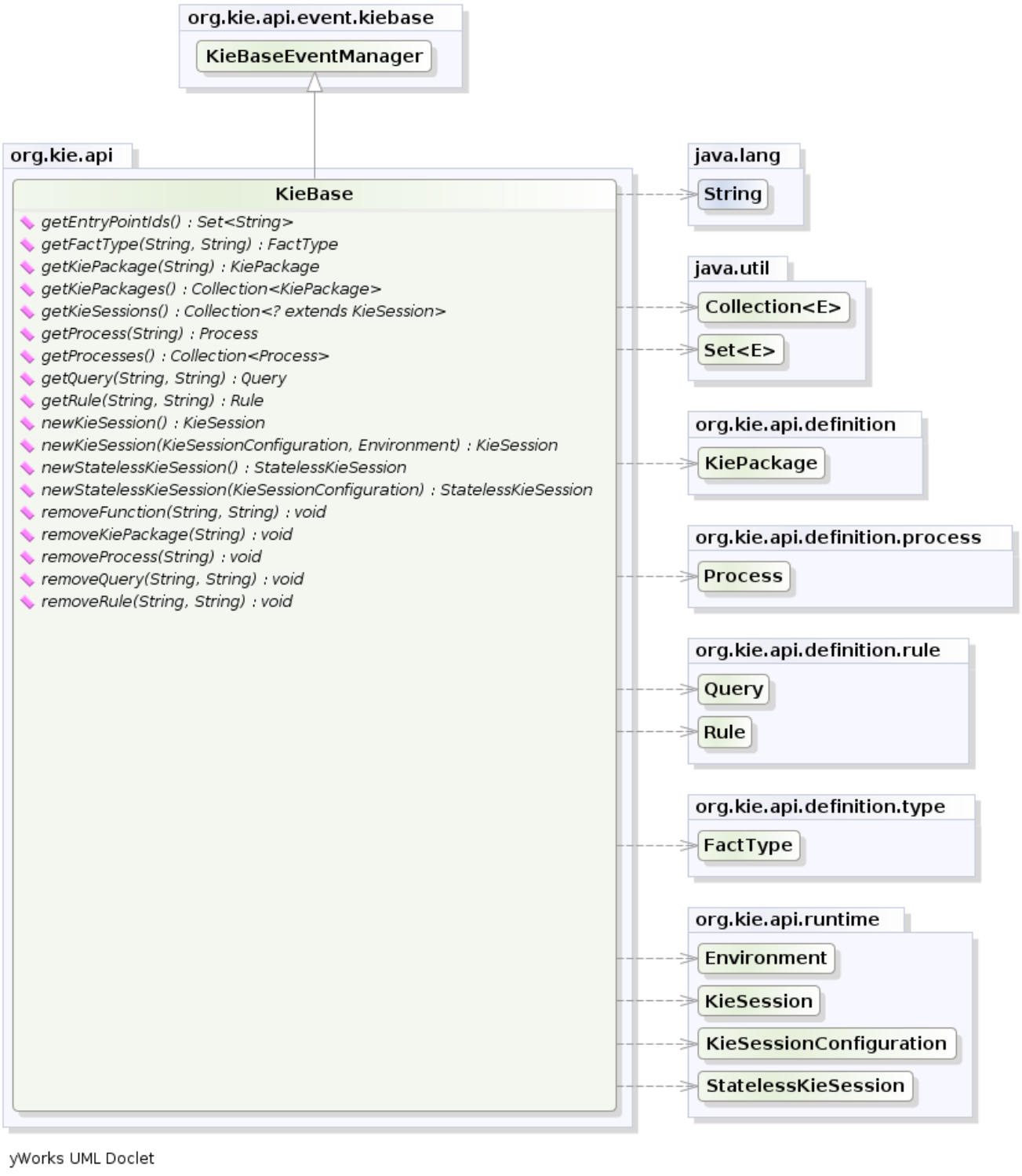


Figure 3.9. KieBase

Report a bug

CHAPTER 4. MAVEN

4.1. LEARN ABOUT MAVEN

4.1.1. About Maven

Apache Maven is a distributed build automation tool used in Java application development to build and manage software projects. Maven uses configuration XML files called POM (Project Object Model) to define project properties and manage the build process. POM files describe the project's module and component dependencies, build order, and targets for the resulting project packaging and output. This ensures that projects are built in a correct and uniform manner.

Maven uses repositories to store Java libraries, plug-ins, and other build artifacts. Repositories can be either local or remote. A local repository is a download of artifacts from a remote repository cached on a local machine. A remote repository is any other repository accessed using common protocols, such as **`http://`** when located on an HTTP server, or **`file://`** when located on a file server. The default repository is the public remote Maven 2 Central Repository.

Configuration of Maven is performed by modifying the **`settings.xml`** file. You can either configure global Maven settings in the ***`M2_HOME`*`/conf/settings.xml`** file, or user-level settings in the ***`USER_HOME`*`/.m2/settings.xml`** file.

For more information about Maven, see Welcome to Apache Maven.

For more information about Maven repositories, see Apache Maven Project - Introduction to Repositories.

For more information about Maven POM files, see the Apache Maven Project POM Reference.

Report a bug

4.1.2. About the Maven POM File

The Project Object Model, or POM, file is a configuration file used by Maven to build projects. It is an XML file that contains information about the project and how to build it, including the location of the source, test, and target directories, the project dependencies, plug-in repositories, and goals it can execute. It can also include additional details about the project including the version, description, developers, mailing list, license, and more. A **`pom.xml`** file requires some configuration options and will default all others. See Section 4.1.3, “Minimum Requirements of a Maven POM File” for details.

The schema for the **`pom.xml`** file can be found at http://maven.apache.org/maven-v4_0_0.xsd.

For more information about POM files, see the Apache Maven Project POM Reference.

Report a bug

4.1.3. Minimum Requirements of a Maven POM File

Minimum requirements

The minimum requirements of a **`pom.xml`** file are as follows:

- project root

- modelVersion
- groupId - the id of the project's group
- artifactId - the id of the artifact (project)
- version - the version of the artifact under the specified group

Sample pom.xml file

A basic **pom.xml** file might look like this:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jboss.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

Report a bug

4.1.4. About the Maven Settings File

The Maven **settings.xml** file contains user-specific configuration information for Maven. It contains information that should not be distributed with the **pom.xml** file, such as developer identity, proxy information, local repository location, and other settings specific to a user.

There are two locations where the **settings.xml** can be found.

In the Maven install

The settings file can be found in the ***M2_HOME*/conf/** directory. These settings are referred to as **global** settings. The default Maven settings file is a template that can be copied and used as a starting point for the user settings file.

In the user's install

The settings file can be found in the ***USER_HOME*/.m2/** directory. If both the Maven and user **settings.xml** files exist, the contents are merged. Where there are overlaps, the user's **settings.xml** file takes precedence.

The following is an example of a Maven **settings.xml** file:

```
<settings>
    <profiles>
        <profile>
            <id>my-profile</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <repositories>
                <repository>
                    <id>fusesource</id>

<url>http://repo.fusesource.com/nexus/content/groups/public/</url>
                    <snapshots>
```

```
                    <enabled>false</enabled>
                </snapshots>
                <releases>
                    <enabled>true</enabled>
                </releases>
            </repository>
            ...
        </repositories>
    </profile>
</profiles>
...
</settings>
```

The schema for the **settings.xml** file can be found at http://maven.apache.org/xsd/settings-1.0.0.xsd.

Report a bug

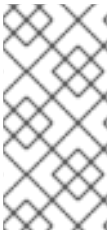
4.1.5. KIE Plugin

The KIE plugin for Maven ensures that artifact resources are validated and pre-compiled, it is recommended that this is used at all times. To use the plugin simply add it to the build section of the Maven pom.xml

Example 4.1. Adding the KIE plugin to a Maven pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.kie</groupId>
      <artifactId>kie-maven-plugin</artifactId>
      <version>${project.version}</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

Building a KIE module without the Maven plugin will copy all the resources, as is, into the resulting JAR. When that JAR is loaded by the runtime, it will attempt to build all the resources then. If there are compilation issues it will return a null KieContainer. It also pushes the compilation overhead to the runtime. In general this is not recommended, and the Maven plugin should always be used.



NOTE

For compiling decision tables and processes, appropriate dependencies must be added either to project dependencies (as compile scope) or as plugin dependencies. For decision tables the dependency is **org.drools:drools-decisiontables** and for processes **org.jbpm:jbpm-bpmn2**.

Report a bug

4.1.6. Maven Versions and Dependencies

Maven supports a number of mechanisms to manage versioning and dependencies within applications. Modules can be published with specific version numbers, or they can use the SNAPSHOT suffix. Dependencies can specify version ranges to consume, or take avantage of SNAPSHOT mechanism.

If you always want to use the newest version, Maven has two keywords you can use as an alternative to version ranges. You should use these options with care as you are no longer in control of the plugins/dependencies you are using.

When you depend on a plugin or a dependency, you can use the a version value of LATEST or RELEASE. LATEST refers to the latest released or snapshot version of a particular artifact, the most recently deployed artifact in a particular repository. RELEASE refers to the last non-snapshot release in the repository. In general, it is not a best practice to design software which depends on a non-specific version of an artifact. If you are developing software, you might want to use RELEASE or LATEST as a convenience so that you don't have to update version numbers when a new release of a third-party library is released. When you release software, you should always make sure that your project depends on specific versions to reduce the chances of your build or your project being affected by a software release not under your control. Use LATEST and RELEASE with caution, if at all.

Here's an example illustrating the various options. In the Maven repository, com.foo:my-foo has the following metadata:

```
<metadata>
  <groupId>com.foo</groupId>
  <artifactId>my-foo</artifactId>
  <version>2.0.0</version>
  <versioning>
    <release>1.1.1</release>
    <versions>
      <version>1.0</version>
      <version>1.0.1</version>
      <version>1.1</version>
      <version>1.1.1</version>
      <version>2.0.0</version>
    </versions>
    <lastUpdated>20090722140000</lastUpdated>
  </versioning>
</metadata>
```

If a dependency on that artifact is required, you have the following options (other version ranges can be specified of course, just showing the relevant ones here): Declare an exact version (will always resolve to 1.0.1):

```
<version>[1.0.1]</version>
```

Declare an explicit version (will always resolve to 1.0.1 unless a collision occurs, when Maven will select a matching version):

```
<version>1.0.1</version>
```

Declare a version range for all 1.x (will currently resolve to 1.1.1):

```
<version>[1.0.0,2.0.0)</version>
```

Declare an open-ended version range (will resolve to 2.0.0):

```
<version>[1.0.0,)</version>
```

Declare the version as LATEST (will resolve to 2.0.0):

```
<version>LATEST</version>
```

Declare the version as RELEASE (will resolve to 1.1.1):

```
<version>RELEASE</version>
```

Note that by default your own deployments will update the "latest" entry in the Maven metadata, but to update the "release" entry, you need to activate the "release-profile" from the Maven super POM. You can do this with either "-Prelease-profile" or "-DperformRelease=true"

Report a bug

4.1.7. Remote Repository Setup

The maven settings.xml is used to configure Maven execution.

The settings.xml file can be located in 3 locations, the actual settings used is a merge of those 3 locations.

- The Maven install: $M2_HOME/conf/settings.xml
- A user's install: ${user.home}/.m2/settings.xml
- Folder location specified by the system propert kie.maven.settings.custom

The settings.xml is used to specify the location of remote repositories. It is important that you activate the profile that specifies the remote repository, typically this can be done using "activeByDefault":

```
<profiles>
  <profile>
    <id>profile-1</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    ...
  </profile>
</profiles>
```

Report a bug

CHAPTER 5. KIE API

5.1. KIE FRAMEWORK

5.1.1. KIE Systems

The various aspects, or life cycles, of KIE systems in the JBoss Rules environment can typically be broken down into the following labels:

- **Author**
  - Knowledge author using UI metaphors such as DRL, BPMN2, decision tables, and class models.
- **Build**
  - Builds the authored knowledge into deployable units.
  - For KIE this unit is a JAR.
- **Test**
  - Test KIE knowledge before it is deployed to the application.
- **Deploy**
  - Deploys the unit to a location where applications may use them.
  - KIE uses Maven style repository.
- **Utilize**
  - The loading of a JAR to provide a KIE session (KieSession), for which the application can interact with.
  - KIE exposes the JAR at runtime via a KIE container (KieContainer).
  - KieSessions, for the runtimes to interact with, are created from the KieContainer.
- **Run**
  - System interaction with the KieSession, via API.
- **Work**
  - User interaction with the KieSession, via command line or UI.
- **Manage**
  - Manage any KieSession or KieContainer.

Report a bug

5.1.2. KieBase

A **`KieBase`** is a repository of all the application's knowledge definitions. It contains rules, processes, functions, and type models. The **`KieBase`** itself does not contain data; instead, sessions are created

from the **`KieBase`** into which data can be inserted, and, ultimately, process instances may be started. Creating the **`KieBase`** can be quite heavy, whereas session creation is very light; therefore, it is recommended that **`KieBase`** be cached where possible to allow for repeated session creation. Accordingly, the caching mechanism is automatically provided by the **`KieContainer`**.

Table 5.1. kbase Attributes

Attribute name	Default value	Admitted values	Meaning
name	none	any	The name which retrieves the KieBase from the KieContainer. This is the only mandatory attribute.
includes	none	any comma separated list	A comma separated list of other KieBases contained in this kmodule. The artifacts of all these KieBases will also be included in this one.
packages	all	any comma separated list	By default all the JBoss Rules artifacts under the resources folder, at any level, are included into the KieBase. This attribute allows to limit the artifacts that will be compiled in this KieBase to only the ones belonging to the list of packages.
default	false	true, false	Defines if this KieBase is the default one for this module, so it can be created from the KieContainer without passing any name to it. There can be at most one default KieBase in each module.

Attribute name	Default value	Admitted values	Meaning
equalsBehavior	identity	identity, equality	Defines the behavior of JBoss Rules when a new fact is inserted into the Working Memory. With identity it always create a new FactHandle unless the same object isn't already present in the Working Memory, while with equality only if the newly inserted object is not equal (according to its equal method) to an already existing fact.
eventProcessingMode	cloud	cloud, stream	When compiled in cloud mode the KieBase treats events as normal facts, while in stream mode allow temporal reasoning on them.
declarativeAgenda	disabled	disabled, enabled	Defines if the Declarative Agenda is enabled or not.

Report a bug

5.1.3. KieSession

The **`KieSession`** stores and executes on runtime data. It is created from the **`KieBase`**, or, more easily, created directly from the **`KieContainer`** if it has been defined in the kmodule.xml file

Table 5.2. ksession Attributes

Attribute name	Default value	Admitted values	Meaning
name	none	any	Unique name of this KieSession. Used to fetch the KieSession from the KieContainer. This is the only mandatory attribute.

Attribute name	Default value	Admitted values	Meaning
type	stateful	stateful, stateless	A stateful session allows to iteratively work with the Working Memory, while a stateless one is a one-off execution of a Working Memory with a provided data set.
default	false	true, false	Defines if this KieSession is the default one for this module, so it can be created from the KieContainer without passing any name to it. In each module there can be at most one default KieSession for each type.
clockType	realtime	realtime, pseudo	Defines if events timestamps are determined by the system clock or by a psuedo clock controlled by the application. This clock is specially useful for unit testing temporal rules.
beliefSystem	simple	simple, jtms, defeasible	Defines the type of belief system used by the KieSession.

Report a bug

5.1.4. KieFileSystem

It is also possible to define the **`KieBase`**s and **`KieSession`**s belonging to a KieModule programmatically instead of the declarative definition in the kmodule.xml file. The same programmatic API also allows in explicitly adding the file containing the Kie artifacts instead of automatically read them from the resources folder of your project. To do that it is necessary to create a **`KieFileSystem`**, a sort of virtual file system, and add all the resources contained in your project to it.

Like all other Kie core components you can obtain an instance of the **`KieFileSystem`** from the **`KieServices`**. The kmodule.xml configuration file must be added to the filesystem. This is a mandatory step. Kie also provides a convenient fluent API, implemented by the **`KieModuleModel`**, to programmatically create this file.

To do this in practice it is necessary to create a **KieModuleModel** from the **KieServices**, configure it with the desired **KieBase**s and **KieSession**s, convert it in XML and add the XML to the **KieFileSystem**. This process is shown by the following example:

Example 5.1. Creating a kmodule.xml programmatically and adding it to a KieFileSystem

```
KieServices kieServices = KieServices.Factory.get();
KieModuleModel kieModuleModel = kieServices.newKieModuleModel();

KieBaseModel kieBaseModel1 = kieModuleModel.newKieBaseModel( "KBase1 ")
        .setDefault( true )
        .setEqualsBehavior( EqualityBehaviorOption.EQUALITY )
        .setEventProcessingMode( EventProcessingOption.STREAM );

KieSessionModel ksessionModel1 = kieBaseModel1.newKieSessionModel(
"KSession1" )
        .setDefault( true )
        .setType( KieSessionModel.KieSessionType.STATEFUL )
        .setClockType( ClockTypeOption.get("realtime") );

KieFileSystem kfs = kieServices.newKieFileSystem();
```

At this point it is also necessary to add to the **KieFileSystem**, through its fluent API, all others Kie artifacts composing your project. These artifacts have to be added in the same position of a corresponding usual Maven project.

Report a bug

5.1.5. KieResources

Example 5.2. Adding Kie artifacts to a KieFileSystem

```
KieFileSystem kfs = ...
kfs.write( "src/main/resources/KBase1/ruleSet1.drl",
stringContainingAValidDRL )
        .write( "src/main/resources/dtable.xls",
                kieServices.getResources().newInputStreamResource(
dtableFileStream ) );
```

This example shows that it is possible to add the Kie artifacts both as plain Strings and as **Resource**s. In the latter case the **Resource**s can be created by the **KieResources** factory, also provided by the **KieServices**. The **KieResources** provides many convenient factory methods to convert an **InputStream**, a **URL**, a **File**, or a **String** representing a path of your file system to a **Resource** that can be managed by the **KieFileSystem**.

Normally the type of a **Resource** can be inferred from the extension of the name used to add it to the **KieFileSystem**. However it also possible to not follow the Kie conventions about file extensions and explicitly assign a specific **ResourceType** to a **Resource** as shown below:

Example 5.3. Creating and adding a Resource with an explicit type

```
KieFileSystem kfs = ...
kfs.write( "src/main/resources/myDrl.txt",
           kieServices.getResources().newInputStreamResource( drlStream
)
                      .setResourceType(ResourceType.DRL) );
```

Add all the resources to the **KieFileSystem** and build it by passing the **KieFileSystem** to a **KieBuilder**

When the contents of a **KieFileSystem** are successfully built, the resulting **KieModule** is automatically added to the **KieRepository**. The **KieRepository** is a singleton acting as a repository for all the available **KieModule**s.

Report a bug

5.2. BUILDING WITH MAVEN

5.2.1. The kmodule

BRMS 6.0 introduces a new configuration and convention approach to building knowledge bases instead of using the programmatic builder approach in 5.x. The builder is still available to fall back on, as it's used for the tooling integration.

Building now uses Maven, and aligns with Maven practices. A KIE project or module is simply a Maven Java project or module; with an additional metadata file META-INF/kmodule.xml. The kmodule.xml file is the descriptor that selects resources to knowledge bases and configures those knowledge bases and sessions. There is also alternative XML support via Spring and OSGi BluePrints.

While standard Maven can build and package KIE resources, it will not provide validation at build time. There is a Maven plugin which is recommended to use to get build time validation. The plugin also generates many classes, making the runtime loading faster too.

KIE uses defaults to minimise the amount of configuration. With an empty kmodule.xml being the simplest configuration. There must always be a kmodule.xml file, even if empty, as it's used for discovery of the JAR and its contents.

Maven can either 'mvn install' to deploy a KieModule to the local machine, where all other applications on the local machine use it. Or it can 'mvn deploy' to push the KieModule to a remote Maven repository. Building the Application will pull in the KieModule and populate the local Maven repository in the process.

JARs can be deployed in one of two ways. Either added to the classpath, like any other JAR in a Maven dependency listing, or they can be dynamically loaded at runtime. KIE will scan the classpath to find all the JARs with a kmodule.xml in it. Each found JAR is represented by the KieModule interface. The terms classpath KieModule and dynamic KieModule are used to refer to the two loading approaches. While dynamic modules supports side by side versioning, classpath modules do not. Further once a module is on the classpath, no other version may be loaded dynamically.

The kmodule.xml allows to define and configure one or more **KieBase**s and for each **KieBase** all the different **KieSession**s that can be created from it, as shown by the following example:

Example 5.4. A sample kmodule.xml file

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg1">
    <ksession name="KSession2_1" type="stateful" default="true/">
    <ksession name="KSession2_1" type="stateless" default="false/"
beliefSystem="jtms">
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateful" default="false"
clockType="realtime">
      <fileLogger file="drools.log" threaded="true" interval="10"/>
      <workItemHandlers>
        <workItemHandler name="name"
type="org.domain.WorkItemHandler"/>
      </workItemHandlers>
      <listeners>
        <ruleRuntimeEventListener
type="org.domain.RuleRuntimeListener"/>
        <agendaEventListener type="org.domain.FirstAgendaListener"/>
        <agendaEventListener type="org.domain.SecondAgendaListener"/>
        <processEventListener type="org.domain.ProcessListener"/>
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```

Here 2 **KieBase**s have been defined and it is possible to instantiate 2 different types of**KieSession**s from the first one, while only one from the second.

Report a bug

5.2.2. Creating a KIE Project

A Kie Project has the structure of a normal Maven project with the only peculiarity of including a kmodule.xml file defining in a declaratively way the **KieBase**s and **KieSession**s that can be created from it. This file has to be placed in the resources/META-INF folder of the Maven project while all the other Kie artifacts, such as DRL or a Excel files, must be stored in the resources folder or in any other subfolder under it.

Since meaningful defaults have been provided for all configuration aspects, the simplest kmodule.xml file can contain just an empty kmodule tag like the following:

Example 5.5. An empty kmodule.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule"/>
```

In this way the kmodule will contain one single default **KieBase**. All Kie assets stored under the resources folder, or any of its subfolders, will be compiled and added to it. To trigger the building of these artifacts it is enough to create a **KieContainer** for them.

Report a bug

5.2.3. Creating a KIE Container

Illustrated below is a simple case example on how to create a **KieContainer** that reads files built from the classpath:

Example 5.6. Creating a KieContainer from the classpath

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();
```

After defining a kmodule.xml, it is possible to simply retrieve the KieBases and KieSessions from the KieContainer using their names.

Example 5.7. Retriving KieBases and KieSessions from the KieContainer

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();

KieBase kBase1 = kContainer.getKieBase("KBase1");
KieSession kieSession1 = kContainer.newKieSession("KSession2_1");
StatelessKieSession kieSession2 =
kContainer.newStatelessKieSession("KSession2_2");
```

It has to be noted that since KSession2_1 and KSession2_2 are of 2 different types (the first is stateful, while the second is stateless) it is necessary to invoke 2 different methods on the **KieContainer** according to their declared type. If the type of the **KieSession** requested to the **KieContainer** doesn't correspond with the one declared in the kmodule.xml file the **KieContainer** will throw a **RuntimeException**. Also since a **KieBase** and a **KieSession** have been flagged as default is it possible to get them from the **KieContainer** without passing any name.

Example 5.8. Retriving default KieBases and KieSessions from the KieContainer

```
KieContainer kContainer = ...

KieBase kBase1 = kContainer.getKieBase(); // returns KBase1
KieSession kieSession1 = kContainer.newKieSession(); // returns
KSession2_1
```

Since a Kie project is also a Maven project the groupId, artifactId and version declared in the pom.xml file are used to generate a **ReleaseId** that uniquely identifies this project inside your application. This allows creation of a new KieContainer from the project by simply passing its **ReleaseId** to the **KieServices**.

Example 5.9. Creating a KieContainer of an existing project by ReleaseId

```
KieServices kieServices = KieServices.Factory.get();
ReleaseId releaseId = kieServices.newReleaseId( "org.acme",
"myartifact", "1.0" );
KieContainer kieContainer = kieServices.newKieContainer( releaseId );
```

Report a bug

5.2.4. KieServices

KieServices is the interface from where it possible to access all the Kie building and runtime facilities:

In this way all the Java sources and the Kie resources are compiled and deployed into the KieContainer which makes its contents available for use at runtime.

Report a bug

5.3. KIE DEPLOYMENT

5.3.1. KieRepository

When the contents of a **KieFileSystem** are successfully built, the resulting **KieModule** is automatically added to the **KieRepository**. The **KieRepository** is a singleton acting as a repository for all the available **KieModule**s.

After this it is possible to create through the **KieServices** a new **KieContainer** for that **KieModule** using its **ReleaseId**. However, since in this case the **KieFileSystem** don't contain any pom.xml file (it is possible to add one using the **KieFileSystem.writePomXML** method), Kie cannot determine the **ReleaseId** of the **KieModule** and assign to it a default one. This default **ReleaseId** can be obtained from the **KieRepository** and used to identify the **KieModule** inside the **KieRepository** itself. The following example shows this whole process.

Example 5.10. Building the contents of a KieFileSystem and creating a KieContainer

```
KieServices kieServices = KieServices.Factory.get();
KieFileSystem kfs = ...
kieServices.newKieBuilder( kfs ).buildAll();
KieContainer kieContainer =
kieServices.newKieContainer(kieServices.getRepository().getDefaultReleas
eId());
```

At this point it is possible to get **KieBase**s and create new **KieSession**s from this **KieContainer** exactly in the same way as in the case of a **KieContainer** created directly from the classpath.

It is a best practice to check the compilation results. The **KieBuilder** reports compilation results of 3 different severities: ERROR, WARNING and INFO. An ERROR indicates that the compilation of the project failed and in the case no **KieModule** is produced and nothing is added to the **KieRepository**. WARNING and INFO results can be ignored, but are available for inspection.

Example 5.11. Checking that a compilation didn't produce any error

```
KieBuilder kieBuilder = kieServices.newKieBuilder( kfs ).buildAll();
assertEquals( 0, kieBuilder.getResults().getMessages(
Message.Level.ERROR ).size() );
```

Report a bug

5.3.2. Session Modification

The **KieBase** is a repository of all the application's knowledge definitions. It will contain rules, processes, functions, and type models. The **KieBase** itself does not contain data; instead, sessions are created from the **KieBase** into which data can be inserted and from which process instances may be started. The **KieBase** can be obtained from the **KieContainer** containing the **KieModule** where the **KieBase** has been defined.

Sometimes, for instance in a OSGi environment, the **KieBase** needs to resolve types that are not in the default class loader. In this case it will be necessary to create a **KieBaseConfiguration** with an additional class loader and pass it to **KieContainer** when creating a new **KieBase** from it.

Example 5.12. Creating a new KieBase with a custom ClassLoader

```
KieServices kieServices = KieServices.Factory.get();
KieBaseConfiguration kbaseConf = kieServices.newKieBaseConfiguration(
null, MyType.class.getClassLoader() );
KieBase kbase = kieContainer.newKieBase( kbaseConf );
```

The **KieBase** creates and returns **KieSession** objects, and it may optionally keep references to those. When **KieBase** modifications occur those modifications are applied against the data in the sessions. This reference is a weak reference and it is also optional, which is controlled by a boolean flag.

Report a bug

5.3.3. KieScanner

The **KieScanner** allows continuous monitoring of your Maven repository to check whether a new release of a Kie project has been installed. A new release is deployed in the **KieContainer** wrapping that project. The use of the **KieScanner** requires kie-ci.jar to be on the classpath.

A **KieScanner** can be registered on a **KieContainer** as in the following example.

Example 5.13. Registering and starting a KieScanner on a KieContainer

```
KieServices kieServices = KieServices.Factory.get();
ReleaseId releaseId = kieServices.newReleaseId( "org.acme",
"myartifact", "1.0-SNAPSHOT" );
KieContainer kContainer = kieServices.newKieContainer( releaseId );
KieScanner kScanner = kieServices.newKieScanner( kContainer );
```

```
// Start the KieScanner polling the Maven repository every 10 seconds
kScanner.start( 10000L );
```

In this example the **KieScanner** is configured to run with a fixed time interval, but it is also possible to run it on demand by invoking the **scanNow()** method on it. If the **KieScanner** finds in the Maven repository an updated version of the Kie project used by that **KieContainer** it automatically downloads the new version and triggers an incremental build of the new project. From this moment all the new **KieBase**s and **KieSession**s created from that **KieContainer** will use the new project version.

Report a bug

5.4. RUNNING IN KIE

5.4.1. KieRuntime

The **KieRuntime** provides methods that are applicable to both rules and processes, such as setting globals and registering channels. ("Exit point" is an obsolete synonym for "channel".)

Report a bug

5.4.2. Globals in KIE

Globals are named objects that are made visible to the rule engine, but in a way that is fundamentally different from the one for facts: changes in the object backing a global do not trigger reevaluation of rules. Still, globals are useful for providing static information, as an object offering services that are used in the RHS of a rule, or as a means to return objects from the rule engine. When you use a global on the LHS of a rule, make sure it is immutable, or, at least, don't expect changes to have any effect on the behavior of your rules.

A global must be declared in a rules file, and then it needs to be backed up with a Java object.

```
global java.util.List list
```

With the Knowledge Base now aware of the global identifier and its type, it is now possible to call **ksession.setGlobal()** with the global's name and an object, for any session, to associate the object with the global. Failure to declare the global type and identifier in DRL code will result in an exception being thrown from this call.

```
List list = new ArrayList();
ksession.setGlobal("list", list);
```

Make sure to set any global before it is used in the evaluation of a rule. Failure to do so results in a **NullPointerException**.

Report a bug

5.4.3. Event Packages

The event package provides means to be notified of rule engine events, including rules firing, objects being asserted, etc. This allows separation of logging and auditing activities from the main part of your application (and the rules).

The **KieRuntimeEventManager** interface is implemented by the **KieRuntime** which provides two interfaces, **RuleRuntimeEventManager** and **ProcessEventManager**. We will only cover the **RuleRuntimeEventManager** here.

The **RuleRuntimeEventManager** allows for listeners to be added and removed, so that events for the working memory and the agenda can be listened to.

The following code snippet shows how a simple agenda listener is declared and attached to a session. It will print matches after they have fired.

Example 5.14. Adding an AgendaEventListener

```
ksession.addEventListener( new DefaultAgendaEventListener() {
    public void afterMatchFired(AfterMatchFiredEvent event) {
        super.afterMatchFired( event );
        System.out.println( event );
    }
});
```

JBoss Rules also provides **DebugRuleRuntimeEventListener** and **DebugAgendaEventListener** which implement each method with a debug print statement. To print all Working Memory events, you add a listener like this:

Example 5.15. Adding a DebugRuleRuntimeEventListener

```
ksession.addEventListener( new DebugRuleRuntimeEventListener() );
```

All emitted events implement the **KieRuntimeEvent** interface which can be used to retrieve the actual **KnowlegeRuntime** the event originated from.

The events currently supported are:

- MatchCreatedEvent
- MatchCancelledEvent
- BeforeMatchFiredEvent
- AfterMatchFiredEvent
- AgendaGroupPushedEvent
- AgendaGroupPoppedEvent
- ObjectInsertEvent
- ObjectDeletedEvent
- ObjectUpdatedEvent
- ProcessCompletedEvent
- ProcessNodeLeftEvent

- ProcessNodeTriggeredEvent
- ProcessStartEvent

Report a bug

5.4.4. KieRuntimeLogger

The KieRuntimeLogger uses the comprehensive event system in JBoss Rules to create an audit log that can be used to log the execution of an application for later inspection, using tools such as the Eclipse audit viewer.

Example 5.16. FileLogger

```
KieRuntimeLogger logger =
KieServices.Factory.get().getLoggers().newFileLogger( session, "audit"
);
...
// Be sure to close the logger otherwise it will not write.
logger.close();
```

Report a bug

5.4.5. CommandExecutor Interface

KIE has the concept of stateful or stateless sessions. Stateful sessions have already been covered, which use the standard KieRuntime, and can be worked with iteratively over time. Stateless is a one-off execution of a KieRuntime with a provided data set. It may return some results, with the session being disposed at the end, prohibiting further iterative interactions. You can think of stateless as treating an engine like a function call with optional return results.

The foundation for this is the **`CommandExecutor`** interface, which both the stateful and stateless interfaces extend. This returns an **`ExecutionResults`**:

The **`CommandExecutor`** allows for commands to be executed on those sessions, the only difference being that the StatelessKieSession executes **`fireAllRules()`** at the end before disposing the session. The commands can be created using the **`CommandExecutor`** .The Javadocs provide the full list of the allowed comands using the **`CommandExecutor`**.

setGlobal and getGlobal are two commands relevant to JBoss Rules.

Set Global calls setGlobal underneath. The optional boolean indicates whether the command should return the global's value as part of the **`ExecutionResults`**. If true it uses the same name as the global name. A String can be used instead of the boolean, if an alternative name is desired.

Example 5.17. Set Global Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ExecutionResults bresults =
    ksession.execute( CommandFactory.newSetGlobal( "stilton", new
Cheese( "stilton" ), true);
Cheese stilton = bresults.getValue( "stilton" );
```

-

Allows an existing global to be returned. The second optional String argument allows for an alternative return name.

