Red Hat JBoss BPM Suite 6.1

Development Guide

For Red Hat JBoss Developers

Last Updated: 2017-11-13

Red Hat JBoss BPM Suite 6.1 Development Guide

For Red Hat JBoss Developers

Kanchan Desai
kadesai@redhat.com

Doug Hoffman

Eva Kopalova

Petr Penicka

Red Hat Content Services

Legal Notice

Copyright © 2015 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

A guide to using API's in Red Hat JBoss BPM Suite for Developers.

Table of Contents

PART I. OVERVIEW 5

CHAPTER 1. ABOUT THIS GUIDE 6
1.1. AUDIENCE 6
1.2. PREREQUISITES 7

CHAPTER 2. JBOSS BRMS AND JBOSS BPM SUITE ARCHITECTURE 8
2.1. JBOSS BUSINESS RULES MANAGEMENT SYSTEM 8
2.2. JBOSS BUSINESS PROCESS MANAGEMENT SUITE 9
2.3. SUPPORTED PLATFORMS 11
2.4. USE CASES 11

CHAPTER 3. MAVEN DEPENDENCIES 14
3.1. MAVEN REPOSITORIES 14
3.2. USING MAVEN REPOSITORY IN YOUR PROJECT 14
3.3. MAVEN CONFIGURATION FILE 14
3.4. MAVEN SETTINGS FILE 15
3.5. DEPENDENCY MANAGEMENT 15
3.6. INTEGRATED MAVEN DEPENDENCIES 16
3.7. UPLOADING ARTIFACTS TO MAVEN REPOSITORY 16
3.8. DEPLOYING RED HAT JBOSS BPM SUITE ARTIFACTS TO RED HAT JBOSS FUSE 19

CHAPTER 4. INSTALL AND SETUP JBOSS DEVELOPER STUDIO 21
4.1. INSTALLING THE JBOSS DEVELOPER STUDIO PLUG-INS 21
4.2. CONFIGURING THE JBOSS BRMS/BPM SUITE SERVER 21
4.3. IMPORTING PROJECTS FROM A GIT REPOSITORY INTO JBOSS DEVELOPER STUDIO 22

PART II. ALL ABOUT RULES 26

CHAPTER 5. RULE ALGORITHMS 27
5.1. PHREAK ALGORITHM 27
5.2. RULE EVALUATION WITH PHREAK ALGORITHM 27
5.3. RETE ALGORITHM 28
5.4. SWITCHING BETWEEN PHREAK AND RETEOO 32

CHAPTER 6. GETTING STARTED WITH RULES AND FACTS 33
6.1. CREATE YOUR FIRST RULE 33
6.2. EXECUTION OF RULES 48
6.3. INFERENCE 50
6.4. TRUTH MAINTENANCE 50
6.5. USING DECISION TABLES IN SPREADSHEETS 55
6.6. LOGGING 65

CHAPTER 7. COMPLEX EVENT PROCESSING 67
7.1. INTRODUCTION TO COMPLEX EVENT PROCESSING 67
7.2. EVENTS 68
7.3. CLOCK IMPLEMENTATION IN COMPLEX EVENT PROCESSING 72
7.4. EVENT PROCESSING MODES 73
7.5. EVENT STREAMS 75
7.6. TEMPORAL OPERATIONS 78
7.7. SLIDING WINDOWS 87
7.8. MEMORY MANAGEMENT FOR EVENTS 89

CHAPTER 8. WORKING WITH RULES 91

8.1. WHAT'S IN A RULE FILE 91
8.2. HOW RULES OPERATE ON FACTS 91
8.3. USING RULE KEYWORDS 92
8.4. ADDING COMMENTS TO A RULE FILE 98
8.5. ERROR MESSAGES IN RULES 98
8.6. PACKAGING 101
8.7. FUNCTIONS IN A RULE 103
8.8. BACKWARD-CHAINING 108
8.9. TYPE DECLARATION 116
8.10. RULE ATTRIBUTES 121
8.11. PATTERNS 125
8.12. ELEMENTS AND VARIABLES 126
8.13. SEARCHING THE WORKING MEMORY USING QUERY 153
8.14. DOMAIN SPECIFIC LANGUAGES (DSLS) 154

CHAPTER 9. USING JBOSS DEVELOPER STUDIO TO CREATE AND TEST RULES 165
9.1. JBOSS DEVELOPER STUDIO DROOLS PERSPECTIVE 165
9.2. JBOSS BRMS RUNTIMES 165
9.3. EXPLORING A JBOSS BRMS APPLICATION 168
9.4. CREATING A JBOSS BRMS PROJECT 168
9.5. USING TEXTUAL RULE EDITOR 169
9.6. RED HAT JBOSS BRMS VIEWS 169
9.7. DEBUGGING RULES 170

PART III. ALL ABOUT PROCESSES 172

CHAPTER 10. GETTING STARTED WITH PROCESSES 173
10.1. THE JBOSS BPM SUITE ENGINE 173
10.2. INTEGRATING BPM SUITE ENGINE WITH OTHER SERVICES 173

CHAPTER 11. WORKING WITH PROCESSES 175
11.1. BPMN 2.0 NOTATION 175
11.2. WHAT COMPRISES A BUSINESS PROCESS 183
11.3. ACTIVITIES 187
11.4. DATA 194
11.5. EVENTS 195
11.6. GATEWAYS 201
11.7. VARIABLES 203
11.8. ASSIGNMENT 204
11.9. ACTION SCRIPTS 204
11.10. CONSTRAINTS 205
11.11. TIMERS 206
11.12. MULTI-THREADING 206
11.13. PROCESS FLUENT API 207
11.14. TESTING BUSINESS PROCESSES 209

CHAPTER 12. HUMAN TASKS MANAGEMENT 217
12.1. HUMAN TASKS 217
12.2. USING USER TASKS IN PROCESSES 217
12.3. DATA MAPPING 218
12.4. TASK LIFECYCLE 218
12.5. TASK PERMISSIONS 220
12.6. TASK PERMISSIONS 221

CHAPTER 13. PERSISTENCE AND TRANSACTIONS 225

13.1. PROCESS INSTANCE STATE 225
13.2. AUDIT LOG 228
13.3. TRANSACTIONS 230
13.4. CONFIGURATION 232

CHAPTER 14. USING JBOSS DEVELOPER STUDIO TO CREATE AND TEST PROCESSES 236
14.1. JBOSS BPM SUITE RUNTIME 236
14.2. IMPORTING PROJECTS FROM A GIT REPOSITORY INTO JBOSS DEVELOPER STUDIO 237
14.3. EXPLORING A JBOSS BPM SUITE APPLICATION 240
14.4. CREATING A JBOSS BPM SUITE PROJECT 241
14.5. CONVERTING AN EXISTING JAVA PROJECT TO A BPM SUITE PROJECT 241
14.6. CREATING A PROCESS USING BPMN2 PROCESS WIZARD 241
14.7. BUILDING A PROCESS USING BPMN2 PROCESS EDITOR 242
14.8. CREATING A PROCESS USING BPMN MAVEN PROCESS WIZARD 242
14.9. DEBUGGING BUSINESS PROCESSES 244
14.10. SYNCHRONIZING JBOSS DEVELOPER STUDIO WORKSPACE WITH BUSINESS CENTRAL REPOSITORIES 247

PART IV. KIE 250

CHAPTER 15. KIE API 251
15.1. KIE 251
15.2. KIE FRAMEWORK 252
15.3. BUILDING WITH MAVEN 265
15.4. KIE DEPLOYMENT 269
15.5. RUNNING IN KIE 272
15.6. KIE CONFIGURATION 294
15.7. KIE SESSIONS 300
15.8. RUNTIME MANAGER 305

CHAPTER 16. REMOTE API 328
16.1. REST API 328
16.2. JMS 360
16.3. EJB INTERFACE 369
16.4. REMOTE JAVA API 371

CHAPTER 17. CDI INTEGRATION 390
17.1. JBOSS BPM SUITE WITH CDI INTEGRATION 390
17.2. DEPLOYMENT SERVICE 390
17.3. CONFIGURING CDI INTEGRATION 392
17.4. RUNTIMEMANAGER AS CDI BEAN 397

CHAPTER 18. SOAP INTERFACE 401
18.1. SOAP API 401
18.2. CLIENT-SIDE JAVA WEBSERVICE CLIENT 401

APPENDIX A. REVISION HISTORY 403

PART I. OVERVIEW

CHAPTER 1. ABOUT THIS GUIDE

This guide is intended for users who are implementing a standalone JBoss BRMS solution or the complete JBoss BPM Suite solution. It discusses the following topics:

- Detailed Architecture of JBoss BRMS and JBoss BPM Suite.
- Detailed description of how to author, test, debug, and package simple and complex business rules and processes using Integrated Development environment (IDE).
- JBoss BRMS runtime environment.
- Domain specific languages (DSLs) and how to use them in a rule.
- Complex event processing.

This guide comprises the following sections:

1. Overview

   This section provides detailed information on JBoss BRMS and JBoss BPM suite, their architecture, key components. It also discusses the role of Maven in project building and deploying.

2. All About Rules

   This section provides details on all you have to know to author rules with JBoss Developer Studio. It describes the rule algorithms, rule structure, components, advanced conditions, constraints, commands, Domain Specific Languages and Complex Event Processing. It provides details on how to use the various views, editors, and perspectives that JBoss Developer Studio offers.

3. All About Processes

   This section describes what comprises a business process and how you can author and test them using JBoss Developer Studio.

4. KIE

   This section highlights the KIE API with detailed description of how to create, build, deploy, and run KIE projects.

5. Appendix

   This section comprises important reference material such as key knowledge terms, and examples.

1.1. AUDIENCE

This book has been designed to be understood by:

- Author of rules and processes who are responsible for authoring and testing business rules and processes using JBoss Developer Studio.
- Java application developers responsible for developing and integrating business rules and processes into Java and Java EE enterprise applications.

1.2. PREREQUISITES

Users of this guide must meet one or more of the following prerequisites:

- Basic Java/Java EE programming experience
- Knowledge of the Eclipse IDE, Maven and GIT

CHAPTER 2. JBOSS BRMS AND JBOSS BPM SUITE ARCHITECTURE

2.1. JBOSS BUSINESS RULES MANAGEMENT SYSTEM

Red Hat JBoss BRMS is an open source business rule management system that provides rules development, access, change, and management capabilities. In today's world, when IT organizations consistently face changes in terms of policies, new products, government imposed regulations, a system like JBoss BRMS makes it easy by separating business logic from the underlying code. It includes a rule engine, a rules development environment, a management system, and a repository. It allows both developers and business analysts to view, manage, and verify business rules as they are executed within an IT application infrastructure.

JBoss BRMS can be executed in any Java EE-compliant container. It supports an open choice of authoring and management consoles and language and decision table inputs.

2.1.1. JBoss BRMS Key Components

JBoss BRMS comprises the following components:

- Drools Expert

  Drools Expert is a pattern matching based rule engine that runs on Java EE application servers, JBoss BRMS platform, or bundled with Java applications. It comprises an inference engine, a production memory , and a working memory. Rules are stored in the production memory and the facts that the inference engine matches the rules against, are stored in the working memory.

- Business Central

  Business Central is a web interface intended for business analysts for creation and maintenance of business rules and rule artifacts. It is designed to ease creation, testing, and packaging of rules for business users.

- Drools Flow

  Drools flow provides business process capabilities to the JBoss BRMS platform. This framework can be embedded into any Java application or can even run standalone on a server. A business process provides stepwise tasks using a flow chart, for the Rule Engine to execute.

- Drools Fusion

  Drools Fusion provides event processing capabilities to the JBoss BRMS platform. Drools Fusion defines a set of goals to be achieved such as:

  - Support events as first class citizens.
  - Support detection, correlation, aggregation and composition of events.
  - Support processing streams of events.
  - Support temporal constraints in order to model the temporal relationships between events.

- Drools Integrated Development Environment (IDE)

  We encourage you to use Red Hat JBoss Developer Studio (JBDS) with JBoss BRMS plug-ins

to develop and test business rules. The JBoss Developer Studio builds upon an extensible, open source Java-based IDE Eclipse providing platform and framework capabilities, making it ideal for JBoss BRMS rules development.

2.1.2. JBoss BRMS Features

The JBoss BRMS provides the following key features:

- Centralized repository of business assets (JBoss BRMS artifacts)
- IDE tools to define and govern decision logic
- Building, deploying, and testing the decision logic
- Packages of business assets
- Categorization of business assets
- Integration with development tools
- Business logic and data separation
- Business logic open to reuse and changes
- Easy to maintain business logic
- Enables several stakeholders (business analysts, developer, administrators) to contribute in defining the business logic

2.2. JBOSS BUSINESS PROCESS MANAGEMENT SUITE

Red Hat JBoss BPM Suite is an open source business process management system that combines business process management and business rules management. JBoss BRMS offers tools to author rules and business processes, but does not provide tools to start or manage the business processes. Red Hat JBoss BPM Suite includes all the JBoss BRMS functionalities, with additional capabilities of business activity monitoring, starting business processes, and managing tasks using Business Central. JBoss BPM Suite also provides a central repository to store rules and processes.

2.2.1. JBoss BPM Suite Key Components

The Red Hat JBoss BPM Suite comprises the following components:

- JBoss BPM Central (Business Central)

  Business Central is a web-based application for creating, editing, building, managing, and monitoring JBoss BPM Suite business assets. It also allows execution of business processes and management of tasks created by those processes.

- Business Activity Monitoring Dashboards

  The Business Activity Monitor (BAM) dashboard provides report generation capabilities. It allows you to use a pre-difined dashboard and even create your own customized dashboard.

- Maven Artifact Repository

JBoss BPM Suite projects are built as Apache Maven projects and the default location of the Maven repository is **`<working-directory>/repositories/kie`**. You can specify an alternate repository location by changing the org.guvnor.m2repo.dir property.

Each project builds a JAR artifact file called a **`kjar`**. You can store your project artifacts and dependent jars in this repository.

- Execution Engine

  The JBoss BPM Suite execution engine is responsible for executing business processes and managing the tasks, which result from these processes. Business Central provides a user interface for executing processes and managing tasks.

  NOTE

  To execute your business processes, you can use Business Central web application that bundles the execution engine, enabling a ready to use process execution environment. Alternatively, you can create your own execution server and embed the JBoss BPM Suite and JBoss BRMS libraries with your application using the standard Java EE way.

  For example, if you are developing a web application, include the JBoss BPM Suite/BRMS libraries in the **`WEB-INF/lib`** folder of your application.

- Business Central Repository

  The business artifacts of a JBoss BPM Suite project such as process models, rules, and forms are stored in Git repositories managed through the Business Central. You can also access these repositories outside of Business Central through the git or ssh protocols.

2.2.2. JBoss BPM Suite Features

JBoss BPM Suite provides the following features:

- Pluggable human task service based on WS-HumanTask for including tasks that need to be performed by human actors.
- Pluggable persistence and transactions (based on JPA / JTA).
- Web-based process designer to support the graphical creation and simulation of your business processes (drag and drop).
- Web-based data modeler and form modeler to support the creation of data models and process and task forms.
- Web-based, customizable dashboards and reporting.
- A web-based workbench called Business Central, supporting the complete BPM life cycle:
  - Modeling and deployment: To author your processes, rules, data models, forms and other assets.
  - Execution: To execute processes, tasks, rules and events on the core runtime engine.
  - Runtime Management: To work on assigned task, manage process instances.

    - Reporting: To keep track of the execution using Business Activity Monitoring capabilities.
- Eclipse-based developer tools to support the modeling, testing and debugging of processes.
- Remote API to process engine as a service (REST, JMS, Remote Java API).
- Integration with Maven, Spring, and OSGi.

2.3. SUPPORTED PLATFORMS

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

- Red Hat JBoss Enterprise Application Platform 6.4
- Red Hat JBoss Web Server 2.1 (Tomcat 7) on JDK 1.7
- IBM WebSphere Application Server 8.5.5.0
- Oracle WebLogic Server 12.1.3 (12c)

2.4. USE CASES

2.4.1. Use Case: Business Decision Management in the Insurance Industry with Red Hat JBoss BRMS

Red Hat JBoss BRMS comprises a high performance rule engine, a rule repository, easy to use rule authoring tools, and complex event processing rule engine extensions. The following use case describes how these features of JBoss BRMS are implemented in insurance industry.

The consumer insurance market is extremely competitive, and it is imperative that customers receive efficient, competitive, and comprehensive services when visiting an online insurance quotation solution. An insurance provider increased revenue from their online quotation solution by upselling relevant, additional products during the quotation process to the visitors of the solution.

The diagram below shows integration of JBoss BRMS with the insurance provider's infrastructure. This integration is fruitful in such a way that when a request for insurance is processed, JBoss BRMS is consulted and appropriate additional products are presented with the insurance quotation.

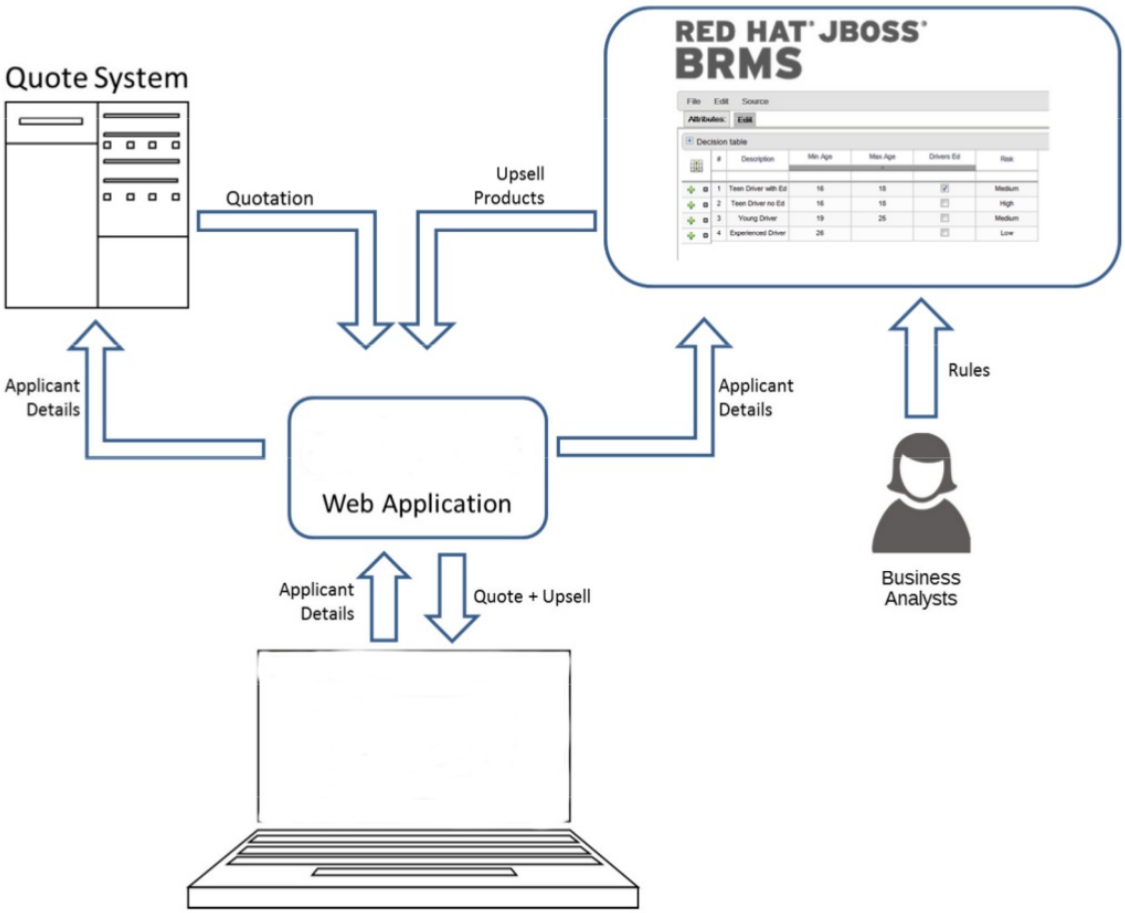


Figure 2.1. JBoss BRMS Use Case: Insurance Industry Decision Making

JBoss BRMS provides the decision management functionality, that automatically determines the products to present to the applicant based on the rules defined by the business analysts. The rules are implemented as decision tables, so they can be easily understood and modified without requiring additional support from IT.

2.4.2. Use Case: Process-based solutions in the loan industry

This section describes a use case of deploying JBoss BPM Suite to automate business processes (such as loan approval process) at a retail bank. This use case is a typical process-based specific deployment that might be the first step in a wider adoption of JBoss BPM Suite throughout an enterprise. It leverages features of both business rules and processes of JBoss BPM Suite.

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. The bank then processes the application in several steps, such as 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 has 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 JBoss BPM Suite. Here is the process flow:

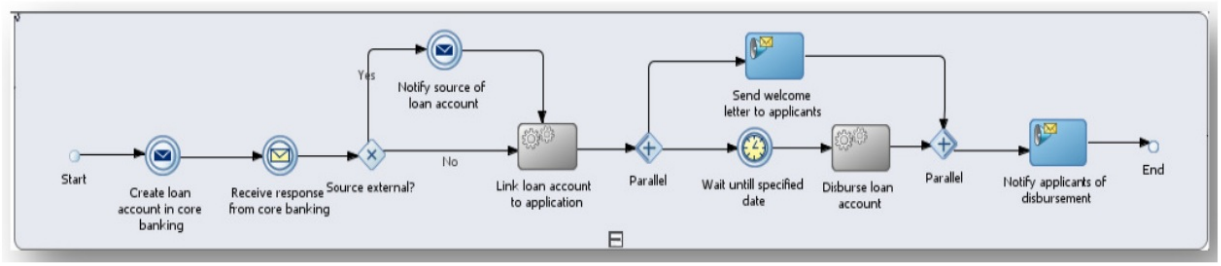


Figure 2.2. High-level loan application process flow

Business rules are developed with the rule authoring tools in JBoss 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 JBoss BPM Suite so that the entire loan application process can be automated.

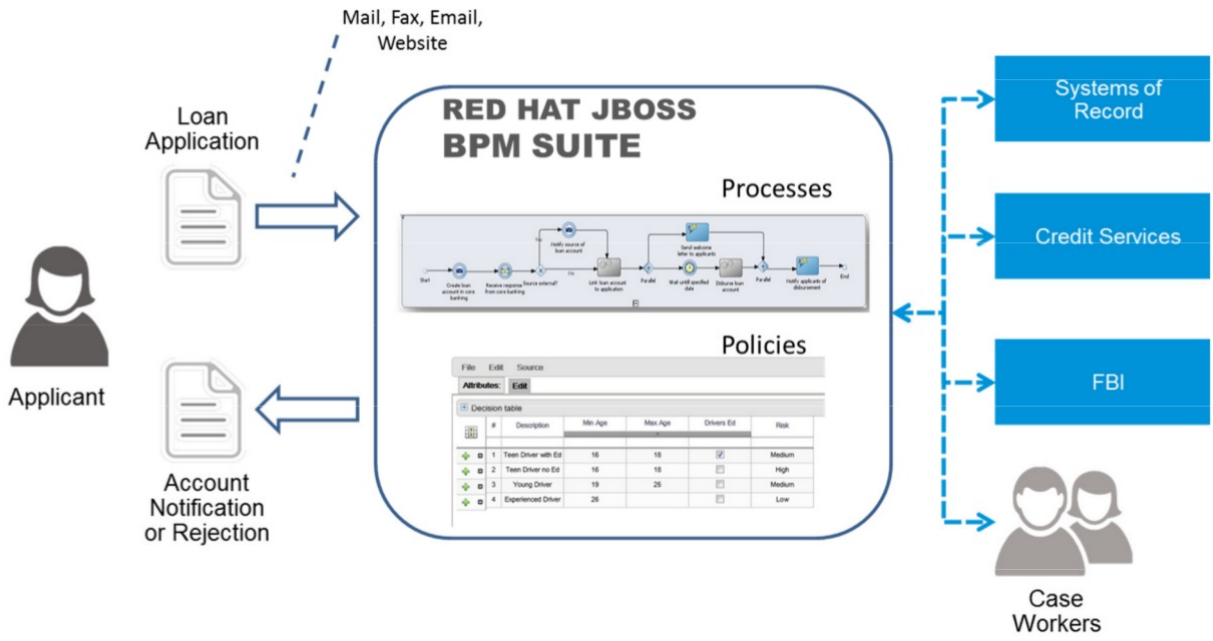


Figure 2.3. Loan Application Process Automation

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.

CHAPTER 3. MAVEN DEPENDENCIES

Apache Maven is a distributed build automation tool used in Java application development to build and manage software projects. Apart from building, publishing, and deploying capabilities, using Maven for your JBoss BRMS and JBoss BPM suite projects ensures the following:

- The build process is easy and a uniform build system is implemented across projects.
- All the required jar files for a project are made available at compile time.
- A proper project structure is set up.
- Dependencies and versions are well managed.
- No need for additional build processing, as Maven builds output into a number of predefined types, such as jar and war.