Example 5.18. Get Global Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ExecutionResults bresults =
    ksession.execute( CommandFactory.getGlobal( "stilton" );
Cheese stilton = bresults.getValue( "stilton" );
```

All the above examples execute single commands. The **`BatchExecution`** represents a composite command, created from a list of commands. It will iterate over the list and execute each command in turn. This means you can insert some objects, start a process, call fireAllRules and execute a query, all in a single **`execute(...)`** call, which is quite powerful.

The StatelessKieSession will execute **`fireAllRules()`** automatically at the end. However the keen-eyed reader probably has already noticed the **`FireAllRules`** command and wondered how that works with a StatelessKieSession. The **`FireAllRules`** command is allowed, and using it will disable the automatic execution at the end; think of using it as a sort of manual override function.

Any command, in the batch, that has an out identifier set will add its results to the returned **`ExecutionResults`** instance.

Example 5.19. BatchExecution Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();

List cmds = new ArrayList();
cmds.add( CommandFactory.newInsertObject( new Cheese( "stilton", 1),
"stilton") );
cmds.add( CommandFactory.newStartProcess( "process cheeses" ) );
cmds.add( CommandFactory.newQuery( "cheeses" ) );
ExecutionResults bresults = ksession.execute(
CommandFactory.newBatchExecution( cmds ) );
Cheese stilton = ( Cheese ) bresults.getValue( "stilton" );
QueryResults qresults = ( QueryResults ) bresults.getValue( "cheeses" );
```

In the above example multiple commands are executed, two of which populate the **`ExecutionResults`**. The query command defaults to use the same identifier as the query name, but it can also be mapped to a different identifier.

All commands support XML and jSON marshalling using XStream, as well as JAXB marshalling. This is covered in the *Rule Commands* section: Section 8.1, “Available API”.

Report a bug

5.5. KIE CONFIGURATION

5.5.1. Build Result Severity

In some cases, it is possible to change the default severity of a type of build result. For instance, when a new rule with the same name of an existing rule is added to a package, the default behavior is to replace the old rule by the new rule and report it as an INFO. This is probably ideal for most use cases, but in some deployments the user might want to prevent the rule update and report it as an error.

Changing the default severity for a result type, configured like any other option in JBoss Rules, can be done by API calls, system properties or configuration files. As of this version, JBoss Rules supports configurable result severity for rule updates and function updates. To configure it using system properties or configuration files, the user has to use the following properties:

Example 5.20. Setting the severity using properties

```
// sets the severity of rule updates
drools.kbuilder.severity.duplicateRule = <INFO|WARNING|ERROR>
// sets the severity of function updates
drools.kbuilder.severity.duplicateFunction = <INFO|WARNING|ERROR>
```

Report a bug

5.5.2. StatelessKieSession

The **`StatelessKieSession`** wraps the **`KieSession`**, instead of extending it. Its main focus is on the decision service type scenarios. It avoids the need to call **`dispose()`**. Stateless sessions do not support iterative insertions and the method call **`fireAllRules()`** from Java code; the act of calling **`execute()`** is a single-shot method that will internally instantiate a **`KieSession`**, add all the user data and execute user commands, call **`fireAllRules()`**, and then call **`dispose()`**. While the main way to work with this class is via the **`BatchExecution`** (a subinterface of **`Command`**) as supported by the **`CommandExecutor`** interface, two convenience methods are provided for when simple object insertion is all that's required. The **`CommandExecutor`** and **`BatchExecution`** are talked about in detail in their own section.

Our simple example shows a stateless session executing a given collection of Java objects using the convenience API. It will iterate the collection, inserting each element in turn.

Example 5.21. Simple StatelessKieSession execution with a Collection

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ksession.execute( collection );
```

If this was done as a single Command it would be as follows:

Example 5.22. Simple StatelessKieSession execution with InsertElements Command

```
ksession.execute( CommandFactory.newInsertElements( collection ) );
```

If you wanted to insert the collection itself, and the collection's individual elements, then **`CommandFactory.newInsert(collection)`** would do the job.

Methods of the **`CommandFactory`** create the supported commands, all of which can be marshalled using XStream and the **`BatchExecutionHelper`**. **`BatchExecutionHelper`** provides details on the

XML format as well as how to use Drools Pipeline to automate the marshalling of **`BatchExecution`** and **`ExecutionResults`**.

`StatelessKieSession` supports globals, scoped in a number of ways. We cover the non-command way first, as commands are scoped to a specific execution call. Globals can be resolved in three ways.

- The StatelessKieSession method **`getGlobals()`** returns a Globals instance which provides access to the session's globals. These are shared for *all* execution calls. Exercise caution regarding mutable globals because execution calls can be executing simultaneously in different threads.

  Example 5.23. Session scoped global

  ```
  StatelessKieSession ksession = kbase.newStatelessKieSession();
  // Set a global hbnSession, that can be used for DB interactions
  in the rules.
  ksession.setGlobal( "hbnSession", hibernateSession );
  // Execute while being able to resolve the "hbnSession"
  identifier.
  ksession.execute( collection );
  ```

- Using a delegate is another way of global resolution. Assigning a value to a global (with **`setGlobal(String, Object)`**) results in the value being stored in an internal collection mapping identifiers to values. Identifiers in this internal collection will have priority over any supplied delegate. Only if an identifier cannot be found in this internal collection, the delegate global (if any) will be used.

- The third way of resolving globals is to have execution scoped globals. Here, a **`Command`** to set a global is passed to the **`CommandExecutor`**.

The **`CommandExecutor`** interface also offers the ability to export data via "out" parameters. Inserted facts, globals and query results can all be returned.

Example 5.24. Out identifiers

```
// Set up a list of commands
List cmds = new ArrayList();
cmds.add( CommandFactory.newSetGlobal( "list1", new ArrayList(), true )
);
cmds.add( CommandFactory.newInsert( new Person( "jon", 102 ), "person" )
);
cmds.add( CommandFactory.newQuery( "Get People" "getPeople" );

// Execute the list
ExecutionResults results =
  ksession.execute( CommandFactory.newBatchExecution( cmds ) );

// Retrieve the ArrayList
results.getValue( "list1" );
// Retrieve the inserted Person fact
results.getValue( "person" );
// Retrieve the query as a QueryResults instance.
results.getValue( "Get People" );
```

Report a bug

5.5.3. Marshalling

The **KieMarshallers** are used to marshal and unmarshal KieSessions.

An instance of the **KieMarshallers** can be retrieved from the **KieServices**. A simple example is shown below:

Example 5.25. Simple Marshaller Example

```
// ksession is the KieSession
// kbase is the KieBase
ByteArrayOutputStream baos = new ByteArrayOutputStream();
Marshaller marshaller =
KieServices.Factory.get().getMarshallers().newMarshaller( kbase );
marshaller.marshall( baos, ksession );
baos.close();
```

However, with marshalling, you will need more flexibility when dealing with referenced user data. To achieve this use the **ObjectMarshallingStrategy** interface. Two implementations are provided, but users can implement their own. The two supplied strategies are **IdentityMarshallingStrategy** and **SerializeMarshallingStrategy**. **SerializeMarshallingStrategy** is the default, as shown in the example above, and it just calls the **Serializable** or **Externalizable** methods on a user instance. **IdentityMarshallingStrategy** creates an integer id for each user object and stores them in a Map, while the id is written to the stream. When unmarshalling it accesses the **IdentityMarshallingStrategy** map to retrieve the instance. This means that if you use the **IdentityMarshallingStrategy**, it is stateful for the life of the Marshaller instance and will create ids and keep references to all objects that it attempts to marshal. Below is the code to use an Identity Marshalling Strategy.

Example 5.26. IdentityMarshallingStrategy

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
KieMarshallers kMarshallers = KieServices.Factory.get().getMarshallers()
ObjectMarshallingStrategy oms =
kMarshallers.newIdentityMarshallingStrategy()
Marshaller marshaller =
        kMarshallers.newMarshaller( kbase, new
ObjectMarshallingStrategy[]{ oms } );
marshaller.marshall( baos, ksession );
baos.close();
```

Im most cases, a single strategy is insufficient. For added flexibility, the **ObjectMarshallingStrategyAcceptor** interface can be used. This Marshaller has a chain of strategies, and while reading or writing a user object it iterates the strategies asking if they accept responsibility for marshalling the user object. One of the provided implementations is **ClassFilterAcceptor**. This allows strings and wild cards to be used to match class names. The default is "*.*", so in the above example the Identity Marshalling Strategy is used which has a default "*.*" acceptor.

Assuming that we want to serialize all classes except for one given package, where we will use identity lookup, we could do the following:

Example 5.27. IdentityMarshallingStrategy with Acceptor

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
KieMarshallers kMarshallers = KieServices.Factory.get().getMarshallers()
ObjectMarshallingStrategyAcceptor identityAcceptor =
        kMarshallers.newClassFilterAcceptor( new String[] {
"org.domain.pkg1.*" } );
ObjectMarshallingStrategy identityStrategy =
        kMarshallers.newIdentityMarshallingStrategy( identityAcceptor
);
ObjectMarshallingStrategy sms =
kMarshallers.newSerializeMarshallingStrategy();
Marshaller marshaller =
        kMarshallers.newMarshaller( kbase,
                                    new ObjectMarshallingStrategy[]{
identityStrategy, sms } );
marshaller.marshall( baos, ksession );
baos.close();
```

Note that the acceptance checking order is in the natural order of the supplied elements.

Also note that if you are using scheduled matches (i.e. some of your rules use timers or calendars) they are marshallable only if, before you use it, you configure your KieSession to use a trackable timer job factory manager as follows:

Example 5.28. Configuring a trackable timer job factory manager

```
KieSessionConfiguration ksconf =
KieServices.Factory.get().newKieSessionConfiguration();
ksconf.setOption(TimerJobFactoryOption.get("trackable"));
KSession ksession = kbase.newKieSession(ksconf, null);
```

Report a bug

5.5.4. KIE Persistence

Longterm out of the box persistence with Java Persistence API (JPA) is possible with JBoss Rules. It is necessary to have some implementation of the Java Transaction API (JTA) installed. For development purposes the Bitronix Transaction Manager is suggested, as it's simple to set up and works embedded, but for production use JBoss Transactions is recommended.

Example 5.29. Simple example using transactions

```
KieServices kieServices = KieServices.Factory.get();
Environment env = kieServices.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY,
         Persistence.createEntityManagerFactory( "emf-name" ) );
env.set( EnvironmentName.TRANSACTION_MANAGER,
         TransactionManagerServices.getTransactionManager() );
```

```
// KieSessionConfiguration may be null, and a default will be used
KieSession ksession =
        kieServices.getStoreServices().newKieSession( kbase, null, env
);
int sessionId = ksession.getId();

UserTransaction ut =
  (UserTransaction) new InitialContext().lookup(
"java:comp/UserTransaction" );
ut.begin();
ksession.insert( data1 );
ksession.insert( data2 );
ksession.startProcess( "process1" );
ut.commit();
```

To use a JPA, the Environment must be set with both the **`EntityManagerFactory`** and the **`TransactionManager`**. If rollback occurs the ksession state is also rolled back, hence it is possible to continue to use it after a rollback. To load a previously persisted KieSession you'll need the id, as shown below:

Example 5.30. Loading a KieSession

```
KieSession ksession =
        kieServices.getStoreServices().loadKieSession( sessionId,
kbase, null, env );
```

To enable persistence several classes must be added to your persistence.xml, as in the example below:

Example 5.31. Configuring JPA

```
<persistence-unit name="org.drools.persistence.jpa" transaction-
type="JTA">
   <provider>org.hibernate.ejb.HibernatePersistence</provider>
   <jta-data-source>jdbc/BitronixJTADataSource</jta-data-source>
   <class>org.drools.persistence.info.SessionInfo</class>
   <class>org.drools.persistence.info.WorkItemInfo</class>
   <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
         <property name="hibernate.max_fetch_depth" value="3"/>
         <property name="hibernate.hbm2ddl.auto" value="update" />
         <property name="hibernate.show_sql" value="true" />
         <property name="hibernate.transaction.manager_lookup_class"

value="org.hibernate.transaction.BTMTransactionManagerLookup" />
   </properties>
</persistence-unit>
```

The jdbc JTA data source would have to be configured first. Bitronix provides a number of ways of doing this, and its documentation should be consulted for details. For a quick start, here is the programmatic approach:

Example 5.32. Configuring JTA DataSource

```
PoolingDataSource ds = new PoolingDataSource();
ds.setUniqueName( "jdbc/BitronixJTADataSource" );
ds.setClassName( "org.h2.jdbcx.JdbcDataSource" );
ds.setMaxPoolSize( 3 );
ds.setAllowLocalTransactions( true );
ds.getDriverProperties().put( "user", "sa" );
ds.getDriverProperties().put( "password", "sasa" );
ds.getDriverProperties().put( "URL", "jdbc:h2:mem:mydb" );
ds.init();
```

Bitronix also provides a simple embedded JNDI service, ideal for testing. To use it, add a jndi.properties file to your META-INF folder and add the following line to it:

Example 5.33. JNDI properties

```
java.naming.factory.initial=bitronix.tm.jndi.BitronixInitialContextFacto
ry
```

Report a bug

CHAPTER 6. RULE SYSTEMS

6.1. FORWARD-CHAINING

Forward-chaining is a production rule system. It is data-driven which means it reacts to the data it is presented. Facts are inserted into the working memory which results in one or more rules being true. They are then placed on the schedule to be executed by the agenda.

JBoss Rules is a forward-chaining engine.

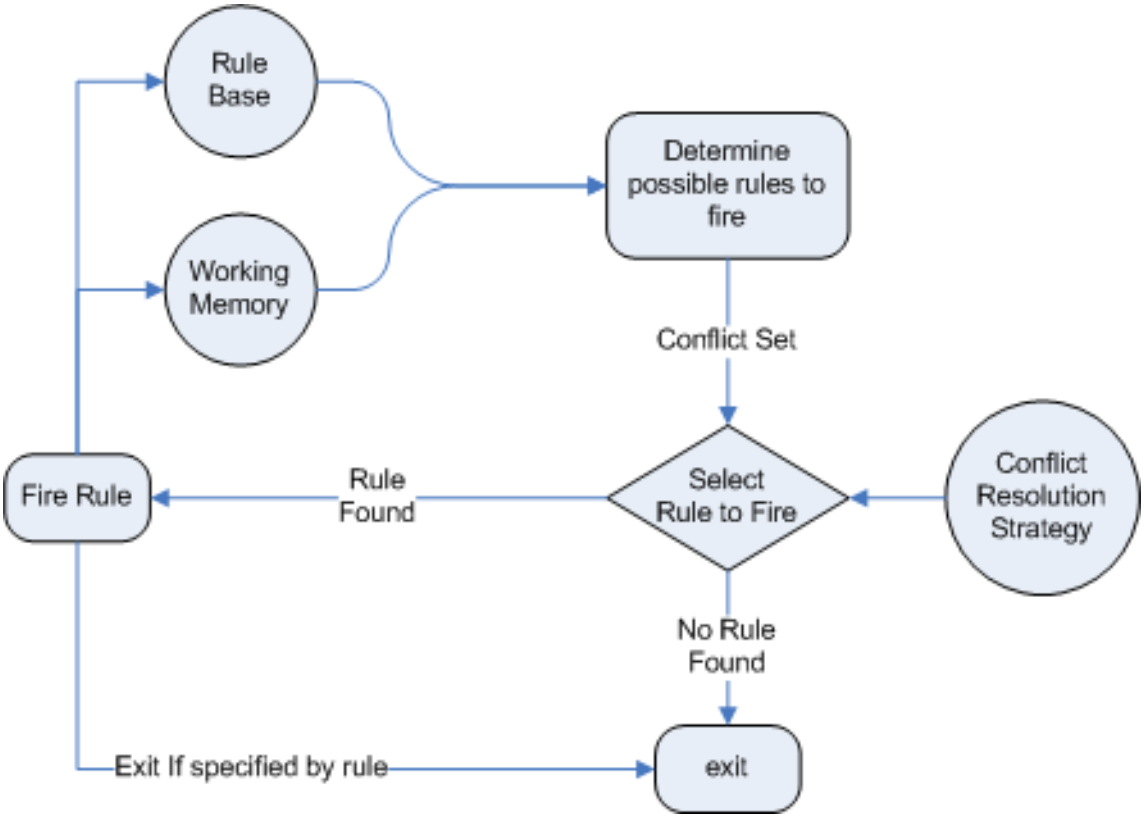


Figure 6.1. Forward Chaining Chart

Report a bug

6.2. BACKWARD-CHAINING

6.2.1. Backward-Chaining

A backward-chaining rule system is goal-driven. This means the system starts with a *conclusion* which the engine tries to satisfy. If it cannot do so it searches for sub-goals, that is, conclusions that will complete part of the current goal. It continues this process until either the initial conclusion is satisfied or there are no more unsatisfied sub-goals. **`Prolog`** is an example of a backward-chaining engine.

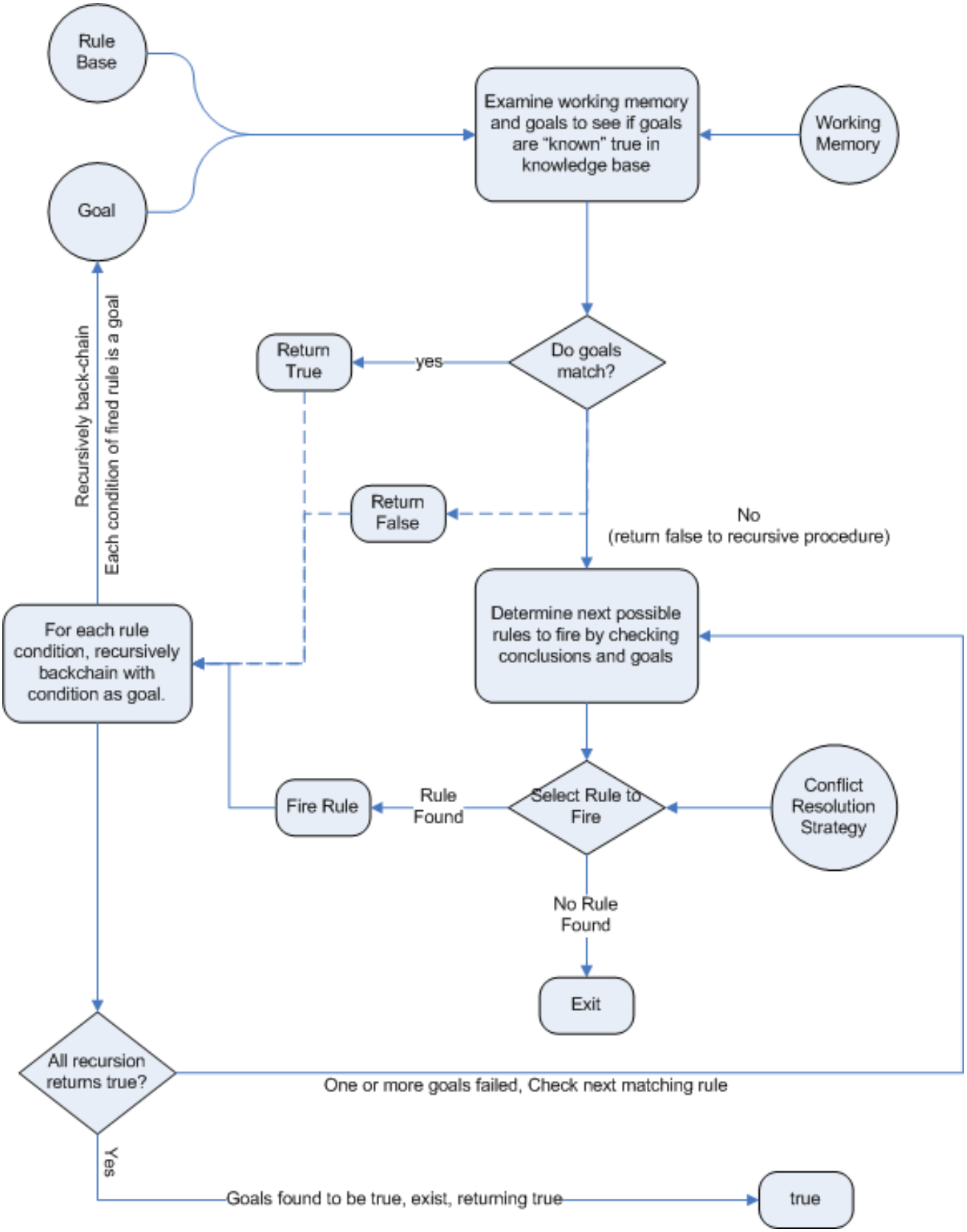


Figure 6.2. Backward Chaining Chart

IMPORTANT

Backward-chaining was implemented in JBoss BRMS 5.2.

Report a bug

6.2.2. Backward-Chaining Systems

Backward-Chaining is a feature recently added to the JBoss Rules Engine. This process is often referred to as derivation queries, and it is not as common compared to reactive systems since JBoss Rules is primarily reactive forward chaining. That is, it responds to changes in your data. The backward-chaining added to the engine is for product-like derivations.

Report a bug

6.2.3. Cloning Transitive Closures

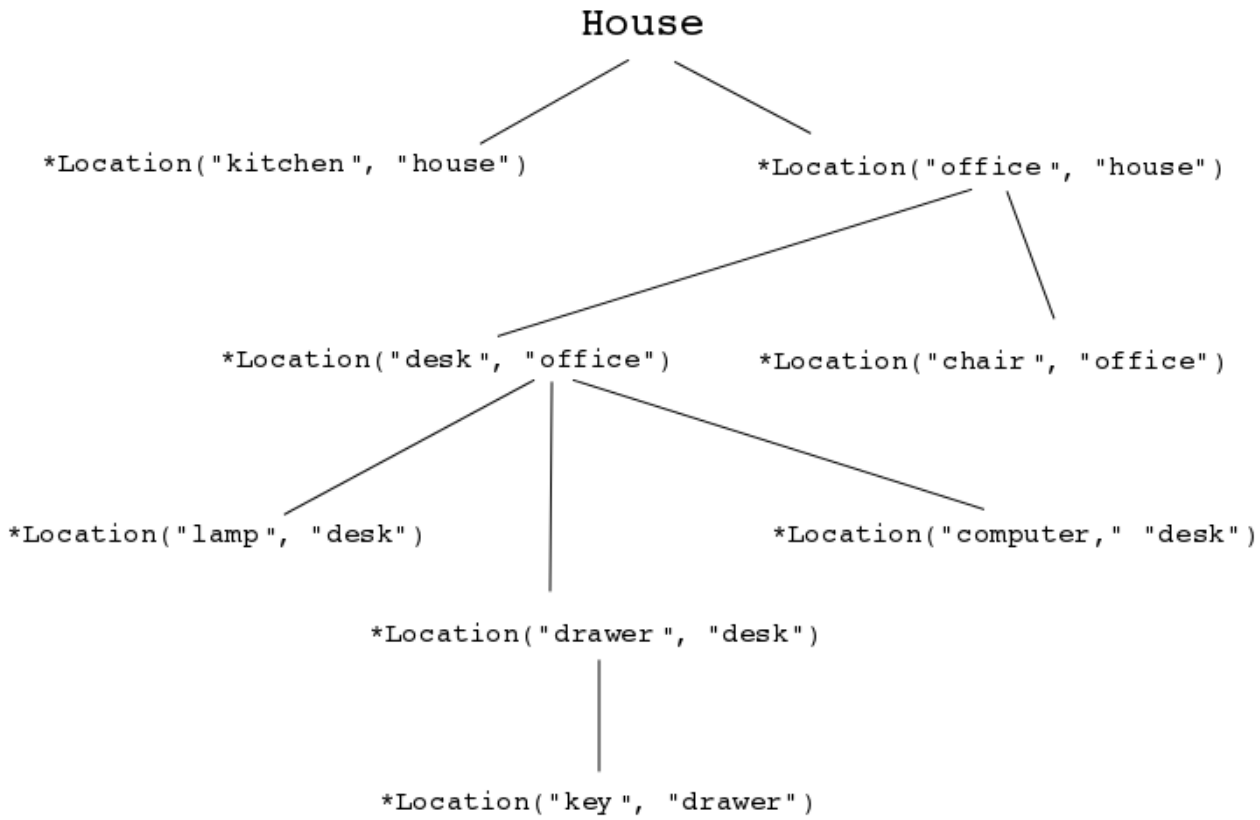


Figure 6.3. Reasoning Graph

The previous chart demonstrates a House example of transitive items. A similar reasoning chart can be created by implementing the following rules:

Procedure 6.1. Configure Transitive Closures

1. First, create some java rules to develop reasoning for transitive items. It inserts each of the locations.
2. Next, create the **`Location`** class; it has the item and where it is located.
3. Type the rules for the House example as depicted below:

```
ksession.insert( new Location("office", "house") );
ksession.insert( new Location("kitchen", "house") );
ksession.insert( new Location("knife", "kitchen") );
```

```
ksession.insert( new Location("cheese", "kitchen") );
ksession.insert( new Location("desk", "office") );
ksession.insert( new Location("chair", "office") );
ksession.insert( new Location("computer", "desk") );
ksession.insert( new Location("drawer", "desk") );
```

4. A transitive design is created in which the item is in its designated location such as a "desk" located in an "office."

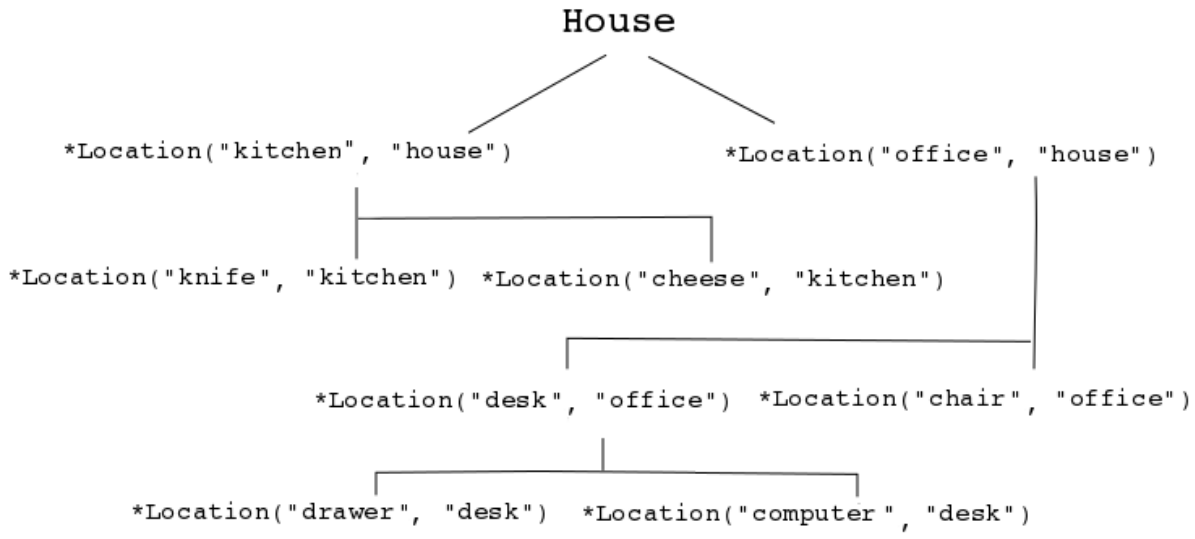


Figure 6.4. Transitive Reasoning Graph of a House.

NOTE

Notice compared to the previous graph, there is no **"key"** item in a **"drawer"** location. This will become evident in a later topic.

Report a bug

6.2.4. Defining a Query

Procedure 6.2. Define a Query

1. Create a query to look at the data inserted into the rules engine:

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

Notice how the query is recursive and is calling "isContainedIn."

2. Create a rule to print out every string inserted into the system to see how things are implemented. The rule should resemble the following format:

```
rule "go" salience 10
when
    $s : String( )
then
    System.out.println( $s );
end
```

3. Using Step 2 as a model, create a rule that calls upon the Step 1 query "isContainedIn."

```
rule "go1"
when
    String( this == "go1" )
    isContainedIn("office", "house"; )
then
    System.out.println( "office is in the house" );
end
```

The "go1" rule will fire when the first string is inserted into the engine. That is, it asks if the item "office" is in the location "house." Therefore, the Step 1 query is evoked by the previous rule when the "go1" String is inserted.

4. Create the "go1," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go1" );
ksession.fireAllRules();
---
go1
office is in the house
```

The --- line indicates the separation of the output of the engine from the firing of the "go" rule and the "go1" rule.

- "go1" is inserted
- Salience ensures it goes first
- The rule matches the query

Report a bug

6.2.5. Transitive Closure Example

Procedure 6.3. Create a Transitive Closure

1. Create a Transitive Closure by implementing the following rule:

```
rule "go2"
when
 String( this == "go2" )
 isContainedIn("drawer", "house"; )
then
 System.out.println( "Drawer in the House" );
end
```

2. Recall from the Cloning Transitive Closure's topic, there was no instance of "drawer" in "house." "drawer" was located in "desk."

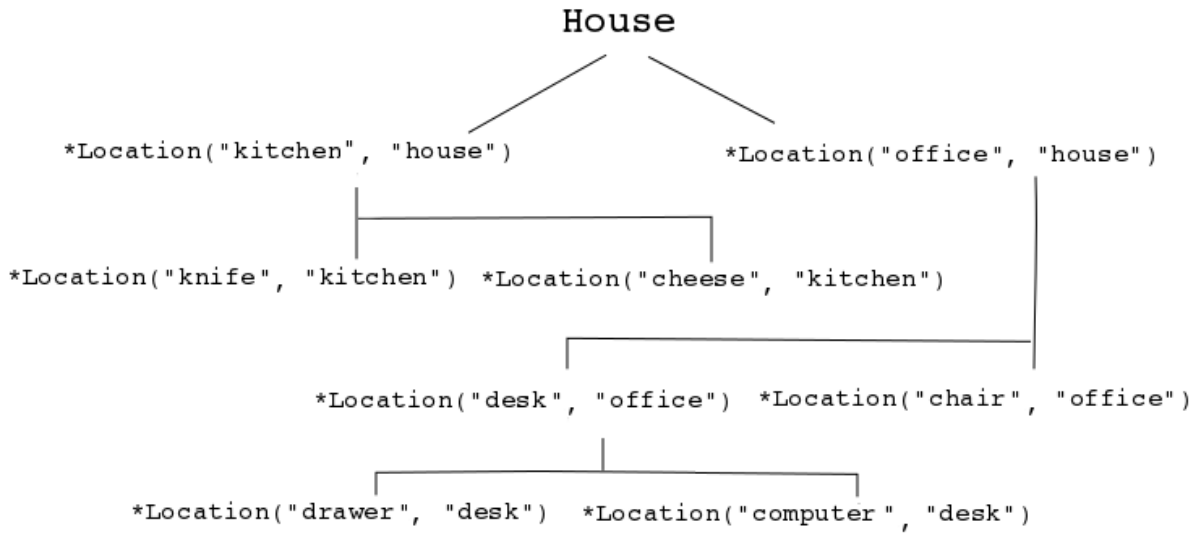


Figure 6.5. Transitive Reasoning Graph of a Drawer.

3. Use the previous query for this recursive information.

```
query isContainedIn( String x, String y )
 Location( x, y; )
 or
 ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

4. Create the "go2," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go2" );
ksession.fireAllRules();
---
go2
Drawer in the House
```

When the rule is fired, it correctly tells you "go2" has been inserted and that the "drawer" is in the "house."

5. Check how the engine determined this outcome.
   - The query has to recurse down several levels to determine this.
   - Instead of using **`Location( x, y; )`**, The query uses the value of **`(z, y; )`** since "drawer" is not in "house."
   - The **`z`** is currently unbound which means it has no value and will return everything that is in the argument.
   - **`y`** is currently bound to "house," so **`z`** will return "office" and "kitchen."

- Information is gathered from "office" and checks recursively if the "drawer" is in the "office." The following query line is being called for these parameters: **`isContainedIn (x ,z; )`**

There is no instance of "drawer" in "office;" therefore, it does not match. With **`z`** being unbound, it will return data that is within the "office," and it will gather that **`z == desk`**.

```
isContainedIn(x==drawer, z==desk)
```

`isContainedIn` recurses three times. On the final recurse, an instance triggers of "drawer" in the "desk."

```
Location(x==drawer, y==desk)
```

This matches on the first location and recurses back up, so we know that "drawer" is in the "desk," the "desk" is in the "office," and the "office" is in the "house;" therefore, the "drawer" is in the "house" and returns **`true`**.

Report a bug

6.2.6. Reactive Transitive Queries

Procedure 6.4. Create a Reactive Transitive Query

1. Create a Reactive Transitive Query by implementing the following rule:

```
rule "go3"
when
  String( this == "go3" )
  isContainedIn("key", "office"; )
then
  System.out.println( "Key in the Office" );
end
```

Reactive Transitive Queries can ask a question even if the answer can not be satisfied. Later, if it is satisfied, it will return an answer.

NOTE

Recall from the Cloning Transitive Closures example that there was no "key" item in the system.

2. Use the same query for this reactive information.

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

3. Create the "go3," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go3" );
```

```
ksession.fireAllRules();
---
go3
```

- "go3" is inserted
- **`fireAllRules();`** is called

The first rule that matches any String returns "go3" but nothing else is returned because there is no answer; however, while "go3" is inserted in the system, it will continuously wait until it is satisfied.

4. Insert a new location of "key" in the "drawer":

```
ksession.insert( new Location("key", "drawer") );
ksession.fireAllRules();
---
Key in the Office
```

This new location satisfies the transitive closure because it is monitoring the entire graph. In addition, this process now has four recursive levels in which it goes through to match and fire the rule.

Report a bug

6.2.7. Queries with Unbound Arguments

Procedure 6.5. Create an Unbound Argument's Query

1. Create a Query with Unbound Arguments by implementing the following rule:

```
rule "go4"
when
  String( this == "go4" )
  isContainedIn(thing, "office"; )
then
  System.out.println( "thing" + thing + "is in the Office"
);
end
```

This rule is asking for everything in the "office," and it will tell everything in all the rows below. The unbound argument (out variable **`thing`**) in this example will return every possible value; accordingly, it is very similar to the **`z`** value used in the Reactive Transitive Query example.

2. Use the query for the unbound arguments.

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

3. Create the "go4," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go4" );
ksession.fireAllRules();
---
go4
thing Key is in the Office
thing Computer is in the Office
thing Drawer is in the Office
thing Desk is in the Office
thing Chair is in the Office
```

When "go4" is inserted, it returns all the previous information that is transitively below "Office."

Report a bug

6.2.8. Multiple Unbound Arguments

Procedure 6.6. Creating Multiple Unbound Arguments

1. Create a query with Mulitple Unbound Arguments by implementing the following rule:

```
rule "go5"
when
  String( this == "go5" )
  isContainedIn(thing, location; )
then
  System.out.println( "thing" + thing + "is in" + location
);
end
```

Both **`thing`** and **`location`** are unbound out variables, and without bound arguments, everything is called upon.

2. Use the query for multiple unbound arguments.

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

3. Create the "go5," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go5" );
ksession.fireAllRules();
---
go5
thing Knife is in House
thing Cheese is in House
thing Key is in House
thing Computer is in House
thing Drawer is in House
thing Desk is in House
thing Chair is in House
thing Key is in Office
```

```
thing Computer is in Office
thing Drawer is in Office
thing Key is in Desk
thing Office is in House
thing Computer is in Desk
thing Knife is in Kitchen
thing Cheese is in Kitchen
thing Kitchen is in House
thing Key is in Drawer
thing Drawer is in Desk
thing Desk is in Office
thing Chair is in Office
```

When "go5" is called, it returns everything within everything.

Report a bug

CHAPTER 7. RULE LANGUAGES

7.1. RULE OVERVIEW

7.1.1. Overview

JBoss Rules has a native rule language. This format is very light in terms of punctuation, and supports natural and domain specific languages via "expanders" that allow the language to morph to your problem domain.

Report a bug

7.1.2. A rule file

A rule file is typically a file with a .drl extension. In a DRL file you can have multiple rules, queries and functions, as well as some resource declarations like imports, globals, and attributes that are assigned and used by your rules and queries. However, you are also able to spread your rules across multiple rule files (in that case, the extension .rule is suggested, but not required) - spreading rules across files can help with managing large numbers of rules. A DRL file is simply a text file.

Report a bug

7.1.3. The structure of a rule file

The overall structure of a rule file is the following:

Example 7.1. Rules file

```
package package-name

imports

globals

functions

queries

rules
```

The order in which the elements are declared is not important, except for the package name that, if declared, must be the first element in the rules file. All elements are optional, so you will use only those you need.

Report a bug

7.1.4. What is a rule

For the inpatients, just as an early view, a rule has the following rough structure:

```
rule "name"
    attributes
```

```
    when
        LHS
    then
        RHS
end
```

Mostly punctuation is not needed, even the double quotes for "name" are optional, as are newlines. Attributes are simple (always optional) hints to how the rule should behave. LHS is the conditional parts of the rule, which follows a certain syntax which is covered below. RHS is basically a block that allows dialect specific semantic code to be executed.

It is important to note that white space is not important, *except* in the case of domain specific languages, where lines are processed one by one and spaces may be significant to the domain language.

Report a bug

7.2. RULE LANGUAGE KEYWORDS

7.2.1. Hard Keywords

Hard keywords are words which you cannot use when naming your domain objects, properties, methods, functions and other elements that are used in the rule text. The hard keywords are **`true`**, **`false`**, and **`null`**.

Report a bug

7.2.2. Soft Keywords

Soft keywords can be used for naming domain objects, properties, methods, functions and other elements. The rules engine recognizes their context and processes them accordingly.

Report a bug

7.2.3. List of Soft Keywords

Rule attributes can be both simple and complex properties that provide a way to influence the behavior of the rule. They are usually written as one attribute per line and can be optional to the rule. Listed below are various rule attributes:

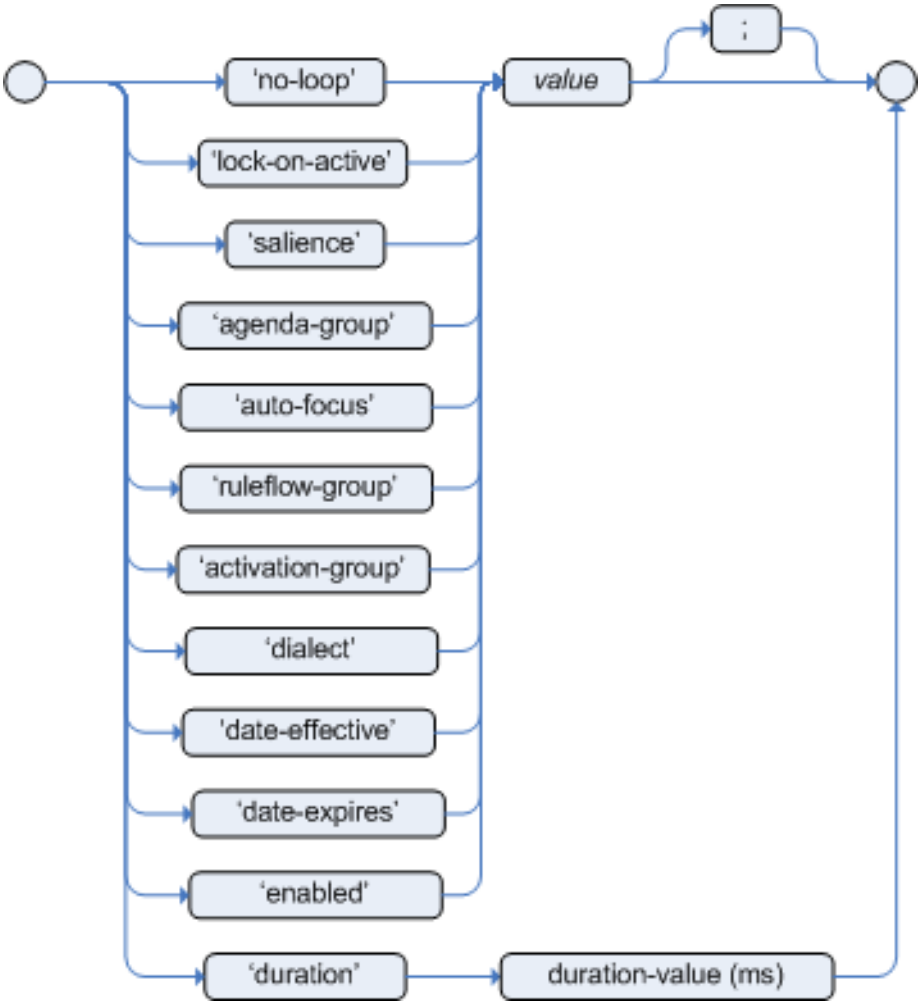


Figure 7.1. Rule Attributes

Table 7.1. Soft Keywords

Name	Default Value	Type	Description
`no-loop`	false	Boolean	When a rule's consequence modifies a fact, it may cause the rule to activate again, causing an infinite loop. Setting 'no-loop' to "true" will skip the creation of another activation for the rule with the current set of facts.

Name	Default Value	Type	Description
`lock-on-active`	false	Boolean	Whenever a 'ruleflow-group' becomes active or an 'agenda-group' receives the focus, any rule within that group that has 'lock-on-active' set to "true" will not be activated any more. Regardless of the origin of the update, the activation of a matching rule is discarded. This is a stronger version of 'no-loop' because the change is not only caused by the rule itself. It is ideal for calculation rules where you have a number of rules that modify a fact, and you do not want any rule re-matching and firing again. Only when the 'ruleflow-group' is no longer active or the 'agenda-group' loses the focus, those rules with 'lock-on-active' set to "true" become eligible again for their activations to be placed onto the agenda.

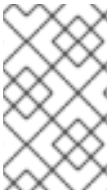
Name	Default Value	Type	Description

salience	0	integer	Each rule has an integer salience attribute which defaults to zero and can be negative or positive. Salience is a form of priority where rules with higher salience values are given higher priority when ordered in the activation queue. BRMS also supports dynamic salience where you can use an expression involving bound variables like the following:
			`rule "Fire in rank order 1,2,.."`
			
			`salience( -$rank )`
			`when`
			
			`Element( $rank : rank,... )`
			`then`
			`...`
			`end`

ruleflow-group	N/A	String	Ruleflow is a BRMS feature that lets you exercise control over the firing of rules. Rules that are assembled by the same "ruleflow-group' identifier fire only when their group is active. This attribute has been merged with 'agenda-group' and the behaviours are basically the same.

Name	Default Value	Type	Description
agenda-group	MAIN	String	Agenda groups allow the user to partition the agenda, which provides more execution control. Only rules in the agenda group that have acquired the focus are allowed to fire. This attribute has been merged with 'ruleflow-group' and the behaviours are basically the same.
auto-focus	false	Boolean	When a rule is activated where the 'auto-focus' value is "true" and the rule's agenda group does not have focus yet, it is automatically given focus, allowing the rule to potentially fire.
activation-group	N/A	String	Rules that belong to the same 'activation-group' identified by this attribute's String value, will only fire exclusively. More precisely, the first rule in an 'activation-group' to fire will cancel all pending activations of all rules in the group, i.e., stop them from firing.
dialect	specified by package	String	Java and MVEL are the possible values of the 'dialect' attribute. This attribute specifies the language to be used for any code expressions in the LHS or the RHS code block. While the 'dialect' can be specified at the package level, this attribute allows the package definition to be overridden for a rule.

Name	Default Value	Type	Description
date-effective	N/A	String, date and time definition	A rule can only activate if the current date and time is after the 'date-effective' attribute. An example 'date-effective' attribute is displayed below: `rule "Start Exercising" date-effective "4-Sep-2014" when $m : org.drools.compiler.Message() then $m.setFired(true); end`
date-expires	N/A	String, date and time definition	A rule cannot activate if the current date and time is after the 'date-expires' attribute. An example 'date-expires' attribute is displayed below: `rule "Run 4km" date-effective "4-Sep-2014" date-expires "9-Sep-2014" when $m : org.drools.compiler.Message() then $m.setFired(true); end`
duration	no default	long	If a rule is still "true", the 'duration' attribute will dictate that the rule will fire after a specified duration.



NOTE

The attributes 'ruleflow-group' and 'agenda-group' have been merged and now behave the same. The GET methods have been left the same, for deprecations reasons, but both attributes return the same underlying data.

Report a bug

7.3. RULE LANGUAGE COMMENTS

7.3.1. Comments

Comments are sections of text that are ignored by the rule engine. They are stripped out when they are encountered, except inside semantic code blocks (like a rule's RHS).

Report a bug

7.3.2. Single Line Comment Example

This is what a single line comment looks like. To create single line comments, you can use '//'. The parser will ignore anything in the line after the comment symbol:

```
rule "Testing Comments"
when
    // this is a single line comment
    eval( true ) // this is a comment in the same line of a pattern
then
    // this is a comment inside a semantic code block
end
```

Report a bug

7.3.3. Multi-Line Comment Example

This is what a multi-line comment looks like. This configuration comments out blocks of text, both in and outside semantic code blocks:

```
rule "Test Multi-line Comments"
when
    /* this is a multi-line comment
       in the left hand side of a rule */
    eval( true )
then
    /* and this is a multi-line comment
       in the right hand side of a rule */
end
```

Report a bug

7.4. RULE LANGUAGE MESSAGES

7.4.1. Error Messages

JBoss Rules introduces standardized error messages. This standardization aims to help users to find and resolve problems in a easier and faster way.

Report a bug

7.4.2. Error Message Format

This is the standard error message format.

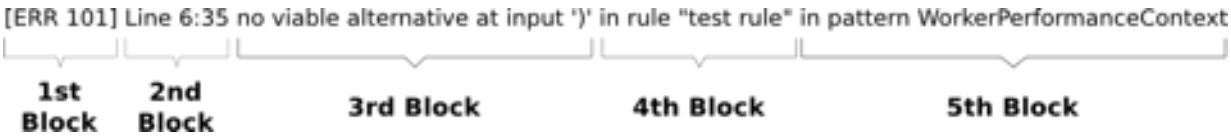


Figure 7.2. Error Message Format Example

1st Block: This area identifies the error code.

2nd Block: Line and column information.

3rd Block: Some text describing the problem.

4th Block: This is the first context. Usually indicates the rule, function, template or query where the error occurred. This block is not mandatory.

5th Block: Identifies the pattern where the error occurred. This block is not mandatory.

Report a bug

7.4.3. Error Messages Description

Table 7.2. Error Messages

Error Message	Description	Example	

[ERR 101] Line 4:4 no viable alternative at input 'exits' in rule one	Indicates when the parser came to a decision point but couldn't identify an alternative.	`1: rule one`
		`2:   when`
		`3:     exists Foo()`
		`4:     exits Bar()`
		`5:   then`
		`6: end`	

[ERR 101] Line 3:2 no viable alternative at input 'WHEN'	This message means the parser has encountered the token **WHEN** (a hard keyword) which is in the wrong place, since the rule name is missing.	`1: package org.drools;`
		`2: rule`
		`3:   when`
		`4:     Object()`
		`5:   then`
		`6:     System.out.println("A RHS");`
		`7: end`	

Error Message	Description	Example	

[ERR 101] Line 0:-1 no viable alternative at input '<eof>' in rule simple_rule in pattern [name]	Indicates an open quote, apostrophe or parentheses.	`1: rule simple_rule`
		`2:   when`
		`3:     Student( name == "Andy )`
		`4:   then`
		`5: end`	

[ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule simple_rule in pattern Bar	Indicates that the parser was looking for a particular symbol that it didn't end at the current input position.	`1: rule simple_rule`
		`2:   when`
		`3:     foo3 : Bar(`	

[ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule simple_rule in pattern [name]	This error is the result of an incomplete rule statement. Usually when you get a 0:-1 position, it means that parser reached the end of source. To fix this problem, it is necessary to complete the rule statement.	`1: package org.drools;`
		`2:`
		`3: rule "Avoid NPE on wrong syntax"`
		`4:   when`
		`5:     not( Cheese( ( type == "stilton", price == 10 ) || ( type == "brie", price == 15 ) ) from $cheeseList )`
		`6:   then`
		`7:     System.out.println("OK");`
		`8: end`	

[ERR 103] Line 7:0 rule 'rule_key' failed predicate: {(validateIdentifierKey(DroolsSoftKeywords.RULE))}? in rule	A validating semantic predicate evaluated to false. Usually these semantic predicates are used to identify soft keywords.	`1: package nesting;`
		` 2: dialect "mvel"`
		` 3:`
		` 4: import org.drools.Person`
		` 5: import org.drools.Address`
		` 6:`
		` 7: fdsfdsfds`
		` 8:`
		` 9: rule "test something"`
		`10:   when`
		`11:     p: Person( name=="Michael" )`
		`12:   then`
		`13:     p.name = "other";`
		`14:`
		`System.out.println(p.name);`
		`15: end`	

Error Message	Description	Example	

[ERR 104] Line 3:4 trailing semi-colon not allowed in rule simple_rule	This error is associated with the **`eval`** clause, where its expression may not be terminated with a semicolon. This problem is simple to fix: just remove the semi-colon.	`1: rule simple_rule`
		`2:   when`
		`3:     eval(abc();)`
		`4:   then`
		`5: end`	

[ERR 105] Line 2:2 required (...)+ loop did not match anything at input 'aa' in template test_error	The recognizer came to a subrule in the grammar that must match an alternative at least once, but the subrule did not match anything. To fix this problem it is necessary to remove the numeric value as it is neither a valid data type which might begin a new template slot nor a possible start for any other rule file construct.	`1: template test_error`
		`2:   aa s  11;`
		`3: end`	

Report a bug

7.4.4. Package

A *package* is a collection of rules and other related constructs, such as imports and globals. The package members are typically related to each other, such as HR rules. A package represents a namespace, which ideally is kept unique for a given grouping of rules. The package name itself is the namespace, and is not related to files or folders in any way.

It is possible to assemble rules from multiple rule sources, and have one top level package configuration that all the rules are kept under (when the rules are assembled). It is not possible to merge into the same package resources declared under different names. A single Rulebase may, however, contain multiple packages built on it. A common structure is to have all the rules for a package in the same file as the package declaration (so that is it entirely self-contained).

Report a bug

7.4.5. Import Statements

Import statements work like import statements in Java. You need to specify the fully qualified paths and type names for any objects you want to use in the rules. JBoss Rules automatically imports classes from the Java package of the same name, and also from the package **`java.lang`**.

Report a bug

7.4.6. Using Globals

In order to use globals you must:

1. Declare the global variable in the rules file and use it in rules. Example:

```
global java.util.List myGlobalList;

rule "Using a global"
when
    eval( true )
then
    myGlobalList.add( "Hello World" );
end
```

2. Set the global value on the working memory. It is best practice to set all global values before asserting any fact to the working memory. Example:

```
List list = new ArrayList();
WorkingMemory wm = rulebase.newStatefulSession();
wm.setGlobal( "myGlobalList", list );
```

Report a bug

7.4.7. The From Element

The *from* element allows you to pass a Hibernate session as a global. It also lets you pull data from a named Hibernate query.

Report a bug

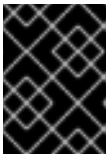
7.4.8. Using Globals with an e-Mail Service

Procedure 7.1. Task

1. Open the integration code that is calling the rule engine.
2. Obtain your emailService object and then set it in the working memory.
3. In the DRL, declare that you have a global of type emailService and give it the name "email".
4. In your rule consequences, you can use things like email.sendSMS(number, message).

WARNING

Globals are not designed to share data between rules and they should never be used for that purpose. Rules always reason and react to the working memory state, so if you want to pass data from rule to rule, assert the data as facts into the working memory.



IMPORTANT

Do not set or change a global value from inside the rules. We recommend to you always set the value from your application using the working memory interface.

Report a bug

7.5. DOMAIN SPECIFIC LANGUAGES (DSLS)

7.5.1. Domain Specific Languages

Domain Specific Languages (or DSLs) are a way of creating a rule language that is dedicated to your problem domain. A set of DSL definitions consists of transformations from DSL "sentences" to DRL constructs, which lets you use of all the underlying rule language and engine features. You can write rules in DSL rule (or DSLR) files, which will be translated into DRL files.

DSL and DSLR files are plain text files and you can use any text editor to create and modify them. There are also DSL and DSLR editors you can use, both in the IDE as well as in the web based BRMS, although they may not provide you with the full DSL functionality.

Report a bug

7.5.2. Using DSLs

DSLs can serve as a layer of separation between rule authoring (and rule authors) and the technical intricacies resulting from the modeling of domain object and the rule engine's native language and methods. A DSL hides implementation details and focuses on the rule logic proper. DSL sentences can also act as "templates" for conditional elements and consequence actions that are used repeatedly in your rules, possibly with minor variations. You may define DSL sentences as being mapped to these repeated phrases, with parameters providing a means for accommodating those variations.

Report a bug

7.5.3. DSL Example

Table 7.3. DSL Example

Example	Description
`[when]Something is {colour}=Something(colour=="{colour}")`	**[when]** indicates the scope of the expression (that is, whether it is valid for the LHS or the RHS of a rule). The part after the bracketed keyword is the expression that you use in the rule. The part to the right of the equal sign ("=") is the mapping of the expression into the rule language. The form of this string depends on its destination, RHS or LHS. If it is for the LHS, then it ought to be a term according to the regular LHS syntax; if it is for the RHS then it might be a Java statement.

Report a bug

7.5.4. How the DSL Parser Works

Whenever the DSL parser matches a line from the rule file written in the DSL with an expression in the DSL definition, it performs three steps of string manipulation:

- The DSL extracts the string values appearing where the expression contains variable names in brackets.
- The values obtained from these captures are interpolated wherever that name occurs on the right hand side of the mapping.
- The interpolated string replaces whatever was matched by the entire expression in the line of the DSL rule file.

NOTE

You can use (for instance) a '?' to indicate that the preceding character is optional. One good reason to use this is to overcome variations in natural language phrases of your DSL. But, given that these expressions are regular expression patterns, this means that all wildcard characters in Java's pattern syntax have to be escaped with a preceding backslash ('\').

Report a bug

7.5.5. The DSL Compiler

The DSL compiler transforms DSL rule files line by line. If you do not wish for this to occur, ensure that the captures are surrounded by characteristic text (words or single characters). As a result, the matching operation done by the parser plucks out a substring from somewhere within the line. In the example below, quotes are used as distinctive characters. (The characters that surround the capture are not included during interpolation, just the contents between them.)

Report a bug

7.5.6. DSL Syntax Examples

Table 7.4. DSL Syntax Examples

Name	Description	Example
Quotes	Use quotes for textual data that a rule editor may want to enter. You can also enclose the capture with words to ensure that the text is correctly matched.	`[when]something is "{color}"=Something(color=="{color}")` `[when]another {state} thing=OtherThing(state=="{state}"`
Braces	In a DSL mapping, the braces "{" and "}" should only be used to enclose a variable definition or reference, resulting in a capture. If they should occur literally, either in the expression or within the replacement text on the right hand side, they must be escaped with a preceding backslash ("\").	`[then]do something= if (foo) \{ doSomething(); \}`

Mapping with correct syntax example	n/a	`# This is a comment to be ignored.`
		`[when]There is a person with name of "{name}"=Person(name=="{name}")`
		`[when]Person is at least {age} years old and lives in "{location}"= Person(age >= {age}, location=="{location}")`
		`[then]Log "{message}"=System.out.println("{message}");`
		`[when]And = and`

Name	Description	Example
Expanded DSL example	n/a	`There is a person with name of "Kitty" ==> Person(name="Kitty") Person is at least 42 years old and lives in "Atlanta" ==> Person(age >= 42, location="Atlanta") Log "boo" ==> System.out.println("boo"); There is a person with name of "Bob" and Person is at least 30 years old and lives in "Utah" ==> Person(name="Bob") and Person(age >= 30, location="Utah")`

NOTE

If you are capturing plain text from a DSL rule line and want to use it as a string literal in the expansion, you must provide the quotes on the right hand side of the mapping.

Report a bug

7.5.7. Chaining DSL Expressions

DSL expressions can be chained together one one line to be used at once. It must be clear where one ends and the next one begins and where the text representing a parameter ends. (Otherwise you risk getting all the text until the end of the line as a parameter value.) The DSL expressions are tried, one after the other, according to their order in the DSL definition file. After any match, all remaining DSL expressions are investigated, too.

Report a bug

7.5.8. Adding Constraints to Facts

Table 7.5. Adding Constraints to Facts

Name	Description	Example
Expressing LHS conditions	The DSL facility allows you to add constraints to a pattern by a simple convention: if your DSL expression starts with a hyphen (minus character, "-") it is assumed to be a field constraint and, consequently, is is added to the last pattern line preceding it. In the example, the class **`Cheese`**, has these fields: type, price, age and country. You can express some LHS condition in normal DRL.	`Cheese(age < 5, price == 20, type=="stilton", country=="ch")`

DSL definitions	The DSL definitions given in this example result in three DSL phrases which may be used to create any combination of constraint involving these fields.	`[when]There is a Cheese with=Cheese()`
		`[when]- age is less than {age}=age<{age}`
		`[when]- type is '{type}'=type=='{type}'`
		`[when]- country equal to '{country}'=country=='{country}'`

"-"	The parser will pick up a line beginning with "-" and add it as a constraint to the preceding pattern, inserting a comma when it is required.	`There is a Cheese with`
		`- age is less than 42`
		`- type is 'stilton'`
		
		`Cheese(age<42, type=='stilton')`

Name	Description	Example

Defining DSL phrases	Defining DSL phrases for various operators and even a generic expression that handles any field constraint reduces the amount of DSL entries.	`[when][]is less than or equal to=<=`
		`[when][]is less than=<`
		`[when][]is greater than or equal to=>=`
		`[when][]is greater than=>`
		`[when][]is equal to===`
		`[when][]equals===`
		`[when][]There is a Cheese with=Cheese()`
		[when][]- {field:\w*} {operator} {value:\d*}={field} {operator} {value}

DSL definition rule	n/a	`There is a Cheese with`
		`  - age is less than 42`
		`  - rating is greater than 50`
		`  - type equals 'stilton'`
		In this specific case, a phrase such as "is less than" is replaced by **<**, and then the line matches the last DSL entry. This removes the hyphen, but the final result is still added as a constraint to the preceding pattern. After processing all of the lines, the resulting DRL text is:
		`Cheese(age<42, rating > 50, type=='stilton')`

NOTE

The order of the entries in the DSL is important if separate DSL expressions are intended to match the same line, one after the other.

Report a bug

7.5.9. Tips for Developing DSLs

- Write representative samples of the rules your application requires and test them as you develop.
- Rules, both in DRL and in DSLR, refer to entities according to the data model representing the application data that should be subject to the reasoning process defined in rules.
- Writing rules is easier if most of the data model's types are facts.
- Mark variable parts as parameters. This provides reliable leads for useful DSL entries.
- You may postpone implementation decisions concerning conditions and actions during this first design phase by leaving certain conditional elements and actions in their DRL form by prefixing a line with a greater sign (">"). (This is also handy for inserting debugging statements.)
- New rules can be written by reusing the existing DSL definitions, or by adding a parameter to an existing condition or consequence entry.
- Keep the number of DSL entries small. Using parameters lets you apply the same DSL sentence for similar rule patterns or constraints.

Report a bug

7.5.10. DSL and DSLR Reference

A DSL file is a text file in a line-oriented format. Its entries are used for transforming a DSLR file into a file according to DRL syntax:

- A line starting with "#" or "//" (with or without preceding white space) is treated as a comment. A comment line starting with "#/" is scanned for words requesting a debug option, see below.
- Any line starting with an opening bracket ("[") is assumed to be the first line of a DSL entry definition.
- Any other line is appended to the preceding DSL entry definition, with the line end replaced by a space.

Report a bug

7.5.11. The Make Up of a DSL Entry

A DSL entry consists of the following four parts:

- A scope definition, written as one of the keywords "when" or "condition", "then" or "consequence", "*" and "keyword", enclosed in brackets ("[" and "]"). This indicates whether the DSL entry is valid for the condition or the consequence of a rule, or both. A scope indication of "keyword" means that the entry has global significance, that is, it is recognized anywhere in a DSLR file.
- A type definition, written as a Java class name, enclosed in brackets. This part is optional unless the next part begins with an opening bracket. An empty pair of brackets is valid, too.
- A DSL expression consists of a (Java) regular expression, with any number of embedded *variable definitions,* terminated by an equal sign ("="). A variable definition is enclosed in braces ("{" and "}"). It consists of a variable name and two optional attachments, separated by colons

(":"). If there is one attachment, it is a regular expression for matching text that is to be assigned to the variable. If there are two attachments, the first one is a hint for the GUI editor and the second one the regular expression.

Note that all characters that are "magic" in regular expressions must be escaped with a preceding backslash ("\") if they should occur literally within the expression.

- The remaining part of the line after the delimiting equal sign is the replacement text for any DSLR text matching the regular expression. It may contain variable references, i.e., a variable name enclosed in braces. Optionally, the variable name may be followed by an exclamation mark ("!") and a transformation function, see below.

  Note that braces ("{" and "}") must be escaped with a preceding backslash ("\") if they should occur literally within the replacement string.

Report a bug

7.5.12. Debug Options for DSL Expansion

Table 7.6. Debug Options for DSL Expansion

Word	Description
result	Prints the resulting DRL text, with line numbers.
steps	Prints each expansion step of condition and consequence lines.
keyword	Dumps the internal representation of all DSL entries with scope "keyword".
when	Dumps the internal representation of all DSL entries with scope "when" or "*".
then	Dumps the internal representation of all DSL entries with scope "then" or "*".
usage	Displays a usage statistic of all DSL entries.

Report a bug

7.5.13. DSL Definition Example

This is what a DSL definition looks like:

```
# Comment: DSL examples

#/ debug: display result and usage

# keyword definition: replaces "regula" by "rule"
[keyword][]regula=rule
```

```
# conditional element: "T" or "t", "a" or "an", convert matched word
[when][][Tt]here is an? {entity:\w+}=
        ${entity!lc}: {entity!ucfirst} ()

# consequence statement: convert matched word, literal braces
[then][]update {entity:\w+}=modify( ${entity!lc} )\{ \}
```

Report a bug

7.5.14. Transformation of a DSLR File

The transformation of a DSLR file proceeds as follows:

1. The text is read into memory.

2. Each of the "keyword" entries is applied to the entire text. The regular expression from the keyword definition is modified by replacing white space sequences with a pattern matching any number of white space characters, and by replacing variable definitions with a capture made from the regular expression provided with the definition, or with the default (".*?"). Then, the DSLR text is searched exhaustively for occurrences of strings matching the modified regular expression. Substrings of a matching string corresponding to variable captures are extracted and replace variable references in the corresponding replacement text, and this text replaces the matching string in the DSLR text.

3. Sections of the DSLR text between "when" and "then", and "then" and "end", respectively, are located and processed in a uniform manner, line by line, as described below.

   For a line, each DSL entry pertaining to the line's section is taken in turn, in the order it appears in the DSL file. Its regular expression part is modified: white space is replaced by a pattern matching any number of white space characters; variable definitions with a regular expression are replaced by a capture with this regular expression, its default being ".*?". If the resulting regular expression matches all or part of the line, the matched part is replaced by the suitably modified replacement text.

   Modification of the replacement text is done by replacing variable references with the text corresponding to the regular expression capture. This text may be modified according to the string transformation function given in the variable reference; see below for details.

   If there is a variable reference naming a variable that is not defined in the same entry, the expander substitutes a value bound to a variable of that name, provided it was defined in one of the preceding lines of the current rule.

4. If a DSLR line in a condition is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a pattern CE, that is, a type name followed by a pair of parentheses. if this pair is empty, the expanded line (which should contain a valid constraint) is simply inserted, otherwise a comma (",") is inserted beforehand.

   If a DSLR line in a consequence is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a "modify" statement, ending in a pair of braces ("{" and "}"). If this pair is empty, the expanded line (which should contain a valid method call) is simply inserted, otherwise a comma (",") is inserted beforehand.

NOTE

It is currently *not* possible to use a line with a leading hyphen to insert text into other conditional element forms (e.g., "accumulate") or it may only work for the first insertion (e.g., "eval").

Report a bug

7.5.15. String Transformation Functions

Table 7.7. String Transformation Functions

Name	Description
uc	Converts all letters to upper case.
lc	Converts all letters to lower case.
ucfirst	Converts the first letter to upper case, and all other letters to lower case.
num	Extracts all digits and "-" from the string. If the last two digits in the original string are preceded by "." or ",", a decimal period is inserted in the corresponding position.
a?*b*/*c*	Compares the string with string *a*, and if they are equal, replaces it with *b*, otherwise with *c*. But *c* can be another triplet *a*, *b*, *c*, so that the entire structure is, in fact, a translation table.

Report a bug

7.5.16. Stringing DSL Transformation Functions

Table 7.8. Stringing DSL Transformation Functions

Name	Description	Example

Name	Description	Example

.dsl	A file containing a DSL definition is customarily given the extension **`.dsl`**. It is passed to the Knowledge Builder with **`ResourceType.DSL`**. For a file using DSL definition, the extension **`.dslr`** should be used. The Knowledge Builder expects **`ResourceType.DSLR`**. The IDE, however, relies on file extensions to correctly recognize and work with your rules file.	`# definitions for conditions`
		`[when][]There is an? {entity}=${entity!lc}: {entity!ucfirst}()`
		`[when][]- with an? {attr} greater than {amount}={attr} <= {amount!num}`
		`[when][]- with a {what} {attr}={attr} {what!positive? >0/negative? %lt;0/zero? ==0/ERROR}`

DSL passing	The DSL must be passed to the Knowledge Builder ahead of any rules file using the DSL.
	
	For parsing and expanding a DSLR file the DSL configuration is read and supplied to the parser. Thus, the parser can "recognize" the DSL expressions and transform them into native rule language expressions.
		`KnowledgeBuilder kBuilder = new KnowledgeBuilder();`
		`Resource dsl = ResourceFactory.newClassPathResource( dslPath, getClass() );`
		`kBuilder.add( dsl, ResourceType.DSL );`
		`Resource dslr = ResourceFactory.newClassPathResource( dslrPath, getClass() );`
		`kBuilder.add( dslr, ResourceType.DSLR );`

Report a bug

CHAPTER 8. RULE COMMANDS

8.1. AVAILABLE API

XML marshalling/unmarshalling of the Drools Commands requires the use of special classes, which are going to be described in the following sections.

The following urls show sample script examples for jaxb, xstream and json marshalling using:

- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD
- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD
- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/xstream.mvt?r=HEAD

XStream

To use the XStream commands marshaller you need to use the DroolsHelperProvider to obtain an XStream instance. We need to use this because it has the commands converters registered.

- Marshalling

  BatchExecutionHelperProviderImpl.newXStreamMarshaller().toXML(command);

- Unmarshalling

  BatchExecutionHelperProviderImpl.newXStreamMarshaller().fromXML(xml)

JSON

JSON API to marshalling/unmarshalling is similar to XStream API:

- Marshalling

  BatchExecutionHelper.newJSonMarshaller().toXML(command);

- Unmarshalling

  BatchExecutionHelper.newJSonMarshaller().fromXML(xml)

JAXB

There are two options for using JAXB, you can define your model in an XSD file or you can have a POJO model. In both cases you have to declare your model inside JAXBContext, and in order to do that you need to use Drools Helper classes. Once you have the JAXBContext you need to create the Unmarshaller/Marshaller as needed.

Using an XSD file to define the model

With your model defined in a XSD file you need to have a KnowledgeBase that has your XSD model added as a resource.

To do this, the XSD file must be added as a XSD ResourceType into the KnowledgeBuilder. Finally you can create the JAXBContext using the KnowledgeBase created with the KnowledgeBuilder

```
Options xjcOpts = new Options();
xjcOpts.setSchemaLanguage(Language.XMLSCHEMA);
```

```
JaxbConfiguration jaxbConfiguration =
KnowledgeBuilderFactory.newJaxbConfiguration( xjcOpts, "xsd" );
kbuilder.add(ResourceFactory.newClassPathResource("person.xsd",
getClass()), ResourceType.XSD, jaxbConfiguration);
KnowledgeBase kbase = kbuilder.newKnowledgeBase();

List<String> classesName = new ArrayList<String>();
classesName.add("org.drools.compiler.test.Person");

JAXBContext jaxbContext =
KnowledgeBuilderHelper.newJAXBContext(classesName.toArray(new
String[classesName.size()]), kbase);
```

Using a POJO model

In this case you need to use DroolsJaxbHelperProviderImpl to create the JAXBContext. This class has two parameters:

1. classNames: A List with the canonical name of the classes that you want to use in the marshalling/unmarshalling process.

2. properties: JAXB custom properties

```
List<String> classNames = new ArrayList<String>();
classNames.add("org.drools.compiler.test.Person");
JAXBContext jaxbContext =
DroolsJaxbHelperProviderImpl.createDroolsJaxbContext(classNames, null);
Marshaller marshaller = jaxbContext.createMarshaller();
```

Report a bug

8.2. COMMANDS SUPPORTED

Currently, the following commands are supported:

- BatchExecutionCommand
- InsertObjectCommand
- RetractCommand
- ModifyCommand
- GetObjectCommand
- InsertElementsCommand
- FireAllRulesCommand
- StartProcessCommand
- SignalEventCommand
- CompleteWorkItemCommand
- AbortWorkItemCommand

- QueryCommand
- SetGlobalCommand
- GetGlobalCommand
- GetObjectsCommand

NOTE

In the next snippets code we are going to use a POJO org.drools.compiler.test.Person that has two fields

- name: String
- age: Integer

NOTE

In the next examples, to marshall the commands we have used the next snippet codes:

- XStream

```
String xml =
BatchExecutionHelper.newXStreamMarshaller().toXML(command
);
```

- JSON

```
String xml =
BatchExecutionHelper.newJSonMarshaller().toXML(command);
```

- JAXB

```
Marshaller marshaller = jaxbContext.createMarshaller();
StringWriter xml = new StringWriter();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
true);
marshaller.marshal(command, xml);
```

Report a bug

8.3. COMMANDS

8.3.1. BatchExecutionCommand

- Description: The command that contains a list of commands, which will be sent and executed.
- Attributes

Table 8.1. BatchExecutionCommand attributes

Name	Description	required
lookup	Sets the knowledge session id on which the commands are going to be executed	true
commands	List of commands to be executed	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
InsertObjectCommand insertObjectCommand = new
InsertObjectCommand(new Person("john", 25));
FireAllRulesCommand fireAllRulesCommand = new FireAllRulesCommand();
command.getCommands().add(insertObjectCommand);
command.getCommands().add(fireAllRulesCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <insert>
    <org.drools.compiler.test.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.compiler.test.Person>
  </insert>
  <fire-all-rules/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":[{"insert":
{"object":{"org.drools.compiler.test.Person":
{"name":"john","age":25}}}},{"fire-all-rules":""}]}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <insert>
        <object xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>25</age>
            <name>john</name>
        </object>
    </insert>
    <fire-all-rules max="-1"/>
</batch-execution>
```

Report a bug

8.3.2. InsertObjectCommand

- Description: Insert an object in the knowledge session.
- Attributes

Table 8.2. InsertObjectCommand attributes

Name	Description	required
object	The object to be inserted	true
outIdentifier	Id to identify the FactHandle created in the object insertion and added to the execution results	false
returnObject	Boolean to establish if the object must be returned in the execution results. Default value: true	false
entryPoint	Entrypoint for the insertion	false

- Command creation

```
List<Command> cmds = ArrayList<Command>();

Command insertObjectCommand = CommandFactory.newInsert(new
Person("john", 25), "john", false, null);
cmds.add( insertObjectCommand );

BatchExecutionCommand command =
CommandFactory.createBatchExecution(cmds, "ksession1" );
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <insert out-identifier="john" entry-point="my stream" return-
object="false">
    <org.drools.compiler.test.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.compiler.test.Person>
  </insert>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"insert":
{"entry-point":"my stream", "out-identifier":"john","return-
object":false,"object":{"org.drools.compiler.test.Person":
{"name":"john","age":25}}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <insert out-identifier="john" entry-point="my stream" >
        <object xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>25</age>
            <name>john</name>
        </object>
    </insert>
</batch-execution>
```

Report a bug

8.3.3. RetractCommand

- Description: Retract an object from the knowledge session.
- Attributes

Table 8.3. RetractCommand attributes

Name	Description	required
handle	The FactHandle associated to the object to be retracted	true

- Command creation: we have two options, with the same output result:

1. Create the Fact Handle from a string

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
RetractCommand retractCommand = new RetractCommand();
retractCommand.setFactHandleFromString("123:234:345:456:567");
command.getCommands().add(retractCommand);
```

2. Set the Fact Handle that you received when the object was inserted

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
RetractCommand retractCommand = new RetractCommand(factHandle);
command.getCommands().add(retractCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <retract fact-handle="0:234:345:456:567"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"retract":
{"fact-handle":"0:234:345:456:567"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <retract fact-handle="0:234:345:456:567"/>
</batch-execution>
```

Report a bug

8.3.4. ModifyCommand

- Description: Allows you to modify a previously inserted object in the knowledge session.
- Attributes

Table 8.4. ModifyCommand attributes

Name	Description	required
handle	The FactHandle associated to the object to be retracted	true
setters	List of setters object's modifications	true

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
ModifyCommand modifyCommand = new ModifyCommand();
modifyCommand.setFactHandleFromString("123:234:345:456:567");
List<Setter> setters = new ArrayList<Setter>();
setters.add(new SetterImpl("age", "30"));
modifyCommand.setSetters(setters);
command.getCommands().add(modifyCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
```

```
  <modify fact-handle="0:234:345:456:567">
    <set accessor="age" value="30"/>
  </modify>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"modify":
{"fact-handle":"0:234:345:456:567","setters":
{"accessor":"age","value":30}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <modify fact-handle="0:234:345:456:567">
        <set value="30" accessor="age"/>
    </modify>
</batch-execution>
```

Report a bug

8.3.5. GetObjectCommand

- Description: Used to get an object from a knowledge session
- Attributes

Table 8.5. GetObjectCommand attributes

Name	Description	required
factHandle	The FactHandle associated to the object to be retracted	true
outIdentifier	Id to identify the FactHandle created in the object insertion and added to the execution results	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetObjectCommand getObjectCommand = new GetObjectCommand();
getObjectCommand.setFactHandleFromString("123:234:345:456:567");
getObjectCommand.setOutIdentifier("john");
command.getCommands().add(getObjectCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <get-object fact-handle="0:234:345:456:567" out-
identifier="john"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-
object":{"fact-handle":"0:234:345:456:567","out-
identifier":"john"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <get-object out-identifier="john" fact-
handle="0:234:345:456:567"/>
</batch-execution>
```

Report a bug

8.3.6. InsertElementsCommand

- Description: Used to insert a list of objects.
- Attributes

Table 8.6. InsertElementsCommand attributes

Name	Description	required
objects	The list of objects to be inserted on the knowledge session	true
outIdentifier	Id to identify the FactHandle created in the object insertion and added to the execution results	false
returnObject	Boolean to establish if the object must be returned in the execution results. Default value: true	false
entryPoint	Entrypoint for the insertion	false

- Command creation

```
List<Command> cmds = ArrayList<Command>();
```

```
List<Object> objects = new ArrayList<Object>();
objects.add(new Person("john", 25));
objects.add(new Person("sarah", 35));

Command insertElementsCommand = CommandFactory.newInsertElements(
objects );
cmds.add( insertElementsCommand );

BatchExecutionCommand command =
CommandFactory.createBatchExecution(cmds, "ksession1" );
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <insert-elements>
    <org.drools.compiler.test.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.compiler.test.Person>
    <org.drools.compiler.test.Person>
      <name>sarah</name>
      <age>35</age>
    </org.drools.compiler.test.Person>
  </insert-elements>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"insert-
elements":{"objects":[{"containedObject":
{"@class":"org.drools.compiler.test.Person","name":"john","age":2
5}},{"containedObject":
{"@class":"Person","name":"sarah","age":35}}]}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <insert-elements return-objects="true">
        <list>
            <element xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                <age>25</age>
                <name>john</name>
            </element>
            <element xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                <age>35</age>
                <name>sarah</name>
            </element>
        <list>
    </insert-elements>
</batch-execution>
```

Report a bug

8.3.7. FireAllRulesCommand

- Description: Allow execution of the rules activations created.
- Attributes

Table 8.7. FireAllRulesCommand attributes

Name	Description	required
max	The max number of rules activations to be executed. default is -1 and will not put any restriction on execution	false
outIdentifier	Add the number of rules activations fired on the execution results	false
agendaFilter	Allow the rules execution using an Agenda Filter	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
FireAllRulesCommand fireAllRulesCommand = new FireAllRulesCommand();
fireAllRulesCommand.setMax(10);
fireAllRulesCommand.setOutIdentifier("firedActivations");
command.getCommands().add(fireAllRulesCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <fire-all-rules max="10" out-identifier="firedActivations"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"fire-all-
rules":{"max":10,"out-identifier":"firedActivations"}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
   <fire-all-rules out-identifier="firedActivations" max="10"/>
```

```
</batch-execution>
```

Report a bug

8.3.8. StartProcessCommand

- Description: Allows you to start a process using the ID. Also you can pass parameters and initial data to be inserted.
- Attributes

Table 8.8. StartProcessCommand attributes

Name	Description	required
processId	The ID of the process to be started	true
parameters	A Map<String, Object> to pass parameters in the process startup	false
data	A list of objects to be inserted in the knowledge session before the process startup	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
StartProcessCommand startProcessCommand = new StartProcessCommand();
startProcessCommand.setProcessId("org.drools.task.processOne");
command.getCommands().add(startProcessCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <start-process processId="org.drools.task.processOne"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"start-
process":{"process-id":"org.drools.task.processOne"}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <start-process processId="org.drools.task.processOne">
```

```
        <parameter/>
    </start-process>
</batch-execution>
```

Report a bug

8.3.9. SignalEventCommand

- Description: Send a signal event.
- Attributes

Table 8.9. SignalEventCommand attributes

Name	Description	required
event-type		true
processInstanceId		false
event		false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
SignalEventCommand signalEventCommand = new SignalEventCommand();
signalEventCommand.setProcessInstanceId(1001);
signalEventCommand.setEventType("start");
signalEventCommand.setEvent(new Person("john", 25));
command.getCommands().add(signalEventCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <signal-event process-instance-id="1001" event-type="start">
    <org.drools.pipeline.camel.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.pipeline.camel.Person>
  </signal-event>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"signal-
event":{"process-instance-id":1001,"@event-type":"start","event-
type":"start","object":{"org.drools.pipeline.camel.Person":
{"name":"john","age":25}}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <signal-event event-type="start" process-instance-id="1001">
        <event xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>25</age>
            <name>john</name>
        </event>
    </signal-event>
</batch-execution>
```

Report a bug

8.3.10. CompleteWorkItemCommand

- Description: Allows you to complete a WorkItem.
- Attributes

Table 8.10. CompleteWorkItemCommand attributes

Name	Description	required
workItemId	The ID of the WorkItem to be completed	true
results		false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
CompleteWorkItemCommand completeWorkItemCommand = new
CompleteWorkItemCommand();
completeWorkItemCommand.setWorkItemId(1001);
command.getCommands().add(completeWorkItemCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <complete-work-item id="1001"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"complete-
work-item":{"id":1001}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <complete-work-item id="1001"/>
</batch-execution>
```

Report a bug

8.3.11. AbortWorkItemCommand

- Description: Allows you abort an WorkItem. The same as session.getWorkItemManager().abortWorkItem(workItemId)
- Attributes

Table 8.11. AbortWorkItemCommand attributes

Name	Description	required
workItemId	The ID of the WorkItem to be completed	true

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
AbortWorkItemCommand abortWorkItemCommand = new
AbortWorkItemCommand();
abortWorkItemCommand.setWorkItemId(1001);
command.getCommands().add(abortWorkItemCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <abort-work-item id="1001"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"abort-work-
item":{"id":1001}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <abort-work-item id="1001"/>
</batch-execution>
```

Report a bug

8.3.12. QueryCommand

- Description: Executes a query defined in knowledge base.
- Attributes

Table 8.12. QueryCommand attributes

Name	Description	required
name	The query name	true
outIdentifier	The identifier of the query results. The query results are going to be added in the execution results with this identifier	false
arguments	A list of objects to be passed as a query parameter	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
QueryCommand queryCommand = new QueryCommand();
queryCommand.setName("persons");
queryCommand.setOutIdentifier("persons");
command.getCommands().add(queryCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <query out-identifier="persons" name="persons"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"query":
{"out-identifier":"persons","name":"persons"}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <query name="persons" out-identifier="persons"/>
</batch-execution>
```

Report a bug

8.3.13. SetGlobalCommand

- Description: Allows you to set a global.
- Attributes

Table 8.13. SetGlobalCommand attributes

Name	Description	required
identifier	The identifier of the global defined in the knowledge base	true
object	The object to be set into the global	false
out	A boolean to add, or not, the set global result into the execution results	false
outIdentifier	The identifier of the global execution result	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
SetGlobalCommand setGlobalCommand = new SetGlobalCommand();
setGlobalCommand.setIdentifier("helper");
setGlobalCommand.setObject(new Person("kyle", 30));
setGlobalCommand.setOut(true);
setGlobalCommand.setOutIdentifier("output");
command.getCommands().add(setGlobalCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <set-global identifier="helper" out-identifier="output">
    <org.drools.compiler.test.Person>
      <name>kyle</name>
      <age>30</age>
    </org.drools.compiler.test.Person>
  </set-global>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"set-
```

```
global":{"identifier":"helper","out-
identifier":"output","object":{"org.drools.compiler.test.Person":
{"name":"kyle","age":30}}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <set-global out="true" out-identifier="output"
identifier="helper">
        <object xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>30</age>
            <name>kyle</name>
        </object>
    </set-global>
</batch-execution>
```

Report a bug

8.3.14. GetGlobalCommand

- Description: Allows you to get a global previously defined.
- Attributes

Table 8.14. GetGlobalCommand attributes

Name	Description	required
identifier	The identifier of the global defined in the knowledge base	true
outIdentifier	The identifier to be used in the execution results	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetGlobalCommand getGlobalCommand = new GetGlobalCommand();
getGlobalCommand.setIdentifier("helper");
getGlobalCommand.setOutIdentifier("helperOutput");
command.getCommands().add(getGlobalCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <get-global identifier="helper" out-identifier="helperOutput"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-
global":{"identifier":"helper","out-
identifier":"helperOutput"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <get-global out-identifier="helperOutput"
identifier="helper"/>
</batch-execution>
```

Report a bug

8.3.15. GetObjectsCommand

- Description: Returns all the objects from the current session as a Collection.
- Attributes

Table 8.15. GetObjectsCommand attributes

Name	Description	required
objectFilter	An ObjectFilter to filter the objects returned from the current session	false
outIdentifier	The identifier to be used in the execution results	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetObjectsCommand getObjectsCommand = new GetObjectsCommand();
getObjectsCommand.setOutIdentifier("objects");
command.getCommands().add(getObjectsCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <get-objects out-identifier="objects"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-
```

```
objects":{"out-identifier":"objects"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <get-objects out-identifier="objects"/>
</batch-execution>
```

Report a bug

CHAPTER 9. XML

9.1. THE XML FORMAT

WARNING

The XML rule language has not been updated to support functionality introduced in Drools 5.x and is consider a deprecated feature.

As an option, JBoss Rules supports a "native" rule language as an alternative to DRL. This allows you to capture and manage your rules as XML data. Just like the non-XML DRL format, the XML format is parsed into the internal "AST" representation as fast as possible (using a SAX parser). There is no external transformation step required.

Report a bug

9.2. XML RULE EXAMPLE

This is what a rule looks like in XML:

```
<?xml version="1.0" encoding="UTF-8"?>

<package name="com.sample"
         xmlns="http://drools.org/drools-5.0"
         xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
         xs:schemaLocation="http://drools.org/drools-5.0 drools-5.0.xsd">

<import name="java.util.HashMap" />
<import name="org.drools.*" />

<global identifier="x" type="com.sample.X" />
<global identifier="yada" type="com.sample.Yada" />

<function return-type="void" name="myFunc">
    <parameter identifier="foo" type="Bar" />
    <parameter identifier="bada" type="Bing" />

    <body>
     System.out.println("hello world");
    </body>
</function>

<rule name="simple_rule">
<rule-attribute name="salience" value="10" />
<rule-attribute name="no-loop" value="true" />
<rule-attribute name="agenda-group" value="agenda-group" />
<rule-attribute name="activation-group" value="activation-group" />

<lhs>
```

```
    <pattern identifier="foo2" object-type="Bar" >
            <or-constraint-connective>
                <and-constraint-connective>
                    <field-constraint field-name="a">
                        <or-restriction-connective>
                            <and-restriction-connective>
                                <literal-restriction evaluator=">"
value="60" />
                                <literal-restriction evaluator="<"
value="70" />
                            </and-restriction-connective>
                            <and-restriction-connective>
                                <literal-restriction evaluator="<"
value="50" />
                                <literal-restriction evaluator=">"
value="55" />
                            </and-restriction-connective>
                        </or-restriction-connective>
                    </field-constraint>

                    <field-constraint field-name="a3">
                        <literal-restriction evaluator="==" value="black"
/>
                    </field-constraint>
                </and-constraint-connective>

                <and-constraint-connective>
                    <field-constraint field-name="a">
                        <literal-restriction evaluator="==" value="40" />
                    </field-constraint>

                    <field-constraint field-name="a3">
                        <literal-restriction evaluator="==" value="pink"
/>
                    </field-constraint>
                </and-constraint-connective>

                <and-constraint-connective>
                    <field-constraint field-name="a">
                        <literal-restriction evaluator="==" value="12"/>
                    </field-constraint>

                    <field-constraint field-name="a3">
                        <or-restriction-connective>
                            <literal-restriction evaluator="=="
value="yellow"/>
                            <literal-restriction evaluator="=="
value="blue" />
                        </or-restriction-connective>
                    </field-constraint>
                </and-constraint-connective>
            </or-constraint-connective>
        </pattern>

        <not>
            <pattern object-type="Person">
```

```
                <field-constraint field-name="likes">
                    <variable-restriction evaluator="==" 
identifier="type"/>
                </field-constraint>
            </pattern>

            <exists>
                <pattern object-type="Person">
                    <field-constraint field-name="likes">
                        <variable-restriction evaluator="==" 
identifier="type"/>
                    </field-constraint>
                </pattern>
            </exists>
        </not>

        <or-conditional-element>
            <pattern identifier="foo3" object-type="Bar" >
                <field-constraint field-name="a">
                    <or-restriction-connective>
                        <literal-restriction evaluator="==" value="3" />
                        <literal-restriction evaluator="==" value="4" />
                    </or-restriction-connective>
                </field-constraint>
                <field-constraint field-name="a3">
                    <literal-restriction evaluator="==" value="hello" />
                </field-constraint>
                <field-constraint field-name="a4">
                    <literal-restriction evaluator="==" value="null" />
                </field-constraint>
            </pattern>

            <pattern identifier="foo4" object-type="Bar" >
                <field-binding field-name="a" identifier="a4" />
                <field-constraint field-name="a">
                    <literal-restriction evaluator="!=" value="4" />
                    <literal-restriction evaluator="!=" value="5" />
                </field-constraint>
            </pattern>
        </or-conditional-element>

        <pattern identifier="foo5" object-type="Bar" >
            <field-constraint field-name="b">
                <or-restriction-connective>
                    <return-value-restriction evaluator="==" >a4 + 
1</return-value-restriction>
                    <variable-restriction evaluator=">" identifier="a4" 
/>
                    <qualified-identifier-restriction evaluator="==">
                        org.drools.Bar.BAR_ENUM_VALUE
                    </qualified-identifier-restriction>
                </or-restriction-connective>
            </field-constraint>
        </pattern>

        <pattern identifier="foo6" object-type="Bar" >
```

```
            <field-binding field-name="a" identifier="a4" />
            <field-constraint field-name="b">
                <literal-restriction evaluator="==" value="6" />
            </field-constraint>
        </pattern>
  </lhs>
 <rhs>
    if ( a == b ) {
      assert( foo3 );
    } else {
      retract( foo4 );
    }
    System.out.println( a4 );
   </rhs>
</rule>

</package>
```

Report a bug

9.3. XML ELEMENTS

Table 9.1. XML Elements

Name	Description
global	Defines global objects that can be referred to in the rules.
function	Contains a function declaration for a function to be used in the rules. You have to specify a return type, a unique name and parameters, in the body goes a snippet of code.
import	Imports the types you wish to use in the rule.

Report a bug

9.4. DETAIL OF A RULE ELEMENT

This example rule has LHS and RHS (conditions and consequence) sections. The RHS is a block of semantic code that will be executed when the rule is activated. The LHS is slightly more complicated as it contains nested elements for conditional elements, constraints and restrictions:

```
<rule name="simple_rule">
<rule-attribute name="salience" value="10" />
<rule-attribute name="no-loop" value="true" />
<rule-attribute name="agenda-group" value="agenda-group" />
<rule-attribute name="activation-group" value="activation-group" />

<lhs>
    <pattern identifier="cheese" object-type="Cheese">
```

```
        <from>
            <accumulate>
                <pattern object-type="Person"></pattern>
                <init>
                    int total = 0;
                </init>
                <action>
                    total += $cheese.getPrice();
                </action>
                <result>
                    new Integer( total ) );
                </result>
            </accumulate>
        </from>
    </pattern>

    <pattern identifier="max" object-type="Number">
        <from>
            <accumulate>
                <pattern identifier="cheese" object-type="Cheese">
</pattern>
                <external-function evaluator="max" expression="$price"/>
            </accumulate>
        </from>
    </pattern>
</lhs>
<rhs>
    list1.add( $cheese );
</rhs>
</rule>
```

Report a bug

9.5. XML RULE ELEMENTS

Table 9.2. XML Rule Elements

Element	Description
Pattern	This allows you to specify a type (class) and perhaps bind a variable to an instance of that class. Nested under the pattern object are constraints and restrictions that have to be met. The Predicate and Return Value constraints allow Java expressions to be embedded.
Conditional elements (not, exists, and, or)	These work like their DRL counterparts. Elements that are nested under and an "and" element are logically "anded" together. Likewise with "or" (and you can nest things further). "Exists" and "Not" work around patterns, to check for the existence or nonexistence of a fact meeting the pattern's constraints.

Element	Description
Eval	Allows the execution of a valid snippet of Java code as long as it evaluates to a boolean (do not end it with a semi-colon, as it is just a fragment). This can include calling a function. The Eval is less efficient than the columns, as the rule engine has to evaluate it each time, but it is a "catch all" feature for when you can express what you need to do with Column constraints.

Report a bug

9.6. AUTOMATIC TRANSFORMING BETWEEN XML AND DRL

JBoss Rules comes with some utility classes to transform between formats. This works by parsing the rules from the source format into the AST and then "dumping" out to the appropriate target format. This allows you to, for example, write rules in DRL and export them to XML.

Report a bug

9.7. CLASSES FOR AUTOMATIC TRANSFORMING BETWEEN XML AND DRL

These are the classes to use when transforming between XML and DRL files. Using combinations of these, you can convert between any format (including round trip):

- DrlDumper - for exporting DRL
- DrlParser - for reading DRL
- XmlPackageReader - for reading XML

NOTE

DSLs will not be preserved (from DRLs that are using a DSL) - but they will be able to be converted.

Report a bug

CHAPTER 10. OBJECTS AND INTERFACES

10.1. GLOBALS

Globals are named objects that are made visible to the rule engine, but unlike facts, changes in the object backing a global do not trigger reevaluation of rules. Globals are useful for providing static information, as an object offering services that are used in the RHS of a rule, or as a means to return objects from the rule engine. When you use a global on the LHS of a rule, make sure it is immutable, or else your changes will not have any effect on the behavior of your rules.

Report a bug

10.2. WORKING WITH GLOBALS

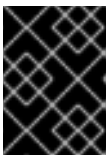
Procedure 10.1. Task

1. To start implementing globals into the Working Memory, declare a global in a rules file and back it up with a Java object:

```
global java.util.List list
```

2. With the Knowledge Base now aware of the global identifier and its type, you can call **`ksession.setGlobal()`** with the global's name and an object (for any session) to associate the object with the global:

```
List list = new ArrayList();
ksession.setGlobal("list", list);
```



IMPORTANT

Failure to declare the global type and identifier in DRL code will result in an exception being thrown from this call.

3. Set the global before it is used in the evaluation of a rule. Failure to do so results in a **`NullPointerException`**.

Report a bug

10.3. RESOLVING GLOBALS

Globals can be resolved in three ways:

getGlobals()

The Stateless Knowledge Session method **`getGlobals()`** returns a Globals instance which provides access to the session's globals. These are shared for all execution calls. Exercise caution regarding mutable globals because execution calls can be executing simultaneously in different threads.

Delegates

Using a delegate is another way of providing global resolution. Assigning a value to a global (with

`setGlobal(String, Object)`) results in the value being stored in an internal collection mapping identifiers to values. Identifiers in this internal collection will have priority over any supplied delegate. If an identifier cannot be found in this internal collection, the delegate global (if any) will be used.

Execution

Execution scoped globals use a **`Command`** to set a global which is then passed to the **`CommandExecutor`**.

Report a bug

10.4. SESSION SCOPED GLOBAL EXAMPLE

This is what a session scoped Global looks like:

```
StatelessKieSession ksession = kbase.newStatelessKnowledgeSession();
// Set a global hbnSession, that can be used for DB interactions in the
rules.
ksession.setGlobal( "hbnSession", hibernateSession );
// Execute while being able to resolve the "hbnSession" identifier.
ksession.execute( collection );
```

Report a bug

10.5. STATEFULRULESESSIONS

The **`StatefulRuleSession`** property is inherited by the **`StatefulKnowledgeSession`** and provides the rule-related methods that are relevant from outside of the engine.

Report a bug

10.6. AGENDAFILTER OBJECTS

`AgendaFilter` objects are optional implementations of the filter interface which are used to allow or deny the firing of an activation. What is filtered depends on the implementation.

Report a bug

10.7. USING THE AGENDAFILTER

Procedure 10.2. Task

- To use a filter specify it while calling **`fireAllRules()`**. The following example permits only rules ending in the string **`"Test"`**. All others will be filtered out:

```
ksession.fireAllRules( new RuleNameEndsWithAgendaFilter( "Test" ) );
```

Report a bug

10.8. RULE ENGINE PHASES

The engine cycles repeatedly through two phases:

Working Memory Actions

This is where most of the work takes place, either in the Consequence (the RHS itself) or the main Java application process. Once the Consequence has finished or the main Java application process calls **`fireAllRules()`** the engine switches to the Agenda Evaluation phase.

Agenda Evaluation

This attempts to select a rule to fire. If no rule is found it exits. Otherwise it fires the found rule, switching the phase back to Working Memory Actions.

The process repeats until the agenda is clear, in which case control returns to the calling application. When Working Memory Actions are taking place, no rules are being fired.

Report a bug

10.9. THE EVENT MODEL

The event package provides means to be notified of rule engine events, including rules firing, objects being asserted, etc. This allows you, for instance, to separate logging and auditing activities from the main part of your application (and the rules).

Report a bug

10.10. THE KNOWLEGERUNTIMEEVENTMANAGER

The **`KnowlegeRuntimeEventManager`** interface is implemented by the **`KnowledgeRuntime`** which provides two interfaces, **`WorkingMemoryEventManager`** and **`ProcessEventManager`**.

Report a bug

10.11. THE WORKINGMEMORYEVENTMANAGER

The **`WorkingMemoryEventManager`** allows for listeners to be added and removed, so that events for the working memory and the agenda can be listened to.

Report a bug

10.12. ADDING AN AGENDAEVENTLISTENER

The following code snippet shows how a simple agenda listener is declared and attached to a session. It will print activations after they have fired:

```
ksession.addEventListener( new DefaultAgendaEventListener() {
   public void afterActivationFired(AfterActivationFiredEvent event) {
       super.afterActivationFired( event );
       System.out.println( event );
   }
});
```

Report a bug

10.13. PRINTING WORKING MEMORY EVENTS

This code lets you print all Working Memory events by adding a listener:

```
ksession.addEventListener( new DebugWorkingMemoryEventListener() );
```

Report a bug

10.14. KNOWLEGERUNTIMEEVENTS

All emitted events implement the **`KnowlegeRuntimeEvent`** interface which can be used to retrieve the actual **`KnowlegeRuntime`** the event originated from.

Report a bug

10.15. SUPPORTED EVENTS FOR THE KNOWLEDGERUNTIMEEVENT INTERFACE

The events currently supported are:

- ActivationCreatedEvent
- ActivationCancelledEvent
- BeforeActivationFiredEvent
- AfterActivationFiredEvent
- AgendaGroupPushedEvent
- AgendaGroupPoppedEvent
- ObjectInsertEvent
- ObjectRetractedEvent
- ObjectUpdatedEvent
- ProcessCompletedEvent
- ProcessNodeLeftEvent
- ProcessNodeTriggeredEvent
- ProcessStartEvent

Report a bug

10.16. THE KNOWLEDGERUNTIMELOGGER

The KnowledgeRuntimeLogger uses the comprehensive event system in JBoss Rules to create an audit log that can be used to log the execution of an application for later inspection, using tools such as the Eclipse audit viewer.

Report a bug

10.17. ENABLING A FILELOGGER

To enable a FileLogger to track your files, use this code:

```
KnowledgeRuntimeLogger logger =
  KnowledgeRuntimeLoggerFactory.newFileLogger(ksession,
"logdir/mylogfile");
...
logger.close();
```

Report a bug

10.18. USING STATELESSKNOWLEDGESESSION IN JBOSS RULES

The **`StatelessKnowledgeSession`** wraps the **`StatefulKnowledgeSession`**, instead of extending it. Its main focus is on decision service type scenarios. It avoids the need to call **`dispose()`**. Stateless sessions do not support iterative insertions and the method call **`fireAllRules()`** from Java code. The act of calling **`execute()`** is a single-shot method that will internally instantiate a **`StatefulKnowledgeSession`**, add all the user data and execute user commands, call **`fireAllRules()`**, and then call **`dispose()`**. While the main way to work with this class is via the **`BatchExecution`** (a subinterface of **`Command`**) as supported by the **`CommandExecutor`** interface, two convenience methods are provided for when simple object insertion is all that's required. The **`CommandExecutor`** and **`BatchExecution`** are talked about in detail in their own section.

Report a bug

10.19. PERFORMING A STATELESSKNOWLEDGESESSION EXECUTION WITH A COLLECTION

This the code for performing a StatelessKnowledgeSession execution with a collection:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newFileSystemResource( fileName ),
ResourceType.DRL );
if (kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
} else {
    Kie kbase = KnowledgeBaseFactory.newKnowledgeBase();
    kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
    StatelessKieSession ksession = kbase.newStatelessKnowledgeSession();
    ksession.execute( collection );
}
```

Report a bug

10.20. PERFORMING A STATELESSKNOWLEDGESESSION EXECUTION WITH THE INSERTELEMENTS COMMAND

This is the code for performing a StatelessKnowledgeSession execution with the InsertElements Command:

```
ksession.execute( CommandFactory.newInsertElements( collection ) );
```

NOTE

To insert the collection and its individual elements, use **`CommandFactory.newInsert(collection)`**.

Report a bug

10.21. THE BATCHEXECUTIONHELPER

Methods of the **`CommandFactory`** create the supported commands, all of which can be marshaled using XStream and the **`BatchExecutionHelper`**. **`BatchExecutionHelper`** provides details on the XML format as well as how to use JBoss Rules Pipeline to automate the marshaling of **`BatchExecution`** and **`ExecutionResults`**.

Report a bug

10.22. THE COMMANDEXECUTOR INTERFACE

The **`CommandExecutor`** interface allows users to export data using "out" parameters. This means that inserted facts, globals and query results can all be returned using this interface.

Report a bug

10.23. OUT IDENTIFIERS

This is an example of what out identifiers look like:

```
// Set up a list of commands
List cmds = new ArrayList();
cmds.add( CommandFactory.newSetGlobal( "list1", new ArrayList(), true ) );
cmds.add( CommandFactory.newInsert( new Person( "jon", 102 ), "person" ) );
cmds.add( CommandFactory.newQuery( "Get People" "getPeople" );

// Execute the list
ExecutionResults results =
  ksession.execute( CommandFactory.newBatchExecution( cmds ) );

// Retrieve the ArrayList
results.getValue( "list1" );
// Retrieve the inserted Person fact
results.getValue( "person" );
// Retrieve the query as a QueryResults instance.
results.getValue( "Get People" );
```

Report a bug

CHAPTER 11. COMPLEX EVENT PROCESSING

11.1. INTRODUCTION TO COMPLEX EVENT PROCESSING

JBoss BRMS Complex Event Processing provides the JBoss Enterprise BRMS Platform with complex event processing capabilities.

For the purpose of this guide, *Complex Event Processing*, or CEP, refers to the ability to process multiple events and detect interesting events from within a collection of events, uncover relationships that exist between events, and infer new data from the events and their relationships.

An *event* can best be described as a record of a significant change of state in the application domain. Depending on how the domain is modeled, the change of state may be represented by a single event, multiple atomic events, or even hierarchies of correlated events. Using a stock broker application as an example, a change in security prices, a change in ownership from seller to buyer, or a change in an account holder's balance are all considered to be events as a change has occurred in the state of the application domain.

Event processing use cases, in general, share several requirements and goals with *business rules use cases*.

From a business perspective, business rule definitions are often defined based on the occurrence of scenarios triggered by events. For example:

- On an algorithmic trading application: Take an action if the security price increases X% above the day's opening price.

  The price increases are denoted by events on a stock trade application.

- On a monitoring application: Take an action if the temperature in the server room increases X degrees in Y minutes.

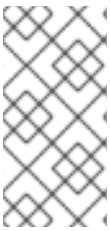
  The sensor readings are denoted by events.

Both business rules and event processing queries change frequently and require an immediate response for the business to adapt to new market conditions, regulations, and corporate policies.

From a technical perspective:

- Both business rules and event processing require seamless integration with the enterprise infrastructure and applications. This is particularly important with regard to life-cycle management, auditing, and security.

- Both business rules and event processing have functional requirements like *pattern matching* and non-functional requirements like response time limits and query/rule explanations.



NOTE

JBoss BRMS Complex Event Processing provides the complex event processing capabilities of JBoss Business Rules Management System. The Business Rules Management and Business Process Management capabilities are provided by other modules.

Complex event processing scenarios share these distinguishing characteristics:

- They usually process large numbers of events, but only a small percentage of the events are of interest.
- The events are usually immutable, as they represent a record of change in state.
- The rules and queries run against events and must react to detected event patterns.
- There are usually strong temporal relationships between related events.
- Individual events are not important. The system is concerned with patterns of related events and the relationships between them.
- It is often necessary to perform composition and aggregation of events.

As such, JBoss BRMS Complex Event Processing supports the following behaviors:

- Support events, with their proper semantics, as *first class citizens*.
- Allow detection, correlation, aggregation, and composition of events.
- Support processing streams of events.
- Support temporal constraints in order to model the temporal relationships between events.
- Support *sliding windows* of interesting events.
- Support a *session-scoped* unified clock.
- Support the required volumes of events for complex event processing use cases.
- Support reactive rules.
- Support adapters for event input into the engine (pipeline).

The rest of this guide describes each of the features that JBoss BRMS Complex Event Processing provides.

Report a bug

CHAPTER 12. FEATURES OF JBOSS BRMS COMPLEX EVENT PROCESSING

12.1. EVENTS

Events are a record of significant change of state in the application domain. From a complex event processing perspective, an event is a special type of fact or object. A fact is a known piece of data. For instance, a fact could be a stock's opening price. A rule is a definition of how to react to the data. For instance, if a stock price reaches $X, sell the stock.

The defining characteristics of events are the following:

Events are *immutable*

An event is a record of change which has occurred at some time in the past, and as such it cannot be changed.

NOTE

The rules engine does not enforce immutability on the Java objects representing events; this makes *event data enrichment* possible.

The application should be able to populate un-populated event attributes, which can be used to enrich the event with inferred data; however, event attributes that have already been populated should not be changed.

Events have strong *temporal constraints*

Rules involving events usually require the correlation of multiple events that occur at different points in time relative to each other.

Events have *managed life-cycles*

Because events are immutable and have temporal constraints, they are usually only of interest for a specified period of time. This means the engine can automatically manage the life-cycle of events.

Events can use *sliding windows*

It is possible to define and use sliding windows with events since all events have timestamps associated with them. Therefore, sliding windows allow the creation of rules on aggregations of values over a time period.

Events can be declared as either *interval-based* events or *point-in-time* events. Interval-based events have a duration time and persist in working memory until their duration time has lapsed. Point-in-time events have no duration and can be thought of as interval-based events with a duration of zero.

Report a bug

12.2. EVENT DECLARATION

To declare a fact type as an event, assign the **`@role`** meta-data tag to the fact with the **`event`** parameter. The @role meta-data tag can accept two possible values:

- **`fact`**: Assigning the fact role declares the type is to be handled as a regular fact. Fact is the default role.
- **`event`**: Assigning the event role declares the type is to be handled as an event.

This example declares that a stock broker application's **`StockTick`** fact type will be handled as an event:

Example 12.1. Declaring a Fact Type as an Event

```
import some.package.StockTick

declare StockTick
    @role( event )
end
```

Facts can also be declared inline. If **`StockTick`** was a fact type declared in the DRL instead of in a pre-existing class, the code would be as follows:

Example 12.2. Declaring a Fact Type and Assigning it to an Event Role

```
declare StockTick
    @role( event )

    datetime : java.util.Date
    symbol : String
    price : double
end
```

For more information on *type declarations*, please refer to the Rule Language section of the *JBoss Rules Reference Guide*.

Report a bug

12.3. EVENT META-DATA

Every event has associated meta-data. Typically, the meta-data is automatically added as each event is inserted into working memory. The meta-data defaults can be changed on an event-type basis using the meta-data tags:

- @role
- @timestamp
- @duration
- @expires

The following examples assume the application domain model includes the following class:

Example 12.3. The VoiceCall Fact Class

```
/**
 * A class that represents a voice call in
 * a Telecom domain model
 */
public class VoiceCall {
    private String  originNumber;
    private String  destinationNumber;
    private Date    callDateTime;
    private long    callDuration;          // in milliseconds

    // constructors, getters, and setters
}
```

@role

The @role meta-data tag indicates whether a given fact type is either a regular fact or an event. It accepts either **fact** or **event** as a parameter. The default is **fact**.

```
@role( <fact|event> )
```

Example 12.4. Declaring VoiceCall as an Event Type

```
declare VoiceCall
    @role( event )
end
```

@timestamp

A timestamp is automatically assigned to every event. By default, the time is provided by the session clock and assigned to the event at insertion into the working memory. Events can have their own timestamp attribute, which can be included by telling the engine to use the attribute's timestamp instead of the session clock.

To use the attribute's timestamp, use the attribute name as the parameter for the **@timestamp** tag.

```
@timestamp( <attributeName> )
```

Example 12.5. Declaring the VoiceCall Timestamp Attribute

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
end
```

@duration

JBoss BRMS Complex Event Processing supports both point-in-time and interval-based events. A point-in-time event is represented as an interval-based event with a duration of zero time units. By default, every event has a duration of zero. To assign a different duration to an event, use the attribute name as the parameter for the **@duration** tag.

```
@duration( <attributeName> )
```

Example 12.6. Declaring the VoiceCall Duration Attribute

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
end
```

@expires

Events may be set to expire automatically after a specific duration in the working memory. By default, this happens when the event can no longer match and activate any of the current rules. You can also explicitly define when an event should expire. The @expires tag is only used when the engine is running in *stream* mode.

```
@expires( <timeOffset> )
```

The value of **`timeOffset`** is a temporal interval that sets the relative duration of the event.

```
[#d][#h][#m][#s][#[ms]]
```

All parameters are optional and the # parameter should be replaced by the appropriate value.

To declare that the **`VoiceCall`** facts should expire one hour and thirty-five minutes after insertion into the working memory, use the following:

Example 12.7. Declaring the Expiration Offset for the VoiceCall Events

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
    @expires( 1h35m )
end
```

See Also:

- Section 12.6, “Event Processing Modes”
- Section 12.14.2, “Explicit Expiration”

Report a bug

12.4. SESSION CLOCK

Events have strong temporal constraints making it is necessary to use a reference clock. If a rule needs to determine the average price of a given stock over the last sixty minutes, it is necessary to compare the stock price event's timestamp with the current time. The reference clock provides the current time.

Because the rules engine can simultaneously run an array of different scenarios that require different clocks, multiple clock implementations can be used by the engine.

Scenarios that require different clocks include the following:

- Rules testing: Testing always requires a controlled environment, and when the tests include rules with temporal constraints, it is necessary to control the input rules, facts, and the flow of time.
- Regular execution: A rules engine that reacts to events in real time needs a real-time clock.
- Special environments: Specific environments may have specific time control requirements. For instance, clustered environments may require clock synchronization or JEE environments may require you to use an application server-provided clock.
- Rules replay or simulation: In order to replay or simulate scenarios, it is necessary that the application controls the flow of time.

Report a bug

12.5. AVAILABLE CLOCK IMPLEMENTATIONS

JBoss BRMS Complex Event Processing comes equipped with two clock implementations:

Real-Time Clock

The real-time clock is the default implementation based on the system clock. The real-time clock uses the system clock to determine the current time for timestamps.

To explicitly configure the engine to use the real-time clock, set the session configuration parameter to ***`realtime`***:

```
KieSessionConfiguration config =
KieServices.Factory.get().newKieSessionConfiguration()
   config.setOption( ClockTypeOption.get("realtime") );
```

Pseudo-Clock

The pseudo-clock is useful for testing temporal rules since it can be controlled by the application.

To explicitly configure the engine to use the pseudo-clock, set the session configuration parameter to ***`pseudo`***:

```
KieSessionConfiguration config =
KieServices.Factory.get().newKieSessionConfiguration();
   config.setOption( ClockTypeOption.get("pseudo") );
```

This example shows how to control the pseudo-clock:

```
KieSessionConfiguration conf =
KieServices.Factory.get().newKieSessionConfiguration();
   conf.setOption( ClockTypeOption.get( "pseudo" ) );
```

```
KieSession session = kbase.newKieSession( conf, null );

SessionPseudoClock clock = session.getSessionClock();

// then, while inserting facts, advance the clock as necessary:
FactHandle handle1 = session.insert( tick1 );
clock.advanceTime( 10, TimeUnit.SECONDS );
FactHandle handle2 = session.insert( tick2 );
clock.advanceTime( 30, TimeUnit.SECONDS );
FactHandle handle3 = session.insert( tick3 );
```

Report a bug

12.6. EVENT PROCESSING MODES

Rules engines process facts and rules to provide applications with results. Regular facts (facts with no temporal constraints) are processed independent of time and in no particular order. JBoss BRMS processes facts of this type in cloud mode. Events (facts which have strong temporal constraints) must be processed in real-time or near real-time. JBoss BRMS processes these events in stream mode. Stream mode deals with synchronization and makes it possible for JBoss BRMS to process events.

Report a bug

12.7. CLOUD MODE

Cloud mode is the default operating mode of JBoss Business Rules Management System.

Running in Cloud mode, the engine applies a many-to-many pattern matching algorithm, which treats the events as an unordered cloud. Events still have timestamps, but there is no way for the rules engine running in Cloud mode to draw relevance from the timestamp because Cloud mode is unaware of the present time.

This mode uses the rules constraints to find the matching tuples, activate, and fire rules.

Cloud mode does not impose any kind of additional requirements on facts; however, because it has no concept of time, it cannot take advantage of temporal features such as *sliding windows* or *automatic life-cycle management*. In Cloud mode, it is necessary to explicitly retract events when they are no longer needed.

Certain requirements that are not imposed include the following:

- No need for clock synchronization since there is no notion of time.
- No requirement on ordering events since the engine looks at the events as an unordered cloud against which the engine tries to match rules.

Cloud mode can be specified either by setting a system property, using configuration property files, or via the API.

The API call follows:

```
KieBaseConfiguration config =
KieServices.Factory.get().newKieBaseConfiguration();
   config.setOption( EventProcessingOption.CLOUD );
```

The equivalent property follows:

```
drools.eventProcessingMode = cloud
```

Report a bug

12.8. STREAM MODE

Stream mode processes events chronologically as they are inserted into the rules engine. Stream mode uses a session clock that enables the rules engine to process events as they occur in time. The session clock enables processing events as they occur based on the age of the events. Stream mode also synchronizes streams of events (so events in different streams can be processed in chronological order), implements sliding windows of interest, and enables automatic life-cycle management.

The requirements for using stream mode are the following:

- Events in each stream must be ordered chronologically.
- A session clock must be present to synchronize event streams.

NOTE

The application does not need to enforce ordering events between streams, but the use of event streams that have not been synchronized may cause unexpected results.

Stream mode can be enabled by setting a system property, using configuration property files, or via the API.

The API call follows:

```
KieBaseConfiguration config =
KieServices.Factory.get().newKieBaseConfiguration();
   config.setOption( EventProcessingOption.STREAM );
```

The equivalent property follows:

```
drools.eventProcessingMode = stream
```

Report a bug

12.9. SUPPORT FOR EVENT STREAMS

Complex event processing use cases deal with streams of events. The streams can be provided to the application via JMS queues, flat text files, database tables, raw sockets, or even web service calls.

Streams share a common set of characteristics:

- Events in the stream are ordered by timestamp. The timestamps may have different semantics for different streams, but they are always ordered internally.

- There is usually a high volume of events in the stream.
- Atomic events contained in the streams are rarely useful by themselves.
- Streams are either homogeneous (they contain a single type of event) or heterogeneous (they contain events of different types).

A stream is also known as an *entry point*.

Facts from one entry point, or stream, may join with facts from any other entry point in addition to facts already in working memory. Facts always remain associated with the entry point through which they entered the engine. Facts of the same type may enter the engine through several entry points, but facts that enter the engine through entry point A will never match a pattern from entry point B.

See Also:

- Section 12.10, “Declaring and Using Entry Points”

Report a bug

12.10. DECLARING AND USING ENTRY POINTS

Entry points are declared implicitly by making direct use of them in rules. Referencing an entry point in a rule will make the engine, at compile time, identify and create the proper internal structures to support that entry point.

For example, a banking application that has transactions fed into the engine via streams could have one stream for all of the transactions executed at ATMs. A rule for this scenario could state, "A withdrawal is only allowed if the account balance is greater than the withdrawal amount the customer has requested."

Example 12.8. Example ATM Rule

```
rule "authorize withdraw"
   when
       WithdrawRequest( $ai : accountId, $am : amount ) from entry-point
"ATM Stream"
       CheckingAccount( accountId == $ai, balance > $am )
   then
       // authorize withdraw
   end
```

When the engine compiles this rule, it will identify that the pattern is tied to the entry point "ATM Stream." The engine will create all the necessary structures for the rule-base to support the "ATM Stream", and this rule will only match **`WithdrawRequest`** events coming from the "ATM Stream."

Note the ATM example rule joins the event (**`WithdrawalRequest`**) from the stream with a fact from the main working memory (**`CheckingAccount`**).

The banking application may have a second rule that states, "A fee of $2 must be applied to a withdraw request made via a branch teller."

Example 12.9. Using Multiple Streams

```
rule "apply fee on withdraws on branches"
   when
       WithdrawRequest( $ai : accountId, processed == true ) from entry-
point "Branch Stream"
       CheckingAccount( accountId == $ai )
   then
       // apply a $2 fee on the account
   end
```

This rule matches events of the same type (**`WithdrawRequest`**) as the example ATM rule but from a different stream. Events inserted into the "ATM Stream" will never match the pattern on the second rule, which is tied to the "Branch Stream;" accordingly, events inserted into the "Branch Stream" will never match the pattern on the example ATM rule, which is tied to the "ATM Stream".

Declaring the stream in a rule states that the rule is only interested in events coming from that stream.

Events can be inserted manually into an entry point instead of directly into the working memory.

Example 12.10. Inserting Facts into an Entry Point

```
   // create your rulebase and your session as usual
   KieSession session = ...

   // get a reference to the entry point
   WorkingMemoryEntryPoint atmStream =
session.getWorkingMemoryEntryPoint( "ATM Stream" );

   // and start inserting your facts into the entry point
   atmStream.insert( aWithdrawRequest );
```

Report a bug

12.11. NEGATIVE PATTERN IN STREAM MODE

A *negative pattern* is concerned with conditions that are not met. Negative patterns make reasoning in the absence of events possible. For instance, a safety system could have a rule that states, "If a fire is detected and the sprinkler is *not* activated, sound the alarm."

In Cloud mode, the engine assumes all facts (regular facts and events) are known in advance and evaluates negative patterns immediately.

Example 12.11. A Rule with a Negative Pattern

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( ) )
then
    // sound the alarm
end
```

An example in stream mode is displayed below. This rule keeps consistency when dealing with negative patterns and temporal constraints at the same time interval.

Example 12.12. A Rule with a Negative Pattern, Temporal Constraints, and an Explicit Duration Parameter.

```
rule "Sound the alarm"
    duration( 10s )
when
    $f : FireDetected( )
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

In stream mode, negative patterns with temporal constraints may force the engine to wait for a set time before activating a rule. A rule may be written for an alarm system that states, "If a fire is detected and the sprinkler is *not* activated after 10 seconds, sound the alarm." Unlike the previous stream mode example, this one does not require the user to calculate and write the duration parameter.

Example 12.13. A Rule with a Negative Pattern with Temporal Constraints

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

The rule depicted below expects one "Heartbeat" event to occur every 10 seconds; if not, the rule fires. What is special about this rule is that it uses the same type of object in the first pattern and in the negative pattern. The negative pattern has the temporal constraint to wait between 0 to 10 seconds before firing, and it excludes the Heartbeat bound to $h. Excluding the bound Heartbeat is important since the temporal constraint [0s, ...] does not exclude by itself the bound event $h from being matched again, thus preventing the rule to fire.

Example 12.14. Excluding Bound Events in Negative Patterns

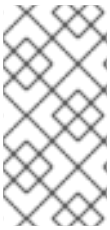
```
rule "Sound the alarm"
when
    $h: Heartbeat( ) from entry-point "MonitoringStream"
    not( Heartbeat( this != $h, this after[0s,10s] $h ) from entry-point
"MonitoringStream" )
then
    // Sound the alarm
end
```

Report a bug

12.12. TEMPORAL REASONING

12.12.1. Temporal Reasoning

Complex Event Processing requires the rules engine to engage in temporal reasoning. Events have strong temporal constraints so it is vital the rules engine can determine and interpret an event's temporal attributes, both as they relate to other events and the 'flow of time' as it appears to the rules engine. This makes it possible for rules to take time into account; for instance, a rule could state, "Calculate the average price of a stock over the last 60 minutes."



NOTE

JBoss BRMS Complex Event Processing implements interval-based time events, which have a duration attribute that is used to indicate how long an event is of interest. Point-in-time events are also supported and treated as interval-based events with a duration of 0 (zero).

Report a bug

12.12.2. Temporal Operations

12.12.2.1. Temporal Operations

JBoss BRMS Complex Event Processing implements 13 temporal operators and their logical complements (negation). The 13 temporal operators are the following:

- After
- Before
- Coincides
- During
- Finishes
- Finishes By
- Includes
- Meets
- Met By
- Overlaps
- Overlapped By
- Starts
- Started By

Report a bug

12.12.2.2. After

The **after** operator correlates two events and matches when the temporal distance (the time between the two events) from the current event to the event being correlated falls into the distance range declared for the operator.

For example:

```
$eventA : EventA( this after[ 3m30s, 4m ] $eventB )
```

This pattern only matches if the temporal distance between the time when **$eventB** finished and the time when **$eventA** started is between the lower limit of three minutes and thirty seconds and the upper limit of four minutes.

This can also be represented as follows:

```
3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
```

The **after** operator accepts one or two optional parameters:

- If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).
- If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.
- If no value is defined, the interval starts at one millisecond and runs indefinitely with no end time.

The **after** operator also accepts negative temporal distances.

For example:

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB )
```

If the first value is greater than the second value, the engine will automatically reverse them.

The following two patterns are equivalent to each other:

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB )
   $eventA : EventA( this after[ -2m, -3m30s ] $eventB )
```

Report a bug

12.12.2.3. Before

The **before** operator correlates two events and matches when the temporal distance (time between the two events) from the event being correlated to the current event falls within the distance range declared for the operator.

For example:

```
$eventA : EventA( this before[ 3m30s, 4m ] $eventB )
```

This pattern only matches if the temporal distance between the time when **$eventA** finished and the time when **$eventB** started is between the lower limit of three minutes and thirty seconds and the upper limit of four minutes.

This can also be represented as follows:

```
3m30s <= $eventB.startTimestamp - $eventA.endTimeStamp <= 4m
```

The **before** operator accepts one or two optional parameters:

- If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).
- If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.
- If no value is defined, the interval starts at one millisecond and runs indefinitely with no end time.

The **before** operator also accepts negative temporal distances.

For example:

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB )
```

If the first value is greater than the second value, the engine will automatically reverse them.

The following two patterns are equivalent to each other:

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB )
   $eventA : EventA( this before[ -2m, -3m30s ] $eventB )
```

Report a bug

12.12.2.4. Coincides

The **coincides** operator correlates two events and matches when both events happen at the same time.

For example:

```
$eventA : EventA( this coincides $eventB )
```

This pattern only matches if both the start timestamps of **$eventA** and **$eventB** are identical and the end timestamps of both **$eventA** and **$eventB** are also identical.

The **coincides** operator accepts optional thresholds for the distance between the events' start times and the events' end times, so the events do not have to start at exactly the same time or end at exactly the same time, but they need to be within the provided thresholds.

The following rules apply when defining thresholds for the **coincides** operator:

- If only one parameter is given, it is used to set the threshold for both the start and end times of both events.

- If two parameters are given, the first is used as a threshold for the start time and the second one is used as a threshold for the end time.

For example:

```
$eventA : EventA( this coincides[15s, 10s] $eventB )
```

This pattern will only match if the following conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 15s &&
   abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 10s
```

WARNING

The **coincides** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance internals.

Report a bug

12.12.2.5. During

The **during** operator correlates two events and matches when the current event happens during the event being correlated.

For example:

```
$eventA : EventA( this during $eventB )
```

This pattern only matches if **$eventA** starts after **$eventB** and ends before **$eventB** ends.

This can also be represented as follows:

```
$eventB.startTimestamp < $eventA.startTimestamp <= $eventA.endTimestamp <
$eventB.endTimestamp
```

The **during** operator accepts one, two, or four optional parameters:

The following rules apply when providing parameters for the **during** operator:

- If one value is defined, this value will represent the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.
- If two values are defined, these values represent a threshold that the current event's start time and end time must occur between in relation to the correlated event's start and end times.

  If the values 5s and 10s are provided, the current event must start between 5 and 10 seconds after the correlated event, and similarly the current event must end between 5 and 10 seconds before the correlated event.

- If four values are defined, the first and second values will be used as the minimum and maximum distances between the starting times of the events, and the third and fourth values will be used as the minimum and maximum distances between the end times of the two events.

Report a bug

12.12.2.6. Finishes

The **`finishes`** operator correlates two events and matches when the current event's start timestamp post-dates the correlated event's start timestamp and both events end simultaneously.

For example:

```
$eventA : EventA( this finishes $eventB )
```

This pattern only matches if **`$eventA`** starts after **`$eventB`** starts and ends at the same time as **`$eventB`** ends.

This can be represented as follows:

```
$eventB.startTimestamp < $eventA.startTimestamp &&
   $eventA.endTimestamp == $eventB.endTimestamp
```

The **`finishes`** operator accepts one optional parameter. If defined, the optional parameter sets the maximum time allowed between the end times of the two events.

For example:

```
$eventA : EventA( this finishes[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
$eventB.startTimestamp < $eventA.startTimestamp &&
   abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

WARNING

The **`finishes`** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.7. Finishes By

The **`finishedby`** operator correlates two events and matches when the current event's start time predates the correlated event's start time but both events end simultaneously. **`finishedby`** is the symmetrical opposite of the **`finishes`** operator.

For example:

```
$eventA : EventA( this finishedby $eventB )
```

This pattern only matches if **$eventA** starts before **$eventB** starts and ends at the same time as **$eventB** ends.

This can be represented as follows:

```
$eventA.startTimestamp < $eventB.startTimestamp &&
   $eventA.endTimestamp == $eventB.endTimestamp
```

The **finishedby** operator accepts one optional parameter. If defined, the optional parameter sets the maximum time allowed between the end times of the two events.