3.1. MAVEN REPOSITORIES

Maven uses repositories to store Java libraries, plug-ins, and other build artifacts. These repositories can be local or remote. JBoss BRMS and JBoss BPM suite products maintain local and remote maven repositories that you can add to your project for accessing the rules, processes, events, and other project dependencies. You must configure Maven to use these repositories and the Maven Central Repository in order to provide correct build functionality.

When building projects and archetypes, Maven dynamically retrieves Java libraries and Maven plug-ins from local repositories or downloads then from remote repositories. This promotes sharing and reuse of dependencies across projects.

3.2. USING MAVEN REPOSITORY IN YOUR PROJECT

You can direct Maven to use the JBoss Enterprise Application Platform Maven repository in your project in one of the following ways:

- By configuring the project's POM file (**`pom.xml`**).
- By modifying the Maven settings file (**`settings.xml`**).

The recommended approach is to direct Maven to use the JBoss Enterprise Application Platform Maven repository across all projects using the Maven global or user settings.

3.3. MAVEN CONFIGURATION FILE

To use Apache Maven for building and managing your JBoss BRMS and JBoss BPM Suite projects, you need to configure your projects to be built with Maven. To do so, Maven provides the Project Object Model or a **`pom.xml`** file that holds configuration details for your project.

The **`pom.xml`** is an XML file that contains information about the project (such as project name, version, description, developers, mailing list, and license), and build details (such as dependencies, location of the source, test, target directories, and plug-ins, repositories).

When you generate a project in Maven, it automatically generates the **`pom.xml`** file. You can edit this file to add more dependencies and new repositories. Maven downloads all the jar files and the dependent jar files from the Maven repository when you compile and package your project.

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 Apache Maven Project POM Reference.

3.4. MAVEN SETTINGS FILE

The Maven settings file (**settings.xml**) is used to configure Maven execution. You can locate this file in the following locations:

- In the Maven install directory at **$M2_HOME/conf/settings.xml**. These settings are called global settings.
- In the user's install directory at **${user.home}/.m2/settings.xml**. These settings are called user settings.
- Folder location specified by the system property kie.maven.settings.custom.

Note that the actual settings used is a merge of the files located in these locations.

Here 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>
```

Here, the activeByDefault tag is used to activate the profile that specifies the remote repository.

3.5. DEPENDENCY MANAGEMENT

In order to use the correct Maven dependencies in your Red Hat JBoss BPM Suite project, you must add relevant Bill Of Materials (BOM) files to the project's **pom.xml** file. Adding the BOM files ensures that the correct versions of transitive dependencies from the provided Maven repositories are included in the project.

The Maven repository in 6.1.0 is designed to be used only in combination with Maven Central and no other repositories are required.

Depending on your project requirements, declare the dependencies in your POM file in the dependencies section:

- org.jboss.bom.brms:jboss-brms-bpmsuite-bom:VERSION: This is the basic BOM without any Java EE6 support.
- org.jboss.bom.brms:jboss-javaee-6.0-with-brms-bpmsuite:VERSION: This provides support for Java EE6.

3.6. 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.

3.7. UPLOADING ARTIFACTS TO MAVEN REPOSITORY

There may be scenarios when your project may fail to fetch dependencies from a remote repository configured in its **pom.xml**. In such cases, you can programmatically upload dependencies to JBoss BPM Suite by uploading artifacts to the embedded maven repository through Business Central. JBoss BPM Suite uses a servlet for the maven repository interactions. This servlet processes a GET request to download an artifact and a POST request to upload one. You can leverage the servlet's POST request to upload an artifact to the repository via REST. To do this, implement the Http basic authentication and issue an HTTP POST request in the following format:

```
[protocol]://[hostname]:[port]/[context-root]/maven2/[groupId replacing
'.' with '/']/[artifactId]/[version]/[artifactId]-[version].jar
```

For example, to upload the **org.slf4j:slf4j-api:1.7.7 jar**, where artifactId is ***slf4j-api***, groupId is ***slf4j***, and version is ***1.7.7***, the URI must be:

```
http://localhost:8080/business-central/maven2/org/slf4j/slf4j-
api/1.7.7/slf4j-api-1.7.7.jar
```

■

The following example illustrates uploading a jar located at **/tmp** directory as a user **bpmsAdmin** with the password **abcd1234!**, to an instance of JBoss BPM Suite running locally:

```
package com.rhc.example;

import java.io.File;
import java.io.IOException;

import org.apache.http.HttpEntity;
import org.apache.http.HttpHost;
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.client.AuthCache;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.CredentialsProvider;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.protocol.HttpClientContext;
import org.apache.http.entity.mime.HttpMultipartMode;
import org.apache.http.entity.mime.MultipartEntityBuilder;
import org.apache.http.entity.mime.content.FileBody;
import org.apache.http.impl.auth.BasicScheme;
import org.apache.http.impl.client.BasicAuthCache;
import org.apache.http.impl.client.BasicCredentialsProvider;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class UploadMavenArtifact {
 private static final Logger LOG =
LoggerFactory.getLogger(UploadMavenArtifact.class);

 public static void main(String[] args) {

  //Maven coordinates
  String groupId = "com.rhc.example";
  String artifactId = "bpms-upload-jar";
  String version = "1.0.0-SNAPSHOT";


  //File to upload
  File file = new File("/tmp/"+artifactId+"-"+version+".jar");

  //Server properties
  String protocol = "http";
  String hostname = "localhost";
  Integer port = 8080;
  String username = "bpmsAdmin";
  String password = "abcd1234!";


  //Create the HttpEntity (body of our POST)
  FileBody fileBody = new FileBody(file);
  MultipartEntityBuilder builder = MultipartEntityBuilder.create();
```

```
  builder.setMode(HttpMultipartMode.BROWSER_COMPATIBLE);
  builder.addPart("upfile", fileBody);
  HttpEntity entity = builder.build();


  //Calculate the endpoint from the maven coordinates
  String resource = "/business-central/maven2/" + groupId.replace('.',
'/') + "/" + artifactId +"/" + version + "/" + artifactId + "-" + version
+ ".jar";

  LOG.info("POST " + hostname + ":" + port + resource);

  //Set up HttpClient to use Basic pre-emptive authentication with the
provided credentials
  HttpHost target = new HttpHost(hostname, port, protocol);
  CredentialsProvider credsProvider = new BasicCredentialsProvider();
        credsProvider.setCredentials(
                new AuthScope(target.getHostName(), target.getPort()),
                new UsernamePasswordCredentials(username,password));
        CloseableHttpClient httpclient = HttpClients.custom()
                .setDefaultCredentialsProvider(credsProvider).build();
        HttpPost httpPost = new HttpPost(resource);
  httpPost.setEntity(entity);
        AuthCache authCache = new BasicAuthCache();
        BasicScheme basicAuth = new BasicScheme();
        authCache.put(target, basicAuth);
        HttpClientContext localContext = HttpClientContext.create();
        localContext.setAuthCache(authCache);

  try {
   //Perform the HTTP POST
   CloseableHttpResponse response = httpclient.execute(target, httpPost,
localContext);
   LOG.info(response.toString());
   //Now check your artifact repository!
  } catch (ClientProtocolException e) {
   LOG.error("Protocol Error", e);
   throw new RuntimeException(e);
  } catch (IOException e) {
   LOG.error("IOException while getting response", e);
   throw new RuntimeException(e);
  }
 }
}
```

Alternative Maven Approach

An alternative maven approach is to configure your projects **`pom.xml`** by adding the repository as shown below:

```
    <distributionManagement>
      <repository>
        <id>guvnor-m2-repo</id>
        <name>maven repo</name>
        <url>http://localhost:8080/business-central/maven2/</url>
```

```
        <layout>default</layout>
      </repository>
    </distributionManagement>
```

Once you specify the repository information in the **pom.xml**, add the corresponding configuration in **settings.xml** as shown below:

```
    <server>
      <id>guvnor-m2-repo</id>
      <username>bpmsAdmin</username>
      <password>abcd1234!</password>
      <configuration>
        <wagonProvider>httpclient</wagonProvider>
        <httpConfiguration>
          <all>
            <usePreemptive>true</usePreemptive>
          </all>
        </httpConfiguration>
      </configuration>
    </server>
```

Now when you run the **mvn deploy** command, the jar file gets uploaded.

3.8. DEPLOYING RED HAT JBOSS BPM SUITE ARTIFACTS TO RED HAT JBOSS FUSE

Red Hat JBoss Fuse is an open source Enterprise Service Bus (ESB) with an elastic footprint and is based on Apache Karaf. The 6.1 version of Red Hat JBoss BPM Suite supports deployment of runtime artifacts to Fuse.

With the 6.1 release, JBoss BPM Suite runtime components (in the form of JARs) are OSGi enabled. The runtime engines JARs **MANIFEST.MF** files describe their dependencies, amongst other things. You can plug these JARs directly into an OSGi environment, like Fuse.

WARNING

JBoss BPM Suite uses a scanner to enable continuous integration and resolution/fetching of artifacts from remote Maven repositories. This scanner, called KIE-CI, uses a native Maven parser called Plexus to parse Maven POMs. However, this parser is not OSGi compatible and fails to instantiate in an OSGi environment. KIE-CI automatically switches to a simpler POM parser called **MinimalPomParser**.

The **MinimalPomParser** is a very simple POM parser implementation provided by Drools and is limited in what it can parse. It ignores some POM file parts, like a kJAR's parent POM. This means that users must not rely on those POM features (such as dependencies declared in parent POM in their kJARs) when using KIE-CI in OSGi environment.

Separating assets and code

One of the main advantage of deploying JBoss BPM Suite artifacts on Fuse is that each bundle is isolated, running in its own classloader. This allows you to separate the logic (code) from the assets. Business users can produce and change the rules and processes (assets) and package them in their own bundle, keeping them separate from the project bundle (code), created by the developer team. Assets can be updated without needing to change the project code.

CHAPTER 4. INSTALL AND SETUP JBOSS DEVELOPER STUDIO

Red Hat JBoss Developer Studio is the JBoss Integrated Development Environment (IDE) based on Eclipse. Get the latest JBoss Developer Studio from the Red Hat customer support portal at https://access.redhat.com. JBoss Developer Studio provides plug-ins with tools and interfaces for Red Hat JBoss BRMS and Red Hat JBoss BPM Suite. These plugins are based on the community version of these products. So, the JBoss BRMS plug-in is called the Drools plug-in and the JBoss BPM Suite plug-in is called the jBPM plug-in.

Refer to the *Red Hat JBoss Developer Studio* documentation for installation and set-up instructions.

WARNING

Due to an issue in the way multi-byte rule names are handled, you must ensure that the instance of JBoss Developer Studio is started with the file encoding set to **`UTF-8`**. You can do this by editing the **`$JBDS_HOME/studio/jbdevstudio.ini`** file and adding the following property: **`"-Dfile.encoding=UTF-8"`**

4.1. INSTALLING THE JBOSS DEVELOPER STUDIO PLUG-INS

Get the latest JBoss Developer version 8 from the Red Hat customer support portal at https://access.redhat.com. The JBoss BRMS and JBoss BPM Suite plug-ins for JBoss Developer Studio are available via the update site.

Procedure 4.1. Install the JBoss BRMS and JBoss BPM Suite Plug-ins in JBoss Developer Studio 8

1. Start JBoss Developer Studio.
2. Select **Help** → **Install New Software**.
3. Click **`Add`** to enter the **`Add Repository`** menu.
4. Provide a name for the software site in the **`Name`** field and add the following url in the **`Location`** field: https://devstudio.jboss.com/updates/8.0/integration-stack/
5. Click **`OK`**.
6. Select the **`JBoss Business Process and Rule Development`** from the available options and click **`Next`** and then **`Next`** again.
7. Read and accept the license by selecting the appropriate radio button, and click **`Finish`**.
8. You must restart JBoss Developer Studio, after the installation of the plug-ins has completed.

4.2. CONFIGURING THE JBOSS BRMS/BPM SUITE SERVER

JBoss Developer Studio can be configured to run the Red Hat JBoss BRMS and BPM Suite Server.

Procedure 4.2. Configure the Server

1. Open the Drools view by selecting **Window** → **Open Perspective** → **Other** and select **`Drools`** and click OK.

   To open the JBoss BPM Suite view, select **Window** → **Open Perspective** → **Other** and select **`jBPM`** and click OK.

2. Add the server view by selecting **Window** → **Show View** → **Other...** and select **Server** → **Servers**.

3. Open the server menu by right clicking the Servers panel and select **New** → **Server**.

4. Define the server by selecting **JBoss Enterprise Middleware** → **JBoss Enterprise Application Platform 6.1+** and clicking **`Next`**.

5. Set the home directory by clicking the **`Browse`** button. Navigate to and select the installation directory for JBoss EAP 6.1.1 which has JBoss BRMS installed. For configuring JBoss BPM Suite server, select the installation directory which has JBoss BPM Suite installed.

6. Provide a name for the server in the **`Name`** field, make sure that the configuration file is set, and click **`Finish`**.

4.3. IMPORTING PROJECTS FROM A GIT REPOSITORY INTO JBOSS DEVELOPER STUDIO

You can configure JBoss Developer Studio to connect to a central Git asset repository. The repository stores rules, models, functions and processes.

You can either clone a remote Git repository or import a local Git repository.

Procedure 4.3. Cloning a Remote Git Repository

1. Start the Red Hat JBoss BRMS/BPM Suite server (whichever is applicable) by selecting the server from the server tab and click the start icon.

2. Simultaneously, start the Secure Shell server, if not running already, by using the following command. The command is Linux and Mac specific only. On these platforms, if sshd has already been started, this command fails. In that case, you may safely ignore this step.

   ```
   /sbin/service sshd start
   ```

3. In JBoss Developer Studio, select **File** → **Import...** and navigate to the Git folder. Open the Git folder to select **Projects from Git** and click **`Next`**.

4. Select the repository source as **Clone URI** and click **`Next`**.

5. Enter the details of the Git repository in the next window and click **`Next`**.

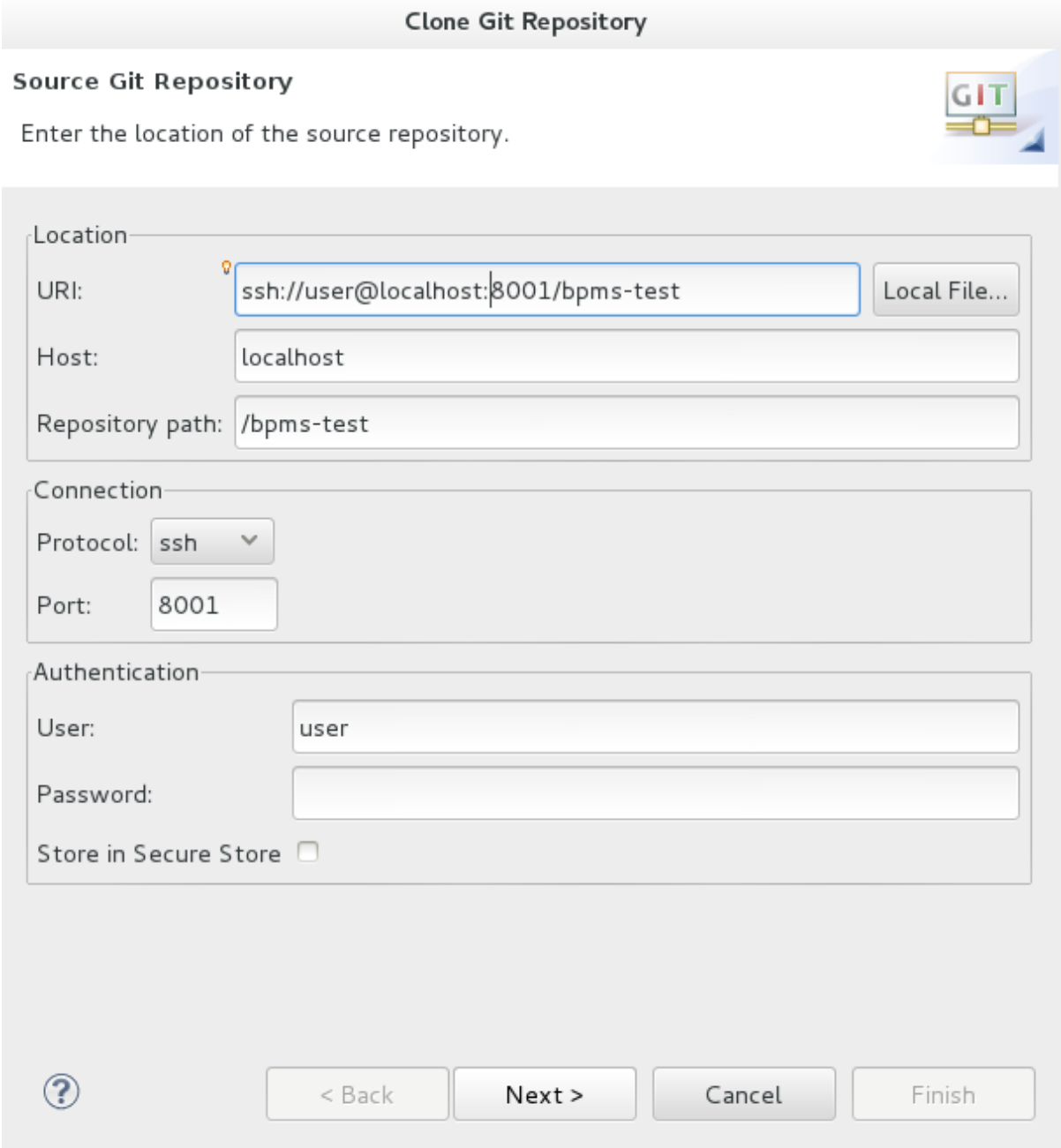


Figure 4.1. Git Repository Details

6. Select the branch you wish to import in the following window and click **Next**.

7. To define the local storage for this project, enter (or select) a non-empty directory, make any configuration changes and click **Next**.

8. Import the project as a general project in the following window and click **Next**. Name the project and click **Finish**.

Procedure 4.4. Importing a Local Git Repository

1. Start the Red Hat JBoss BRMS/BPM Suite server (whichever is applicable) by selecting the server from the server tab and click the start icon.

2. In JBoss Developer Studio, select **File** → **Import...** and navigate to the Git folder. Open the Git folder to select **Projects from Git** and click **Next**.

3. Select the repository source as **Existing local repository** and click **Next**.

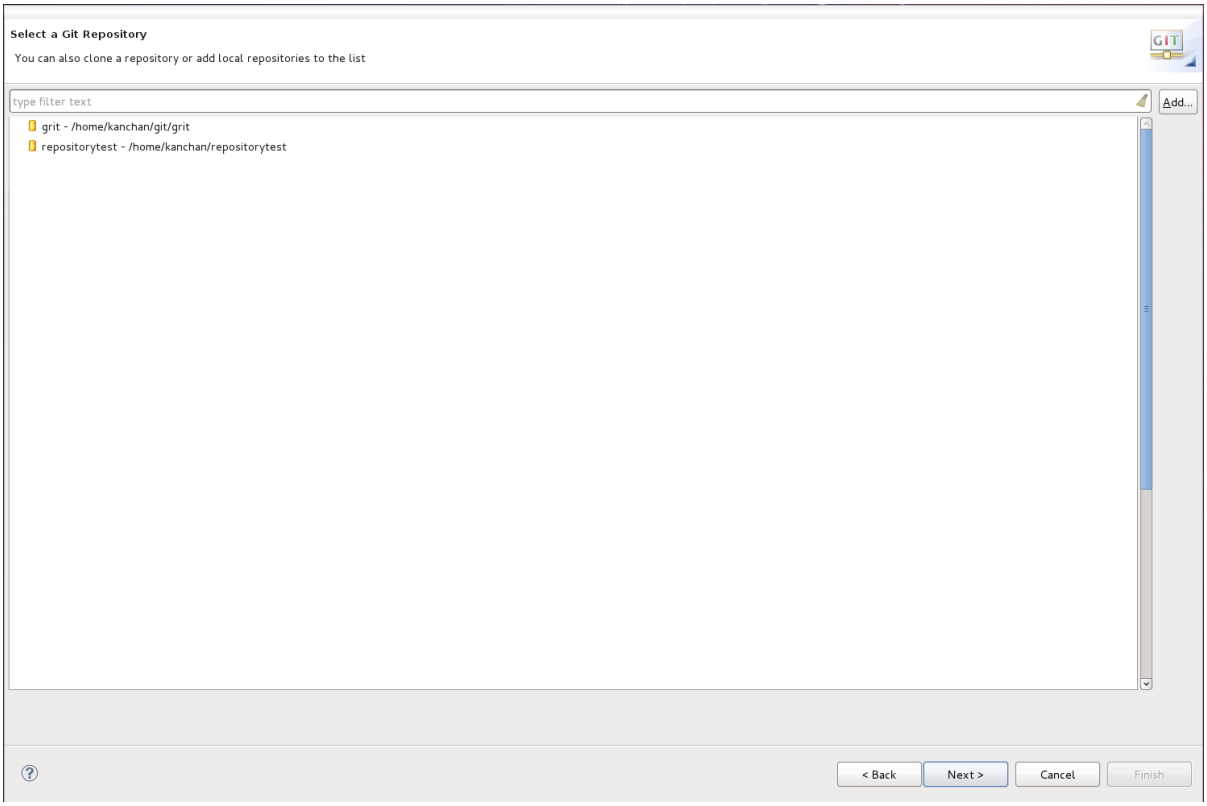


Figure 4.2. Git Repository Details

4. Select the repository that is to be configured from the list of available repositories and click **`Next`**.

5. In the dialog that opens, select the radio button **Import as general project** from the **Wizard for project import group** and click **Next**. Name the project and click **Finish**.

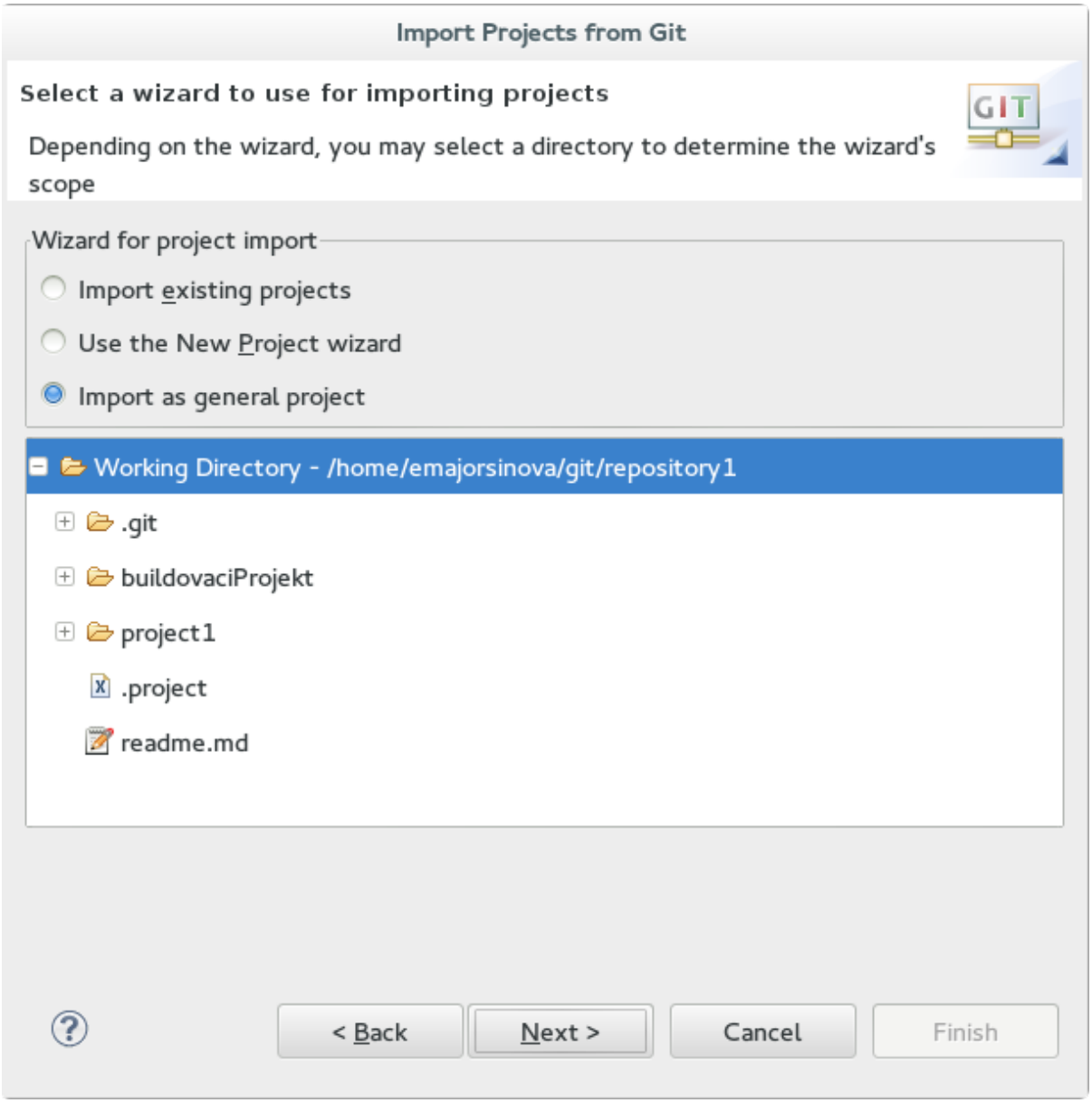


Figure 4.3. Wizard for Project Import

PART II. ALL ABOUT RULES

CHAPTER 5. RULE ALGORITHMS

5.1. PHREAK ALGORITHM

The new PHREAK algorithm is evolved from the RETE algorithm. While RETE is considered eager and data oriented, PHREAK on the other hand follows lazy and goal oriented approach. The RETE algorithm does a lot of work during the insert, update and delete actions in order to find partial matches for all rules. In case of PHREAK, this partial matching of rule is delayed deliberately.

The eagerness of RETE algorithm during rule matching wastes a lot of time in case of large systems as it does result in a rule firing eventually. PHREAK algorithm addresses this issue and therefore is able to handle large data more efficiently.

PHREAK is derived from a number of algorithms including the following LEAPS, RETE/UL and Collection-Oriented Match algorithms.

In addition to the enhancements listed in the Rete00 algorithm, PHREAK algorithm 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.

5.2. RULE EVALUATION WITH PHREAK ALGORITHM

When the rule engine starts, all the rules are unlinked. At this stage, there is no rule evaluation. The insert, update and deletes actions are queued before entering the beta network. The rule engine uses a simple heuristic, based on the rule most likely to result in firings, to calculate and select the next rule for evaluation. This delays the evaluation and firing of the other rules. When a rule has all the right input values populated, it gets linked in. That means, a goal representing this rule is created and placed into a priority queue, which is ordered by salience. Each queue is associated with an AgendaGroup. The engine only evaluates rules for the active AgendaGroup by inspecting the queue and popping the goal for the rule with the highest salience. So the work done shifts from the insert, update, delete phase to the fireAllRules phase. Only the rule for which the goal was created is evaluated and other potential rule evaluations are delayed. While individual rules are evaluated, node sharing is still achieved through the process of segmentation.

Unlike the tuple oriented RETE ,the PHREAK propagation is collection-oriented. So for the rule being evaluated, the engine accesses the first node and processes all queued insert, update and deletes. 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, adding the results to the same set. Once finished, this set is propagated to the next child node, and so on until the terminal node is reached. This creates a batch process effect which can provide performance advantages for certain rule constructs.

This linking and unlinking of rules happens through a layered bit mask system, based on 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. In case when there is no sharing of

the node, it becomes a single segment.

A bit-mask offset is assigned to each node in the segment. Also another bit mask is assigned to each segment in the rule's path. When there is at least one input, the node's bit is set to on state. When each node has its bit set to on state, the segment's bit is also set to on state. If any node's bit is set to off state, the segment is also set to off state. If each segment in the rule's path is set to on state, the rule is said to be linked in and a goal is created to schedule the rule for evaluation. The same bit-mask technique is used to also track dirty node, segments and rules. This allows for an already linked rule to be scheduled for evaluation if it is considered dirty since it was last evaluated. This ensures that no rule will ever evaluate partial matches.

As opposed to a single unit of memory in RETE, PHREAK has three levels of memory. This allows for much more contextual understanding during evaluation of a rule.

5.3. RETE ALGORITHM

5.3.1. ReteOO

The Rete implementation used in BRMS 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.

5.3.2. The Rete Root Node

Figure 5.1. ReteNode

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

5.3.3. The ObjectTypeNode

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

In JBoss BRMS, an inserted object retrieves a list of valid ObjectTypesNodes through a lookup in a HashMap from the object's class. If this list does not exist, it scans all the ObjectTypeNodes to find valid matches. It then caches these matched nodes in the list. This enables JBoss BRMS to match against any class type that matches with an **`instanceof`** check.

5.3.4. 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.

5.3.5. Hashing

JBoss BRMS 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 retrieves the correct AlphaNode from the HashMap. This avoids unnecessary literal checks.

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

5.3.6. BetaNodes

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

5.3.7. Alpha Memory

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

5.3.8. Beta Memory

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

5.3.9. Lookups with BetaNodes

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

5.3.10. LeftInputNodeAdapters

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

5.3.11. Terminal Nodes

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

5.3.12. Node Sharing

Node sharing is used to prevent redundancy. As many rules repeat the same patterns, node sharing allows users to collapse those patterns so that the patterns need not be reevaluated for every single instance.

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

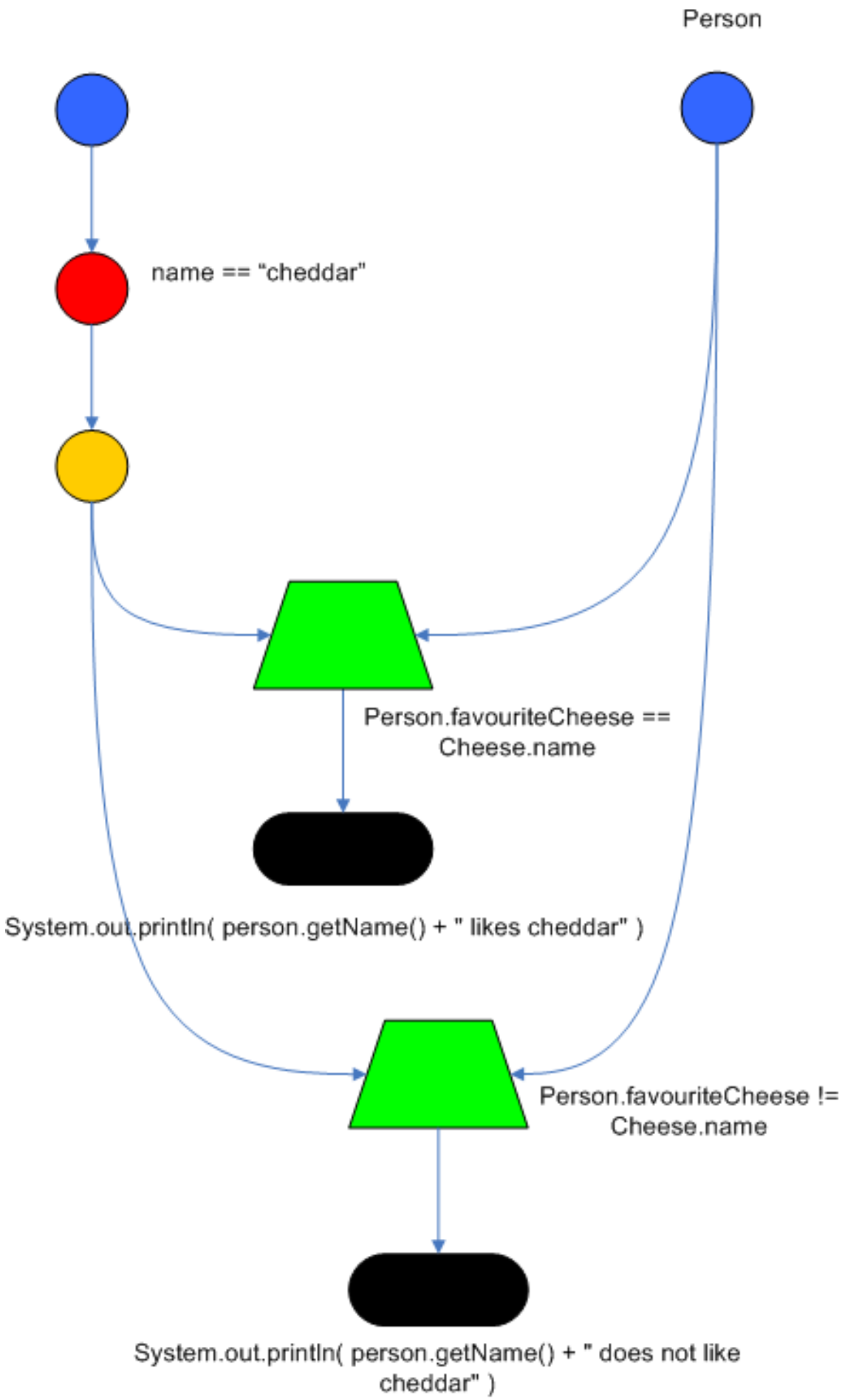


Figure 5.2. Node Sharing

5.4. SWITCHING BETWEEN PHREAK AND RETEOO

Switching Using System Properties

To switch between the PHREAK algorithm and the ReteOO algorithm, you need to edit the **`drools.ruleEngine`** system properties with the following values:

```
drools.ruleEngine=phreak
```

or

```
drools.ruleEngine=reteoo
```

The previous value of phreak is the default value.

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, you can 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

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

CHAPTER 6. GETTING STARTED WITH RULES AND FACTS

To create business rules, you need an appropriate fact model on which your business rules will operate. A fact is an instance of an application object represented as a POJO. You then author rules containing the business logic using either the Business Central web interface or your JBoss Developer Studio.

The conditions on the WHEN clause of a rule, query for fact combinations that match it's criteria. So, when a particular set of conditions occur as specified in your rule's WHEN clause, then the specified list of actions in the THEN clause are executed. A rule's action asserts a fact, retracts a fact, or updates a fact on to the Rule engine. As a result, other rules may then be fired.

This is how rules are processed:

1. BRMS parses all the .drl rule files into the knowledge base.
2. Each fact is asserted into the working memory. As the facts are asserted, BRMS uses PHREAK or RETE algorithm to infer how the facts relate to the rules. So the working memory now contains a copy of the parsed rules and a reference to the facts.
3. The **`fireAllRules()`** method is called. This triggers all the interactions between facts and rules and the rule engine evaluates all the rules against aal the facts and concludes which rules should be fired against which facts.
4. All the rule-facts combination (when a particular rule matches against one or more sets of facts), are queued within a data construct called an agenda.
5. Finally, activations are processed one-by-one from the agenda, calling the consequence of the rule on the facts that activated it. Note that the firing of an activation on the agenda can modify the contents of the agenda before the next activation is fired. The PHREAK or RETE algorithm are used to handle such situatiobs edfficiently.

6.1. CREATE YOUR FIRST RULE

In this section, you will learn to create and execute your first rule.

As with most Java applications, you can create a rule in *plain Java* and that is what we will start with. This will allow you to get comfortable with the idea of using rules without getting distracted with tooling.

Since a lot of developers are more comfortable with using Maven, we will next show you how to create the same rule using *Maven*.

We will then move on to creating (and executing) the same rule using *JBoss Developer Studio with the JBoss BRMS plug-in*.

Finally, we will move to creating and executing the same rule in the *Business Central* environment of JBoss BRMS.

This will help you decide which environment is right for you to learn about rules. Let's get started.

6.1.1. Create and Execute Your First Rule Using Plain Java

Procedure 6.1. Create and Execute your First Rule using plain Java

1. **Create your fact model**

Create a POJO based on which your rule runs. For example, create a **`Person.java`** file in a directory called **`my-project`**. The **`Person`** class contains the getter and setter methods to retrieve and set values of first name, last name, hourly rate, and wage of a person:

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

  public class Person {
    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;

      public String getFirstName() {
       return firstName;
     }

     public void setFirstName(String firstName) {
       this.firstName = firstName;
     }

     public String getLastName() {
       return lastName;
     }

     public void setLastName(String lastName) {
       this.lastName = lastName;
     }

     public Integer getHourlyRate() {
       return hourlyRate;
     }

     public void setHourlyRate(Integer hourlyRate) {
       this.hourlyRate = hourlyRate;
     }

     public Integer getWage(){
       return wage;
     }

     public void setWage(Integer wage){
       this.wage = wage;
     }
    }
```

2. **Create your rule**
   Create your rule file in **`.drl`** format under **`my-project`** directory. Here is the simple rule file called **`Person.drl`**, which does a calculation on the wage and hourly rate values and displays a message based on the result.

```
dialect "java"

rule "Wage"
```

```
  when

    Person(hourlyRate*wage > 100)
    Person(name : firstName, surname : lastName)

  then
    System.out.println( "Hello" + " " + name + " " + surname + "!"
);
    System.out.println( "You are rich!" );

end
```

3. **Create a main class**
   Create your main class (say, **`DroolsTest.java`**) and save it in the same **`my-project`** directory as your POJO. This file will load the knowledge base and fire your rules:

   In the **`DroolsTest.java`** file:

   a. Add the following import statements to import the KIE services, container, and session:

   ```
   import org.kie.api.KieServices;
   import org.kie.api.runtime.KieContainer;
   import org.kie.api.runtime.KieSession;
   ```

   b. Load the knowledge base and fire your rule from the **`main()`** method:

   ```
   public class DroolsTest {
       public static final void main(String[] args) {
         try {
           // load up the knowledge base
           KieServices ks = KieServices.Factory.get();
           KieContainer kContainer = ks.getKieClasspathContainer();
           KieSession kSession = kContainer.newKieSession();

           // go !
           Person p = new Person();
           p.setWage(12);
           p.setFirstName("Tom");
           p.setLastName("Summers");
           p.setHourlyRate(10);

           kSession.insert(p);
           kSession.fireAllRules();
           }

         catch (Throwable t) {
           t.printStackTrace();
         }
       }
   }
   ```

   The **`main()`** method passes the model to the rule, which contains the first name, last name, wage, and hourly rate.

4. **Download the BRMS Engine jar files**
   Download the BRMS engine jar files and save them under **`my-project/BRMS-engine-jars/`**. These files are available from the Red Hat Customer Portal under the generic deployable version.

5. **Create the `kmodule.xml` metadata file**
   Create a file called **`kmodule.xml`** under **`my-project/META-INF`** to create the default session. At the minimum, this file contains the following:

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

6. **Build your example**
   Navigate to the **`my-project`** directory and execute the following command from the command line:

```
javac -classpath "./BRMS-engine-jars/*:." DroolsTest.java
```

   This will compile and build your java files.

7. **Run your example**
   If there were no compilation errors, you can now run the **`DroolsTest`** to execute your rule:

```
java -classpath "./BRMS-engine-jars/*:." DroolsTest
```

   The expected output is:

```
Hello Tom Summers!
You are rich!
```

6.1.2. Create and Execute Your First Rule Using Maven

Procedure 6.2. Create and Execute your First Rule using Maven

1. **Create a basic Maven archetype**
   Navigate to a directory of choice in your system and execute the following command:

```
mvn archetype:generate -DgroupId=com.sample.app -DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart -
DinteractiveMode=false
```

   This creates a directory called **`my-app`** with the following structure:

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
```

```
|                   `-- App.java
`-- test
    `-- java
        `-- com
            `-- mycompany
                `-- app
                    `-- AppTest.java
```

The **my-app** directory comprises:

- A **src/main** directory for storing your application's sources.
- A **src/test** directory for storing your test sources.
- A **pom.xml** file containing the Project Object Model (POM) for your project. At this stage, the **pom.xml** file contains the following:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sample.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

2. **Create your Fact Model**
   Once you are done with the archetype, create a class based on which your rule runs. Create the POJO called **Person.java** file under **my-app/src/main/java/com/mycompany/app** folder. This class contains the getter and setter methods to retrieve and set values of first name, last name, hourly rate, and wage of a person.

```
package com.mycompany.app;

  public class Person {

    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;


    public String getFirstName() {
      return firstName;
```

```
    }

    public void setFirstName(String firstName) {
      this.firstName = firstName;
    }

    public String getLastName() {
      return lastName;
    }

    public void setLastName(String lastName) {
      this.lastName = lastName;
    }

    public Integer getHourlyRate() {
      return hourlyRate;
    }

    public void setHourlyRate(Integer hourlyRate) {
      this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
      return wage;
    }

    public void setWage(Integer wage){
      this.wage = wage;
    }

  }
```

3. **Create your rule**
   Create your rule file in **`.drl`** format under **`my-app/src/main/resources/rules`**.

   Here is the simple rule file called **`Person.drl`**, which imports the **`Person`** class:

```
package com.mycompany.app;
import com.mycompany.app.Person;

dialect "java"

rule "Wage"

  when
    Person(hourlyRate*wage > 100)
    Person(name : firstName, surname : lastName)

  then
    System.out.println( "Hello " + name + " " + surname + "!" );
    System.out.println( "You are rich!" );

end
```

As before, this rule does a simple calculation on the wage and hourly rate values and displays a message based on the result.

4. **Create the `kmodule.xml` metadata file**
   Create an empty file called **`kmodule.xml`** under **`my-app/src/main/resources/META-INF`** to create the default session. This file contains the following:

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

5. **Set project dependencies in the `pom.xml` configuration file**
   As Maven manages the classpath through this configuration file, you must declare in it the libraries your application requires. Edit the **`my-app/pom.xml`** file to set the JBoss BRMS dependencies and setup the GAV values for your application, as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0.0</version>
<repositories>
  <repository>
    <id>jboss-ga-repository</id>
    <url>http://maven.repository.redhat.com/techpreview/all</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>LATEST</version>
  </dependency>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-api</artifactId>
    <version>LATEST</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

6. **Test it!**
   After you add the dependencies in the **`pom.xml`** file, use the **`testApp`** method of the **`my-app/src/test/java/com/mycompany/app/AppTest.java`** (which is created by default by Maven) to instantiate and test the rule.

In the **`AppTest.java`** file:

a. Add the following import statements to import the KIE services, container, and session:

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
```

b. Load the knowledge base and fire your rule from the **`testApp()`** method:

```
public void testApp() {

  // load up the knowledge base
  KieServices ks = KieServices.Factory.get();
  KieContainer kContainer = ks.getKieClasspathContainer();
  KieSession kSession = kContainer.newKieSession();

  // set up our Person fact model
  Person p = new Person();
  p.setWage(12);
  p.setFirstName("Tom");
  p.setLastName("Summers");
  p.setHourlyRate(10);

  // insert him into the session
  kSession.insert(p);

  // and fire all rules on him
  kSession.fireAllRules();

  // we can assert here, but the rule itself should output
something since the person's wage is more than our baseline rule
}
```

The **`testApp()`** method passes the model to the rule, which contains the first name, last name, wage, and hourly rate.

7. **Build your example**
Navigate to the **`my-app`** directory and execute the following command from the command line:

```
mvn clean install
```

When you run this command for the first time, it may take a while as Maven downloads all the artifacts required for this project such as JBoss BRMS jar files.

The expected output is:

```
Hello Tom Summers!
You are rich!
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
1.194 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

```
[INFO]
...

[INFO] ------------------------------------------------------------
--------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
--------------
[INFO] Total time: 6.393 s
...
[INFO] ------------------------------------------------------------
--------------
```

That is it! You have run the rule using Maven!

6.1.3. Create and Execute Your First Rule Using JBoss Developer Studio

Procedure 6.3. Create and Execute your First Rule using JBoss Developer Studio

To execute a rule project in JBoss Developer Studio successfully, ensure that you have installed the JBoss BRMS tools plug-in support, and configured the JBoss EAP 6 running BRMS and the BRMS runtime.

1. **Create a BRMS Project.**

   a. Start JBoss Developer Studio and navigate to **File** → **New** → **Project**.

      This opens a **`New Project`** dialog box.

   b. In the New Project dialog box, select **Drools** → **Drools Project** and click **`Next`**.

   c. Type a name for your project and click **`Next`**.

      The New Project dialog box provides you choice to add some default artifacts to your project, such as sample rules, decision tables and Java classes for them. Let us select the first two check boxes and click **`Next`**.

   d. Select the configured BRMS runtime in the **`Drools Runtime`** dialog box. If you have not already configured your BRMS runtime, click **`Configure Workspace Settings...`** link and configure the BRMS runtime jars.

   e. Select *Drools 6.0.x* for **`Generate code compatible with:`** field and provide values for *groupId*, *artifactId*, and *version*. These values form your project's fully qualified artifact name. Let us provide the following values:

      - groupId: com.mycompany.app
      - artifactId: my-app
      - version: 1.0.0

   f. Click Finish.

      This sets up a basic project structure, classpath and sample rules for you to get started with.

```
My-Project
`-- src/main/java
    |-- com.sample
    |   `-- DroolsTest.java
    |
 `-- src/main/rules
    | -- Sample.drl
    |
  `-- JRE System Library
    |
  `-- Drools Library
    |
  `-- src
    |
  `-- pom.xml
```

This newly created project called *My-Project* comprises the following:

- A rule file called **`Sample.drl`** under **`src/main/rules`** directory.
- An example java file called **`DroolsTest.java`** under **`src/main/java`** in the com.sample package. You can use the **`DroolsTest`** class to execute your rules in the BRMS engine.
- The Drools Library directory. This acts as a custom classpath container that contains all the other jar files necessary for execution.

2. **Create your fact model**
   The sample **`DroolsTest.java`** file contains a sample POJO called **`Message`** with getter and setter methods. You can edit this class or create another similar POJO. Let us remove this POJO and create a new POJO called **`Person`**, which sets and retrieves values of first name, last name, hourly rate, and wage of a person:

```
public static class Person {
          private String firstName;
          private String lastName;
          private Integer hourlyRate;
          private Integer wage;

          public String getFirstName() {
            return firstName;
          }

          public void setFirstName(String firstName) {
            this.firstName = firstName;
          }

          public String getLastName() {
            return lastName;
          }

          public void setLastName(String lastName) {
            this.lastName = lastName;
          }

          public Integer getHourlyRate() {
```

```
            return hourlyRate;
          }

          public void setHourlyRate(Integer hourlyRate) {
            this.hourlyRate = hourlyRate;
          }

          public Integer getWage(){
            return wage;
          }

          public void setWage(Integer wage){
            this.wage = wage;
          }
```

3. **Update the main method**
   The sample **`DroolsTest.java`** file contains a **`main()`** method that loads up the knowledge base and fires the rules. Update this **`main()`** method to pass the *Person* object to the rule:

```
  public static final void main(String[] args) {
      try {
        // load up the knowledge base
          KieServices ks = KieServices.Factory.get();
          KieContainer kContainer = ks.getKieClasspathContainer();
          KieSession kSession = kContainer.newKieSession("ksession-
rules");

        // go !
          Person p = new Person();
          p.setWage(12);
          p.setFirstName("Tom");
          p.setLastName("Summers");
          p.setHourlyRate(10);

          kSession.insert(p);
          kSession.fireAllRules();

          }

        catch (Throwable t) {
          t.printStackTrace();
        }
    }
```

NOTE

To load the knowledge base, you first get the KieServices instance and the classpath based KieContainer. Then you build your KieSession with the KieContainer. Here, we are passing the session name *ksession-rules* that matches the one defined in **`kmodule.xml`** file.

4. **Create your rule**
   The sample rule file **`Sample.drl`** contains a basic skeleton of a rule. You can edit this file or create a new one to write your own rule.

In your rule file:

a. Include the package name:

```
package com.sample
```

b. Import facts into the rule:

```
import com.sample.DroolsTest.Person;
```

c. Create the rule in "when", "then" format.