```
$eventA : EventA( this finishedby[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
$eventA.startTimestamp < $eventB.startTimestamp &&
   abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

WARNING

The **finishedby** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.8. Includes

The **includes** operator examines two events and matches when the event being correlated happens during the current event. It is the symmetrical opposite of the **during** operator.

For example:

```
$eventA : EventA( this includes $eventB )
```

This pattern only matches if **$eventB** starts after **$eventA** and ends before **$eventA** ends.

This can be represented as follows:

```
$eventA.startTimestamp < $eventB.startTimestamp <= $eventB.endTimestamp <
$eventA.endTimestamp
```

The **includes** operator accepts 1, 2 or 4 optional parameters:

- If one value is defined, this value will represent the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.
- If two values are defined, these values represent a threshold that the current event's start time and end time must occur between in relation to the correlated event's start and end times.

  If the values 5s and 10s are provided, the current event must start between 5 and 10 seconds after the correlated event, and similarly the current event must end between 5 and 10 seconds before the correlated event.
- If four values are defined, the first and second values will be used as the minimum and maximum distances between the starting times of the events, and the third and fourth values will be used as the minimum and maximum distances between the end times of the two events.

Report a bug

12.12.2.9. Meets

The **`meets`** operator correlates two events and matches when the current event ends at the same time as the correlated event starts.

For example:

```
$eventA : EventA( this meets $eventB )
```

This pattern matches if **`$eventA`** ends at the same time as **`$eventB`** starts.

This can be represented as follows:

```
abs( $eventB.startTimestamp - $eventA.endTimestamp ) == 0
```

The **`meets`** operator accepts one optional parameter. If defined, it determines the maximum time allowed between the end time of the current event and the start time of the correlated event.

For example:

```
$eventA : EventA( this meets[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventB.startTimestamp - $eventA.endTimestamp) <= 5s
```

WARNING

The **`meets`** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.10. Met By

The **metby** operator correlates two events and matches when the current event starts at the same time as the correlated event ends.

For example:

```
$eventA : EventA( this metby $eventB )
```

This pattern matches if **$eventA** starts at the same time as **$eventB** ends.

This can be represented as follows:

```
abs( $eventA.startTimestamp - $eventB.endTimestamp ) == 0
```

The **metby** operator accepts one optional parameter. If defined, it sets the maximum distance between the end time of the correlated event and the start time of the current event.

For example:

```
$eventA : EventA( this metby[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.endTimestamp) <= 5s
```

WARNING

The **metby** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.11. Overlaps

The **overlaps** operator correlates two events and matches when the current event starts before the correlated event starts and ends after the correlated event starts, but it ends before the correlated event ends.

For example:

```
$eventA : EventA( this overlaps $eventB )
```

This pattern matches if these conditions are met:

```
$eventA.startTimestamp < $eventB.startTimestamp < $eventA.endTimestamp <
$eventB.endTimestamp
```

The **overlaps** operator accepts one or two optional parameters:

- If one parameter is defined, it will define the maximum distance between the start time of the correlated event and the end time of the current event.
- If two values are defined, the first value will be the minimum distance, and the second value will be the maximum distance between the start time of the correlated event and the end time of the current event.

Report a bug

12.12.2.12. Overlapped By

The **overlappedby** operator correlates two events and matches when the correlated event starts before the current event, and the correlated event ends after the current event starts but before the current event ends.

For example:

```
$eventA : EventA( this overlappedby $eventB )
```

This pattern matches if these conditions are met:

```
$eventB.startTimestamp < $eventA.startTimestamp < $eventB.endTimestamp <
$eventA.endTimestamp
```

The **overlappedby** operator accepts one or two optional parameters:

- If one parameter is defined, it sets the maximum distance between the start time of the correlated event and the end time of the current event.
- If two values are defined, the first value will be the minimum distance, and the second value will be the maximum distance between the start time of the correlated event and the end time of the current event.

Report a bug

12.12.2.13. Starts

The **starts** operator correlates two events and matches when they start at the same time, but the current event ends before the correlated event ends.

For example:

```
$eventA : EventA( this starts $eventB )
```

This pattern matches if **$eventA** and **$eventB** start at the same time, and **$eventA** ends before **$eventB** ends.

This can be represented as follows:

```
$eventA.startTimestamp == $eventB.startTimestamp &&
   $eventA.endTimestamp < $eventB.endTimestamp
```

The **starts** operator accepts one optional parameter. If defined, it determines the maximum distance between the start times of events in order for the operator to still match:

```
$eventA : EventA( this starts[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s &&
   $eventA.endTimestamp < $eventB.endTimestamp
```

WARNING

The **starts** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.14. Started By

The **startedby** operator correlates two events. It matches when both events start at the same time and the correlating event ends before the current event.

For example:

```
$eventA : EventA( this startedby $eventB )
```

This pattern matches if **$eventA** and **$eventB** start at the same time, and **$eventB** ends before **$eventA** ends.

This can be represented as follows:

```
$eventA.startTimestamp == $eventB.startTimestamp &&
   $eventA.endTimestamp > $eventB.endTimestamp
```

The **startedby** operator accepts one optional parameter. If defined, it sets the maximum distance between the start time of the two events in order for the operator to still match:

```
$eventA : EventA( this starts[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s &&
   $eventA.endTimestamp > $eventB.endTimestamp
```

WARNING

The **`startsby`** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.13. SLIDING WINDOWS

12.13.1. Sliding Time Windows

Stream mode allows events to be matched over a sliding time window. A *sliding window* is a time period that stretches back in time from the present. For instance, a sliding window of two minutes includes any events that have occurred in the past two minutes. As events fall out of the sliding time window (in this case because they occurred more than two minutes ago), they will no longer match against rules using this particular sliding window.

For example:

```
StockTick() over window:time( 2m )
```

JBoss BRMS Complex Event Processing uses the **`over`** keyword to associate windows with patterns.

Sliding time windows can also be used to calculate averages and over time. For instance, a rule could be written that states, "If the average temperature reading for the last ten minutes goes above a certain point, sound the alarm."

Example 12.15. Average Value over Time

```
rule "Sound the alarm in case temperature rises above threshold"
when
    TemperatureThreshold( $max : max )
    Number( doubleValue > $max ) from accumulate(
        SensorReading( $temp : temperature ) over window:time( 10m ),
        average( $temp ) )
then
    // sound the alarm
end
```

The engine will automatically discard any **`SensorReading`** more than ten minutes old and keep re-calculating the average.

Report a bug

12.13.2. Sliding Length Windows

Similar to Time Windows, Sliding Length Windows work in the same manner; however, they consider events based on order of their insertion into the session instead of flow of time.

The pattern below demonstrates this order by only considering the last 10 RHT Stock Ticks independent of how old they are. Unlike the previous StockTick from the Sliding Time Windows pattern, this pattern uses window:length.

```
StockTick( company == "RHT" ) over window:length( 10 )
```

The example below portrays window length instead of window time; that is, it allows the user to sound an alarm in case the average temperature over the last 100 readings from a sensor is above the threshold value.

Example 12.16. Average Value over Length

```
rule "Sound the alarm in case temperature rises above threshold"
when
    TemperatureThreshold( $max : max )
    Number( doubleValue > $max ) from accumulate(
        SensorReading( $temp : temperature ) over window:length( 100 ),
        average( $temp ) )
then
    // sound the alarm
end
```

NOTE

The engine disregards events that fall off a window when calculating that window, but it does not remove the event from the session based on that condition alone as there might be other rules that depend on that event.

NOTE

Length based windows do not define temporal constraints for event expiration from the session, and the engine will not consider them. If events have no other rules defining temporal constraints and no explicit expiration policy, the engine will keep them in the session indefinitely.

Report a bug

12.14. MEMORY MANAGEMENT FOR EVENTS

12.14.1. Memory Management for Events

Automatic memory management for events is available when running the rules engine in Stream mode. Events that no longer match any rule due to their temporal constraints can be safely retracted from the session by the rules engine without any side effects, releasing any resources held by the retracted events.

The rules engine has two ways of determining if an event is still of interest:

Explicitly

Event expiration can be explicitly set with the @expires

Implicitly

The rules engine can analyze the temporal constraints in rules to determine the window of interest for events.

Report a bug

12.14.2. Explicit Expiration

Explicit expiration is set with a **`declare`** statement and the metadata @expires tag.

For example:

Example 12.17. Declaring Explicit Expiration

```
declare StockTick
       @expires( 30m )
   end
```

Declaring expiration against an event-type will, in the above example **`StockTick`** events, remove any StockTick events from the session automatically after the defined expiration time if no rules still need the events.

Report a bug

12.14.3. Inferred Expiration

The rules engine can calculate the expiration offset for a given event implicitly by analyzing the temporal constraints in the rules.

For example:

Example 12.18. A Rule with Temporal Constraints

```
rule "correlate orders"
   when
       $bo : BuyOrder( $id : id )
       $ae : AckOrder( id == $id, this after[0,10s] $bo )
   then
       // do something
   end
```

For the example rule, the rules engine automatically calculates that whenever a **`BuyOrder`** event occurs it needs to store the event for up to ten seconds to wait for the matching **`AckOrder`** event, making the implicit expiration offset for **`BuyOrder`** events ten seconds. An **`AckOrder`** event can only match an existing **`BuyOrder`** event making its implicit expiration offset zero seconds.

The engine analyzes the entire rule-base to find the offset for every event-type. Whenever an implicit expiration clashes with an explicit expiration the engine uses the greater value of the two.

Report a bug

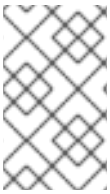
CHAPTER 13. REST API

Representational State Transfer (REST) is a style of software architecture of distributed systems (applications). It allows for a highly abstract client-server communication: clients initiate requests to servers to a particular URL with parameters if needed and servers process the requests and return appropriate responses based on the requested URL. The requests and responses are built around the transfer of representations of resources. A resource can be any coherent and meaningful concept that may be addressed (such as a repository, a Process, a Rule, etc.).

Red Hat JBoss BPM Suite provides REST API for individual application components. The REST API implementations differ slightly:

- Knowledge Store (Artifact Repository) REST API calls are calls to the static data (definitions) and are asynchronous, that is, they continue running after the call as a job. These calls return a job ID, which can be used after the REST API call was performed to request the job status and verify whether the job finished successfully. Parameters of these calls are provided in the form of JSON entities.
- Deployment REST API calls are asynchronous or synchronous, depending on the operation performed. These calls perform actions on the deployments or retrieve information about one ore more deployments.
- Runtime REST API calls are calls to the Execution Server and to the Process Execution Engine, Task Execution Engine, and Business Rule Engine. They are synchronous and return the requested data as JAXB objects.

All REST API calls to Red Hat JBoss BPM Suite use the following URL with the request body:
`http://SERVER_ADDRESS:PORT/business-central/rest/REQUEST_BODY`



NOTE

Note that it is not possible to issue REST API calls over project resources, such as, rules files, work item definitions, process definition files, etc. are not supported. Perform operation over such files with Git and its REST API directly.

Report a bug

13.1. KNOWLEDGE STORE REST API

REST API calls to Knowledge Store allow you to manage the Knowledge Store content and manipulate the static data in the repositories of the Knowledge Store.

The calls are asynchronous; that is, they continue their execution after the call was performed as a job. The job ID is returned by most of the Knowledge Store REST calls, and this is used to request the job status and verify whether the job finished successfully. Other operations return objects like repository lists and organizational units.

Parameters and results of these calls are provided in the form of JSON entities.

Report a bug

13.1.1. Job calls

Most Knowledge Store REST calls return a job ID after it is sent. This is necessary as the calls are asynchronous and you need to be able to reference the job to check its status as it goes through its lifecycle. During its lifecycle, a job can have the following statuses:

- ACCEPTED: the job was accepted and is being processed.
- BAD_REQUEST: the request was not accepted as it contained incorrect content.
- RESOURCE_NOT_EXIST: the requested resource (path) does not exist.
- DUPLICATE_RESOURCE: the resource already exists.
- SERVER_ERROR: an error on the server occurred.
- SUCCESS: the job finished successfully.
- FAIL: the job failed.
- APPROVED: the job was approved.
- DENIED: the job was denied.
- GONE: the job ID could not be found.

  A job can be GONE in the following cases:

  - The job was explicitly removed.
  - The job finished and has been deleted from the status cache (the job is removed from status cache after the cache has reached its maximum capacity).
  - The job never existed.

The following **`job`** calls are provided:

[GET] /jobs/{jobID}

returns the job status - [GET]

Example 13.1. Response of the job call on a repository clone request

```
"{"status":"SUCCESS","jodId":"1377770574783-27","result":"Alias:
testInstallAndDeployProject, Scheme: git, Uri:
git://testInstallAndDeployProject","lastModified":1377770578194,"deta
iledResult":null}"
```

[DELETE] /jobs/{jobID}

removes the job - [DELETE]

Report a bug

13.1.2. Repository calls

Repository calls are calls to the Knowledge Store that allow you to manage its Git repositories and their projects.

The following **`repositories`** calls are provided:

[GET] /repositories

This returns a list of the repositories in the Knowledge Store as a JSON entity - [GET]

Example 13.2. Response of the repositories call

```
[{"name":"bpms-assets","description":"generic
assets","userName":null,"password":null,"requestType":null,"gitURL":"
git://bpms-assets"},{"name":"loanProject","description":"Loan
processes and
rules","userName":null,"password":null,"requestType":null,"gitURL":"g
it://loansProject"}]
```

[DELETE] /repositories/{repositoryName}

This removes the repository from the Knowledge Store - [DELETE]

[POST] /repositories/

This creates or clones the repository defined by the JSON entity - [POST]

Example 13.3. JSON entity with repository details of a repository to be cloned

```
{"name":"myClonedRepository", "description":"", "userName":"",
"password":"", "requestType":"clone",
"gitURL":"git://localhost/example-repository"}
```

[POST] /repositories/{repositoryName}/projects/

This creates a project in the repository - [POST]

Example 13.4. Request body that defines the project to be created

```
"{"name":"myProject","description": "my project"}"
```

[DELETE] /repositories/{repositoryName}/projects/

This deletes the project in the repository - [DELETE]

Example 13.5. Request body that defines the project to be deleted

```
"{"name":"myProject","description": "my project"}"
```

Report a bug

13.1.3. Organizational unit calls

Organizational unit calls are calls to the Knowledge Store that allow you to manage its organizational units.

The following **`organizationalUnits`** calls are provided:

[GET] /organizationalunits/

This returns a list of all the organizational units - [GET].

[POST] /organizationalunits/

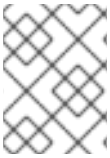
This creates an organizational unit in the Knowledge Store - [POST]. The organizational unit is defined as a JSON entity. This consumes an **`OrganizationalUnit`** instance and returns a **`CreateOrganizationalUnitRequest`** instance.

Example 13.6. Organizational unit in JSON

```
{
  "name":"testgroup",
  "description":"",
  "owner":"tester",
  "repositories":["testGroupRepository"]
}
```

[POST] /organizationalunits/{organizationalUnitName}/repositories/{repositoryName}

This adds the repository to the organizational unit - [POST]. It also returns a **`AddRepositoryToOrganizationalUnitRequest`** instance.



NOTE

Deleting an organizational unit is not supported via the REST API. The removal of an organizational unit is only possible through the Business Central.

Report a bug

13.1.4. Maven calls

Maven calls are calls to a Project in the Knowledge Store that allow you to compile and deploy the Project resources.

The following **`maven`** calls are provided below:

[POST] /repositories/{repositoryName}/projects/{projectName}/maven/compile/

This compiles the project (equivalent to **`mvn compile`**) - [POST]. It consumes a **`BuildConfig`** instance, which must be supplied but is not needed for the operation and may be left blank. It also returns a **`CompileProjectRequest`** instance.

[POST] /repositories/{repositoryName}/projects/{projectName}/maven/install/

This installs the project (equivalent to **`mvn install`**) - [POST]. It consumes a **`BuildConfig`** instance, which must be supplied but is not needed for the operation and may be left blank. It also returns a **`InstallProjectRequest`** instance.

[POST] /repositories/{repositoryName}/projects/{projectName}/maven/test/

This compiles and runs the tests - [POST]. It consumes a **`BuildConfig`** instance and returns a **`TestProjectRequest`** instance.

[POST] /repositories/{repositoryName}/projects/{projectName}/maven/deploy/

This deploys the project (equivalent to mvn deploy) - [POST]. It consumes a **`BuildConfig`** instance, which must be supplied but is not needed for the operation and may be left blank. It also returns a **`DeployProjectRequest`** instance.

Report a bug

13.2. DEPLOYMENT REST API

The kieModule jar files can be deployed or undeployed using the UI or REST API calls. This section details about the REST API deployment calls and their components.

NOTE

Configuration options like the runtime strategy should be defined before deploying the JAR files and cannot be changed post deployment.

A standard regular expression for a deploymentid call is:

```
[\w\.-]+(:[\w\.-]+){2,2}(:[\w\.-]*){0,2}
```

Where the "\w" refers to a character set that can contain the following character sets:

- [A-Z]
- [a-z]
- [0-9]
- _
- .
- -

Following are the elements of a Deployment ID and are separated from each other by a (:) character:

1. Group Id
2. Artifact Id
3. Version
4. kbase Id (optional)

5. ksession Id (optional)

Report a bug

13.2.1. Asynchronous calls

Deployment calls perform 2 [POST] asynchronous REST operations:

1. **`/deployment/{deploymentId}/deploy`**

2. **`/deployment/{deploymentId}/undeploy`**

Asynchronous calls can allow a user to issue a request and jump to the next task before the previous task in the queue is finished. So the information received after posting a call does not reflect the actual state or eventual status of the operation. This returns a status 202 upon the completion of the request which says that "The request has been accepted for processing, but the processing has not been completed.".

This even means that:

- The posted request would have been successfully accepted but the actual operation (deploying or undeploying the deployment unit) may have failed.

- The deployment information retrieved on calling the GET operations may even have changed (including the status of the deployment unit).

Report a bug

13.2.2. Deployment calls

The following **`deployment`** calls are provided:

/deployment/

returns a list of all available deployed instances [GET]

/deployment/{deploymentId}

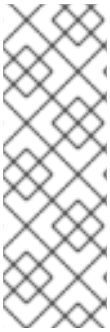
returns a JaxbDeploymentUnit instance containing the information (including the configuration) of the deployment unit [GET]

/deployment/{deploymentId}/deploy

deploys the deployment unit which is referenced by the deploymentid and returns a JaxbDeploymentJobResult instance with the status of the request [POST]

/deployment/{deploymentId}/undeploy

Undeploys the deployment unit referenced by the deploymentId and returns a JaxbDeploymentJobResult instance with the status of the request [POST]



NOTE

The deploy and undeploy operations can fail if one of the following is true:

- An identical job has already been submitted to the queue and has not yet completed.
- The amount of (deploy/undeploy) jobs submitted but not yet processed exceeds the job cache size.

Report a bug

13.3. RUNTIME REST API

Runtime REST API are calls to the Execution Server and to the Process Execution Engine, Task Execution Engine, and Business Rule Engine. As such they allow you to request and manipulate runtime data.

Except the Execute calls, all other REST calls can either use JAXB or JSON

The calls are synchronous and return the requested data as JAXB objects.

While using JSON, the JSON media type ("application/json") should be added to the ACCEPT header of the REST Call.

Their parameters are defined as query string parameters. To add a query string parameter to a runtime REST API call, add the **?** symbol to the REQUEST_BODY and the parameter with the parameter value; for example, **rest/task/query?workItemId=393** returns a TaskSumamry list of all tasks based on the work item with ID 393. Note that parameters and their values are case-sensitive.

When a runtime REST API call requires a Map parameter, you can submit key-value pairs to the operation using a query parameter prefixed with the keyword **map_** keyword; for example,

```
map_age=5000
```

is translated as

```
{ "age" => Long.parseLong("5000") }
```

Note that all runtime calls return a JAXB object.

Example 13.7. A GET call that returns all tasks to a locally running application using curl

```
curl -v -H 'Accept: application/json' -u eko
'localhost:8080/kie/rest/tasks/'
```

To perform runtime REST calls from your Java application you need to create a RemoteRestSessionFactory object and request a **newRuntimeEngine()** object from the RemoteRestSessionFactory. The RuntimeEngine can be then used to create a **KieSession**.

Example 13.8. A GET call that returns a task details to a locally running application in Java with the direct tasks/TASKID request

```
public Task getTaskInstanceInfo(long taskId) throws Exception {
    URL address = new URL(url + "/task/" + taskId);
    ClientRequest restRequest = createRequest(address);

    ClientResponse<JaxbTaskResponse> responseObj =
restRequest.get(JaxbTaskResponse.class);
    ClientResponse<InputStream> taskResponse =
responseObj.get(InputStream.class);
    JAXBContext jaxbTaskContext =
JAXBContext.newInstance(JaxbTaskResponse.class);
    StreamSource source  = new StreamSource(taskResponse.getEntity());
    return jaxbTaskContext.createUnmarshaller().unmarshal(source,
JaxbTaskResponse.class).getValue();

}

private ClientRequest createRequest(URL address) {
    return
getClientRequestFactory().createRequest(address.toExternalForm());
    }

private ClientRequestFactory getClientRequestFactory() {
    DefaultHttpClient httpClient = new DefaultHttpClient();
    httpClient.getCredentialsProvider().setCredentials(new
AuthScope(AuthScope.ANY_HOST,
    AuthScope.ANY_PORT, AuthScope.ANY_REALM), new
UsernamePasswordCredentials(userId, password));
    ClientExecutor clientExecutor = new
ApacheHttpClient4Executor(httpClient);
    return new ClientRequestFactory(clientExecutor,
ResteasyProviderFactory.getInstance());
}
```

Note that if you want to send multiple commands to an entity, in this case **task**, consider using the execute call (refer to Section 13.3.4, “Execute operations”).

While interacting with the Remote API, some classes are to be included in the deployment to enable a user to pass instances of their own classes as parameters to certain operations. REST calls that start with /task often do not contain any information about the associated deployment. In such a case and extra query parameter (deploymentId) is added to the REST call allowing the server to find the appropriate deployment class and deserialize the information passed with the call.

Report a bug

13.3.1. Usage Information

13.3.1.1. Pagination

The pagination parameters allow you to define pagination of the results a REST call returns. The following pagination parameters are available:

page or p

number of the page to be returned (by default set to **1**, that is, page number **1** is returned)

`pageSize` or `s`

number of items per page (default value **`10`**)

If both, the long option and the short option, are included in a URL, the longer version of the parameter takes precedence. When no pagination parameters are included, the returned results are not paginated.

Pagination parameters can be applied to the following REST requests:

```
/task/query
/history/instance
/history/instance/{id: [0-9]+}
/history/instance/{id: [0-9]+}/child
/history/instance/{id: [0-9]+}/node
/history/instance/{id: [0-9]+}/node/{id: [a-zA-Z0-9-:\\.]+}
/history/instance/{id: [0-9]+}/variable/
/history/instance/{id: [0-9]+}/variable/{id: [a-zA-Z0-9-:\\.]+}
/history/process/{id: [a-zA-Z0-9-:\\.]+}
```

Example 13.9. REST request body with the pagination parameter

```
/history/instance?page=3&pageSize=20
/history/instance?p=3&s=20
```

Report a bug

13.3.1.2. Object data type parameters

By default, any object parameters provided in a REST call are considered to be Strings. If you need to explicitly define the data type of a parameter of a call, you can do so by adding one of the following values to the parameter:

- **`\d+i`**: Integer
- **`\d+l`**: Long

Example 13.10. REST request body with the Integer mySignal parameter

```
/rest/runtime/business-central/process/org.jbpm.test/instance/2/signal?
mySignal=1234i
```

Note that the intended use of these object parameters is to define data types of send Signal and Process variable values (consider for example the use in the **`startProcess`** command in the execute call; refer to Section 13.3.4, “Execute operations”).

Report a bug

13.3.2. Runtime calls

Runtime REST calls allow you to acquire and manage data related to the runtime environment; you can provide direct REST calls to the Process Engine and Task Engine of the Process Server (refer to the *Components* section of the *Administration and Configuration Guide*.

To send calls to other Execution Engine components or issue calls that are not available as direct REST calls, use the generic execute call to runtime (**`/runtime/{deploymentId}/execute/{CommandObject}`**; refer to Section 13.3.4, “Execute operations”).

Report a bug

13.3.2.1. Process calls

The REST **`/runtime/{deploymentId}/process/`** calls are send to the Process Execution Engine.

The following **`process`** calls are provided:

/runtime/{deploymentId}/process/{procDefID}/start

creates and starts a Process instance of the provided Process definition [POST]

/runtime/{deploymentId}/process/instance/{procInstanceID}

returns the details of the given Process instance [GET]

/runtime/{deploymentId}/process/instance/{procInstanceID}/signal

sends a signal event to the given Process instance [POST]

The call accepts query map parameter with the Signal details.

> **Example 13.11. A local signal invocation and its REST version**
>
> ```
> ksession.signalEvent("MySignal", "value", 23l);
> ```
>
> ```
> curl -v -u admin 'localhost:8080/business-
> central/rest/runtime/myDeployment/process/instance/23/signal?
> signal=MySignal&event=value'
> ```

/runtime/{deploymentId}/process/instance/{procInstanceID}/abort

aborts the Process instance [POST]

/runtime/{deploymentId}/process/instance/{procInstanceID}/variables

returns variable of the Process instance [GET]

Variables are returned as JaxbVariablesResponse objects. Note that the returned variable values are strings.

Report a bug

13.3.2.2. Signal calls

The REST **`signal/`** calls send a signal defined by the provided query map parameters either to the deployment or to a particular process instance.

The following **`signal`** calls are provided:

/runtime/{deploymentId}/process/instance/{procInstanceID}/signal

sends a signal to the given process instance [POST]

See the previous subsection for an example of this call.

/runtime/{deploymentId}/signal

This operation takes a signal and a event query parameter and sends a signal to the deployment [POST].

- The ***`signal`*** parameter value is used as the name of the signal. This parameter is required.

- The ***`event`*** parameter value is used as the value of the event. This value may use the number query parameter syntax described earlier.

Example 13.12. Signal Call Example

```
/runtime/{deploymentId}/signal?signal={signalCode}
```

This call is equivalent to the **`ksession.signal("signalName", eventValue)`** method.

Report a bug

13.3.2.3. Work item calls

The REST **`/runtime/{deploymentId}/workitem/`** calls allow you to complete or abort a particular work item.

The following **`task`** calls are provided:

/runtime/{deploymentId}/workitem/{workItemID}/complete

completes the given work item [POST]

The call accepts query map parameters containing information about the results.

Example 13.13. A local invocation and its REST version

```
Map<String, Object> results = new HashMap<String, Object>();
results.put("one", "done");
results.put("two", 2);
kieSession.getWorkItemManager().completeWorkItem(23l, results);
```

```
curl -v -u admin 'localhost:8080/business-
central/rest/runtime/myDeployment/workitem/23/complete?
map_one=done&map_two=2i'
```

/runtime/{deploymentId}/workitem/{workItemID}/abort

aborts the given work item [POST]

Report a bug

13.3.2.4. History calls

The REST `/history/` calls administer logs of process instances, their nodes, and process variables.

NOTE

While the REST `/history/`calls specified in 6.0.0.GA of BPMS are still available, as of 6.0.1.GA, the `/history/` calls have been made independent of any deployment, which is also reflected in the URLS used.

The following `history` calls are provided:

/history/clear

clears all process, variable, and node logs [POST]

/history/instances

returns logs of all Process instances [GET]

/history/instance/{procInstanceID}

returns all logs of Process instance (including child logs) [GET]

/history/instance/{procInstanceID}/child

returns logs of child Process instances [GET]

/history/instance/{procInstanceID}/node

returns logs of all nodes of the Process instance [GET]

/history/instance/{procInstanceID}/node/{nodeID}

returns logs of the node of the Process instance [GET]

/history/instance/{procInstanceID}/variable

returns variables of the Process instance with their values [GET]

/history/instance/{procInstanceID}/variable/{variableID}

returns the log of the process instance that have the given variable id [GET]

/history/process/{procInstanceID}

returns the logs of the given Process instance excluding logs of its nodes and variables [GET]

History calls that search by variable

The following REST calls can be used using variables to search process instance, variables and their values.

The REST calls below also accept an optional **`activeProcesses`** parameter that limits the selection to information from active process instances.

/history/variable/{varId}

returns the variable logs of the specified process variable [GET]

/history/variable/{varId}/instances

returns the process instance logs for processes that contain the specified process variable [GET]

/history/variable/{varId}/value/{value}

returns the variable logs for specified process variable with the specified value [GET]

Example 13.14. A local invocation and its REST version

```
auditLogService.findVariableInstancesByNameAndValue("countVar",
"three", true);
```

```
curl -v -u admin 'localhost:8080/business-
central/rest/history/variable/countVar/value/three?
activeProcesses=true'
```

/history/variable/{varId}/value/{value}/instances

returns the process instance logs for process instances that contain the specified process variable with the specified value [GET]

Report a bug

13.3.2.5. Calls to process variables

The REST **`/runtime/{deploymentId}/withvars/`** calls allow you to work with Process variables. Note that all variable values are returned as strings in the JaxbVariablesResponse object.

The following **`withvars`** calls are provided:

/runtime/{deploymentId}/withvars/process/{procDefinitionID}/start

creates and starts Process instance and returns the Process instance with its variables Note that even if a passed variable is not defined in the underlying Process definition, it is created and initialized with the passed value. [POST]

/runtime/{deploymentId}/withvars/process/instance/{procInstanceID}

returns Process instance with its variables [GET]

/runtime/{deploymentId}/withvars/process/instance/{procInstanceID}/signal

sends signal event to Process instance (accepts query map parameters).

Report a bug

13.3.3. Task calls

The REST task calls are send to the Task Execution Engine.

The following **`task`** calls are provided:

/task/{taskId: \\d+}

returns the task in JAXB format [GET]

Further call paths are provided to perform other actions on tasks; refer to Section 13.3.3.1, “Task ID operations”)

/task/query

returns a TaskSummary list returned [GET]

Further call paths are provided to perform other actions on task/query; refer to Section 13.3.3.3, “Query operations”).

/task/content/{contentId: \\d+}

returns the task content in the JAXB format [GET]

For further information, refer to Section 13.3.3.2, “Content operations”)

Report a bug

13.3.3.1. Task ID operations

The **`task/{taskId: \\d+}/ACTION`** calls allow you to execute an action on the given task (if no action is defined, the call is a GET call that returns the JAXB representation of the task).

The following actions can be invoked on a task using the call:

Table 13.1. Task Actions

Task	Action
`activate`	activate task (taskId as query param..)
`claim`	claim task [POST] (The user used in the authentication of the REST url call claims it.)
`claimnextavailable`	claim next available task [POST] (This operation claims the next available task assigned to the user.)
`complete`	complete task [POST] (accepts "query map parameters".)
`delegate`	delegate task [POST] (Requires a **`targetIdquery`** parameter, which identifies the user or group to which the task is delegated.)

Task	Action
exit	exit task [POST]
fail	fail task [POST]
forward	forward task [POST]
release	release task [POST]
resume	resume task [POST]
skip	skip task [POST]
start	start task [POST]
stop	stop task [POST]
suspend	suspend task [POST]
nominate	nominate task [POST] (Requires at least one of either the user or group query parameter, which identify the user(s) or group(s) that are nominated for the task.)

Report a bug

13.3.3.2. Content operations

The **task/content/{contentId: \\d+}** and **task/{taskId: \\d+}/content** operations return the serialized content associated with the given task.

The content associated with a task is stored in the human-task database schema in serialized form either as a string with XML content or a map with several different key-value pairs. The content is serialized using the protobuf based algorithm. This serialization process is normally carried out by the static methods in the **org.jbpm.services.task.utils.ContentMarshallerHelper** class.

If the client that call the REST operation do not have access to the **org.jbpm.services.task.utils.ContentMarshallerHelper** class, they cannot deserialize the task content. When using the REST call to obtain task content, the content is first deserialized usint the **ContentMarshallerHelper** class and then serialized with the common Java serialization mechanism.

Due to restrictions of REST operations, only the objects for which the following is true can be returned to the task content operations:

- The requested objects are instances of a class that implements the **Serializable** interface. In the case of Map objects, they only contain values that implement the **Serializable** interface.
- The objects are *not* instances of a local class, an anonymous class or arrays of a local or anonymous class.
- The object classes are present on the class path of the server .

Report a bug

13.3.3.3. Query operations

The **`/task/query`** call is a GET call that returns a TaskSummary list of the tasks that meet the criteria defined in the call parameters. Note that you can use the pagination feature to define the amount of data to be return.

Parameters
The following parameters can be used with the **`task/query`** call:

- **`workItemId`**: returns only tasks based on the work item.
- **`taskId`**: returns only the task with the particular ID.
- **`businessAdministrator`**: returns task with an identified business administrator.
- **`potentialOwner`**: returns tasks that can be claimed by the potentialOwner user.
- **`status`**: returns tasks that are in the given status (**`Created`**, **`Ready`**, **`Reserved`**, **`InProgress`**, **`Completed`** or **`Failed`**);
- **`taskOwner`**: returns tasks assigned to the particular user (**`Created`**, **`Ready`**, **`Reserved`**, **`InProgress`**, **`Suspended`**, **`Completed`**, **`Failed`**, **`Error`**, **`Exited`**, or **`Obsolete`**).
- **`processInstanceId`**: returns tasks generated by the Process instance.
- **`union`**: specifies whether the query should query the union or intersection of the parameters.

At the moment, although the name of a parameter is interpreted regardless of case, please make sure use the appropriate case for the *values* of the parameters.

> **Example 13.15. Query usage**
>
> This call retrieves the task summaries of all tasks that have a work item id of 3, 4, *or* 5. If you specify the *same* parameter multiple times, the query will select tasks that match *any* of that parameter.
>
> - **`http://server:port/rest/task/query?workItemId=3&workItemId=4&workItemId=5`**
>
> The next call will retrieve any task summaries for which the task id is 27 *and* for which the work item id is 11. Specifying different parameters will result in a set of tasks that match both (all) parameters.
>
> - **`http://server:port/rest/task/query?workItemId=11&taskId=27`**
>
> The next call will retrieve any task summaries for which the task id is 27 *or* the work item id is 11. While these are different parameters, the **`union`** parameter is being used here so that the union of the two queries (the work item id query and the task id query) is returned.
>
> - **`http://server:port/rest/task/query?workItemId=11&taskId=27&union=true`**
>
> The next call will retrieve any task summaries for which the status is **`Created`** *and* the potential owner of the task is `Bob`. Note that the letter case for the status parameter value is case-*in*sensitve.
>
> - **`http://server:port/rest/task/query?status=creAted&potentialOwner=Bob`**

The next call will return any task summaries for which the status is `**Created**` *and* the potential owner of the task is `bob`. Note that the potential owner parameter is case-*sensitive*. `bob` is not the same user id as `Bob`!

- **`http://server:port/rest/task/query?status=created&potentialOwner=bob`**

The next call will return the *intersection* of the set of task summaries for which the process instance is 201, the potential owner is `bob` and for which the status is `**Created**` *or* `**Ready**`.

- **`http://server:port/rest/task/query?status=created&status=ready&potentialOwner=bob&processInstanceId=201`**

That means that the task summaries that have the following characteristics would be included:

- process instance id 201, potential owner `bob`, status `Ready`
- process instance id 201, potential owner `bob`, status `Created`

And that following task summaries will *not* be included:

- process instance id *183*, potential owner `bob`, status `Created`
- process instance id 201, potential owner `mary`, status `Ready`
- process instance id 201, potential owner `bob`, status `*Complete*`

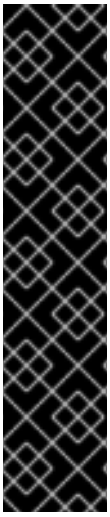
Usage
The parameters can be used by themselves or in certain combinations. If an unsupported parameter combination is used, the system returns an empty list.

You can use certain parameters multiple times with different values: the returned result will contain the union of the entities that met at least one of the defined parameter value. This applies to the ***`workItemId`***, ***`taskId`***, ***`businessAdministrator`***, ***`potentialOwner`***, ***`taskOwner`***, and ***`processInstanceId`***. If entering the ***`status`*** parameter multiple times, the intersection of tasks that have any of the status values and union of tasks that satisfy the other criteria.

Note that the ***`language`*** parameter is required and if not defined the **`en-UK`** value is used. The parameter can be defined only once.

Report a bug

13.3.4. Execute operations



IMPORTANT

The **execute** operations were created in order to support the Java Remote Runtime API. As a result, these calls are also available to the user. However, all of the functionality that these operations expose can be more easily accessed either via other REST operations or via the Java Remote Runtime API.

The Java Remote Runtime API, described in a following section, provides a programmatic interface to the REST and JMS APIs and takes care of the underlying details of sending and receiving commands via REST or JMS.

Advanced users looking to send a batch of commands via the REST API can use the **execute** operation. This is the only way to have the REST API process multiple commands in one operation.

To execute multiple commands in a single REST call, it is convenient to use the **execute** call: the call takes the **JaxbCommandsRequest** object as its parameter. The **JaxbCommandsRequest** object contains a List of **org.kie.api.*COMMAND*.Command** objects (the commands are "stored" in the **JaxbCommandsRequest** object as Strings and send via the **execute** REST call). The JaxbCommandsRequest parameters are ***deploymentId***, if applicable ***processInstanceId***, and a ***Command*** object.

```
  public List<JaxbCommandResponse<?>> executeCommand(String deploymentId,
List<Command<?>> commands) throws Exception {
         URL address = new URL(url + "/runtime/" + deploymentId +
"/execute");
         ClientRequest restRequest = createRequest(address);

         JaxbCommandsRequest commandMessage = new
JaxbCommandsRequest(deploymentId, commands);
         String body =
JaxbSerializationProvider.convertJaxbObjectToString(commandMessage);
         restRequest.body(MediaType.APPLICATION_XML, body);

         ClientResponse<JaxbCommandsResponse> responseObj =
restRequest.post(JaxbCommandsResponse.class);
         checkResponse(responseObj);
         JaxbCommandsResponse cmdsResp = responseObj.getEntity();
         return cmdsResp.getResponses();
     }
          private ClientRequest createRequest(URL address) {
         return
getClientRequestFactory().createRequest(address.toExternalForm());
     }

     private ClientRequestFactory getClientRequestFactory() {
         DefaultHttpClient httpClient = new DefaultHttpClient();
         httpClient.getCredentialsProvider().setCredentials(new
AuthScope(AuthScope.ANY_HOST,
                 AuthScope.ANY_PORT, AuthScope.ANY_REALM), new
UsernamePasswordCredentials(userId, password));
         ClientExecutor clientExecutor = new
ApacheHttpClient4Executor(httpClient);
         return new ClientRequestFactory(clientExecutor,
ResteasyProviderFactory.getInstance());
     }
```

Figure 13.1. Method implementing the execute REST call

The following is a list of commands that the **`execute`** operation will accept. See the constructor and set methods on the actual command classes for further information about which parameters these commands will accept.

The following is the list of accepted commands used to interact with the process engine:

AbortWorkItemCommand	SignalEventCommand
CompleteWorkItemCommand	StartCorrelatedProcessCommand
GetWorkItemCommand	StartProcessCommand
AbortProcessInstanceCommand	GetVariableCommand
GetProcessIdsCommand	GetFactCountCommand
GetProcessInstanceByCorrelationKeyCommand	GetGlobalCommand

GetProcessInstanceCommand	GetIdCommand
GetProcessInstancesCommand	FireAllRulesCommand
SetProcessInstanceVariablesCommand	

The following is the list of accepted commands that interact with task instances:

ActivateTaskCommand	GetTaskAssignedAsPotentialOwnerCommand
AddTaskCommand	GetTaskByWorkItemIdCommand
CancelDeadlineCommand	GetTaskCommand
ClaimNextAvailableTaskCommand	GetTasksByProcessInstanceIdCommand
ClaimTaskCommand	GetTasksByStatusByProcessInstanceIdCommand
CompleteTaskCommand	GetTasksOwnedCommand
CompositeCommand	NominateTaskCommand
DelegateTaskCommand	ProcessSubTaskCommand
ExecuteTaskRulesCommand	ReleaseTaskCommand
ExitTaskCommand	ResumeTaskCommand
FailTaskCommand	SkipTaskCommand
ForwardTaskCommand	StartTaskCommand
GetAttachmentCommand	StopTaskCommand
GetContentCommand	SuspendTaskCommand
GetTaskAssignedAsBusinessAdminCommand	

The following is the list of accepted commands for managing and retrieving historical (audit) information:

ClearHistoryLogsCommand	FindSubProcessInstancesCommand
FindActiveProcessInstancesCommand	FindSubProcessInstancesCommand
FindNodeInstancesCommand	FindVariableInstancesByNameCommand

FindProcessInstanceCommand	FindVariableInstancesCommand
FindProcessInstancesCommand	

Report a bug

13.4. REST SUMMARY

The URL templates in the table below are relative to the following URL:

- `http://server:port/business-central/rest`

Table 13.2. Knowledge Store REST calls

URL Template	Type	Description
/jobs/{jobID}	GET	return the job status
/jobs/{jobID}	DELETE	remove the job
/organizationalunits	GET	return a list of organizational units
/organizationalunits	POST	create an organizational unit in the Knowledge Store described by the JSON **`OrganizationalUnit`** entity
/organizationalunits/{organizationalUnitName}/repositories/{repositoryName}	POST	add a repository to an organizational unit
/repositories/	POST	add the repository to the organizational unit described by the JSON **`RepositoryReqest`** entity
/repositories	GET	return the repositories in the Knowledge Store
/repositories/{repositoryName}	DELETE	remove the repository from the Knowledge Store
/repositories/	POST	create or clone the repository defined by the JSON **`RepositoryRequest`** entity

URL Template	Type	Description
/repositories/{repositoryName}/projects/	POST	create the project defined by the JSON entity in the repository
/repositories/{repositoryName}/projects/{projectName}/maven/compile/	POST	compile the project
/repositories/{repositoryName}/projects/{projectName}/maven/install	POST	install the project
/repositories/{repositoryName}/projects/{projectName}/maven/test/	POST	compile the project and run tests as part of compilation
/repositories/{repositoryName}/projects/{projectName}/maven/deploy/	POST	deploy the project

Table 13.3. runtime REST calls

URL Template	Type	Description
/runtime/{deploymentId}/process/{procDefID}/start	POST	start a process instance based on the Process definition (accepts query map parameters)
/runtime/{deploymentId}/process/instance/{procInstanceID}	GET	return a process instance details
/runtime/{deploymentId}/process/instance/{procInstanceID}/abort	POST	abort the process instance
/runtime/{deploymentId}/process/instance/{procInstanceID}/signal	POST	send a signal event to process instance (accepts query map parameters)
/runtime/{deploymentId}/process/instance/{procInstanceID}/variable/{varId}	GET	return a variable from a process instance
/runtime/{deploymentId}/signal/{signalCode}	POST	send a signal event to deployment
/runtime/{deploymentId}/workitem/{workItemID}/complete	POST	complete a work item (accepts query map parameters)
/runtime/{deploymentId}/workitem/{workItemID}/abort	POST	abort a work item

URL Template	Type	Description
/runtime/{deploymentId}/withvars/process/{procDefinitionID}/start	POST	start a process instance and return the process instance with its variables Note that even if a passed variable is not defined in the underlying process definition, it is created and initialized with the passed value.
/runtime/{deploymentId}/withvars/process/instance/{procInstanceID}/	GET	return a process instance with its variables
/runtime/{deploymentId}/withvars/process/instance/{procInstanceID}/signal	POST	send a signal event to the process instance (accepts query map parameters) The following query parameters are accepted: • The **`signal`** parameter specifies the name of the signal to be sent • The **`event`** parameter specifies the (optional) value of the signal to be sent

Table 13.4. task REST calls

URL Template	Type	Description
/task/query	GET	return a TaskSummary list
/task/content/{contentID}	GET	returns the content of a task
/task/{taskID}	GET	return the task
/task/{taskID}/activate	POST	activate the task
/task/{taskID}/claim	POST	claim the task
/task/{taskID}/claimnextavailable	POST	claim the next available task

URL Template	Type	Description
/task/{taskID}/complete	POST	complete the task (accepts query map paramaters)
/task/{taskID}/delegate	POST	delegate the task
/task/{taskID}/exit	POST	exit the task
/task/{taskID}/fail	POST	fail the task
/task/{taskID}/forward	POST	forward the task
/task/{taskID}/nominate	POST	nominate the task
/task/{taskID}/release	POST	release the task
/task/{taskID}/resume	POST	resume the task (after suspending)
/task/{taskID}/skip	POST	skip the task
/task/{taskID}/start	POST	start the task
/task/{taskID}/stop	POST	stop the task
/task/{taskID}/suspend	POST	suspend the task
/task/{taskID}/content	GET	returns the content of a task

Table 13.5. history REST calls

URL Template	Type	Description
/history/clear/	POST	delete all process, node and history records

URL Template	Type	Description
/history/instances	GET	return the list of all process instance history records
/history/instance/{procInstId}	GET	return a list of process instance history records for a process instance
/history/instance/{procInstId}/child	GET	return a list of process instance history records for the subprocesses of the process instance
/history/instance/{procInstId}/node	GET	return a list of node history records for a process instance
/history/instance/{procInstId}/node/{nodeId}	GET	return a list of node history records for a node in a process instance
/history/instance/{procInstId}/variable	GET	return a list of variable history records for a process instance
/history/instance/{procInstId}/variable/{variableId}	GET	return a list of variable history records for a variable in a process instance
/history/process/{procDefId}	GET	return a list of process instance history records for process instances using a given process definition
/history/variable/{varId}	GET	return a list of variable history records for a variable
/history/variable/{varId}/instances	GET	return a list of process instance history records for process instances that contain a variable with the given variable id
/history/variable/{varId}/value/{value}	GET	return a list of variable history records for variable(s) with the given variable id and given value

URL Template	Type	Description
/history/variable/{varId}/value/{value}/instances	GET	return a list of process instance history records for process instances with the specified variable that contains the specified variable value

Table 13.6. deployment REST calls

URL Template	Type	Description
/deployment	GET	return a list of (deployed) deployments
/deployment/{deploymentId}	GET	return the status and information about the deployment
/deployment/{deploymentId}/deploy	POST	submit a request to deploy a deployment
/deployment/{deploymentId}/undeploy	POST	submit a request to undeploy a deployment

Report a bug

13.5. JMS

The Java Message Service (JMS) is an API that allows Java Enterprise components to communicate with each other asynchronously and reliably.

Operations on the runtime engine and tasks can be done via the JMS API exposed by the jBPM console and KIE workbench. However, it's not possible to manage deployments or the knowledge base via this JMS API.

Unlike the REST API, it is possible to send a batch of commands to the JMS API that will all be processed in one request after which the responses to the commands will be collected and return in one response message.

Report a bug

13.5.1. JMS Queue Setup

When the Workbench is deployed on the JBoss AS or EAP server, it automatically creates 3 queues:

- **`jms/queue/KIE.SESSION`**
- **`jms/queue/KIE.TASK`**

- **jms/queue/KIE.RESPONSE**

The **KIE.SESSION** and **KIE.TASK** queues should be used to send request messages to the JMS API. Command response messages will be then placed on the **KIE.RESPONSE** queues. Command request messages that involve starting and managing business processes should be sent to the **KIE.SESSION** and command request messages that involve managing human tasks, should be sent to the **KIE.TASK** queue.

Although there are 2 different input queues, **KIE.SESSION** and **KIE.TASK**, this is only in order to provide multiple input queues so as to optimize processing: command request messages will be processed in the same manner regardless of which queue they're sent to. However, in some cases, users may send many more requests involving human tasks than requests involving business processes, but then not want the processing of business process-related request messages to be delayed by the human task messages. By sending the appropriate command request messages to the appropriate queues, this problem can be avoided.

The term "*command request message*" used above refers to a JMS byte message that contains a serialized **JaxbCommandsRequest** object. At the moment, only XML serialization (as opposed to, JSON or protobuf, for example) is supported.

Report a bug

13.5.2. Serialization issues

Sometimes, users may wish to pass instances of their own classes as parameters to commands sent in a REST request or JMS message. In order to do this, there are a number of requirements.

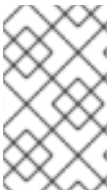
1. The user-defined class satisfy the following in order to be property serialized and deserialized by the JMS or REST API:
    - The user-defined class must be correctly annotated with JAXB annotations, including the following:
        - The user-defined class must be annotated with a **javax.xml.bind.annotation.XmlRootElement** annotation with a non-empty **name** value
        - All fields or getter/setter methods must be annotated with a **javax.xml.bind.annotation.XmlElement** or **javax.xml.bind.annotation.XmlAttribute** annotations.

      Furthermore, the following usage of JAXB annotations is recommended:

        - Annotate the user-defined class with a **javax.xml.bind.annotation.XmlAccessorType** annotation specifying that fields should be used, (**javax.xml.bind.annotation.XmlAccessType.FIELD**). This also means that you should annotate the fields (instead of the getter or setter methods) with **@XmlElement** or **@XmlAttribute** annotations.
        - Fields annotated with **@XmlElement** or **@XmlAttribute** annotations should also be annotated with **javax.xml.bind.annotation.XmlSchemaType** annotations specifying the type of the field, even if the fields contain primitive values.
        - Use objects to store primitive values. For example, use the **java.lang.Integer** class for storing an integer value, and not the **int** class. This way it will always be obvious if the field is storing a value.

- The user-defined class definition must implement a no-arg constructor.
- Any fields in the user-defined class must either be object primitives (such as a **Long** or **String**) or otherwise be objects that satisfy the first 2 requiremends in this list (correct usage of JAXB annotations and a no-arg constructor).

2. The class definition must be included in the deployment jar of the deployment that the JMS message content is meant for.
3. The sender must set a “deploymentId” string property on the JMS bytes message to the name of the deploymentId. This property is necessary in order to be able to load the proper classes from the deployment itself before deserializing the message on the server side.



NOTE

While *submitting* an instance of a user-defined class is possible via both the JMS and REST API's, *retrieving* an instance of the process variable is only possible via the REST API.

Report a bug

13.5.3. Example JMS Usage

The following is an example that shows how to use the JMS API. The numbers ("callouts") along the side of the example refer to notes below that explain particular parts of the example. It's supplied for those advanced users that do not wish to use the BPMS Remote Java API.

The BPMS Remote Java API, described here, will otherwise take care of all of the logic shown below.

```
// normal java imports skipped

import org.drools.core.command.runtime.process.StartProcessCommand;
import
org.jbpm.services.task.commands.GetTaskAssignedAsPotentialOwnerCommand;
import org.kie.api.command.Command;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.api.task.model.TaskSummary;
import
org.kie.services.client.api.command.exception.RemoteCommunicationException
                                                                    1
;
import org.kie.services.client.serialization.JaxbSerializationProvider;
import org.kie.services.client.serialization.SerializationConstants;
import org.kie.services.client.serialization.SerializationException;
import
org.kie.services.client.serialization.jaxb.impl.JaxbCommandResponse;
import
org.kie.services.client.serialization.jaxb.impl.JaxbCommandsRequest;
import
org.kie.services.client.serialization.jaxb.impl.JaxbCommandsResponse;
import
org.kie.services.client.serialization.jaxb.rest.JaxbExceptionResponse;

import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;

public class DocumentationJmsExamples {

  protected static final Logger logger =
LoggerFactory.getLogger(DocumentationJmsExamples.class);

  public void sendAndReceiveJmsMessage() {

    String USER = "charlie";
    String PASSWORD = "ch0c0licious";

    String DEPLOYMENT_ID = "test-project";
    String PROCESS_ID_1 = "oompa-processing";
    URL serverUrl;
    try {
      serverUrl = new URL("http://localhost:8080/business-central/");
    } catch (MalformedURLException murle) {
      logger.error("Malformed URL for the server instance!", murle);
      return;
    }

    // Create JaxbCommandsRequest instance and add commands
    Command<?> cmd = new StartProcessCommand(PROCESS_ID_1);
    int oompaProcessingResultIndex = 0;
```

5

```
    JaxbCommandsRequest req = new JaxbCommandsRequest(DEPLOYMENT_ID, cmd);
```

2

```
    req.getCommands().add(new GetTaskAssignedAsPotentialOwnerCommand(USER,
"en-UK"));
    int loompaMonitoringResultIndex = 1;
```

5

```
    // Get JNDI context from server
    InitialContext context = getRemoteJbossInitialContext(serverUrl, USER,
PASSWORD);

    // Create JMS connection
    ConnectionFactory connectionFactory;
    try {
      connectionFactory = (ConnectionFactory)
context.lookup("jms/RemoteConnectionFactory");
    } catch (NamingException ne) {
      throw new RuntimeException("Unable to lookup JMS connection
factory.", ne);
    }

    // Setup queues
    Queue sendQueue, responseQueue;
    try {
      sendQueue = (Queue) context.lookup("jms/queue/KIE.SESSION");
      responseQueue = (Queue) context.lookup("jms/queue/KIE.RESPONSE");
```

```
    } catch (NamingException ne) {
      throw new RuntimeException("Unable to lookup send or response
queue", ne);
    }

    // Send command request
    Long processInstanceId = null; // needed if you're doing an operation
on a PER_PROCESS_INSTANCE deployment
    String humanTaskUser = USER;
    JaxbCommandsResponse cmdResponse = sendJmsCommands(
        DEPLOYMENT_ID, processInstanceId, humanTaskUser, req,
        connectionFactory, sendQueue, responseQueue,
        USER, PASSWORD, 5);

    // Retrieve results
    ProcessInstance oompaProcInst = null;
    List<TaskSummary> charliesTasks = null;
    for (JaxbCommandResponse<?> response : cmdResponse.getResponses()) {
```

6

```
      if (response instanceof JaxbExceptionResponse) {
        // something went wrong on the server side
        JaxbExceptionResponse exceptionResponse = (JaxbExceptionResponse)
response;
        throw new RuntimeException(exceptionResponse.getMessage());
      }

      if (response.getIndex() == oompaProcessingResultIndex) {
```

5

```
        oompaProcInst = (ProcessInstance) response.getResult();
```

6

```
      } else if (response.getIndex() == loompaMonitoringResultIndex) {
```

5

```
        charliesTasks = (List<TaskSummary>) response.getResult();
```

6

```
      }
    }
  }

  private JaxbCommandsResponse sendJmsCommands(String deploymentId, Long
processInstanceId, String user, JaxbCommandsRequest req,
      ConnectionFactory factory, Queue sendQueue, Queue responseQueue,
String jmsUser, String jmsPassword, int timeout) {
    req.setProcessInstanceId(processInstanceId);
    req.setUser(user);

    Connection connection = null;
    Session session = null;
    String corrId = UUID.randomUUID().toString();
    String selector = "JMSCorrelationID = '" + corrId + "'";
```

```
    JaxbCommandsResponse cmdResponses = null;
    try {

      // setup
      MessageProducer producer;
      MessageConsumer consumer;
      try {
        if (jmsPassword != null) {
          connection = factory.createConnection(jmsUser, jmsPassword);
        } else {
          connection = factory.createConnection();
        }
        session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

        producer = session.createProducer(sendQueue);
        consumer = session.createConsumer(responseQueue, selector);

        connection.start();
      } catch (JMSException jmse) {
        throw new RemoteCommunicationException("Unable to setup a JMS
connection.", jmse);
      }

      JaxbSerializationProvider serializationProvider = new
JaxbSerializationProvider();
      // if necessary, add user-created classes here:
      // xmlSerializer.addJaxbClasses(MyType.class,
AnotherJaxbAnnotatedType.class);

      // Create msg
      BytesMessage msg;
      try {
        msg = session.createBytesMessage();
3


        // set properties
        msg.setJMSCorrelationID(corrId);
3


msg.setIntProperty(SerializationConstants.SERIALIZATION_TYPE_PROPERTY_NAME

                                                    3
, JaxbSerializationProvider.JMS_SERIALIZATION_TYPE);
        Collection<Class<?>> extraJaxbClasses =
serializationProvider.getExtraJaxbClasses();
        if (!extraJaxbClasses.isEmpty()) {
          String extraJaxbClassesPropertyValue = JaxbSerializationProvider
              .classSetToCommaSeperatedString(extraJaxbClasses);

msg.setStringProperty(SerializationConstants.EXTRA_JAXB_CLASSES_PROPERTY_N
AME, extraJaxbClassesPropertyValue);

msg.setStringProperty(SerializationConstants.DEPLOYMENT_ID_PROPERTY_NAME,
```

```
deploymentId);
        }

        // serialize request
        String xmlStr = serializationProvider.serialize(req);
```

```
        msg.writeUTF(xmlStr);
      } catch (JMSException jmse) {
        throw new RemoteCommunicationException("Unable to create and fill
a JMS message.", jmse);
      } catch (SerializationException se) {
        throw new RemoteCommunicationException("Unable to deserialze JMS
message.", se.getCause());
      }

      // send
      try {
        producer.send(msg);
      } catch (JMSException jmse) {
        throw new RemoteCommunicationException("Unable to send a JMS
message.", jmse);
      }

      // receive
      Message response;
      try {
        response = consumer.receive(timeout);
      } catch (JMSException jmse) {
        throw new RemoteCommunicationException("Unable to receive or
retrieve the JMS response.", jmse);
      }

      if (response == null) {
        logger.warn("Response is empty, leaving");
        return null;
      }
      // extract response
      assert response != null : "Response is empty.";
      try {
        String xmlStr = ((BytesMessage) response).readUTF();
        cmdResponses = (JaxbCommandsResponse)
serializationProvider.deserialize(xmlStr);
```

```
      } catch (JMSException jmse) {
        throw new RemoteCommunicationException("Unable to extract " +
JaxbCommandsResponse.class.getSimpleName()
            + " instance from JMS response.", jmse);
      } catch (SerializationException se) {
        throw new RemoteCommunicationException("Unable to extract " +
JaxbCommandsResponse.class.getSimpleName()
            + " instance from JMS response.", se.getCause());
      }
      assert cmdResponses != null : "Jaxb Cmd Response was null!";
```

```
    } finally {
      if (connection != null) {
        try {
          connection.close();
          session.close();
        } catch (JMSException jmse) {
          logger.warn("Unable to close connection or session!", jmse);
        }
      }
    }
    return cmdResponses;
  }

  private InitialContext getRemoteJbossInitialContext(URL url, String
user, String password) {
    Properties initialProps = new Properties();
    initialProps.setProperty(InitialContext.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
    String jbossServerHostName = url.getHost();
    initialProps.setProperty(InitialContext.PROVIDER_URL, "remote://"+
jbossServerHostName + ":4447");
    initialProps.setProperty(InitialContext.SECURITY_PRINCIPAL, user);
    initialProps.setProperty(InitialContext.SECURITY_CREDENTIALS,
password);

    for (Object keyObj : initialProps.keySet()) {
      String key = (String) keyObj;
      System.setProperty(key, (String) initialProps.get(key));
    }
    try {
      return new InitialContext(initialProps);
    } catch (NamingException e) {
      throw new RemoteCommunicationException("Unable to create " +
InitialContext.class.getSimpleName(), e);
    }
  }
}
```

1	These classes can all be found in the **`kie-services-client`** and the **`kie-services-jaxb`** JAR.
2	The **`JaxbCommandsRequest`** instance is the "holder" object in which you can place all of the commands you want to execute in a particular request. By using the **`JaxbCommandsRequest.getCommands()`** method, you can retrieve the list of commands in order to add more commands to the request. A deployment id is required for command request messages that deal with business processes. Command request messages that only contain human task-related commands do not require a deployment id.

3	Note that the JMS message sent to the remote JMS API *must* be constructed as follows: • It must be a JMS byte message. • It must have a filled JMS Correlation ID property. • It must have an int property with the name of "serialization" set to an acceptable value (only 0 at the moment). • It must contain a serialized instance of a **`JaxbCommandsRequest`**, added to the message as a UTF string
4	The same serialization mechanism used to serialize the request message will be used to serialize the response message.
5	In order to match the response to a command, to the initial command, use the **`index`** field of the returned **`JaxbCommandResponse`** instances. This **`index`** field will match the index of the initial command. Because not all commands will return a result, it's possible to send 3 commands with a command request message, and then receive a command response message that only includes one **`JaxbCommandResponse`** message with an **`index`** value of 1. That 1 then identifies it as the response to the second command.
6	Since many of the results returned by various commands are not serializable, the jBPM JMS Remote API converts these results into JAXB equivalents, all of which implement the **`JaxbCommandResponse`** interface. The **`JaxbCommandResponse.getResult()`** method then returns the JAXB equivalent to the actual result, which will conform to the interface of the result. For example, in the code above, the **`StartProcessCommand`** returns a **`ProcessInstance`**. In order to return this object to the requester, the **`ProcessInstance`** is converted to a **`JaxbProcessInstanceResponse`** and then added as a **`JaxbCommandResponse`** to the command response message. The same applies to the **`List<TaskSummary>`** that's returned by the **`GetTaskAssignedAsPotentialOwnerCommand`**. *However, not all methods that can be called on a normal* ***`ProcessInstance`*** *can be called on the* ***`JaxbProcessInstanceResponse`*** because the **`JaxbProcessInstanceResponse`** is simply a representation of a **`ProcessInstance`** object. This applies to various other command response as well. In particular, methods which require an active (backing) **`KieSession`**, such as **`ProcessInstance.getProess()`** or **`ProcessInstance.signalEvent(String type, Object event)`** will throw an **`UnsupportedOperationException`**.

Report a bug

13.6. REMOTE JAVA API

The Remote Java API provides **`KieSession`**, **`TaskService`** and **`AuditLogService`** interfaces to the JMS and REST APIs.

The interface implementations provided by the Remote Java API take care of the underlying logic needed to communicate with the JMS or REST APIs. In other words, these implementations will allow you to interact with a remote workbench instance (i.e. KIE workbench or the jBPM Console) via known interfaces such as the **`KieSession`** or **`TaskService`** interface, without having to deal with the underlying transport and serialization details.

IMPORTANT

While the **KieSession**, **TaskSerivce** and **AuditLogService** instances provided by the Remote Java API may "look" and "feel" like local instances of the same interfaces, please make sure to remember that these instances are only wrappers around a REST or JMS client that interacts with a remote REST or JMS API.

This means that if a requested operation fails on the *server*, the Remote Java API client instance on the *client* side will throw a **RuntimeException** indicating that the REST call failed. This is different from the behaviour of a "real" (or local) instance of a **KieSession**, **TaskSerivce** and **AuditLogService** instance because the exception the local instances will throw will relate to how the operation failed. Also, while local instances require different handling (such as having to dispose of a **KieSession**), client instances provided by the Remote Java API hold no state and thus do not require any special handling.

Lastly, operations on a Remote Java API client instance that would normally throw other exceptions (such as the **TaskService.claim(taskId, userId)** operation when called by a user who is not a potential owner), will now throw a **RuntimeException** instead when the requested operation fails on the *server*.