```
dialect "java"

rule "Wage"

    when
      Person(hourlyRate*wage > 100)
      Person(name : firstName, surname : lastName)

    then
      System.out.println( "Hello" + " " + name + " " + surname +
"!" );
      System.out.println( "You are rich!" );

end
```

5. **Test your rule**
   Right-click the **`DroolsTest.java`** file and select **Run As** → **Java Application**.

Expected output at the console view:

```
Hello Tom Summers!
You are rich!
```

6.1.4. Create and Execute Your First Rule Using Business Central

Prerequisite

Ensure that you have successfully installed JBoss BPM Suite before you run this simple rule example using Business Central interface.

Procedure 6.4. Create and Execute your First Rule using Business Central

1. **Login to Business Central**

   a. On the command line, move into the **`$SERVER_HOME/bin/`** directory and execute the following command:

   ```
   ./standalone.sh
   ```

   b. Once your server is up and running, open the following in a web browser:

   ```
   http://localhost:8080/business-central
   ```

This opens the Business Central login page.

c. Log in to the Business Central with the user credentials created during installation.

2. **Create a repository structure and create a project under it**

   a. On the main menu of Business Central, go to **Authoring** → **Administration**.

   b. In the displayed **`Organizational Unit Manager`** view, click **`Add`**.

   c. In the displayed **`Add New Organizational Unit`** dialog box, define the unit properties. For example:

      - Name: EmployeeWage
      - Owner: Employee

      Click **`OK`**.

   d. On the perspective menu, click **Repositories** → **New repository**.

   e. In the displayed **`Create Repository`** dialog box, define the repository properties. For example:

      - Repository Name: EmployeeRepo
      - Organizational Unit: EmployeeWage

      Click **`Finish`**.

   f. Go to **Authoring** → **Project Authoring**.

   g. In the Project Explorer, under the organizational unit drop-down box, select *EmployeeWage*, and in the repository drop-down box select *EmployeeRepo*.

   h. On the perspective menu, go to **New Item** → **Project**.

   i. In the displayed **`Create new Project`** dialog box, provide a name (for example, *MyProject*) for your project properties and click **`Ok`**.

   j. In the **`New Project`** dialog box, define the maven properties of the Project. For example:

      - Group ID: org.bpms
      - Artifact ID: MyProject
      - Version ID: 1.0.0

      Click **`Finish`**.

3. **Create a fact model**

   a. On the perspective menu, go to **New Item** → **Data Object**.

   b. In the displayed **`Create new Data Object`** dialog box, provide the values for object name and package. For example:

      - Data Object: Person

- Package: org.bpms.myproject

Click **Ok**.

c. In the displayed **Create new field** window of the newly created **Person** data object, add a variable name in the **Id** field, select data type for the variable in the **Type** field. For example:

- Id: firstName

  Type: String

- Id: lastName

  Type: String

- Id: hourlyRate

  Type: Integer

- Id: wage

  Type: Integer

Click **Create** and then **Save**.

4. **Create a rule**

a. On the perspective menu, click **New Item** → **DRL File**.

b. In the Create new dialog box, provide the name and package name of your rule file. For example:

- DRL file name: MyRule

- Package: org.bpms.myproject

Click **Ok**.

c. In the displayed DRL editor with the **MyRule.drl** file, write your rule as shown below:

```
package org.bpms.myproject;
rule "MyRule"
ruleflow-group "MyProjectGroup"
  when

    Person(hourlyRate*wage > 100)
    Person(name : firstName, surname : lastName)

  then
    System.out.println( "Hello" + " " + name + " " + surname +
"!" );
    System.out.println( "You are rich!" );

end
```

d. Click **Save**.

5. **Create a Business Process and add a Business Rule Task**

   a. On the main menu of Business Central, go to **Authoring** → **Project Authoring**.

   b. In the Create new Business Process dialog box, provide values for Business Process name and package. For example:

      - Business Process: MyProcess
      - Package: org.bpms.myproject

      Click **`Ok`**

      The Process Designer with the canvas of the created Process definition opens.

   c. Expand the **`Object Library`** palette with Process Elements.

      A Start Event element appears on the canvas.

   d. From the **`Object Library`**, navigate to **`Tasks`** and drag a Business Rule Task to the canvas. Then, integrate the Business Rule task into the process workflow.

   e. Select the Business Rule Task and set the following properties in the **`Properties`** panel under **`Core Properties`**:

      - Name:Rule_Task
      - DataInputSet

        When you click on the **`DataInputSet`** field, an editor for Data Input opens. Click **`Add Data Input`** and provide the data input elements. For example:

        - Name:person_Task
        - Defined Types:org.bpms.myproject.Person

          Click **`Ok`**.

      - Assigments

        When you click on the **`Assigments`** field, an editor for Data Assignments opens. Provide the assignment values here. For example:

        - Assignment Type: DataInput
        - From Object:person_proc
        - Assignment Type: is mapped to
        - To Object:person_Task
        - Ruleflow Group: MyProjectGroup

        Click **`Ok`**.

        Now you have successfully created an object that maps to the variables you have set in your fact model. Your business process passes this object as an input to your rule.

- Click **`Generate all Forms`**.
- Save the Process.

6. **Build and deploy your rule**

   a. Open Project Editor and click **`Build & Deploy`**.

   A green notification appears in the upper part of the screen informing you that the project has been built and deployed successfully to the Execution Server.

   b. Go to **Process Management** → **Process Definitions**.

   You can see your newly built process listed in the **`Process Definitions`** window.

   c. Click button under **`Actions`** to start your Process.

   A **`MyProcess`** dialog box opens.

   d. In the **`MyProcess`** dialog box, provide the following values of the variables defined in your fact model and click **`Submit`**:

   - firstName: Tom
   - hourlyRate: 12
   - lastName: Summers
   - wage: 10

   As these values satisfy the rule condition, the expected output at the console is:

```
16:19:58,479 INFO
[org.jbpm.kie.services.impl.store.DeploymentSynchronizer] (http-
/127.0.0.1:8080-1) Deployment unit org.bpms:MyProject:1.0 stored
successfully
16:26:56,119 INFO  [stdout] (http-/127.0.0.1:8080-5) Hello Tom
Summers!
16:26:56,119 INFO  [stdout] (http-/127.0.0.1:8080-5) You are
rich!
```

6.2. EXECUTION OF RULES

6.2.1. Agenda

The Agenda is a *Rete* feature. During actions on the **`WorkingMemory`**, rules may become fully matched and eligible for execution. A single Working Memory Action can result in multiple eligible rules. When a rule is fully matched an Activation is created, referencing the rule and the matched facts, and placed onto the Agenda. The Agenda controls the execution order of these Activations using a Conflict Resolution strategy.

6.2.2. Agenda Processing

The engine cycles repeatedly through two phases:

1. 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.

2. 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.

6.2.3. Conflict Resolution

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).

6.2.4. AgendaGroup

Agenda groups are a way to partition rules on the agenda. At any one time, only one group has "focus" which means that activations for rules in that group only will take effect. You can also have rules with "auto focus" which means that the focus is taken for its agenda group when that rule's conditions are true.

Agenda groups are known as "modules" in CLIPS terminology. Agenda groups provide a way to create a "flow" between grouped rules. You can switch the group which has focus either from within the rule engine, or via the API. If your rules have a clear need for multiple "phases" or "sequences" of processing, consider using agenda-groups for this purpose.

6.2.5. setFocus()

Each time **`setFocus()`** is called it pushes the specified Agenda Group onto a stack. When the focus group is empty it is popped from the stack and the focus group that is now on top evaluates. An Agenda Group can appear in multiple locations on the stack. The default Agenda Group is "MAIN", with all rules which do not specify an Agenda Group being in this group. It is also always the first group on the stack, given focus initially, by default.

6.2.6. setFocus() Example

This is what the setFocus() element looks like:

```
ksession.getAgenda().getAgendaGroup( "Group A" ).setFocus();
```

6.2.7. ActivationGroup

An activation group is a set of rules bound together by the same "activation-group" rule attribute. In this group only one rule can fire, and after that rule has fired all the other rules are cancelled from the agenda. The **`clear()`** method can be called at any time, which cancels all of the activations before one has had a chance to fire.

6.2.8. ActivationGroup Example

This is what an ActivationGroup looks like:

```
ksession.getAgenda().getActivationGroup( "Group B" ).clear();
```

6.3. INFERENCE

6.3.1. The Inference Engine

The *inference engine* is the part of the Jboss BRMS engine which matches production facts and data to rules. It is often called the brain of a Production Rules System as it is able to scale to a large number of rules and facts. It makes inferences based on its existing knowledge and performs the actions based on what it infers from the information.

The rules are stored in the production memory and the facts that the inference engine matches against, are stored in the working memory. Facts are asserted into the working memory where they may get modified or retracted. A system with a large number of rules and facts may result in many rules being true for the same fact assertion. Such conflicting rules are managed using a conflict resolution strategy. This strategy determines the order of execution of the rules by assigning a priority level to each rule.

Inferences can be forward chaining or backward chaining. In a forward chaining inference machanism, when some data gets inserted into the working memory, the related rules are triggered and if the data satisfies the rule conditions, corresponding actions are taken. These actions may insert new data into the working memory and therefore trigger more rules and so on. Thus, the forward chaining inference is data driven. On the contrary, the backward chaining inference is goal driven. In this case, the system looks for a particular goal, 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. Correct use of inference can create agile and less error prone business rules, which are easier to maintain.

6.3.2. Inference Example

The following example illustrates how an inference is made about whether a person is eligible to have a bus pass based on the rule conditions. Here is a rule that provides the age policy for a person to hold a bus pass:

```
rule "Infer Adult"
when
  $p : Person( age >= 18 )
then
  insert( new IsAdult( $p ) )
end
```

Based on this rule, a rule engine infers whether a person is an adult or a child and act on it. Every person who is 18 years or above will have an instance of IsAdult inserted for them in the working memory. This inferred relation of age and bus pass can be inferred in any rule, such as:

```
$p : Person()
IsAdult( person == $p )
```

6.4. TRUTH MAINTENANCE

The inference engine is responsible for logical decisions on assertions and retraction of facts. After regular insertions, facts are generally retracted explicitly. However, in case of logical assertions, the fact that was asserted are automatically retracted when the conditions that asserted it in the first place are no

longer true. In other words, the facts are retracted when there is no single condition that supports the logical assertion.

The inference engine uses a the mechanism of truth maintenance to efficiently handle the inferred information from rules. A *Truth Maintenance System* (TMS) refers to an inference engine's ability to enforce truthfulness when applying rules. It provides justified reasoning for each and every action taken by the inference engine. It validates the conclusions of an inference engine. If the inference engine asserts some data as a result of firing a rule, it uses the truth maintenance to justify the assertion.

A Truth Maintenance System also helps identify inconsistencies and handle contradictions. For example, if there are two rules to be fired, each resulting in a contradictory action, the Truth Maintenance System enables the inference engine to decide its actions based on assumptions and derivations of previously calculated conclusions. Truth maintenance plays an important role in enabling the inference engine to logically insert or retract facts. With logical assertions, the fact that was asserted are automatically retracted when the conditions that asserted it in the first place are no longer true.

The normal insertion of facts, referred to as stated insertions, are straight forward and do not need a reasoning. However, the logical assertions need to be justified. If the inference engine tries to logically insert an object when there is an equal stated object, it fails as it can not justify a stated fact. If the inference engine tries for a stated insertion of an existing equal object that is justified, then it overrides the justified insertion, and removes the justifications.

The following flowcharts illustrate the lifecycle of stated and logical insertions:

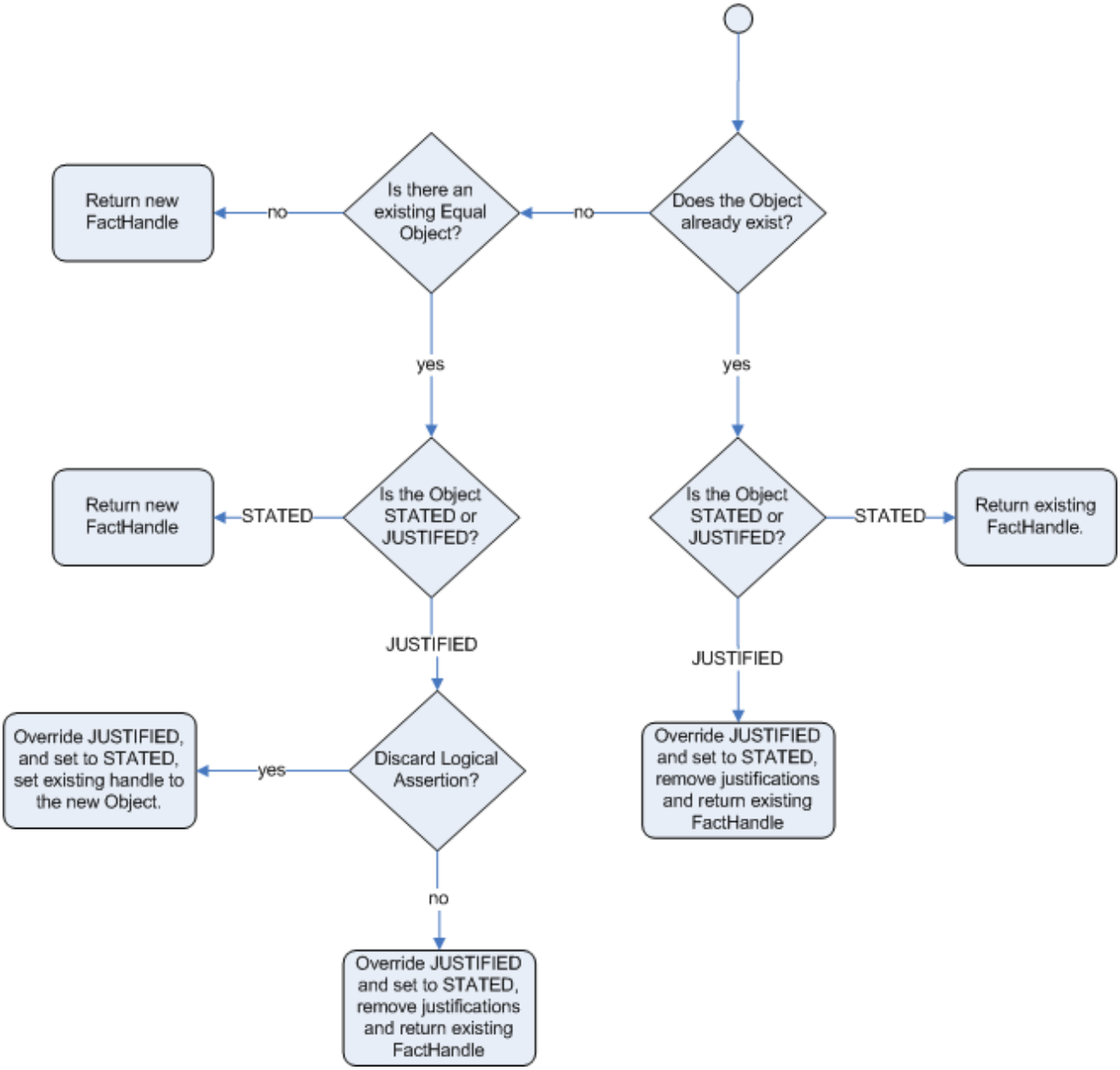


Figure 6.1. Stated Assertion

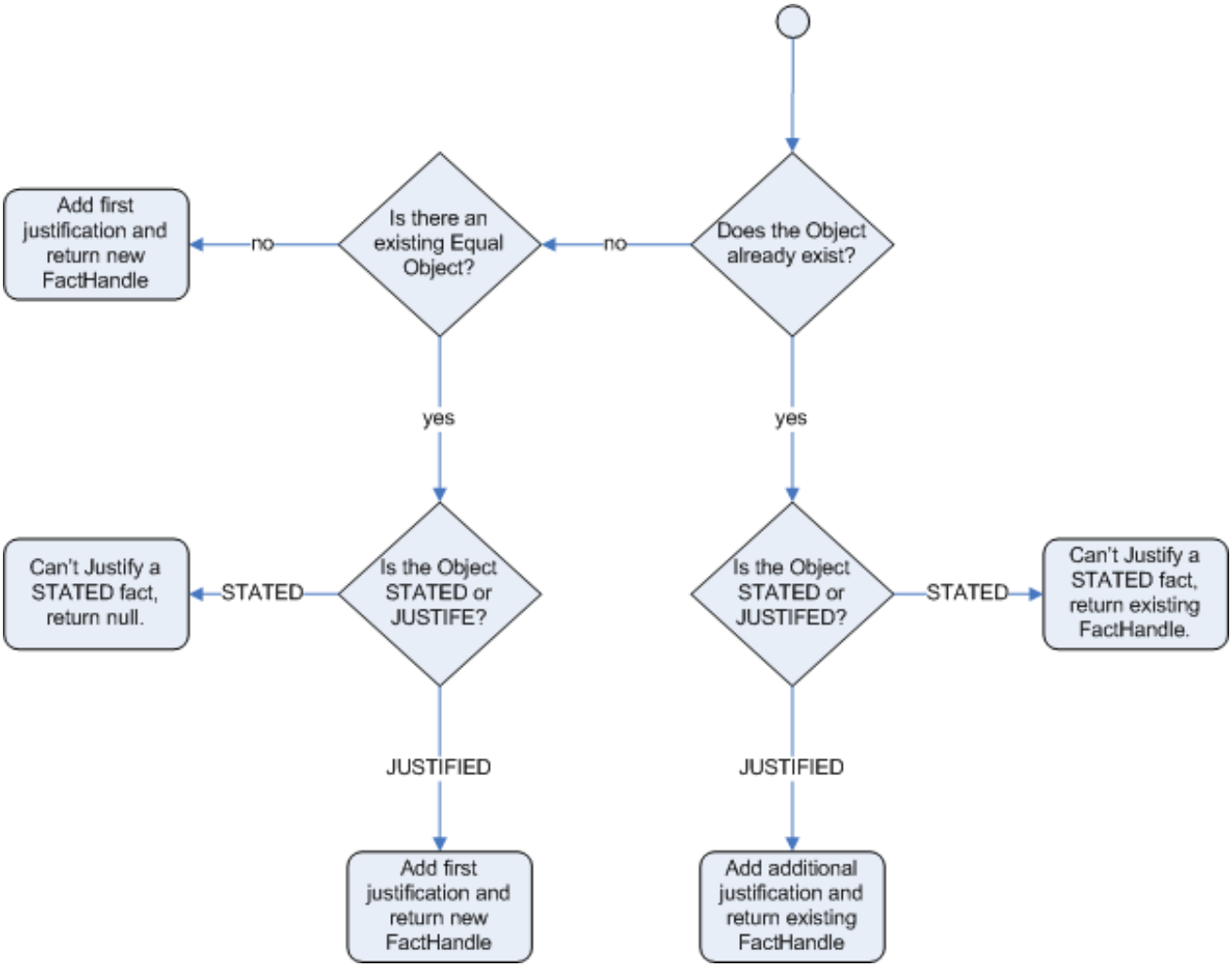


Figure 6.2. Logical Assertion

IMPORTANT

For the Truth Maintenance System and logical assertions to work, your fact objects (POJOs) must override **`equals`** and **`hashCode`** methods from **`java.lang.Object`** as per the Java standard.Two objects are equal if and only if their equals methods return true for each other and if their **`hashCode`** methods return the same values. For more information, refer the Java API documentation.

6.4.1. Example Illustrating Truth Maintenance

This example illustrates how the Truth Maintenance System helps in the inference mechanism. The following rules provides information on basic policies on issuing child and adult bus passes.

```
rule "Issue Child Bus Pass"
when
  $p : Person( age < 16 )
then
  insert(new ChildBusPass( $p ) );
end

rule "Issue Adult Bus Pass"
when
  $p : Person( age >= 16 )
```

```
then
  insert(new AdultBusPass( $p ) );
end
```

These rules are monolithic and provide poor separation of concerns. The truth maintenance mechanism in an inference engine makes the system become more robust and have a clear separation of concerns. For example, the following rule uses logical insertion of facts, which makes the fact dependent on the truth of the when clause:

```
rule "Infer Child"
when
  $p : Person( age < 16 )
then
  insertLogical( new IsChild( $p ) )
end

rule "Infer Adult" when
  $p : Person( age >= 16 )
then
  insertLogical( new IsAdult( $p ) )
end
```

When the condition in the rule is false, the fact is automatically retracted. This works particularly well as the two rules are mutually exclusive. So in the above rules, if the person is under 16 years, it inserts an ***IsChild*** fact. Once the person is 16 years or above, the ***IsChild*** fact is automatically retracted and the ***IsAdult*** fact inserted.

Now the two rules for issuing child and adult bus pass can logically insert the ***ChildBusPass*** and ***AdultBusPass*** facts, as the Truth Maintenance System supports chaining of logical insertions for a cascading set of retracts. Here is how the logical insertion is done:

```
rule "Issue Child Bus Pass"
when
  $p : Person( )
    IsChild( person == $p )
then
  insertLogical(new ChildBusPass( $p ) );
end

rule "Issue Adult Bus Pass"
when
  $p : Person( age >= 16 )
    IsAdult( person =$p )
then
    insertLogical(new AdultBusPass( $p ) );
end
```

When a person turns 16 years old, the ***IsChild*** fact as well as the person's ***ChildBusPass*** fact is retracted. To these set of conditions, you can relate another rule which states that a person must return the child pass after turning 16 years old. So when the Truth Maintenance System automatically retracts the ***ChildBusPass*** object, this rule triggers and sends a request to the person:

```
rule "Return ChildBusPass Request"
when
  $p : Person( )
```

```
        not( ChildBusPass( person == $p ) )
then
  requestChildBusPass( $p );
end
```

6.5. USING DECISION TABLES IN SPREADSHEETS

Decision tables are a way of representing conditional logic in a precise manner, and they are well suited to *business* level rules.

6.5.1. Decision Tables in Spreadsheets

JBoss BRMS supports managing rules in a spreadsheet format. Supported formats are Excel (XLS) and CSV. This means that a variety of spreadsheet programs (such as Microsoft Excel, OpenOffice.org Calc, and others) can be utilized.

NOTE

Use XLS format for decision tables if you are building and uploading them using Business Central. Business Central does not support decision tables in CSV format.

6.5.2. OpenOffice Example

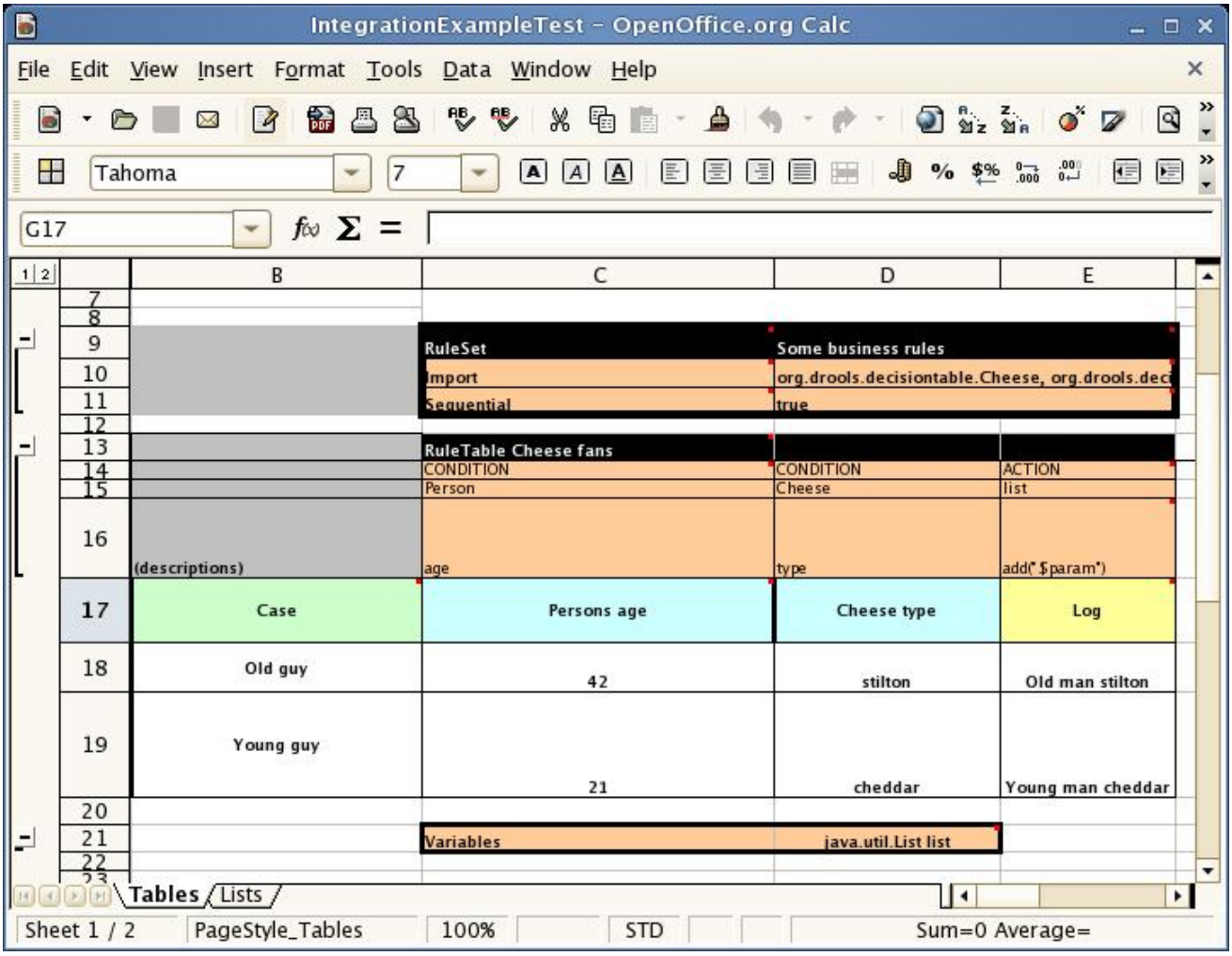
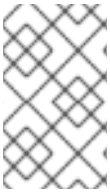


Figure 6.3. OpenOffice Screenshot

In the above examples, the technical aspects of the decision table have been collapsed away (using a standard spreadsheet feature).

The rules start from row 17, with each row resulting in a rule. The conditions are in columns C, D, E, etc., and the actions are off-screen. The values' meanings are indicated by the headers in Row 16. (Column B is just a description.)



NOTE

Although the decision tables look like they process top down, this is not necessarily the case. Ideally, rules are authored without regard for the order of rows. This makes maintenance easier, as rows will not need to be shifted around all the time.

6.5.3. Rules and Spreadsheets

Rules inserted into rows

As each row is a rule, the same principles apply as with written code. As the rule engine processes the facts, any rules that match may fire.

Agendas

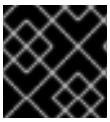
It is possible to clear the agenda when a rule fires and simulate a very simple decision table where only the first match effects an action.

Multiple tables

You can have multiple tables on one spreadsheet. This way, rules can be grouped where they share common templates, but are still all combined into one rule package.

6.5.4. The RuleTable Keyword

When using decision tables, the spreadsheet searches for the *RuleTable* keyword to indicate the start of a rule table (both the starting row and column).



IMPORTANT

Keywords should all be in the same column.

6.5.5. The RuleSet Keyword

The *RuleSet* keyword indicates the name to be used in the *rule package* that will encompass all the rules. This name is optional, using a default, but it *must* have the *RuleSet* keyword in the cell immediately to the right.

6.5.6. Rule Template Example

Rule Templates use tabular data source as a source of rule data and populate a template to generate many rules. With Rule Templates, the data is separated from the rule and there are no restrictions on which part of the rule is data-driven. So it allows you to do everything that is possible in decision tables, and in addition to that, you can also:

- Store your data in a database (or any other format)

- Conditionally generate rules based on the values in the data
- Use data for any part of your rules (such as condition operator, class name, and property name)
- Run different templates over the same data

Consider the sample template data below. In case of a regular decision table, there would be hidden rows before row 1 and between rows 1 and 2 containing rule metadata. With rule templates, the data is completely separate from the rules.

Case	Persons age	Cheese type	Log
Old guy	42	stilton	Old man stilton
Young guy	21	cheddar	Young man cheddar

Figure 6.4. Template Data

So you can apply multiple rule templates to the same data and your data is not tied to your rules at all. Here is a sample rule template:

6.5.7. Rule Templates