The first step in interacting with the remote runtime is to create either the **RemoteRestRuntimeEngineFactory** or **RemoteJmsRuntimeEngineFactory**, both of which are instances of the **RemoteRuntimeEngineFactory** interface.

The configuration for the Remote Java API is done when creating the **RemoteRuntimeEngineFactory** instance: there are a number of different constructors for both the JMS and REST implementations that allow the configuration of such things as the base URL of the REST API, JMS queue location or timeout while waiting for responses.

Once the factory instances have been created, there are a couple of methods that can then be used to instantiate the client instance that you want to use:

Remote Java API Methods

RemoteRuntimeEngine RemoteRuntimeEngineFactory.newRuntimeEngine()

This method instantiates a new **RemoteRuntimeEngine** (client) instance.

KieSession RemoteRuntimeEngine.getKieSession()

This method instantiates a new (client) **KieSession** instance.

TaskService RemoteRuntimeEngine.getTaskService()

This method instantiates a new (client) **TaskService** instance.

AuditLogService RemoteRuntimeEngine.getAuditLogService()

This method instantiates a new (client) **AuditLogService** instance.

NOTE

`RemoteRuntimeEngineFactory.addExtraJaxbClasses(Collection<Class<?>> extraJaxbClasses );` method can only be called on builder now. This method adds extra classes to the classpath available to the serialization mechanisms. When passing instances of user-defined classes in a Remote Java API call, it's important to have added the classes via this method first so that the class instances can be serialized correctly.

Report a bug

13.6.1. The REST Remote Java RuntimeEngine Factory

The **`RemoteRestRuntimeEngineFactory`** class is the starting point for building and configuring a new **`RuntimeEngine`** instance that can interact with the remote API. The main use for this class is to create builder instances of REST using the **`newBuilder()`** method. These builder instances are then used to either directly create a **`RuntimeEngine`** instance that will act as a client to the remote REST API or to create an instance of this factory. Illustrated in the table below are the various methods available in the **`RemoteRestRuntimeEngineBuilder`** class:

Table 13.7. RemoteRestRuntimeEngineBuilder Methods

Method Name	Parameter Type	Description
`addDeploymentId`	**`java.lang.String`**	This is the name (id) of the deployment the **`RuntimeEngine`** should interact with.
`addUrl`	**`java.net.URL`**	This is the URL of the deployed business-central or BPMS instance. For example: **`http://localhost:8080/business-central/`**
`addUserName`	**`java.lang.String`**	This is the user name needed to access the REST API.
`addPassword`	**`java.lang.String`**	This is the password needed to access the REST API.
`addProcessInstanceId`	**`long`**	This is the name (id) of the process the **`RuntimeEngine`** should interact with.
`addTimeout`	**`int`**	This maximum number of seconds to wait when waiting for a response from the server.
`addExtraJaxbClasses`	**`class`**	This adds extra classes to the classpath available to the serialization mechanisms.

Example usage

The following example illustrates how the Remote Java API can be used with the REST API.

```
import org.kie.api.runtime.KieSession;
import org.kie.api.task.TaskService;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.services.client.api.RemoteRestRuntimeEngineFactory;
import
org.kie.services.client.api.RemoteRestRuntimeEngineFactoryBuilderImpl;
import org.kie.services.client.api.command.RemoteRuntimeEngine;


public void javaRemoteApiRestExample(String deploymentId, URL baseUrl,
String user, String password) {
  // The serverRestUrl should contain a URL similar to
"http://localhost:8080/business-central/"

  RemoteRestRuntimeEngineFactory remoteRestRuntimeEngineFactory =
RemoteRestRuntimeEngineFactory.newBuilder()
   .addDeploymentId(deploymentId)
  .addUrl(url)
  .addUserName(userName)
  .addPassword(passWord)
  .addTimeout(timeOut)
  .build();

 RemoteRuntimeEngine engine =
remoteRestRuntimeEngineFactory.newRuntimeEngine();

  // Create KieSession and TaskService instances and use them
   KieSession ksession = engine.getKieSession();
   TaskService taskService = engine.getTaskService();

  // Each opertion on a KieSession, TaskService or AuditLogService
(client) instance
  // sends a request for the operation to the server side and waits for
the response
  // If something goes wrong on the server side, the client will throw an
exception.
   ProcessInstance processInstance
       = ksession.startProcess("com.burns.reactor.maintenance.cycle");
   long procId = processInstance.getId();

   String taskUserId = user;
   taskService = engine.getTaskService();
   List<TaskSummary> tasks =
taskService.getTasksAssignedAsPotentialOwner(user, "en-UK");

   long taskId = -1;
   for (TaskSummary task : tasks) {
        if (task.getProcessInstanceId() == procId) {
          taskId = task.getId();
        }
    }

    if (taskId == -1) {
```

```
      throw new IllegalStateException("Unable to find task for " + user +
" in process instance " + procId);
    }

    taskService.start(taskId, taskUserId);
  }
}
```

Report a bug

13.6.2. Custom Model Objects and Remote API

Usage of custom model objects from a client application using the Remote API is supported in BPMS. Custom model objects are the model objects that you create using the Data Modeler within Business Central. Once built and deployed successfully into a project, these objects are part of the project in the local Maven repository.

NOTE

It is recommended that the model objects are reused instead of being recreated locally in the client application.

The process to access and manipulate these objects from the client application is detailed here:

Procedure 13.1. Accessing custom model objects using the Remote API

1. Make sure that the custom model objects have been installed into the local Maven repository of the project that they are a part of (by a process of building the project successfully).
2. If your client application is a Maven based project include the custom model objects project as a Maven dependency in the **pom.xml** configuration file of the client application.

   ```
   <dependency>
         <groupId>${groupid}</groupId>
         <artifactId>${artifactid}</artifactId>
         <version>${version}</version>
   </dependency>
   ```

   The value of these fields can be found in your Project Editor within Business Central: **Authoring → Project Authoring** on the main menu and then **Tools → Project Editor** from the perspective menu.

   - If the client application is NOT a Maven based project download the BPMS project, which includes the model classes, from Business Central by clicking on **Authoring → Artifact Repository**. Add this jar file of the project on the build path of your client application so that the model object classes can be found and used.
3. You can now use the custom model objects within your client application and invoke methods on them using the Remote API. The following listing shows an example of this, where **Person** is a custom model object.

   ```
   import org.jbpm.services.task.utils.ContentMarshallerHelper;
   import org.kie.api.runtime.KieSession;
   import org.kie.api.runtime.process.ProcessInstance;
   ```

```
import org.kie.api.task.TaskService;
import org.kie.api.task.model.Content;
import org.kie.api.task.model.Task;
import org.kie.services.client.api.RemoteRestRuntimeEngineFactory;
import org.kie.services.client.api.command.RemoteRuntimeEngine;

// the rest of the code here
.
.
.
// the following code in a method
RemoteRestRuntimeEngineFactory factory =
RemoteRestRuntimeEngineFactory.newBuilder().addUrl(url).addUserName(
username).addPassword(password).addDeploymentId(deploymentId).addExt
raJaxbClasses(new Class[]{UserDefinedClass.class,
AnotherUserDefinedClass.class}).build();
        runtimeEngine = factory.newRuntimeEngine();
        ksession = runtimeEngine.getKieSession();

        Map<String, Object> params = new HashMap<String, Object>();
        Person person = new Person();
        person.setName("anton");
        params.put("pVar", person);
    ProcessInstance pi = kieSession.startProcess(PROCESS_ID,
params);
```

Make sure that your client application has imported the correct BPMS libraries for the example to work.

Report a bug

13.6.3. The JMS Remote Java RuntimeEngine Factory

The **`RemoteJmsRuntimeEngineFactory`** works similar to the REST variation in that it is a starting point for building and configuring a new **`RuntimeEngine`** instance that can interact with the remote API. The main use for this class is to create builder instances of JMS using the **`newBuilder()`** method. These builder instances are then used to either directly create a **`RuntimeEngine`** instance that will act as a client to the remote JMS API or to create an instance of this factory. Illustrated in the table below are the various methods available for the **`RemoteJmsRuntimeEngineFactoryBuilder`**:

Table 13.8. RemoteJmsRuntimeEngineFactoryBuilder Methods

Method Name	Parameter Type	Description
`addDeploymentId`	**`java.lang.String`**	This is the name (id) of the deployment the **`RuntimeEngine`** should interact with.
`addProcessInstanceId`	**`long`**	This is the name (id) of the process the **`RuntimeEngine`** should interact with.

Method Name	Parameter Type	Description
addUserName	**java.lang.String**	This is the user name needed to access the JMS queues (in your application server configuration).
addPassword	**java.lang.String**	This is the password needed to access the JMS queues (in your application server configuration).
addTimeout	**int**	This maximum number of seconds to wait when waiting for a response from the server.
addExtraJaxbClasses	**class**	This adds extra classes to the classpath available to the serialization mechanisms.
addRemoteInitialContext	**javax.jms.InitialContext**	This is a remote InitialContext instance (created using JNDI) from the server.
addConnectionFactory	**javax.jms.ConnectionFactory**	This is a **ConnectionFactory** instance used to connect to the **ksessionQueue** or **taskQueue**.
addKieSessionQueue	**javax.jms.Queue**	This is an instance of the **Queue** for requests relating to the process instance.
addTaskServiceQueue	**javax.jms.Queue**	This is an instance of the **Queue** for requests relating to task service usage.
addResponseQueue	**javax.jms.Queue**	This is an instance of the **Queue** used to receive responses.
addJbossServerUrl	**java.net.URL**	This is the url for the JBoss Server.
addJbossServerHostName	**java.lang.String**	This is the hostname for the JBoss Server.
addHostName	**java.lang.String**	This is the hostname of the JMS queues.
addJmsConnectorPort	**int**	This is the port for the JMS Connector.
addKeystorePassword	**java.lang.String**	This is the JMS Keystore Password.

Method Name	Parameter Type	Description
addKeystoreLocation	**java.lang.String**	This is the JMS Keystore Location.
addTruststorePassword	**java.lang.String**	This is the JMS Truststore Password.
addTruststoreLocation	**java.lang.String**	This is the JMS Truststore Location.

Example Usage

The following example illustrates how the Remote Java API can be used with the JMS API.

```
import org.kie.api.runtime.KieSession;
import org.kie.api.task.TaskService;
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.services.client.api.RemoteJmsRuntimeEngineFactory;
import org.kie.services.client.api.command.RemoteRuntimeEngine;


public void javaRemoteApiJmsExample(String deploymentId, Long
processInstanceId, String user, String password) {

  // create a factory class with all the values
  RemoteJmsRuntimeEngineFactory jmsRuntimeFactory =
          RemoteJmsRuntimeEngineFactory.newBuilder()
          .addDeploymentId(deploymentId)
          .addProcessInstanceId(processInstanceId)
          .addUserName(user)
          .addPassword(password)
          .addRemoteInitialContext(remoteInitialContext)
          .addTimeout(3)
          .addExtraJaxbClasses(MyType.class)
          .useSsl(false)
          .build();

 RemoteRuntimeEngine engine = jmsRuntimeFactory.newRuntimeEngine();

  // Create KieSession and TaskService instances and use them
   KieSession ksession = engine.getKieSession();
   TaskService taskService = engine.getTaskService();

  // Each opertion on a KieSession, TaskService or AuditLogService
(client) instance
  // sends a request for the operation to the server side and waits for
the response
  // If something goes wrong on the server side, the client will throw an
exception.
   ProcessInstance processInstance
       = ksession.startProcess("com.burns.reactor.maintenance.cycle");
   long procId = processInstance.getId();

   String taskUserId = user;
```

```
   taskService = engine.getTaskService();
   List<TaskSummary> tasks =
taskService.getTasksAssignedAsPotentialOwner(user, "en-UK");

   long taskId = -1;
   for (TaskSummary task : tasks) {
        if (task.getProcessInstanceId() == procId) {
          taskId = task.getId();
        }
    }

    if (taskId == -1) {
      throw new IllegalStateException("Unable to find task for " + user +
" in process instance " + procId);
    }

    taskService.start(taskId, taskUserId);
  }
}
```

Sending and receiving JMS messages

The **sendAndReceiveJmsMessage** example below creates the **JaxbCommandsRequest** instance and adds commands from the user. In addition, it retrieves JNDI context from the server, creates a JMS connection, etc.

```
import org.kie.api.runtime.process.ProcessInstance;
import org.kie.api.task.model.TaskSummary;

public void sendAndReceiveJmsMessage() {

    String USER = "charlie";
    String PASSWORD = "ch0c0licious";

    String DEPLOYMENT_ID = "test-project";
    String PROCESS_ID_1 = "oompa-processing";
    URL serverUrl;
    try {
        serverUrl = new URL("http://localhost:8080/business-central/");
    } catch (MalformedURLException murle) {
        logger.error("Malformed URL for the server instance!", murle);
        return;
    }

// Create JaxbCommandsRequest instance and add commands
    Command<?> cmd = new StartProcessCommand(PROCESS_ID_1);
    int oompaProcessingResultIndex = 0;
    JaxbCommandsRequest req = new JaxbCommandsRequest(DEPLOYMENT_ID, cmd);
    req.getCommands().add(new
GetTaskAssignedAsPotentialOwnerCommand(USER));
    int loompaMonitoringResultIndex = 1;

// Get JNDI context from server
    InitialContext context = getRemoteJbossInitialContext(serverUrl, USER,
PASSWORD);

// Create JMS connection
```

```
    ConnectionFactory connectionFactory;
    try {
        connectionFactory = (ConnectionFactory)
context.lookup("jms/RemoteConnectionFactory");
    } catch (NamingException ne) {
        throw new RuntimeException("Unable to lookup JMS connection
factory.", ne);
    }

// Setup queues
    Queue sendQueue, responseQueue;
    try {
        sendQueue = (Queue) context.lookup("jms/queue/KIE.SESSION");
        responseQueue = (Queue) context.lookup("jms/queue/KIE.RESPONSE");
    } catch (NamingException ne) {
        throw new RuntimeException("Unable to lookup send or response
queue", ne);
    }

// Send command request
    Long processInstanceId = null; // needed if you're doing an operation
on a PER_PROCESS_INSTANCE deployment
    String humanTaskUser = USER;
    JaxbCommandsResponse cmdResponse = sendJmsCommands(
          DEPLOYMENT_ID, processInstanceId, humanTaskUser, req,
          connectionFactory, sendQueue, responseQueue,
          USER, PASSWORD, 5);

// Retrieve results
    ProcessInstance oompaProcInst = null;
    List<TaskSummary> charliesTasks = null;
    for (JaxbCommandResponse<?> response : cmdResponse.getResponses()) {
        if (response instanceof JaxbExceptionResponse) {

          // something went wrong on the server side
          JaxbExceptionResponse exceptionResponse =
(JaxbExceptionResponse) response;
          throw new RuntimeException(exceptionResponse.getMessage());
        }
        if (response.getIndex() == oompaProcessingResultIndex) {
          oompaProcInst = (ProcessInstance) response.getResult();
        } else if (response.getIndex() == loompaMonitoringResultIndex) {
          charliesTasks = (List<TaskSummary>) response.getResult();
        }
    }
}
```

Sending JMS commands

The **sendJmsCommands** example below is a continuation of the previous example. It sets up user created classes and sends, receives, and extracts responses.

```
private JaxbCommandsResponse sendJmsCommands(String deploymentId, Long
processInstanceId, String user,
  JaxbCommandsRequest req, ConnectionFactory factory, Queue sendQueue,
Queue responseQueue, String jmsUser,
  String jmsPassword, int timeout) {
```

```
    req.setProcessInstanceId(processInstanceId);
    req.setUser(user);

    Connection connection = null;
    Session session = null;
    String corrId = UUID.randomUUID().toString();
    String selector = "JMSCorrelationID = '" + corrId + "'";
    JaxbCommandsResponse cmdResponses = null;
    try {

        // setup
        MessageProducer producer;
        MessageConsumer consumer;
        try {
          if (jmsPassword != null) {
            connection = factory.createConnection(jmsUser, jmsPassword);
          } else {
            connection = factory.createConnection();
          }
          session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

          producer = session.createProducer(sendQueue);
          consumer = session.createConsumer(responseQueue, selector);

          connection.start();
        } catch (JMSException jmse) {
            throw new RemoteCommunicationException("Unable to setup a JMS
connection.", jmse);
        }

        JaxbSerializationProvider serializationProvider = new
JaxbSerializationProvider();
        // if necessary, add user-created classes here:
        // xmlSerializer.addJaxbClasses(MyType.class,
AnotherJaxbAnnotatedType.class);

        // Create msg
        BytesMessage msg;
        try {
          msg = session.createBytesMessage();

          // serialize request
          String xmlStr = serializationProvider.serialize(req);
          msg.writeUTF(xmlStr);

          // set properties
         msg.setJMSCorrelationID(corrId);

msg.setIntProperty(SerializationConstants.SERIALIZATION_TYPE_PROPERTY_NAME
, JaxbSerializationProvider.JMS_SERIALIZATION_TYPE);
         Collection<Class<?>> extraJaxbClasses =
serializationProvider.getExtraJaxbClasses();
         if (!extraJaxbClasses.isEmpty()) {
           String extraJaxbClassesPropertyValue =
JaxbSerializationProvider
```

```
            .classSetToCommaSeperatedString(extraJaxbClasses);

msg.setStringProperty(SerializationConstants.EXTRA_JAXB_CLASSES_PROPERTY_N
AME, extraJaxbClassesPropertyValue);

msg.setStringProperty(SerializationConstants.DEPLOYMENT_ID_PROPERTY_NAME,
deploymentId);
          }
        } catch (JMSException jmse) {
          throw new RemoteCommunicationException("Unable to create and fill
a JMS message.", jmse);
        } catch (SerializationException se) {
          throw new RemoteCommunicationException("Unable to deserialze JMS
message.", se.getCause());
        }

        // send
        try {
          producer.send(msg);
        } catch (JMSException jmse) {
          throw new RemoteCommunicationException("Unable to send a JMS
message.", jmse);
        }

        // receive
        Message response;
        try {
          response = consumer.receive(timeout);
        } catch (JMSException jmse) {
          throw new RemoteCommunicationException("Unable to receive or
retrieve the JMS response.", jmse);
        }

        if (response == null) {
          logger.warn("Response is empty, leaving");
          return null;
        }
        // extract response
        assert response != null : "Response is empty.";
        try {
          String xmlStr = ((BytesMessage) response).readUTF();
          cmdResponses = (JaxbCommandsResponse)
serializationProvider.deserialize(xmlStr);
        } catch (JMSException jmse) {
          throw new RemoteCommunicationException("Unable to extract " +
JaxbCommandsResponse.class.getSimpleName()
              + " instance from JMS response.", jmse);
        } catch (SerializationException se) {
          throw new RemoteCommunicationException("Unable to extract " +
JaxbCommandsResponse.class.getSimpleName()
              + " instance from JMS response.", se.getCause());
        }
        assert cmdResponses != null : "Jaxb Cmd Response was null!";
        } finally {
          if (connection != null) {
            try {
```

```
            connection.close();
            if( session != null ) {
              session.close();
            }
          } catch (JMSException jmse) {
            logger.warn("Unable to close connection or session!", jmse);
          }
        }
    }
    return cmdResponses;
}
```

Configuration using an `InitialContext` instance

When configuring the **`RemoteJmsRuntimeEngineFactory`** with an **`InitialContext`** instance as a parameter for Red Hat JBoss EAP 6, it is necessary to retrieve the (remote) **`InitialContext`** instance first from the remote server. The following code illustrates how to do this.

```
private InitialContext getRemoteJbossInitialContext(URL url, String user,
String password) {
    Properties initialProps = new Properties();
    initialProps.setProperty(InitialContext.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
    String jbossServerHostName = url.getHost();
    initialProps.setProperty(InitialContext.PROVIDER_URL, "remote://"+
jbossServerHostName + ":4447");
    initialProps.setProperty(InitialContext.SECURITY_PRINCIPAL, user);
    initialProps.setProperty(InitialContext.SECURITY_CREDENTIALS,
password);

    for (Object keyObj : initialProps.keySet()) {
        String key = (String) keyObj;
        System.setProperty(key, (String) initialProps.get(key));
    }
    try {
        return new InitialContext(initialProps);
    } catch (NamingException e) {
        throw new RemoteCommunicationException("Unable to create " +
InitialContext.class.getSimpleName(), e);
    }
}
```

Report a bug

13.6.4. Supported Methods

As mentioned above, the Remote Java API provides client-like instances of the **`RuntimeEngine`**, **`KieSession`**, **`TaskService`** and **`AuditLogService`** interfaces.

This means that while many of the methods in those interfaces are available, some are not. The following tables lists the methods which are available. Methods not listed in the below table will throw an **`UnsupportedOperationException`** explaining that the called method is not available.

Table 13.9. Available process-related `KieSession` methods

Returns	Method signature	Description
`void`	`abortProcessInstance(long processInstanceId)`	Abort the process instance
`ProcessInstance`	`getProcessInstance(long processInstanceId)`	Return the process instance
`ProcessInstance`	`getProcessInstance(long processInstanceId, boolean readonly)`	Return the process instance
`Collection<ProcessInstance>`	`getProcessInstances()`	Return all (active) process instances
`void`	`signalEvent(String type, Object event)`	Signal all (active) process instances
`void`	`signalEvent(String type, Object event, long processInstanceId)`	Signal the process instance
`ProcessInstance`	`startProcess(String processId)`	Start a new process and return the process instance (if the process instance has not immediately completed)
`ProcessInstance`	`startProcess(String processId, Map<String, Object> parameters);`	Start a new process and return the process instance (if the process instance has not immediately completed)

Table 13.10. Available rules-related `KieSession` methods

Returns	Method signature	Description
`Long`	`getFactCount()`	Return the total fact count
`Object`	`getGlobal(String identifier)`	Return a global fact
`void`	`setGlobal(String identifier, Object value)`	Set a global fact

Table 13.11. Available `WorkItemManager` methods

Returns	Method signature	Description
`void`	`abortWorkItem(long id)`	Abort the work item

Returns	Method signature	Description
`void`	`completeWorkItem(long id, Map<String, Object> results)`	Complete the work item
`void`	`registerWorkItemHandler(String workItemName, WorkItemHandler handler)`	Register the work items
`WorkItem`	`getWorkItem(long workItemId)`	Return the work item

Table 13.12. Available task operation `TaskService` methods

Returns	Method signature	Description
`Long`	`addTask(Task task, Map<String, Object> params)`	Add a new task
`void`	`activate(long taskId, String userId)`	Activate a task
`void`	`claim(long taskId, String userId)`	Claim a task
`void`	`claimNextAvailable(String userId, String language)`	Claim the next available task for a user
`void`	`complete(long taskId, String userId, Map<String, Object> data)`	Complete a task
`void`	`delegate(long taskId, String userId, String targetUserId)`	Delegate a task
`void`	`exit(long taskId, String userId)`	Exit a task
`void`	`fail(long taskId, String userId, Map<String, Object> faultData)`	Fail a task

Returns	Method signature	Description
`void`	`forward(long taskId, String userId, String targetEntityId)`	Forward a task
`void`	`nominate(long taskId, String userId, List<OrganizationalEntity> potentialOwners)`	Nominate a task
`void`	`release(long taskId, String userId)`	Release a task
`void`	`resume(long taskId, String userId)`	Resume a task
`void`	`skip(long taskId, String userId)`	Skip a task
`void`	`start(long taskId, String userId)`	Start a task
`void`	`stop(long taskId, String userId)`	Stop a task
`void`	`suspend(long taskId, String userId)`	Suspend a task

Table 13.13. Available task retrieval and query `TaskService` methods

Returns	Method signature
`Task`	`getTaskByWorkItemId(long workItemId)`
`Task`	`getTaskById(long taskId)`
`List<TaskSummary>`	`getTasksAssignedAsBusinessAdministrator(String userId, String language)`
`List<TaskSummary>`	`getTasksAssignedAsPotentialOwner(String userId, String language)`
`List<TaskSummary>`	`getTasksAssignedAsPotentialOwnerByStatus(String userId, List<Status>gt; status, String language)`
`List<TaskSummary>`	`getTasksOwned(String userId, String language)`

Returns	Method signature
`List<TaskSummary>`	`getTasksOwnedByStatus(String userId, List<Status> status, String language)`
`List<TaskSummary>`	`getTasksByStatusByProcessInstanceId(long processInstanceId, List<Status> status, String language)`
`List<TaskSummary>`	`getTasksByProcessInstanceId(long processInstanceId)`
`Content`	`getContentById(long contentId)`
`Attachment`	`getAttachmentById(long attachId)`

Table 13.14. Available `AuditLogService` methods

Returns	Method signature
`List<ProcessInstanceLog>`	`findProcessInstances()`
`List<ProcessInstanceLog>`	`findProcessInstances(String processId)`
`List<ProcessInstanceLog>`	`findActiveProcessInstances(String processId)`
`ProcessInstanceLog`	`findProcessInstance(long processInstanceId)`
`List<ProcessInstanceLog>`	`findSubProcessInstances(long processInstanceId)`
`List<NodeInstanceLog>`	`findNodeInstances(long processInstanceId)`
`List<NodeInstanceLog>`	`findNodeInstances(long processInstanceId, String nodeId)`
`List<VariableInstanceLog>`	`findVariableInstances(long processInstanceId)`
`List<VariableInstanceLog>`	`findVariableInstances(long processInstanceId, String variableId)`

Returns	Method signature
`List<VariableInstanceLog>`	`findVariableInstancesByName(String variableId, boolean onlyActiveProcesses)`
`List<VariableInstanceLog>`	`findVariableInstancesByNameAndValue(String variableId, String value, boolean onlyActiveProcesses)`
`void`	`clear()`

Report a bug

APPENDIX A. JARS AND LIBRARIES INCLUDED IN RED HAT JBOSS BPM SUITE

The following is a comprehensive list of available JARs included in Red Hat JBoss BPM Suite 6.0.3.

Table A.1. Drools JARs

JAR Name	Description
org.drools:drools-compiler:jar:6.0.3-redhat-6	The Drools compiler jar provides facilities to compile various rule representations (DRL, DSL, etc.) into a corresponding internal structure. In addition, it provides the CDI extension allowing usage of @KSession and other annotations. It is the main artifact used when embedding Drools engine.
org.drools:drools-persistence-jpa:jar:6.0.3-redhat-6	Provides Drools with the ability to persist a KieSession into a DB using Java Persistence API (JPA).
org.drools:drools-core:jar:6.0.3-redhat-6	The core Drools engine jar, contains classes required for runtime execution of rules.
org.drools:drools-decisiontables:jar:6.0.3-redhat-6	Includes classes which are responsible for parsing and compiling Decision Table into the plain DRL. This jar has to be included if you are building a kjar with decision tables present in the resources.
org.drools:drools-verifier:jar:6.0.3-redhat-6	Drools Verifier analyses the quality of Drools rules and reports any issues. Internally used by business-central.
org.drools:drools-templates:jar:6.0.3-redhat-6	Includes classes which are responsible for parsing and compiling provided Data Provider and Rule Template into a final DRL. This Library is required for using decision table/score card.

Table A.2. jBPM Libraries

Library Name	Description
org.jbpm:jbpm-audit:jar:6.0.3-redhat-6	This library audits history information regarding processes into the database. Among others, it includes entity classes - i.e. ProcessInstanceLog which can be found in the database, specific AuditService implementation - JPAAuditLogService - which allows user to query history tables and also Database Logger which stores audit related information into database.

Library Name	Description
org.jbpm:jbpm-bpmn2:jar:6.0.3-redhat-6	Internal representation of BPMN elements, including BPMN validator and parser. It also includes few basic WorkItemHandlers. When building a kjar which includes BPMN processes this library has to be explicitly added on the classpath. The process is understood as a static model.
org.jbpm:jbpm-executor:jar:6.0.3-redhat-6	Executor service which can be used for Asynchronous Task execution. This library is mandatory if you use Asynchronous Task execution in your project.
org.jbpm:jbpm-flow-builder:jar:6.0.3-redhat-6	Compiler of BPMN processes. When building by kie-maven-plugin, the jbpm-bpmn2 project already brings this dependency on classpath.
org.jbpm:jbpm-flow:jar:6.0.3-redhat-6	Internal representation of instantiated process / workflow / ruleflow. This is actually the core engine library that performs the workflow execution.
org.jbpm:jbpm-human-task-audit:jar:6.0.3-redhat-6	Library which audits Task related events - start, claim, complete, etc. into the database (table TaskEvent).
org.jbpm:jbpm-human-task-core:jar:6.0.3-redhat-6	Includes core Human Task Services. API, its implementation, listeners, persistence, commands and more.
org.jbpm:jbpm-human-task-workitems:jar:6.0.3-redhat-6	Implementation of Human Task Work Item handler with necessary utility and helper classes.
org.jbpm:jbpm-kie-services:jar:6.0.3-redhat-6	core implementation of services that encapsulate core engine, task service and Runtime manager APIs into service oriented components to easier pluggability into custom system. Base of execution server brought by kie-wb/business central.
org.jbpm:jbpm-persistence-jpa:jar:6.0.3-redhat-6	Provides classes which store process runtime information into the database using JPA.
org.jbpm:jbpm-runtime-manager:jar:6.0.3-redhat-6	Provides Runtime Manager API which allows developer to interact with processes and tasks.
org.jbpm:jbpm-shared-services:jar:6.0.3-redhat-6	Part of services subsystem that simplifies interaction with process engine within a dynamic environment. This library includes some classes which can be used for interactions with Business Central asynchronously. Please add this library if you use jbpm-executor.

Library Name	Description
org.jbpm:jbpm-test:jar:6.0.3-redhat-6	jBPM Test Framework for unit testing of processes. Please add this jar if you implement your test code using jBPM Test Framework.
org.jbpm:jbpm-workitems:jar:6.0.3-redhat-6	Includes every WorkItemHandlers provided by jBPM - i.e. RestWorkItemHandler, WebServiceWorkItemHandler, etc.Some of them are supported also in JBoss BPM Suite (those which has WorkItems visible by default in the Designer palette) and some are not.

Table A.3. KIE Libraries

Library Name	Description
org.kie:kie-api:jar:6.0.3-redhat-6	The Drools and jBPM public API which is backwards compatible between releases.
org.kie:kie-ci:jar:6.0.3-redhat-6	Allows loading a KIE module for further usage - i.e. KieBase / KieSession can be created from Kie Module. KIE module in its essence is a standard jar project built by Maven, so kie-ci library embeds maven in order to load KIE Module. This library has to be included if you use kjar's functionality.
org.kie:kie-internal:jar:6.0.3-redhat-6	The Drools and jBPM internal API which *might not be* backwards compatible between releases. Any usage of classes located in this library should be consulted with Red Hat in order to determine whether the usage is supported or not.
org.kie:kie-spring:jar:6.0.3-redhat-6	This library has to be included if you want to use jBPM/Spring integration capabilities.
org.kie.remote:kie-services-client:jar:6.0.3-redhat-6	Native Java Remote Client for remote interaction with busines-central server (REST, JMS). This library has to be included if you use Native Java Remote Client for remote interaction with Business Central server (REST, JMS).
org.kie.remote:kie-services-jaxb:jar:6.0.3-redhat-6	JAXB version of various entity classes. This library is required for remote interactions with the Business Central server.
org.kie.remote:kie-services-remote:jar:6.0.3-redhat-6	Server side implementation of the REST API. This library is required for remote interaction with busines-central server.

Report a bug

APPENDIX B. REVISION HISTORY

Revision 1.0.0-40 **Thu Jul 23 2015** **Vidya Iyengar**

Built from Content Specification: 22684, Revision: 765778 by viyengar