```
1  template header
2  age
3  type
4  log
5
6  package org.drools.examples.templates;
7
8  global java.util.List list;
9
10 template "cheesefans"
11
12 rule "Cheese fans_@{row.rowNumber}"
13 when
14    Person(age == @{age})
15    Cheese(type == "@{type}")
16 then
17    list.add("@{log}");
18 end
19
20 end template
```

Figure 6.5. Rule Template

- Line 1: All rule templates start with template header.
- Lines 2-4: Following the header is the list of columns in the order they appear in the data. In this case we are calling the first column age, the second type and the third log.
- Line 5: An empty line signifies the end of the column definitions.

- Lines 6-9: Standard rule header text. This is standard rule DRL and will appear at the top of the generated DRL. Put the package statement and any imports and global and function definitions into this section.
- Line 10: The keyword template signals the start of a rule template. There can be more than one template in a template file, but each template must have a unique name.
- Lines 11-18: The rule template.
- Line 20: The keywords end template signify the end of the template.

The rule templates rely on MVEL to do substitution using the syntax **@{token_name}**. The built-in expression **@{row.rowNumber}** gives a unique number for each row of data and enables you to generate unique rule names. For each row of data, a rule is generated with the values in the data substituted for the tokens in the template. With the example data above, the following rule file is generated:

```
package org.drools.examples.templates;

global java.util.List list;

rule "Cheese fans_1"
when
  Person(age == 42)
  Cheese(type == "stilton")
then
  list.add("Old man stilton");
end

rule "Cheese fans_2"
when
  Person(age == 21)
  Cheese(type == "cheddar")
then
  list.add("Young man cheddar");
end
```

Use the following code to run the rule:

```
DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
dtableconfiguration.setInputType( DecisionTableInputType.XLS );

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

kbuilder.add( ResourceFactory.newClassPathResource( getSpreadsheetName(),
                                                    getClass() ),
              ResourceType.DTABLE,
              dtableconfiguration );
```

6.5.7. Data-Defining Cells

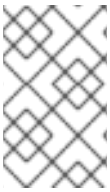
There are two types of rectangular areas *defining data* that is used for generating a DRL file. One, marked by a cell labelled **RuleSet**, defines all DRL items except rules. The other one may occur repeatedly and is to the right and below a cell whose contents begin with **RuleTable**. These areas

represent the actual decision tables, each area resulting in a set of rules of similar structure.

A Rule Set area may contain cell pairs, one below the **`RuleSet`** cell and containing a keyword designating the kind of value contained in the other one that follows in the same row.

6.5.8. Rule Table Columns

The columns of a Rule Table area define patterns and constraints for the left hand sides of the rules derived from it, actions for the consequences of the rules, and the values of individual rule attributes. A Rule Table area should contain one or more columns, both for conditions and actions, and an arbitrary selection of columns for rule attributes, at most one column for each of these. The first four rows following the row with the cell marked with **`RuleTable`** are earmarked as header area, mostly used for the definition of code to construct the rules. It is any additional row below these four header rows that spawns another rule, with its data providing for variations in the code defined in the Rule Table header.



NOTE

All keywords are case insensitive.

Only the first worksheet is examined for decision tables.

6.5.9. Rule Set Entries

Entries in a Rule Set area may define DRL constructs (except rules), and specify rule attributes. While entries for constructs may be used repeatedly, each rule attribute may be given at most once, and it applies to all rules unless it is overruled by the same attribute being defined within the Rule Table area.

Entries must be given in a vertically stacked sequence of cell pairs. The first one contains a keyword and the one to its right the value. This sequence of cell pairs may be interrupted by blank rows or even a Rule Table, as long as the column marked by **`RuleSet`** is upheld as the one containing the keyword.

6.5.10. Entries in the Rule Set Area

Table 6.1. Entries in the Rule Set area

Keyword	Value	Usage
RuleSet	The package name for the generated DRL file. Optional, the default is **`rule_table`**.	Must be First entry.
Sequential	"true" or "false". If "true", then salience is used to ensure that rules fire from the top down.	Optional, at most once. If omitted, no firing order is imposed.
EscapeQuotes	"true" or "false". If "true", then quotation marks are escaped so that they appear literally in the DRL.	Optional, at most once. If omitted, quotation marks are escaped.
Import	A comma-separated list of Java classes to import.	Optional, may be used repeatedly.

Keyword	Value	Usage
Variables	Declarations of DRL globals, i.e., a type followed by a variable name. Multiple global definitions must be separated with a comma.	Optional, may be used repeatedly.
Functions	One or more function definitions, according to DRL syntax.	Optional, may be used repeatedly.
Queries	One or more query definitions, according to DRL syntax.	Optional, may be used repeatedly.
Declare	One or more declarative types, according to DRL syntax.	Optional, may be used repeatedly.

6.5.11. Rule Attribute Entries in the Rule Set Area

IMPORTANT

Rule attributes specified in a Rule Set area will affect all rule assets in the same package (not only in the spreadsheet). Unless you are sure that the spreadsheet is the only one rule asset in the package, the recommendation is to specify rule attributes not in a Rule Set area but in a Rule Table columns for each rule instead.

Table 6.2. Rule Attribute Entries in the Rule Set Area

Keyword	Initial	Value
PRIORITY	P	An integer defining the "salience" value for the rule. Overridden by the "Sequential" flag.
DURATION	D	A long integer value defining the "duration" value for the rule.
TIMER	T	A timer definition. See "Timers" section.
CALENDARS	E	A calendars definition. See "Calendars" section.
NO-LOOP	U	A Boolean value. "true" inhibits looping of rules due to changes made by its consequence.

Keyword	Initial	Value
LOCK-ON-ACTIVE	L	A Boolean value. "true" inhibits additional activations of all rules with this flag set within the same ruleflow or agenda group.
AUTO-FOCUS	F	A Boolean value. "true" for a rule within an agenda group causes activations of the rule to automatically give the focus to the group.
ACTIVATION-GROUP	X	A string identifying an activation (or XOR) group. Only one rule within an activation group will fire, i.e., the first one to fire cancels any existing activations of other rules within the same group.
AGENDA-GROUP	G	A string identifying an agenda group, which has to be activated by giving it the "focus", which is one way of controlling the flow between groups of rules.
RULEFLOW-GROUP	R	A string identifying a rule-flow group.
DATE-EFFECTIVE	V	A string containing a date and time definition. A rule can only activate if the current date and time is after DATE-EFFECTIVE attribute.
DATE-EXPIRES	Z	A string containing a date and time definition. A rule cannot activate if the current date and time is after the DATE-EXPIRES attribute.

6.5.12. The RuleTable Cell

All Rule Tables begin with a cell containing "RuleTable", optionally followed by a string within the same cell. The string is used as the initial part of the name for all rules derived from this Rule Table, with the row number appended for distinction. (This automatic naming can be overridden by using a NAME column.) All other cells defining rules of this Rule Table are below and to the right of this cell.

6.5.13. Column Types

The next row after the RuleTable cell defines the column type. Each column results in a part of the condition or the consequence, or provides some rule attribute, the rule name or a comment. Each attribute column may be used at most once.

6.5.14. Column Headers in the Rule Table

Table 6.3. Column Headers in the Rule Table

Keyword	Initial	Value	Usage
NAME	N	Provides the name for the rule generated from that row. The default is constructed from the text following the RuleTable tag and the row number.	At most one column
DESCRIPTION	I	A text, resulting in a comment within the generated rule.	At most one column
CONDITION	C	Code snippet and interpolated values for constructing a constraint within a pattern in a condition.	At least one per rule table
ACTION	A	Code snippet and interpolated values for constructing an action for the consequence of the rule.	At least one per rule table
METADATA	@	Code snippet and interpolated values for constructing a metadata entry for the rule.	Optional, any number of columns

6.5.15. Conditional Elements

Given a column headed CONDITION, the cells in successive lines result in a conditional element.

- Text in the first cell below CONDITION develops into a pattern for the rule condition, with the snippet in the next line becoming a constraint. If the cell is merged with one or more neighbours, a single pattern with multiple constraints is formed: all constraints are combined into a parenthesized list and appended to the text in this cell. The cell may be left blank, which means that the code snippet in the next row must result in a valid conditional element on its own.

  To include a pattern without constraints, you can write the pattern in front of the text for another pattern.

The pattern may be written with or without an empty pair of parentheses. A "from" clause may be appended to the pattern.

If the pattern ends with "eval", code snippets are supposed to produce boolean expressions for inclusion into a pair of parentheses after "eval".

- Text in the second cell below CONDITION is processed in two steps.

  1. The code snippet in this cell is modified by interpolating values from cells farther down in the column. If you want to create a constraint consisting of a comparison using "==" with the value from the cells below, the field selector alone is sufficient. Any other comparison operator must be specified as the last item within the snippet, and the value from the cells below is appended. For all other constraint forms, you must mark the position for including the contents of a cell with the symbol **`$param`**. Multiple insertions are possible by using the symbols **`$1`**, **`$2`**, etc., and a comma-separated list of values in the cells below.

     A text according to the pattern **`forall(`***delimiter***`){`***snippet***`}`** is expanded by repeating the *snippet* once for each of the values of the comma-separated list of values in each of the cells below, inserting the value in place of the symbol **`$`** and by joining these expansions by the given *delimiter*. Note that the forall construct may be surrounded by other text.

  2. If the cell in the preceding row is not empty, the completed code snippet is added to the conditional element from that cell. A pair of parentheses is provided automatically, as well as a separating comma if multiple constraints are added to a pattern in a merged cell.

     If the cell above is empty, the interpolated result is used as is.

- Text in the third cell below CONDITION is for documentation only. It should be used to indicate the column's purpose to a human reader.

- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the conditional element or constraint for this rule.

6.5.16. Action Statements

Given a column headed ACTION, the cells in successive lines result in an action statement:

- Text in the first cell below ACTION is optional. If present, it is interpreted as an object reference.

- Text in the second cell below ACTION is processed in two steps.

  1. The code snippet in this cell is modified by interpolating values from cells farther down in the column. For a singular insertion, mark the position for including the contents of a cell with the symbol **`$param`**. Multiple insertions are possible by using the symbols **`$1`**, **`$2`**, etc., and a comma-separated list of values in the cells below.

     A method call without interpolation can be achieved by a text without any marker symbols. In this case, use any non-blank entry in a row below to include the statement.

     The forall construct is available here, too.

  2. If the first cell is not empty, its text, followed by a period, the text in the second cell and a terminating semicolon are stringed together, resulting in a method call which is added as an action statement for the consequence.

     If the cell above is empty, the interpolated result is used as is.

- Text in the third cell below ACTION is for documentation only. It should be used to indicate the column's purpose to a human reader.
- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the action statement for this rule.

NOTE

Using **`$1`** instead of **`$param`** will fail if the replacement text contains a comma.

6.5.17. Metadata Statements

Given a column headed METADATA, the cells in successive lines result in a metadata annotation for the generated rules:

- Text in the first cell below METADATA is ignored.
- Text in the second cell below METADATA is subject to interpolation, as described above, using values from the cells in the rule rows. The metadata marker character @ is prefixed automatically, and should not be included in the text for this cell.
- Text in the third cell below METADATA is for documentation only. It should be used to indicate the column's purpose to a human reader.
- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the metadata annotation for this rule.

6.5.18. Interpolating Cell Data Example

- If the template is **`Foo(bar == $param)`** and the cell is **`42`**, then the result is **`Foo(bar == 42)`**.
- If the template is **`Foo(bar < $1, baz == $2)`** and the cell contains **`42,43`**, the result will be **`Foo(bar < 42, baz ==43)`**.
- The template **`forall(&&){bar != $}`** with a cell containing **`42,43`** results in **`bar != 42 && bar != 43`**.

6.5.19. Tips for Working Within Cells

- Multiple package names within the same cell must be comma-separated.
- Pairs of type and variable names must be comma-separated.
- Functions must be written as they appear in a DRL file. This should appear in the same column as the "RuleSet" keyword. It can be above, between or below all the rule rows.
- You can use Import, Variables, Functions and Queries repeatedly instead of packing several definitions into a single cell.
- Trailing insertion markers can be omitted.
- You can provide the definition of a binding variable.

- Anything can be placed in the object type row. Apart from the definition of a binding variable, it could also be an additional pattern that is to be inserted literally.
- The cell below the ACTION header can be left blank. Using this style, anything can be placed in the consequence, not just a single method call. (The same technique is applicable within a CONDITION column.)

6.5.20. The SpreadsheetCompiler Class

The **`SpreadsheetCompiler`** class is the main class used with API spreadsheet-based decision tables in the drools-decisiontables module. This class takes spreadsheets in various formats and generates rules in DRL.

The **`SpreadsheetCompiler`** can be used to generate partial rule files and assemble them into a complete rule package after the fact. This allows the separation of technical and non-technical aspects of the rules if needed.

6.5.21. Using Spreadsheet-Based Decision Tables

Procedure 6.5. Task

1. Generate a sample spreadsheet that you can use as the base.
2. If the JBoss BRMS plug-in is being used, use the wizard to generate a spreadsheet from a template.
3. Use an XSL-compatible spreadsheet editor to modify the XSL.

6.5.22. Lists

In Excel, you can create **`lists`** of values. These can be stored in other worksheets to provide valid lists of values for cells.

6.5.23. Revision Control

When changes are being made to rules over time, older versions are archived. Some applications in JBoss BRMS provide a limited ability to keep a history of changes, but it is recommended to use an alternative means of revision control.

6.5.24. Tabular Data Sources

A tabular data source can be used as a source of rule data. It can populate a template to generate many rules. This can allow both for more flexible spreadsheets, but also rules in existing databases for instance (at the cost of developing the template up front to generate the rules).

6.6. LOGGING

The logging feature enables you to investigate what the Rule Engine does at the back-end. The rule engine uses Java logging API SLF4J for logging. The underlying logging backend can be Logback, Apache Commons Logging, Log4j, or java.util.logging. You can add a dependency to the logging adaptor for your logging framework of choice.

Here is an example of how to use Logback by adding a Maven dependency:

-

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.x</version>
</dependency>
```

NOTE

If you are developing for an ultra light environment, use slf4j-nop or slf4j-simple.

6.6.1. Configuring Logging Level

Here is an example of how you can configure the logging level on the package org.drools in your **`logback.xml`** file when you are using Logback:

```
<configuration>
   <logger name="org.drools" level="debug"/>
    ...
    ...
<configuration>
```

Here is an example of how you can configure the logging level in your **`log4j.xml`** file when you are using Log4J:

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

    <category name="org.drools">
      <priority value="debug" />
    </category>
    ...

    ...

</log4j:configuration>
```

CHAPTER 7. COMPLEX EVENT PROCESSING

7.1. 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).

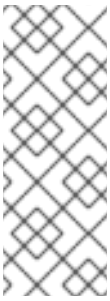
7.2. EVENTS

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.

7.2.1. 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 7.1. 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 7.2. 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 Languages section.

7.2.2. 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 7.3. 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 7.4. 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 7.5. 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 7.6. 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 7.7. Declaring the Expiration Offset for the VoiceCall Events

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

7.3. CLOCK IMPLEMENTATION IN COMPLEX EVENT PROCESSING

7.3.1. 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.

7.3.2. 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***:

```
import org.kie.api.KieServices.Factory;
import org.kie.api.runtime.KieSessionConfiguration;

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***:

```
import org.kie.api.runtime.KieSessionConfiguration;
import org.kie.api.KieServices.Factory;

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

This example shows how to control the pseudo-clock:

```
import org.kie.api.runtime.KieSessionConfiguration;
import org.kie.api.KieServices.Factory;
import org.kie.api.runtime.KieSession;
import org.kie.api.time.SessionClock;
import org.kie.api.runtime.rule.FactHandle;

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 );
```

7.4. 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.

7.4.1. 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:

```
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices.Factory;

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

The equivalent property follows:

```
drools.eventProcessingMode = cloud
```

7.4.2. 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:

```
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices.Factory;

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

The equivalent property follows:

```
drools.eventProcessingMode = stream
```

7.5. 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.

7.5.1. 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 7.8. 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 7.9. 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 7.10. Inserting Facts into an Entry Point

```
import org.kie.api.runtime.KieSession;

   // 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 );
```

7.5.2. 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 7.11. 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 7.12. 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 7.13. 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 7.14. 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
```

7.6. TEMPORAL OPERATIONS

7.6.1. 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).

7.6.2. Temporal Operations

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

- After
- Before
- Coincides
- During
- Finishes
- Finishes By
- Includes
- Meets

- Met By
- Overlaps
- Overlapped By
- Starts
- Started By

7.6.3. 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 )
```

7.6.4. 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 )
```

7.6.5. 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.

7.6.6. 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.

7.6.7. 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.

7.6.8. 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.

7.6.9. 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.

7.6.10. 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.

7.6.11. 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.

7.6.12. 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.

7.6.13. 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.

7.6.14. 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.

7.6.15. 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.

7.7. SLIDING WINDOWS

7.7.1. 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 7.15. 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.

7.7.2. 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 7.16. 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.

7.8. 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.

7.8.1. Explicit Expiration

Explicit expiration is set with a **`declare`** statement and the metadata @expires tag.

For example:

Example 7.17. 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.

7.8.2. 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 7.18. 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.

CHAPTER 8. WORKING WITH RULES

8.1. WHAT'S IN A RULE FILE

8.1.1. 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.

8.1.2. The structure of a rule file

The overall structure of a rule file is the following:

Example 8.1. 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.

8.2. HOW RULES OPERATE ON FACTS

Facts are domain model objects that BRMS uses to evaluate conditions and execute consequences. A rule specifies that when a particular set of conditions occur, then the specified list of actions must be executed. The inference engine matches facts against rules, and when matches are found, rule actions are placed on the agenda. The agenda is the place where rules are queued ready to have their actions fired. The rule engine then determines which eligible rules on the agenda must fire.

8.2.1. Rule files Accessing the Working Memory

The working memory is a stateful object that provides temporary storage and manipulation of facts. The working memory includes an API that contains the following functions that allow access to working memory from rules files:

- **`update(object, handle)`**

This method is used to tell the engine that an object has changed and rules may need to be reconsidered.

- **`update(object)`**

  In this method, the KieSession looks up the fact handle, via an identity check, for the passed object. Although, if property change listeners are provided to the JavaBeans that are inserted into the engine, it is possible to avoid the need to call **`update()`** method when the object changes.

- **`insert(new <method name>())`**

  This method places a new object into the working memory.

- **`retract(handle)`**

  This method removes an object from working memory. It is mapped to the delete method in a KieSession.

- **`insertLogical(new <method name>())`**

  This method is similar to insert, but the object is automatically retracted from the working memory when there are no more facts to support the truth of the currently firing rule.

- **`halt()`**

  This method terminates rule execution immediately. This is required for returning control to the point where the current session is put to work with **`fireUntilHalt()`** method.

- **`getKieRuntime()`**

  The full KIE API is exposed through a predefined variable, kcontext, of type RuleContext. Its method **`getKieRuntime()`** delivers an object of type KieRuntime, which in turn provides access to a wealth of methods, many of which are useful for coding the rule logic. The call **`kcontext.getKieRuntime().halt()`** terminates rule execution immediately.

8.3. USING RULE KEYWORDS

8.3.1. 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`**.

8.3.2. 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.

8.3.3. 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:

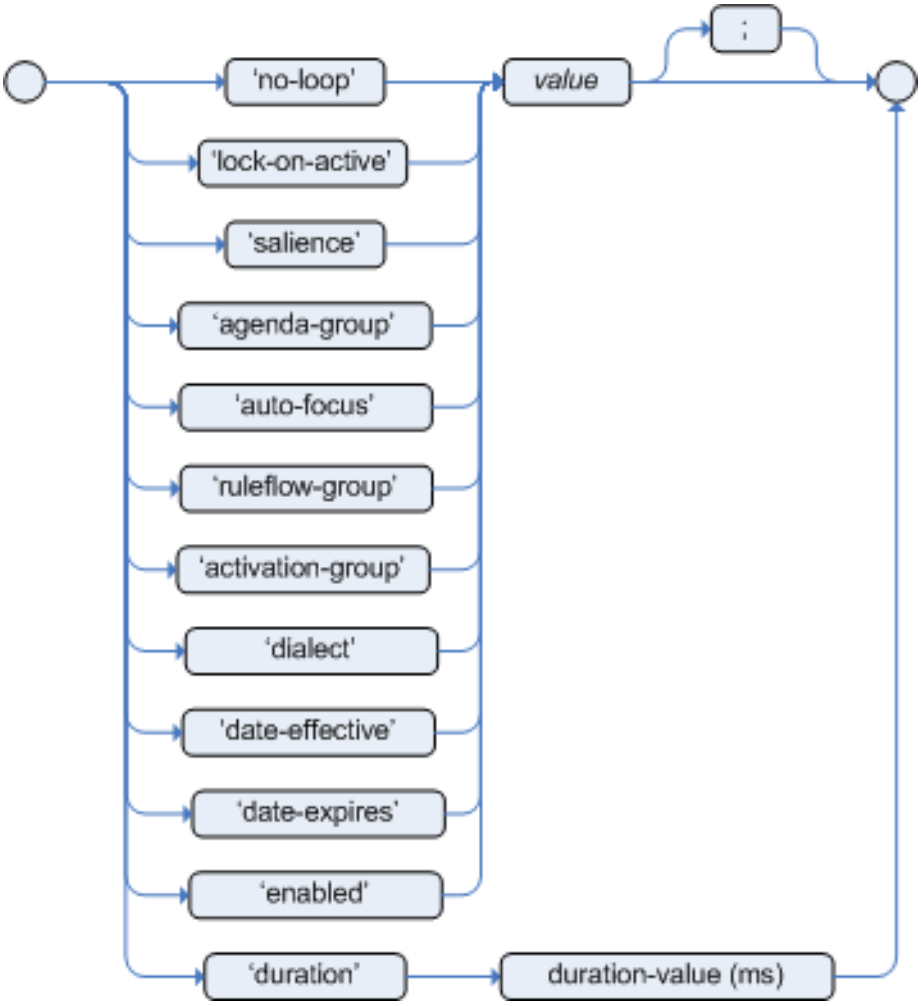


Figure 8.1. Rule Attributes

Table 8.1. 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.

8.4. ADDING COMMENTS TO A RULE FILE

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).

8.4.1. 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
```

8.4.2. 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
```

8.5. ERROR MESSAGES IN RULES

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

8.5.1. Error Message Format

This is the standard error message format.

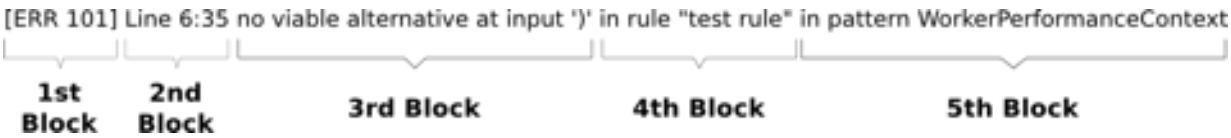


Figure 8.2. Error Message Format Example

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.

8.5.2. Error Messages Description

Table 8.2. 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`	

[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`	

Error Message	Description	Example	

[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`	

8.6. PACKAGING

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).

8.6.1. 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. BRMS automatically imports classes from the Java package of the same name, and also from the package **`java.lang`**.

8.6.2. 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 );
```

8.6.3. 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.

8.6.4. Using Globals with an e-Mail Service

Procedure 8.1. 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.

8.7. FUNCTIONS IN A RULE

Functions are a way to put semantic code in a rule source file, as opposed to in normal Java classes. The main advantage of using functions in a rule is that you can keep the logic all in one place. You can change the functions as needed.

Functions are most useful for invoking actions on the consequence (**`then`**) part of a rule, especially if that particular action is used repeatedly.

A typical function declaration looks like the following:

```
function String hello(String name) {
    return "Hello "+name+"!";
}
```

NOTE

Note that the **`function`** keyword is used, even though it's not technically part of Java. Parameters to the function are defined as for a method. You don't have to have parameters if they are not needed. The return type is defined just like in a regular method.

8.7.1. Function Declaration with Static Method Example

This example of a function declaration shows the static method in a helper class (**`Foo.hello()`**. JBoss BRMS supports the use of function imports, so the following code is all you would need to enter the following:

```
import function my.package.Foo.hello
```

8.7.2. Calling a Function Declaration Example

Irrespective of the way the function is defined or imported, you use a function by calling it by its name, in the consequence or inside a semantic code block. This is shown below:

```
rule "using a static function"
when
    eval( true )
then
    System.out.println( hello( "Bob" ) );
end
```

8.7.3. Type Declarations

Type declarations have two main goals in the rules engine: to allow the declaration of new types, and to allow the declaration of metadata for types.

Table 8.3. Type Declaration Roles

Role	Description
Declaring new types	JBoss BRMS uses plain Java objects as facts out of the box. However, if you wish to define the model directly to the rules engine, you can do so by declaring a new type. You can also declare a new type when there is a domain model already built and you want to complement this model with additional entities that are used mainly during the reasoning process.
Declaring metadata	Facts may have meta information associated to them. Examples of meta information include any kind of data that is not represented by the fact attributes and is consistent among all instances of that fact type. This meta information may be queried at runtime by the engine and used in the reasoning process.

8.7.4. Declaring New Types

To declare a new type, the keyword **`declare`** is used, followed by the list of fields and the keyword **`end`**. A new fact must have a list of fields, otherwise the engine will look for an existing fact class in the classpath and raise an error if not found.

8.7.5. Declaring a New Fact Type Example

In this example, a new fact type called **`Address`** is used. This fact type will have three attributes: **`number`**, **`streetName`** and **`city`**. Each attribute has a type that can be any valid Java type, including any other class created by the user or other fact types previously declared:

```
declare Address
   number : int
   streetName : String
   city : String
end
```

8.7.6. Declaring a New Fact Type Additional Example

This fact type declaration uses a **`Person`** example. **`dateOfBirth`** is of the type **`java.util.Date`** (from the Java API) and **`address`** is of the fact type Address.

```
declare Person
    name : String
    dateOfBirth : java.util.Date
    address : Address
end
```

8.7.7. Using Import Example

This example illustrates how to use the **`import`** feature to avoid he need to use fully qualified class names:

```
import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

8.7.8. Generated Java Classes

When you declare a new fact type, JBoss BRMS generates bytecode that implements a Java class representing the fact type. The generated Java class is a one-to-one Java Bean mapping of the type definition.

8.7.9. Generated Java Class Example

This is an example of a generated Java class using the **`Person`** fact type:

```
public class Person implements Serializable {
    private String name;
    private java.util.Date dateOfBirth;
    private Address address;

    // empty constructor
    public Person() {...}

    // constructor with all fields
    public Person( String name, Date dateOfBirth, Address address ) {...}

    // if keys are defined, constructor with keys
    public Person( ...keys... ) {...}

    // getters and setters
    // equals/hashCode
    // toString
}
```

8.7.10. Using the Declared Types in Rules Example

Since the generated class is a simple Java class, it can be used transparently in the rules like any other fact:

```
rule "Using a declared Type"
when
    $p : Person( name == "Bob" )
then
    // Insert Mark, who is Bob's manager.
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
end
```

8.7.11. Declaring Metadata

Metadata may be assigned to several different constructions in JBoss BRMS, such as fact types, fact attributes and rules. JBoss BRMS uses the at sign ('@') to introduce metadata and it always uses the form:

```
@metadata_key( metadata_value )
```

The parenthesized *metadata_value* is optional.

8.7.12. Working with Metadata Attributes

JBoss BRMS allows the declaration of any arbitrary metadata attribute. Some have special meaning to the engine, while others are available for querying at runtime. JBoss BRMS allows the declaration of metadata both for fact types and for fact attributes. Any metadata that is declared before the attributes of a fact type are assigned to the fact type, while metadata declared after an attribute are assigned to that particular attribute.

8.7.13. Declaring a Metadata Attribute with Fact Types Example

This is an example of declaring metadata attributes for fact types and attributes. There are two metadata items declared for the fact type (**`@author`** and **`@dateOfCreation`**) and two more defined for the name attribute (**`@key`** and **`@maxLength`**). The **`@key`** metadata has no required value, and so the parentheses and the value were omitted:

```
import java.util.Date

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )

    name : String @key @maxLength( 30 )
    dateOfBirth : Date
    address : Address
end
```

8.7.14. The @position Attribute

The **`@position`** attribute can be used to declare the position of a field, overriding the default declared order. This is used for positional constraints in patterns.

8.7.15. @position Example

This is what the @position attribute looks like in use:

```
declare Cheese
    name : String @position(1)
    shop : String @position(2)
    price : int @position(0)
end
```

8.7.16. Predefined Class Level Annotations

Table 8.4. Predefined Class Level Annotations

Annotation	Description
@role(<fact | event>)	This attribute can be used to assign roles to facts and events.
@typesafe(<boolean>)	By default, all type declarations are compiled with type safety enabled. **`@typesafe( false )`** provides a means to override this behavior by permitting a fall-back, to type unsafe evaluation where all constraints are generated as MVEL constraints and executed dynamically. This is useful when dealing with collections that do not have any generics or mixed type collections.
@timestamp(<attribute name>)	Creates a timestamp.
@duration(<attribute name>)	Sets a duration for the implementation of an attribute.
@expires(<time interval>)	Allows you to define when the attribute should expire.
@propertyChangeSupport	Facts that implement support for property changes as defined in the Javabean spec can now be annotated so that the engine register itself to listen for changes on fact properties. .
@propertyReactive	Makes the type property reactive.

8.7.17. @key Attribute Functions

Declaring an attribute as a key attribute has two major effects on generated types:

1. The attribute is used as a key identifier for the type, and thus the generated class implements the **`equals()`** and **`hashCode()`** methods taking the attribute into account when comparing instances of this type.

2. JBoss BRMS generates a constructor using all the key attributes as parameters.

8.7.18. @key Declaration Example

This is an example of @key declarations for a type. JBoss BRMS generates **`equals()`** and **`hashCode()`** methods that checks the ***`firstName`*** and ***`lastName`*** attributes to determine if two instances of *Person* are equal to each other. It does not check the age attribute. It also generates a constructor taking ***`firstName`*** and ***`lastName`*** as parameters:

```
declare Person
    firstName : String @key
    lastName : String @key
    age : int
end
```

8.7.19. Creating an Instance with the Key Constructor Example

This is what creating an instance using the key constructor looks like:

```
Person person = new Person( "John", "Doe" );
```

8.7.20. Positional Arguments

Patterns support positional arguments on type declarations and are defined by the **@position** attribute.

Positional arguments are when you don't need to specify the field name, as the position maps to a known named field. (That is, Person(name == "mark") can be rewritten as Person("mark";).) The semicolon ';' is important so that the engine knows that everything before it is a positional argument. You can mix positional and named arguments on a pattern by using the semicolon ';' to separate them. Any variables used in a positional that have not yet been bound will be bound to the field that maps to that position.

8.7.21. Positional Argument Example

Observe the example below:

```
declare Cheese
    name : String
    shop : String
    price : int
end
```

The default order is the declared order, but this can be overridden using @position

```
declare Cheese
    name : String @position(1)
    shop : String @position(2)
    price : int @position(0)
end
```

8.7.22. The @Position Annotation

The @Position annotation can be used to annotate original pojos on the classpath. Currently only fields on classes can be annotated. Inheritance of classes is supported, but not interfaces of methods.

8.7.23. Example Patterns

These example patterns have two constraints and a binding. The semicolon ';' is used to differentiate the positional section from the named argument section. Variables and literals and expressions using just literals are supported in positional arguments, but not variables:

```
Cheese( "stilton", "Cheese Shop", p; )
Cheese( "stilton", "Cheese Shop"; p : price )
Cheese( "stilton"; shop == "Cheese Shop", p : price )
Cheese( name == "stilton"; shop == "Cheese Shop", p : price )
```

8.8. BACKWARD-CHAINING

8.8.1. Backward-Chaining Systems

Backward-Chaining is a feature recently added to the BRMS Engine. This process is often referred to as derivation queries, and it is not as common compared to reactive systems since BRMS 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.

8.8.2. Cloning Transitive Closures

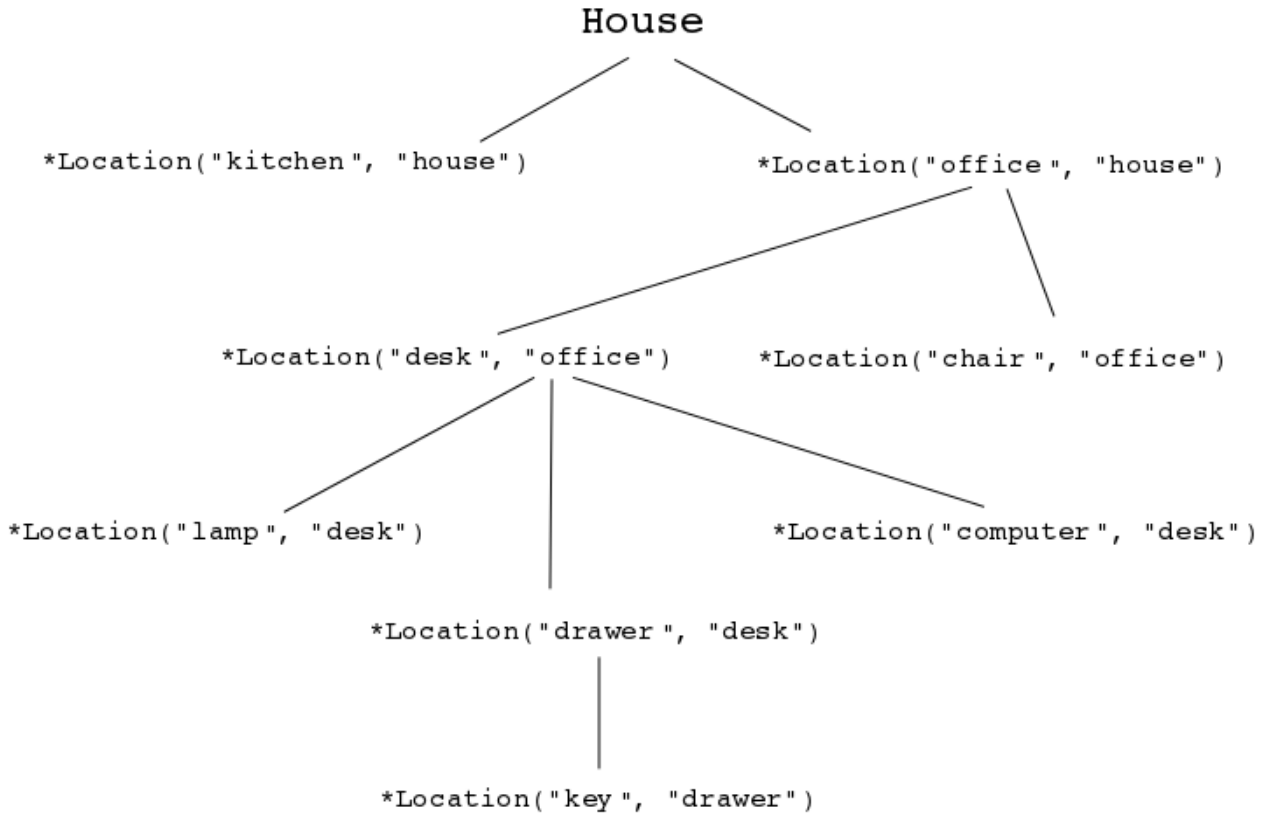


Figure 8.3. Reasoning Graph

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

Procedure 8.2. 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."

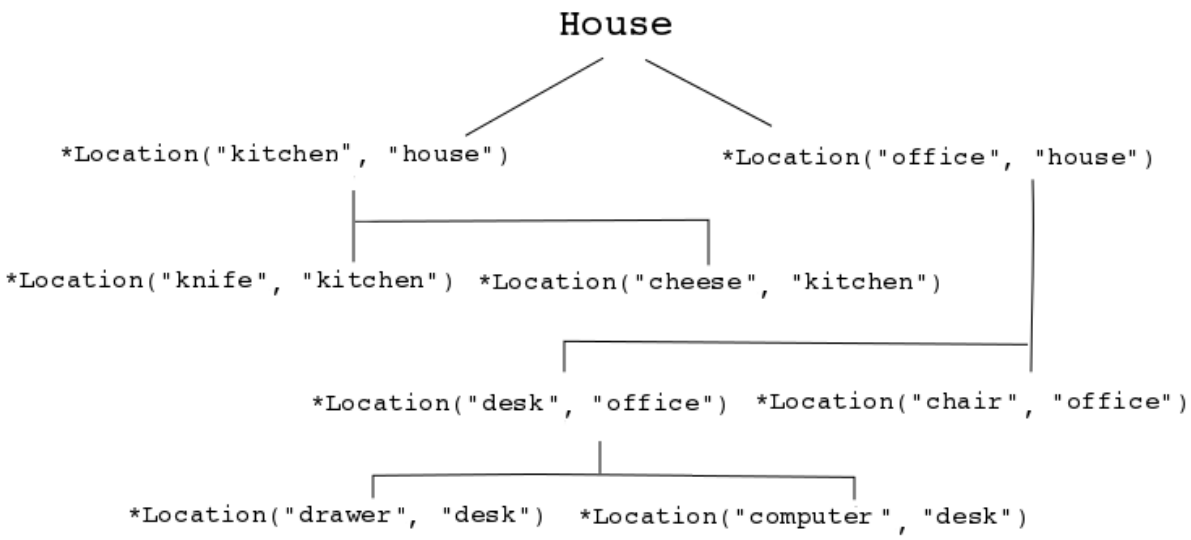


Figure 8.4. Transitive Reasoning Graph of a House.

NOTE

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

8.8.3. Defining a Query

Procedure 8.3. 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

8.8.4. Transitive Closure Example

Procedure 8.4. 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."

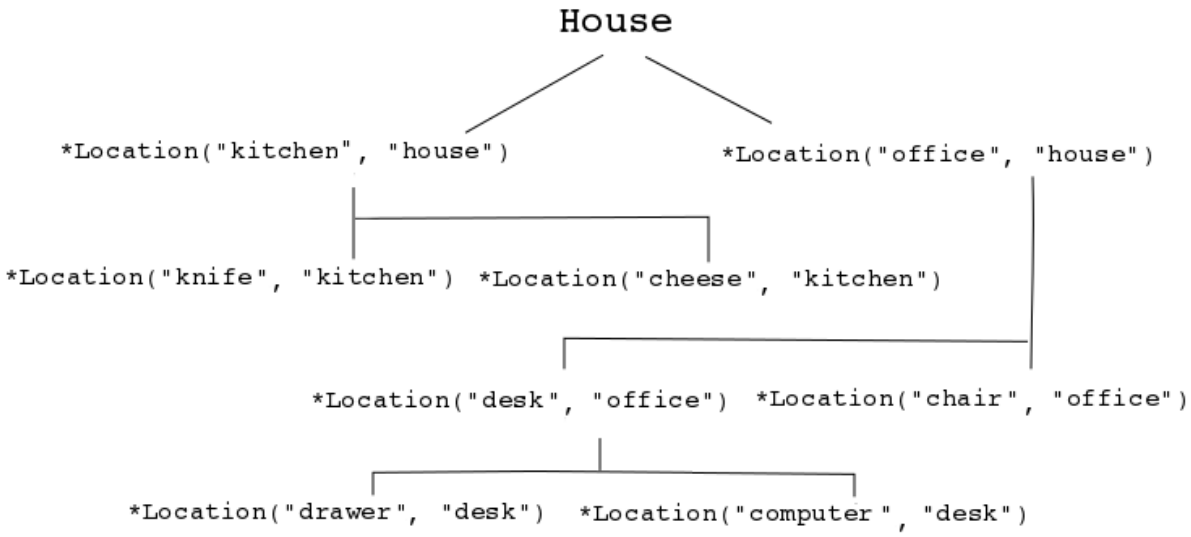


Figure 8.5. Transitive Reasoning Graph of a Drawer.

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**.

8.8.5. Reactive Transitive Queries

Procedure 8.5. 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.

8.8.6. Queries with Unbound Arguments

Procedure 8.6. 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."

8.8.7. Multiple Unbound Arguments

Procedure 8.7. 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.

8.9. TYPE DECLARATION

8.9.1. Declaring Metadata for Existing Types

JBoss BRMS allows the declaration of metadata attributes for existing types in the same way as when declaring metadata attributes for new fact types. The only difference is that there are no fields in that declaration.

8.9.2. Declaring Metadata for Existing Types Example

This example shows how to declare metadata for an existing type:

```
import org.drools.examples.Person

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

8.9.3. Declaring Metadata Using a Fully Qualified Class Name Example

This example shows how you can declare metadata using the fully qualified class name instead of using the import annotation:

```
declare org.drools.examples.Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

8.9.4. Parametrized Constructors for Declared Types Example

For a declared type like the following:

```
declare Person
    firstName : String @key
    lastName : String @key
    age : int
end
```

The compiler will implicitly generate 3 constructors: one without parameters, one with the @key fields and one with all fields.

```
Person() // parameterless constructor
Person( String firstName, String lastName )
Person( String firstName, String lastName, int age )
```

8.9.5. Non-Typesafe Classes

The @typesafe(<boolean>) annotation has been added to type declarations. By default all type declarations are compiled with type safety enabled. @typesafe(false) provides a means to override this behaviour by permitting a fall-back, to type unsafe evaluation where all constraints are generated as MVEL constraints and executed dynamically. This is useful when dealing with collections that do not have any generics or mixed type collections.

8.9.6. Accessing Declared Types from the Application Code

Sometimes applications need to access and handle facts from the declared types. In such cases, JBoss BRMS provides a simplified API for the most common fact handling the application wishes to do. A declared fact belongs to the package where it is declared.

8.9.7. Declaring a Type

This illustrates the process of declaring a type:

```
package org.drools.examples

import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

8.9.8. Handling Declared Fact Types Through the API Example

This example illustrates the handling of declared fact types through the API:

```
// get a reference to a knowledge base with a declared type:
Kie kbase = ...

// get the declared FactType
FactType personType = kbase.getFactType( "org.drools.examples",
                                         "Person" );

// handle the type as necessary:
// create instances:
Object bob = personType.newInstance();

// set attributes values
personType.set( bob,
                "name",
                "Bob" );
personType.set( bob,
                "age",
                42 );

// insert fact into a session
KieSession ksession = ...
ksession.insert( bob );
```

```
ksession.fireAllRules();

// read attributes
String name = personType.get( bob, "name" );
int age = personType.get( bob, "age" );
```

The API also includes other helpful methods, like setting all the attributes at once, reading values from a Map, or reading all attributes at once, into a Map.

8.9.9. Type Declaration Extends

Type declarations support the 'extends' keyword for inheritance. To extend a type declared in Java by a DRL declared subtype, repeat the supertype in a declare statement without any fields.

8.9.10. Type Declaration Extends Example

This illustrates the use of the **`extends`** annotation:

```
import org.people.Person

declare Person
end

declare Student extends Person
    school : String
end

declare LongTermStudent extends Student
    years : int
    course : String
end
```

8.9.11. Traits

Traits allow you to model multiple dynamic types which do not fit naturally in a class hierarchy. A trait is an interface that can be applied (and eventually removed) to an individual object at runtime. To create a trait out of an interface, a **`@format(trait)`** annotation is added to its declaration in DRL.

8.9.12. Traits Example

```
declare GoldenCustomer
    @format(trait)
    // fields will map to getters/setters
    code     : String
    balance  : long
    discount : int
    maxExpense : long
end
```

In order to apply a trait to an object, the new don keyword is added:

```
when
```

```
    $c : Customer()
then
    GoldenCustomer gc = don( $c, Customer.class );
end
```

8.9.13. Core Objects and Traits

When a core object dons a trait, a proxy class is created on the fly (one such class will be generated lazily for each core/trait class combination). The proxy instance, which wraps the core object and implements the trait interface, is inserted automatically and will possibly activate other rules. An immediate advantage of declaring and using interfaces, getting the implementation proxy for free from the engine, is that multiple inheritance hierarchies can be exploited when writing rules. The core classes, however, need not implement any of those interfaces statically, also facilitating the use of legacy classes as cores. Any object can don a trait. For efficiency reasons, however, you can add the @Traitable annotation to a declared bean class to reduce the amount of glue code that the compiler will have to generate. This is optional and will not change the behavior of the engine.

8.9.14. @Traitable Example

This illustrates the use of the @traitable annotation:

```
declare Customer
    @Traitable
    code    : String
    balance : long
end
```

8.9.15. Writing Rules with Traits

The only connection between core classes and trait interfaces is at the proxy level. (That is, a trait is not specifically tied to a core class.) This means that the same trait can be applied to totally different objects. For this reason, the trait does not transparently expose the fields of its core object. When writing a rule using a trait interface, only the fields of the interface will be available, as usual. However, any field in the interface that corresponds to a core object field, will be mapped by the proxy class.

8.9.16. Rules with Traits Example

This example illustrates the trait interface being mapped to a field:

```
when
    $o: OrderItem( $p : price, $code : custCode )
    $c: GoldenCustomer( code == $code, $a : balance, $d: discount )
then
    $c.setBalance( $a - $p*$d );
end
```

8.9.17. Hidden Fields

Hidden fields are fields in the core class not exposed by the interface.

8.9.18. The Two-Part Proxy

The *two-part proxy* has been developed to deal with soft and hidden fields which are not processed intuitively. Internally, proxies are formed by a proper proxy and a wrapper. The former implements the interface, while the latter manages the core object fields, implementing a name/value map to supports soft fields. The proxy uses both the core object and the map wrapper to implement the interface, as needed.

8.9.19. Wrappers

The *wrapper* provides a looser form of typing when writing rules. However, it has also other uses. The wrapper is specific to the object it wraps, regardless of how many traits have been attached to an object. All the proxies on the same object will share the same wrapper. Additionally, the wrapper contains a back-reference to all proxies attached to the wrapped object, effectively allowing traits to see each other.

8.9.20. Wrapper Example

This is an example of using the wrapper:

```
when
    $sc : GoldenCustomer( $c : code, // hard getter
                          $maxExpense : maxExpense > 1000 // soft getter
    )
then
    $sc.setDiscount( ... ); // soft setter
end
```

8.9.21. Wrapper with isA Annotation Example

This illustrates a wrapper in use with the isA annotation:

```
$sc : GoldenCustomer( $maxExpense : maxExpense > 1000,
                      this isA "SeniorCustomer"
)
```

8.9.22. Removing Traits

The business logic may require that a trait is removed from a wrapped object. There are two ways to do so:

Logical don

Results in a logical insertion of the proxy resulting from the traiting operation.

```
then
    don( $x, // core object
         Customer.class, // trait class
         true // optional flag for logical insertion
    )
```

The shed keyword

The shed keyword causes the retraction of the proxy corresponding to the given argument type

```
then
    Thing t = shed( $x, GoldenCustomer.class )
```

–

This operation returns another proxy implementing the org.drools.factmodel.traits.Thing interface, where the getFields() and getCore() methods are defined. Internally, all declared traits are generated to extend this interface (in addition to any others specified). This allows to preserve the wrapper with the soft fields which would otherwise be lost.

8.9.23. Rule Syntax Example

This is an example of the syntax you should use when creating a rule:

```
rule "<name>"
    <attribute>*
when
    <conditional element>*
then
    <action>*
end
```

8.10. RULE ATTRIBUTES

Table 8.5. Rule Attributes

Attribute 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.
ruleflow-group	N/A	String	Ruleflow is a Drools 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.
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; irrespective 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 could now be caused not only by the rule itself. It's ideal for calculation rules where you have a number of rules that modify a fact and you don't 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.
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.
agenda-group	MAIN	String	Agenda groups allow the user to partition the Agenda providing more execution control. Only rules in the agenda group that has acquired the focus are allowed to fire.

Attribute Name	Default Value	Type	Description
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, then it is 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. In other words, the first rule in an activation-group to fire will cancel the other rules' activations, i.e., stop them from firing.
dialect	As specified by the package	String	The dialect species the language to be used for any code expressions in the LHS or the RHS code block. Currently two dialects are available, Java and MVEL. While the dialect can be specified at the package level, this attribute allows the package definition to be overridden for a rule.
date-effective	N/A	String, containing a date and time definition	A rule can only activate if the current date and time is after date-effective attribute.
date-expires	N/A	String, containing a date and time definition	A rule cannot activate if the current date and time is after the date-expires attribute.
duration	no default value	long	The duration dictates that the rule will fire after a specified duration, if it is still true.

8.10.1. Rule Attribute Example

This is an example for using a rule attribute:

```
rule "my rule"
    salience 42
    agenda-group "number-1"
     when ...
```

8.10.2. Timer Attribute Example

This is what the **`timer`** attribute looks like:

```
timer ( int: <initial delay> <repeat interval>? )
timer ( int: 30s )
timer ( int: 30s 5m )

timer ( cron: <cron expression> )
timer ( cron:* 0/15 * * * ? )
```

8.10.3. Timers

The following timers are available in JBoss BRMS:

Interval

Interval (indicated by "int:") timers follow the semantics of **`java.util.Timer`** objects, with an initial delay and an optional repeat interval.

Cron

Cron (indicated by "cron:") timers follow standard Unix cron expressions.

A rule controlled by a timer becomes active when it matches, and once for each individual match. Its consequence is executed repeatedly, according to the timer's settings. This stops as soon as the condition doesn't match any more.

Consequences are executed even after control returns from a call to fireUntilHalt. Moreover, the Engine remains reactive to any changes made to the Working Memory. For instance, removing a fact that was involved in triggering the timer rule's execution causes the repeated execution to terminate, or inserting a fact so that some rule matches will cause that rule to fire. But the Engine is not continually active, only after a rule fires, for whatever reason. Thus, reactions to an insertion done asynchronously will not happen until the next execution of a timer-controlled rule.

Disposing a session puts an end to all timer activity.

8.10.4. Cron Timer Example

This is what the Cron timer looks like:

```
rule "Send SMS every 15 minutes"
    timer (cron:* 0/15 * * * ?)
when
    $a : Alarm( on == true )
then
    channels[ "sms" ].insert( new Sms( $a.mobileNumber, "The alarm is
still on" );
end
```

8.10.5. Calendars

Calendars are used to control when rules can fire. JBoss BRMS uses the Quartz calendar.

8.10.6. Quartz Calendar Example

This is what the Quartz calendar looks like:

```
Calendar weekDayCal =
QuartzHelper.quartzCalendarAdapter(org.quartz.Calendar quartzCal)
```

8.10.7. Registering a Calendar

Procedure 8.8. Task

1. Start a StatefulKnowledgeSession.
2. Use the following code to register the calendar:

```
ksession.getCalendars().set( "weekday", weekDayCal );
```

3. If you wish to utilize the calendar and a timer together, use the following code:

```
rule "weekdays are high priority"
   calendars "weekday"
   timer (int:0 1h)
when
    Alarm()
then
    send( "priority high - we have an alarm” );
end

rule "weekend are low priority"
   calendars "weekend"
   timer (int:0 4h)
when
    Alarm()
then
    send( "priority low - we have an alarm” );
end
```

8.10.8. Left Hand Side

The Left Hand Side (LHS) is a common name for the conditional part of the rule. It consists of zero or more Conditional Elements. If the LHS is empty, it will be considered as a condition element that is always true and it will be activated once, when a new WorkingMemory session is created.

8.10.9. Conditional Elements

Conditional elements work on one or more *patterns*. The most common conditional element is **and**. It is implicit when you have multiple patterns in the LHS of a rule that is not connected in any way.

8.10.10. Rule Without a Conditional Element Example

This is what a rule without a conditional element looks like:

```
rule "no CEs"
when
    // empty
then
    ... // actions (executed once)
end

// The above rule is internally rewritten as:

rule "eval(true)"
when
    eval( true )
```

```
then
    ... // actions (executed once)
end
```

8.11. PATTERNS

A pattern element is the most important Conditional Element. It can potentially match on each fact that is inserted in the working memory. A pattern contains constraints and has an optional pattern binding.

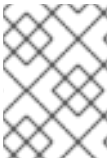
8.11.1. Pattern Example

This is what a pattern looks like:

```
rule "2 unconnected patterns"
when
    Pattern1()
    Pattern2()
then
    ... // actions
end

// The above rule is internally rewritten as:

rule "2 and connected patterns"
when
    Pattern1()
    and Pattern2()
then
    ... // actions
end
```



NOTE

An **and** cannot have a leading declaration binding. This is because a declaration can only reference a single fact at a time, and when the **and** is satisfied it matches both facts.

8.11.2. Pattern Matching

A pattern matches against a fact of the given type. The type need not be the actual class of some fact object. Patterns may refer to superclasses or even interfaces, thereby potentially matching facts from many different classes. The constraints are defined inside parentheses.

8.11.3. Pattern Binding

Patterns can be bound to a matching object. This is accomplished using a pattern binding variable such as **$p**.

8.11.4. Pattern Binding with Variable Example

This is what pattern binding using a variable looks like:

```
rule ...
```

```
when
    $p : Person()
then
    System.out.println( "Person " + $p );
end
```

NOTE

The prefixed dollar symbol (**$**) is not mandatory.

8.11.5. Constraints

A constraint is an expression that returns **true** or **false**. For example, you can have a constraint that states five is smaller than six.

8.12. ELEMENTS AND VARIABLES

8.12.1. Property Access on Java Beans (POJOs)

Any bean property can be used directly. A bean property is exposed using a standard Java bean getter: a method **getMyProperty()** (or **isMyProperty()** for a primitive boolean) which takes no arguments and return something.

JBoss BRMS uses the standard JDK **Introspector** class to do this mapping, so it follows the standard Java bean specification.

WARNING

Property accessors must not change the state of the object in a way that may effect the rules. The rule engine effectively caches the results of its matching in between invocations to make it faster.

8.12.2. POJO Example

This is what the bean property looks like:

```
Person( age == 50 )

// this is the same as:
Person( getAge() == 50 )
```

The age property

The age property is written as **age** in DRL instead of the getter **getAge()**

Property accessors

You can use property access (**age**) instead of getters explicitly (**getAge()**) because of performance enhancements through field indexing.

8.12.3. Working with POJOs

Procedure 8.9. Task

1. Observe the example below:

```
public int getAge() {
    Date now = DateUtil.now(); // Do NOT do this
    return DateUtil.differenceInYears(now, birthday);
}
```

2. To solve this, insert a fact that wraps the current date into working memory and update that fact between **`fireAllRules`** as needed.

8.12.4. POJO Fallbacks

When working with POJOs, a fallback method is applied. If the getter of a property cannot be found, the compiler will resort to using the property name as a method name and without arguments. Nested properties are also indexed.

8.12.5. Fallback Example

This is what happens when a fallback is implemented:

```
Person( age == 50 )

// If Person.getAge() does not exists, this falls back to:
Person( age() == 50 )
```

This is what it looks like as a nested property:

```
Person( address.houseNumber == 50 )

// this is the same as:
Person( getAddress().getHouseNumber() == 50 )
```

WARNING

In a stateful session, care should be taken when using nested accessors as the Working Memory is not aware of any of the nested values and does not know when they change. Consider them immutable while any of their parent references are inserted into the Working Memory. If you wish to modify a nested value you should mark all of the outer facts as updated. In the above example, when the **`houseNumber`** changes, any **`Person`** with that **`Address`** must be marked as updated.

8.12.6. Java Expressions

Table 8.6. Java Expressions

Capability	Example
You can use any Java expression that returns a **`boolean`** as a constraint inside the parentheses of a pattern. Java expressions can be mixed with other expression enhancements, such as property access.	`Person( age == 50 )`
You can change the evaluation priority by using parentheses, as in any logic or mathematical expression.	`Person( age > 100 && ( age % 10 == 0 ) )`
You can reuse Java methods.	`Person( Math.round( weight / ( height * height ) ) < 25.0 )`
Type coercion is always attempted if the field and the value are of different types; exceptions will be thrown if a bad coercion is attempted.	`Person( age == "10" ) // "10" is coerced to 10`

WARNING

Methods must not change the state of the object in a way that may affect the rules. Any method executed on a fact in the LHS should be a *read only* method.

WARNING

The state of a fact should not change between rule invocations (unless those facts are marked as updated to the working memory on every change):

```
Person( System.currentTimeMillis() % 1000 == 0 ) // Do NOT
do this
```

IMPORTANT

All operators have normal Java semantics except for **==** and **!=**.

The **==** operator has null-safe **equals()** semantics:

```
// Similar to: java.util.Objects.equals(person.getFirstName(),
"John")
// so (because "John" is not null) similar to:
// "John".equals(person.getFirstName())
Person( firstName == "John" )
```

The **!=** operator has null-safe **!equals()** semantics:

```
// Similar to: !java.util.Objects.equals(person.getFirstName(),
"John")
Person( firstName != "John" )
```

8.12.7. Comma-Separated Operators

The comma character ('**,**') is used to separate constraint groups. It has implicit and connective semantics.

The comma operator is used at the top level constraint as it makes them easier to read and the engine will be able to optimize them.

8.12.8. Comma-Separated Operator Example

The following illustrates comma-separated scenarios in implicit and connective semantics:

```
// Person is at least 50 and weighs at least 80 kg
Person( age > 50, weight > 80 )
```

```
// Person is at least 50, weighs at least 80 kg and is taller than 2
meter.
Person( age > 50, weight > 80, height > 2 )
```

NOTE

The comma (**,**) operator cannot be embedded in a composite constraint expression, such as parentheses.

8.12.9. Binding Variables

You can bind properties to variables in JBoss BRMS. It allows for faster execution and performance.

8.12.10. Binding Variable Examples

This is an example of a property bound to a variable:

```
// 2 persons of the same age
Person( $firstAge : age ) // binding
Person( age == $firstAge ) // constraint expression
```

NOTE

For backwards compatibility reasons, it's allowed (but not recommended) to mix a constraint binding and constraint expressions as such:

```
// Not recommended
Person( $age : age * 2 < 100 )
```

```
// Recommended (separates bindings and constraint expressions)
Person( age * 2 < 100, $age : age )
```

8.12.11. Unification

You can *unify* arguments across several properties. While positional arguments are always processed with unification, the unification symbol, ':=', exists for named arguments.

8.12.12. Unification Example

This is what unifying two arguments looks like:

```
Person( $age := age )
Person( $age := age)
```

8.12.13. Options and Operators in Red Hat JBoss BRMS

Table 8.7. Options and Operators in Red Hat JBoss BRMS

Option	Description	Example
Date literal	The date format **`dd-mmm-yyyy`** is supported by default. You can customize this by providing an alternative date format mask as the System property named **`drools.dateformat`**. If more control is required, use a restriction.	`Cheese( bestBefore < "27-Oct-2009" )`
List and Map access	You can directly access a **`List`** value by index.	`// Same as childList(0).getAge( ) == 18` `Person( childList[0].age == 18 )`

Option	Description	Example
Value key	You can directly access a **`Map`** value by key.	`// Same as credentialMap.get("jsmith").isValid()` `Person( credentialMap["jsmith"].valid )`

Abbreviated combined relation condition	This allows you to place more than one restriction on a field using the restriction connectives **&&** or **||**. Grouping via parentheses is permitted, resulting in a recursive syntax pattern.	`// Simple abbreviated combined relation condition using a single && Person( age > 30 && < 40 )`
		`// Complex abbreviated combined relation using groupings Person( age ( (> 30 && < 40) || (> 20 && < 25) ) )`
		`// Mixing abbreviated combined relation with constraint connectives Person( age > 30 && < 40 || location == "london" )`

Operators	Operators can be used on properties with natural ordering. For example, for Date fields, **<** means *before*, for **`String`** fields, it means alphabetically lower.	`Person( firstName < $otherFirstName )` `Person( birthDate < $otherBirthDate )`

Option	Description	Example
Operator matches	Matches a field against any valid Java **`regular expression`**. Typically that regexp is a string literal, but variables that resolve to a valid regexp are also allowed. It only applies on **`String`** properties. Using **`matches`** against a **`null`** value always evaluates to false.	`Cheese( type matches "(Buffalo)? \\S*Mozarella" )`
Operator not matches	The operator returns true if the String does not match the regular expression. The same rules apply as for the **`matches`** operator. It only applies on **`String`** properties.	`Cheese( type not matches "(Buffulo)? \\S*Mozarella" )`
The operator contains	The operator **`contains`** is used to check whether a field that is a Collection or array contains the specified value. It only applies on **`Collection`** properties.	`CheeseCounter( cheeses contains "stilton" ) // contains with a String literal CheeseCounter( cheeses contains $var ) // contains with a variable`
The operator not contains	The operator **`not contains`** is used to check whether a field that is a **`Collection`** or array does *not* contain the specified value. It only applies on **`Collection`** properties.	`CheeseCounter( cheeses not contains "cheddar" ) // not contains with a String literal CheeseCounter( cheeses not contains $var ) // not contains with a variable`
The operator memberOf	The operator **`memberOf`** is used to check whether a field is a member of a collection or array; that collection must be a variable.	`CheeseCounter( cheese memberOf $matureCheeses )`

Option	Description	Example
The operator not memberOf	The operator **`not memberOf`** is used to check whether a field is not a member of a collection or array. That collection must be a variable.	`CheeseCounter( cheese not memberOf $matureCheeses )`
The operator soundslike	This operator is similar to **`matches`**, but it checks whether a word has almost the same sound (using English pronunciation) as the given value.	`// match cheese "fubar" or "foobar" Cheese( name soundslike 'foobar' )`

The operator str	The operator **`str`** is used to check whether a field that is a **`String`** starts with or ends with a certain value. It can also be used to check the length of the String.	`Message( routingValue str[startsWith] "R1" )`
		`Message( routingValue str[endsWith] "R2" )`
		`Message( routingValue str[length] 17 )`

Compound Value Restriction	Compound value restriction is used where there is more than one possible value to match. Currently only the **`in`** and **`not in`** evaluators support this. The second operand of this operator must be a comma-separated list of values, enclosed in parentheses. Values may be given as variables, literals, return values or qualified identifiers. Both evaluators are actually *syntactic sugar*, internally rewritten as a list of multiple restrictions using the operators **`!=`** and **`==`**.	`Person( $cheese : favouriteCheese ) Cheese( type in ( "stilton", "cheddar", $cheese ) )`

Option	Description	Example
Inline Eval Operator (deprecated)	An inline eval constraint can use any valid dialect expression as long as it results to a primitive boolean. The expression must be constant over time. Any previously bound variable, from the current or previous pattern, can be used; autovivification is also used to auto-create field binding variables. When an identifier is found that is not a current variable, the builder looks to see if the identifier is a field on the current object type, if it is, the field binding is auto-created as a variable of the same name. This is called autovivification of field variables inside of inline eval's.	`Person( girlAge : age, sex = "F" )` `Person( eval( age == girlAge + 2 ), sex = 'M' ) // eval() is actually obsolete in this example`

8.12.14. Operator Precedence

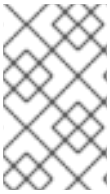
Table 8.8. Operator precedence

Operator type	Operators	Notes
(nested) property access	`.`	Not normal Java semantics
List/Map access	`[ ]`	Not normal Java semantics
constraint binding	`:`	Not normal Java semantics
multiplicative	`* /%`	
additive	`+ -`	
shift	`<< >>>>>`	
relational	`< ><= >=instanceof`	
equality	`== !=`	Does not use normal Java (*not*) *same* semantics: uses (*not*) *equals* semantics instead.
non-short circuiting AND	`&`	
non-short circuiting exclusive OR	`^`	
non-short circuiting inclusive OR	`|`	

Operator type	Operators	Notes
logical AND	**&&**	
logical OR	**||**	
ternary	**? :**	
Comma separated AND	**,**	Not normal Java semantics

8.12.15. Fine Grained Property Change Listeners

This feature allows the pattern matching to only react to modification of properties actually constrained or bound inside of a given pattern. This helps with performance and recursion and avoid artificial object splitting.



NOTE

By default this feature is off in order to make the behavior of the rule engine backward compatible with the former releases. When you want to activate it on a specific bean you have to annotate it with @propertyReactive.

8.12.16. Fine Grained Property Change Listener Example

DRL example

```
declare Person
          @propertyReactive
          firstName : String
          lastName : String
          end
```

Java class example

```
@PropertyReactive
          public static class Person {
          private String firstName;
          private String lastName;
          }
```

8.12.17. Working with Fine Grained Property Change Listeners

Using these listeners means you do not need to implement the no-loop attribute to avoid an infinite recursion. The engine recognizes that the pattern matching is done on the property while the RHS of the rule modifies other the properties. On Java classes, you can also annotate any method to say that its invocation actually modifies other properties.

8.12.18. Using Patterns with @watch

Annotating a pattern with @watch allows you to modify the inferred set of properties for which that pattern will react. The properties named in the @watch annotation are added to the ones automatically inferred. You can explicitly exclude one or more of them by beginning their name with a **!** and to make the pattern to listen for all or none of the properties of the type used in the pattern respectively with the wildcards ***** and **!***.

8.12.19. @watch Example

This is the @watch annotation in a rule's LHS:

```
// listens for changes on both firstName (inferred) and lastName
          Person( firstName == $expectedFirstName ) @watch( lastName )

          // listens for all the properties of the Person bean
          Person( firstName == $expectedFirstName ) @watch( * )

          // listens for changes on lastName and explicitly exclude
firstName
          Person( firstName == $expectedFirstName ) @watch( lastName,
!firstName )

          // listens for changes on all the properties except the age one
          Person( firstName == $expectedFirstName ) @watch( *, !age )
```

NOTE

Since doesn't make sense to use this annotation on a pattern using a type not annotated with @PropertyReactive the rule compiler will raise a compilation error if you try to do so. Also the duplicated usage of the same property in @watch (for example like in: @watch(firstName, ! firstName)) will end up in a compilation error.

8.12.20. Using @PropertySpecificOption

You can enable @watch by default or completely disallow it using the **on** option of the KnowledgeBuilderConfiguration. This new PropertySpecificOption can have one of the following 3 values:

```
- DISABLED => the feature is turned off and all the other related
annotations are just ignored
          - ALLOWED => this is the default behavior: types are not
property reactive unless they are not annotated with @PropertySpecific
          - ALWAYS => all types are property reactive by default
```

8.12.21. Basic Conditional Elements

Table 8.9. Basic Conditional Elements

Name	Description	Example	Additional options

Name	Description	Example	Additional options

and	The Conditional Element **and** is used to group other Conditional Elements into a logical conjunction. JBoss BRMS supports both prefix **and** and infix **and**. It supports explicit grouping with parentheses. You can also use traditional infix and prefix **and**.	`//infixAnd Cheese( cheeseType : type ) and Person( favouriteCheese == cheeseType )`
		
		`//infixAnd with grouping ( Cheese( cheeseType : type ) and ( Person( favouriteCheese == cheeseType ) or Person( favouriteCheese == cheeseType ) )`
			Prefix **and** is also supported:
			
			`(and Cheese( cheeseType : type ) Person( favouriteCheese == cheeseType ) )`
			
			The root element of the LHS is an implicit prefix **and** and doesn't need to be specified:
			
			`when Cheese( cheeseType : type ) Person( favouriteCheese == cheeseType ) then ...`

Name	Description	Example	Additional options

or	This is a shortcut for generating two or more similar rules. JBoss BRMS supports both prefix **`or`** and infix **`or`**. You can use traditional infix, prefix and explicit grouping parentheses.	`//infixOr Cheese( cheeseType : type ) or Person( favouriteCheese == cheeseType )`
		
		`//infixOr with grouping ( Cheese( cheeseType : type ) or ( Person( favouriteCheese == cheeseType ) and Person( favouriteCheese == cheeseType ) )`
		
		`(or Person( sex == "f", age > 60 ) Person( sex == "m", age > 65 )`
			Allows for optional pattern binding. Each pattern must be bound separately, using eponymous variables:
			
			`pensioner : ( Person( sex == "f", age > 60 ) or Person( sex == "m", age > 65 ) )`
			
			`(or pensioner : Person( sex == "f", age > 60 ) pensioner : Person( sex == "m", age > 65 ) )`

Name	Description	Example	Additional options

not	This checks to ensure an object specified as absent is not included in the Working Memory. It may be followed by parentheses around the condition elements it applies to. (In a single pattern you can omit the parentheses.)	`// Brackets are optional: not Bus(color == "red")`
		`// Brackets are optional: not ( Bus(color == "red", number == 42) )`
		`// "not" with nested infix and - two patterns,`
		`// brackets are requires: not ( Bus(color == "red") and`
		
		`Bus(color == "blue") )`	

exists	This checks the working memory to see if a specified item exists. The keyword **exists** must be followed by parentheses around the CEs that it applies to. (In a single pattern you can omit the parentheses.)	`exists Bus(color == "red")`
		`// brackets are optional: exists ( Bus(color == "red", number == 42) )`
		`// "exists" with nested infix and,`
		`// brackets are required: exists ( Bus(color == "red") and`
		
		`Bus(color == "blue") )`	

NOTE

The behavior of the Conditional Element **`or`** is different from the connective **`||`** for constraints and restrictions in field constraints. The engine cannot interpret the Conditional Element **`or`**. Instead, a rule with **`or`** is rewritten as a number of subrules. This process ultimately results in a rule that has a single **`or`** as the root node and one subrule for each of its CEs. Each subrule can activate and fire like any normal rule; there is no special behavior or interaction between these subrules.

8.12.22. The Conditional Element Forall

This element evaluates to true when all facts that match the first pattern match all the remaining patterns. It is a *scope delimiter*. Therefore, it can use any previously bound variable, but no variable bound inside it will be available for use outside of it.

`Forall` can be nested inside other CEs. For instance, **`forall`** can be used inside a **`not`** CE. Only single patterns have optional parentheses, so with a nested **`forall`** parentheses must be used.

8.12.23. Forall Examples

Evaluating to true

```
rule "All English buses are red"
when
    forall( $bus : Bus( type == 'english')
                   Bus( this == $bus, color = 'red' ) )
then
    // all English buses are red
end
```

Single pattern forall

```
rule "All Buses are Red"
when
    forall( Bus( color == 'red' ) )
then
    // all Bus facts are red
end
```

Multi-pattern forall

```
rule "all employees have health and dental care programs"
when
    forall( $emp : Employee()
            HealthCare( employee == $emp )
            DentalCare( employee == $emp )
          )
then
    // all employees have health and dental care
end
```

Nested forall

-

```
rule "not all employees have health and dental care"
when
    not ( forall( $emp : Employee()
                  HealthCare( employee == $emp )
                  DentalCare( employee == $emp ) )
        )
then
    // not all employees have health and dental care
end
```

8.12.24. The Conditional Element From

The Conditional Element **from** enables users to specify an arbitrary source for data to be matched by LHS patterns. This allows the engine to reason over data not in the Working Memory. The data source could be a sub-field on a bound variable or the results of a method call. It is a powerful construction that allows out of the box integration with other application components and frameworks. One common example is the integration with data retrieved on-demand from databases using hibernate named queries.

The expression used to define the object source is any expression that follows regular MVEL syntax. Therefore, it allows you to easily use object property navigation, execute method calls and access maps and collections elements.

IMPORTANT

Using **from** with **lock-on-active** rule attribute can result in rules not being fired.

There are several ways to address this issue:

- Avoid the use of **from** when you can assert all facts into working memory or use nested object references in your constraint expressions (shown below).
- Place the variable assigned used in the modify block as the last sentence in your condition (LHS).
- Avoid the use of **lock-on-active** when you can explicitly manage how rules within the same rule-flow group place activations on one another.

8.12.25. From Examples

Reasoning and binding on patterns

```
rule "validate zipcode"
when
    Person( $personAddress : address )
    Address( zipcode == "23920W") from $personAddress
then
    // zip code is ok
end
```

Using a graph notation

```
rule "validate zipcode"
```

```
when
    $p : Person( )
    $a : Address( zipcode == "23920W") from $p.address
then
    // zip code is ok
end
```

Iterating over all objects

```
rule "apply 10% discount to all items over US$ 100,00 in an order"
when
    $order : Order()
    $item  : OrderItem( value > 100 ) from $order.items
then
    // apply discount to $item
end
```

Use with lock-on-active

```
rule "Assign people in North Carolina (NC) to sales region 1"
ruleflow-group "test"
lock-on-active true
when
    $p : Person(address.state == "NC" )
then
    modify ($p) {} // Assign person to sales region 1 in a modify block
end

rule "Apply a discount to people in the city of Raleigh"
ruleflow-group "test"
lock-on-active true
when
    $p : Person(address.city == "Raleigh" )
then
    modify ($p) {} //Apply discount to person in a modify block
end
```

8.12.26. The Conditional Element Collect

The Conditional Element **`collect`** allows rules to reason over a collection of objects obtained from the given source or from the working memory. In First Oder Logic terms this is the cardinality quantifier.

The result pattern of **`collect`** can be any concrete class that implements the **`java.util.Collection`** interface and provides a default no-arg public constructor. You can use Java collections like ArrayList, LinkedList and HashSet or your own class, as long as it implements the **`java.util.Collection`** interface and provide a default no-arg public constructor.

Variables bound before the **`collect`** CE are in the scope of both source and result patterns and therefore you can use them to constrain both your source and result patterns. Any binding made inside **`collect`** is not available for use outside of it.

8.12.27. The Conditional Element Accumulate

The Conditional Element **accumulate** is a more flexible and powerful form of **collect**, in the sense that it can be used to do what **collect** does and also achieve results that the CE **collect** is not capable of doing. It allows a rule to iterate over a collection of objects, executing custom actions for each of the elements. At the end it returns a result object.

Accumulate supports both the use of pre-defined accumulate functions, or the use of inline custom code. Inline custom code should be avoided though, as it is harder for rule authors to maintain, and frequently leads to code duplication. Accumulate functions are easier to test and reuse.

The Accumulate CE also supports multiple different syntaxes. The preferred syntax is the top level accumulate, as noted bellow, but all other syntaxes are supported for backward compatibility.

8.12.28. Syntax for the Conditional Element Accumulate

Top level accumulate syntax

```
accumulate( <source pattern>; <functions> [;<constraints>] )
```

Syntax example

```
rule "Raise alarm"
when
    $s : Sensor()
    accumulate( Reading( sensor == $s, $temp : temperature );
                $min : min( $temp ),
                $max : max( $temp ),
                $avg : average( $temp );
                $min < 20, $avg > 70 )
then
    // raise the alarm
end
```

In the above example, min, max and average are Accumulate Functions and will calculate the minimum, maximum and average temperature values over all the readings for each sensor.

8.12.29. Functions of the Conditional Element Accumulate

- average
- min
- max
- count
- sum
- collectList
- collectSet

These common functions accept any expression as input. For instance, if someone wants to calculate the average profit on all items of an order, a rule could be written using the average function:

```
rule "Average profit"
when
    $order : Order()
    accumulate( OrderItem( order == $order, $cost : cost, $price : price
);
                $avgProfit : average( 1 - $cost / $price ) )
then
    // average profit for $order is $avgProfit
end
```

8.12.30. The Conditional Element accumulate and Pluggability

Accumulate functions are all pluggable. That means that if needed, custom, domain specific functions can easily be added to the engine and rules can start to use them without any restrictions. To implement a new Accumulate Function all one needs to do is to create a Java class that implements the **`org.drools.runtime.rule.TypedAccumulateFunction`** interface and add a line to the configuration file or set a system property to let the engine know about the new function.

8.12.31. The Conditional Element accumulate and Pluggability Example

As an example of an Accumulate Function implementation, the following is the implementation of the **`average`** function:

```
/**
 * An implementation of an accumulator capable of calculating average
values
 */
public class AverageAccumulateFunction implements
org.drools.runtime.rule.TypedAccumulateFunction {

    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {

    }

    public void writeExternal(ObjectOutput out) throws IOException {

    }

    public static class AverageData implements Externalizable {
        public int    count = 0;
        public double total = 0;

        public AverageData() {}

        public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
            count   = in.readInt();
            total   = in.readDouble();
        }

        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeInt(count);
            out.writeDouble(total);
        }
```

```
    }

    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#createContext()
     */
    public Serializable createContext() {
        return new AverageData();
    }

    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#init(java.lang.Object)
     */
    public void init(Serializable context) throws Exception {
        AverageData data = (AverageData) context;
        data.count = 0;
        data.total = 0;
    }

    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#accumulate(java.lang.Objec
t, java.lang.Object)
     */
    public void accumulate(Serializable context,
                           Object value) {
        AverageData data = (AverageData) context;
        data.count++;
        data.total += ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#reverse(java.lang.Object,
java.lang.Object)
     */
    public void reverse(Serializable context,
                        Object value) throws Exception {
        AverageData data = (AverageData) context;
        data.count--;
        data.total -= ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#getResult(java.lang.Object
)
     */
    public Object getResult(Serializable context) throws Exception {
        AverageData data = (AverageData) context;
        return new Double( data.count == 0 ? 0 : data.total / data.count
);
    }
```

```
    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#supportsReverse()
     */
    public boolean supportsReverse() {
        return true;
    }

    /**
     * {@inheritDoc}
     */
    public Class< ? > getResultType() {
        return Number.class;
    }

}
```

8.12.32. Code for the Conditional Element Accumulate's Functions

Code for plugging in functions (to be entered into the config file)

```
jbossrules.accumulate.function.average =
   org.jbossrules.base.accumulators.AverageAccumulateFunction
```

Alternate Syntax: single function with return type

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
             from accumulate( OrderItem( order == $order, $value :
value ),
                              sum( $value ) )
then
    # apply discount to $order
end
```

8.12.33. Accumulate with Inline Custom Code

WARNING

The use of accumulate with inline custom code is not a good practice for several reasons, including difficulties on maintaining and testing rules that use them, as well as the inability of reusing that code. Implementing your own accumulate functions allows for simpler testing. This form of accumulate is supported for backward compatibility only.

The general syntax of the **accumulate** CE with inline custom code is:

```
<result pattern> from accumulate( <source pattern>,
                                  init( <init code> ),
                                  action( <action code> ),
                                  reverse( <reverse code> ),
                                  result( <result expression> ) )
```

The meaning of each of the elements is the following:

- *<source pattern>*: the source pattern is a regular pattern that the engine will try to match against each of the source objects.
- *<init code>*: this is a semantic block of code in the selected dialect that will be executed once for each tuple, before iterating over the source objects.
- *<action code>*: this is a semantic block of code in the selected dialect that will be executed for each of the source objects.
- *<reverse code>*: this is an optional semantic block of code in the selected dialect that if present will be executed for each source object that no longer matches the source pattern. The objective of this code block is to undo any calculation done in the *<action code>* block, so that the engine can do decremental calculation when a source object is modified or retracted, hugely improving performance of these operations.
- *<result expression>*: this is a semantic expression in the selected dialect that is executed after all source objects are iterated.
- *<result pattern>*: this is a regular pattern that the engine tries to match against the object returned from the *<result expression>*. If it matches, the **accumulate** conditional element evaluates to *true* and the engine proceeds with the evaluation of the next CE in the rule. If it does not matches, the **accumulate** CE evaluates to *false* and the engine stops evaluating CEs for that rule.

8.12.34. Accumulate with Inline Custom Code Examples

Inline custom code

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
             from accumulate( OrderItem( order == $order, $value :
value ),
                              init( double total = 0; ),
                              action( total += $value; ),
                              reverse( total -= $value; ),
                              result( total ) )
then
    # apply discount to $order
end
```

In the above example, for each **Order** in the Working Memory, the engine will execute the *init code* initializing the total variable to zero. Then it will iterate over all **OrderItem** objects for that order, executing the *action* for each one (in the example, it will sum the value of all items into the total

variable). After iterating over all **`OrderItem`** objects, it will return the value corresponding to the *result expression* (in the above example, the value of variable **`total`**). Finally, the engine will try to match the result with the **`Number`** pattern, and if the double value is greater than 100, the rule will fire.

Instantiating and populating a custom object

```
rule "Accumulate using custom objects"
when
    $person   : Person( $likes : likes )
    $cheesery : Cheesery( totalAmount > 100 )
                from accumulate( $cheese : Cheese( type == $likes ),
                                 init( Cheesery cheesery = new
Cheesery(); ),
                                 action( cheesery.addCheese( $cheese
); ),
                                 reverse( cheesery.removeCheese(
$cheese ); ),
                                 result( cheesery ) );
then
    // do something
end
```

8.12.35. Conditional Element Eval

The conditional element **`eval`** is essentially a catch-all which allows any semantic code (that returns a primitive boolean) to be executed. This code can refer to variables that were bound in the LHS of the rule, and functions in the rule package. Overuse of eval reduces the declarativeness of your rules and can result in a poorly performing engine. While **`eval`** can be used anywhere in the patterns, the best practice is to add it as the last conditional element in the LHS of a rule.

Evals cannot be indexed and thus are not as efficient as Field Constraints. However this makes them ideal for being used when functions return values that change over time, which is not allowed within Field Constraints.

8.12.36. Conditional Element Eval Examples

This is what **`eval`** looks like in use:

```
p1 : Parameter()
p2 : Parameter()
eval( p1.getList().containsKey( p2.getItem() ) )
```

```
p1 : Parameter()
p2 : Parameter()
// call function isValid in the LHS
eval( isValid( p1, p2 ) )
```

8.12.37. The Right Hand Side

The Right Hand Side (RHS) is a common name for the consequence or action part of the rule. The main purpose of the RHS is to insert, retractor modify working memory data. It should contain a list of actions to be executed. The RHS part of a rule should also be kept small, thus keeping it declarative and

readable.

NOTE

If you find you need imperative and/or conditional code in the RHS, break the rule down into multiple rules.

8.12.38. RHS Convenience Methods

Table 8.10. RHS Convenience Methods

Name	Description
`update(`*object, handle***`);`**	Tells the engine that an object has changed (one that has been bound to something on the LHS) and rules that need to be reconsidered.
`update(`*object***`);`**	Using **`update()`**, the Knowledge Helper will look up the facthandle via an identity check for the passed object. (If you provide Property Change Listeners to your Java beans that you are inserting into the engine, you can avoid the need to call **`update()`** when the object changes.). After a fact's field values have changed you must call update before changing another fact, or you will cause problems with the indexing within the rule engine. The modify keyword avoids this problem.
`insert(new`*object***`());`**	Places a new object of your creation into the Working Memory.
`insertLogical(new`*object***`());`**	Similar to insert, but the object will be automatically retracted when there are no more facts to support the truth of the currently firing rule.
`retract(`*handle***`);`**	Removes an object from Working Memory.

8.12.39. Convenience Methods using the Drools Variable

- The call **`drools.halt()`** terminates rule execution immediately. This is required for returning control to the point whence the current session was put to work with **`fireUntilHalt()`**.
- Methods **`insert(Object o)`**, **`update(Object o)`** and **`retract(Object o)`** can be called on **`drools`** as well, but due to their frequent use they can be called without the object reference.
- **`drools.getWorkingMemory()`** returns the **`WorkingMemory`** object.
- **`drools.setFocus( String s)`** sets the focus to the specified agenda group.
- **`drools.getRule().getName()`**, called from a rule's RHS, returns the name of the rule.
- **`drools.getTuple()`** returns the Tuple that matches the currently executing rule, and **`drools.getActivation()`** delivers the corresponding Activation. (These calls are useful for logging and debugging purposes.)

8.12.40. Convenience Methods using the Kcontext Variable

- The call **`kcontext.getKieRuntime().halt()`** terminates rule execution immediately.
- The accessor **`getAgenda()`** returns a reference to the session's **`Agenda`**, which in turn provides access to the various rule groups: activation groups, agenda groups, and rule flow groups. A fairly common paradigm is the activation of some agenda group, which could be done with the lengthy call:

  ```
  // give focus to the agenda group CleanUp
  kcontext.getKieRuntime().getAgenda().getAgendaGroup( "CleanUp"
  ).setFocus();
  ```

  (You can achieve the same using **`drools.setFocus( "CleanUp" )`**.)
- To run a query, you call **`getQueryResults(String query)`**, whereupon you may process the results.
- A set of methods dealing with event management lets you add and remove event listeners for the Working Memory and the Agenda.
- Method **`getKieBase()`** returns the **`KieBase`** object, the backbone of all the Knowledge in your system, and the originator of the current session.
- You can manage globals with **`setGlobal(...)`**, **`getGlobal(...)`** and **`getGlobals()`**.
- Method **`getEnvironment()`** returns the runtime's **`Environment`**.

8.12.41. The Modify Statement

Table 8.11. The Modify Statement

Name	Description	Syntax	Example

Name	Description	Syntax	Example
modify	This provides a structured approach to fact updates. It combines the update operation with a number of setter calls to change the object's fields.	**modify (** *<fact-expression>* **)** **{** *<expression>* [, *<expression>*]* **}** The parenthesized *<fact-expression>* must yield a fact object reference. The expression list in the block should consist of setter calls for the given object, to be written without the usual object reference, which is automatically prepended by the compiler.	`rule "modify stilton" when $stilton : Cheese(type == "stilton") then modify( $stilton ){ setPrice( 20 ), setAge( "overripe" ) } end`

8.12.42. Query Examples

NOTE

To return the results use **`ksession.getQueryResults("name")`**, where "name" is the query's name. This returns a list of query results, which allow you to retrieve the objects that matched the query.

Query for people over the age of 30

```
query "people over the age of 30"
    person : Person( age > 30 )
end
```

Query for people over the age of X, and who live in Y

```
query "people over the age of x"  (int x, String y)
    person : Person( age > x, location == y )
end
```

8.12.43. QueryResults Example

We iterate over the returned QueryResults using a standard "for" loop. Each element is a QueryResultsRow which we can use to access each of the columns in the tuple. These columns can be accessed by bound declaration name or index position:

```
QueryResults results = ksession.getQueryResults( "people over the age of
30" );
System.out.println( "we have " + results.size() + " people over the age  of
30" );

System.out.println( "These people are are over 30:" );

for ( QueryResultsRow row : results ) {
    Person person = ( Person ) row.get( "person" );
    System.out.println( person.getName() + "\n" );
}
```

8.12.44. Queries Calling Other Queries

Queries can call other queries. This combined with optional query arguments provides derivation query style backward chaining. Positional and named syntax is supported for arguments. It is also possible to mix both positional and named, but positional must come first, separated by a semi colon. Literal expressions can be passed as query arguments, but you cannot mix expressions with variables.

NOTE

Using the '?' symbol in this process means the query is pull only and once the results are returned you will not receive further results as the underlying data changes.

8.12.45. Queries Calling Other Queries Example

Query calling another query

```
declare Location
    thing : String
    location : String
end

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

Using live queries to reactively receive changes over time from query results

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

rule look when
    Person( $l : likes )
    isContainedIn( $l, 'office'; )
then
   insertLogical( $l 'is in the office' );
end
```

8.12.46. Unification for Derivation Queries

JBoss BRMS supports unification for derivation queries. This means that arguments are optional. It is possible to call queries from Java leaving arguments unspecified using the static field **`org.drools.runtime.rule.Variable.v`**. (You must use *v* and not an alternative instance of Variable.) These are referred to as *out* arguments.

NOTE

The query itself does not declare at compile time whether an argument is in or an out. This can be defined purely at runtime on each use.

8.13. SEARCHING THE WORKING MEMORY USING QUERY

8.13.1. Queries

Queries are used to retrieve fact sets based on patterns, as they are used in rules. Patterns may make use of optional parameters. Queries can be defined in the Knowledge Base, from where they are called up to return the matching results. While iterating over the result collection, any identifier bound in the query can be used to access the corresponding fact or fact field by calling the **`get`** method with the binding variable's name as its argument. If the binding refers to a fact object, its FactHandle can be retrieved by calling **`getFactHandle`**, again with the variable's name as the parameter. Illustrated below is a Query example:

```
QueryResults results =
    ksession.getQueryResults( "my query", new Object[] { "string" } );
for ( QueryResultsRow row : results ) {
    System.out.println( row.get( "varName" ) );
}
```

8.13.2. Live Queries

Invoking queries and processing the results by iterating over the returned set is not a good way to monitor changes over time.

To alleviate this, JBoss BRMS provides Live Queries, which have a listener attached instead of returning an iterable result set. These live queries stay open by creating a view and publishing change events for the contents of this view. To activate, start your query with parameters and listen to changes in the resulting view. The **`dispose`** method terminates the query and discontinues this reactive scenario.

8.13.3. ViewChangedEventListener Implementation Example

```
final List updated = new ArrayList();
final List removed = new ArrayList();
final List added = new ArrayList();

ViewChangedEventListener listener = new ViewChangedEventListener() {
 public void rowUpdated(Row row) {
  updated.add( row.get( "$price" ) );
 }

 public void rowRemoved(Row row) {
  removed.add( row.get( "$price" ) );
```

```
 }

 public void rowAdded(Row row) {
  added.add( row.get( "$price" ) );
 }
};

// Open the LiveQuery
LiveQuery query = ksession.openLiveQuery( "cars",
                                          new Object[] { "sedan",
"hatchback" },
                                          listener );
...
...
query.dispose() // calling dispose to terminate the live query
```

NOTE

For an example of Glazed Lists integration for live queries, visit http://blog.athico.com/2010/07/glazed-lists-examples-for-drools-live.html

8.14. DOMAIN SPECIFIC LANGUAGES (DSLS)

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.

8.14.1. The DSL Editor

The DSL editor provides a tabular view of the mapping of Language to Rule Expressions. The Language Expression feeds the content assistance for the rule editor so that it can suggest Language Expressions from the DSL configuration. (The rule editor loads the DSL configuration when the rule resource is loaded for editing.

NOTE

DSL feature is useful for simple use cases for non technical users to easily define rules based on sentence snippets. For more complex use cases, we recommend you to use other advanced features like decision tables and DRL rules, that are more expressive and flexible.

8.14.2. 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.

8.14.3. DSL Example

Table 8.12. 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.

8.14.4. 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 ('\').

8.14.5. 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.)

8.14.6. DSL Syntax Examples

Table 8.13. 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.

8.14.7. 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.

8.14.8. Adding Constraints to Facts

Table 8.14. Adding Constraints to Facts

Name	Description	Example

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")`

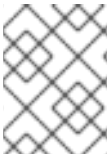
DSL definitions	The DSL definitions given in this example result in three DSL phrases which may be used to create any combination of constraint involving these fields.	`[when]There is a Cheese with=Cheese()`
		`[when]- age is less than {age}=age<{age}`
		`[when]- type is '{type}'=type=='{type}'`
		`[when]-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

8.14.9. 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.

8.14.10. 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.

8.14.11. 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.

8.14.12. Debug Options for DSL Expansion

Table 8.15. 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.

8.14.13. 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} )\{ \}
```

8.14.14. 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.

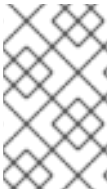
   For a line, each DSL entry pertaining to the line's section is taken in turn, in the order it appears in the DSL file. Its regular expression part is modified: white space is replaced by a pattern matching any number of white space characters; variable definitions with a regular expression are replaced 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").

8.14.15. String Transformation Functions

Table 8.16. String Transformation Functions

Name	Description
uc	Converts all letters to upper case.
lc	Converts all letters to lower case.

Name	Description
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.

8.14.16. Stringing DSL Transformation Functions

Table 8.17. Stringing DSL Transformation Functions

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}`

Name	Description	Example
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 );`

CHAPTER 9. USING JBOSS DEVELOPER STUDIO TO CREATE AND TEST RULES

There are many ways to author rules in BRMS, however as a developer you would prefer an Integrated Development Environment (IDE) such as JBoss Developer Studio that offers you advanced tooling and content assistance. JBoss BRMS and JBoss BPM Suite tooling are compatible with JBoss Developer Studio version 7 and above. The JBoss Developer Studio with JBoss BPM Suite/BRMS plug-ins simplify your development tasks. These plug-ins provide the following features:

- Simple wizards for rule and project creation
- Content assistance for generating the basic rule structure. For example, If you open a **`.drl`** file in the JBoss Developer Studio editor and type **`ru`**, and press **`Ctrl+Space`**, the template rule structure is created.
- Syntax coloring
- Error highlighting
- IntelliSense code completion
- Outline view to display an outline of your structured rule project
- Debug perspective for Rules/Process debugging
- Rete tree view to display Rete network
- Editor for modifying business process diagram
- Support for unit testing via JUnit and TestNG

9.1. JBOSS DEVELOPER STUDIO DROOLS PERSPECTIVE

JBoss Developer Studio comes with all the BRMS and BPM Suite plug-in requirements pre-packaged with it. It offers the following perspectives:

- Drools: allows you to work with JBoss BRMS specific resources
- Business Central Repository Exploring
- jBPM: allows you to work with JBoss BPM Suite resources

9.2. JBOSS BRMS RUNTIMES

A Drools runtime is a collection of jar files on your file system that represent one specific release of the Drools project jars. While creating a new runtime, you must either point to the release of your choice or create a new runtime on your file system from the jars included in the Drools plug-in. For creating a new runtime, you need to specify a default Drools runtime for your Eclipse workspace, but each individual project can override the default and select the appropriate runtime for that project specifically. You can add as many Drools runtimes as you need. In order to use the JBoss BRMS plug-in with Red Hat JBoss Developer Studio, it is necessary to set up the runtime.

9.2.1. Defining a JBoss BRMS Runtime

Procedure 9.1. Task

1. Extract the runtime jar files located in the **`jboss-brms-engine.zip`** archive of the JBoss BRMS Generic Deployable zip archive (not the EAP6 deployable zip archive) available from Red Hat Customer Portal.

2. From the JBoss Developer Studio menu, go to **Window** → **Preferences**.

   The **`Preferences`** dialog opens displaying all your preferences.

3. Navigate to **Drools** → **Installed Drools runtimes**.

4. To define a new Drools runtime, click the add button.

   The **`Drools Runtime`** dialog opens.

5. In the **`Drools Runtime`** dialog, you have the following options to provide the name for your runtime and the its location on your file system:

   - Use the default JAR files included in the Drools Eclipse plug-in to create a new Drools runtime automatically:

     1. Click the **`Create a new Drools c runtime ...`** button.

     2. Browse and select the folder on your file system where you would like this runtime to be created.

        The plug-in automatically copies all required dependencies to the specified folder.

   - Use one specific release of the Drools project,

     1. Create a folder on your file system and copy all the necessary Drools libraries and dependencies into it.

     2. Provide a name for your runtime in the Drools Runtime dialog in the **`Name`** field and browse to the location of this folder containing all the required JARs in the Path field.

6. Click **`OK`**.

   The runtime appears in your table of installed Drools runtimes.

7. Click the checkbox in front of the newly created runtime to make it the default Drools runtime.

   This default Drools runtime will be used as the runtime of all your Drools project that does not have a project-specific runtime selected.

8. Restart JBoss Developer Studio if you have changed the default runtime to ensure that all the projects that are using the default runtime update their classpath accordingly.

9.2.2. Selecting a Runtime for Your JBoss BRMS Project

Whenever you create a Drools project either by using the **`New Drools Project`** wizard or by converting an existing Java project to a Drools project, the Drools plug-in automatically adds all the required JAR files to the classpath of your project.

If you are creating a new Drools project, the plug-in uses the default Drools runtime for that project, unless you specify a project-specific one.

Procedure 9.2. Task

To define a project-specific runtime:

1. Create a new Drools project and in the final step of the **`New Drools Project`** wizard and uncheck the **`Use default Drools runtime`** checkbox.

2. Click the **`Configure workspace settings ...`** link.

   The workspace preferences showing the currently installed Drools runtimes opens.

3. Click **`Add`** to add new runtimes.

9.2.3. Changing the Runtime of Your JBoss BRMS Project

Procedure 9.3. Task

To change the runtime of a Drools project:

1. In the Drools perspective, right-click the project and select **`Properties`**.

   The project properties dialog opens.

2. Navigate and select the Drools category.

3. Check the **`Enable project specific settings`** checkbox and select the appropriate runtime from the drop-down box.

   If you click the **`Configure workspace settings ...`** link, the workspace preferences showing the currently installed Drools runtimes opens. You can add new runtimes there if required. If you uncheck the **`Enable project specific settings`** checkbox, it uses the default runtime as defined in your global preferences.

4. Click **`OK`**.

9.2.4. Configuring the JBoss BRMS Server

JBoss Developer Studio can be configured to run the Red Hat JBoss BRMS\BPM Suite Server.

Procedure 9.4. Configure the Server

1. Open the Drools view by selecting **Window** → **Open Perspective** → **Other** and select Drools and click **`OK`**.

2. Add the server view by selecting **Window** → **Show View** → **Other...** and select **Server** → **Servers**.

3. Open the server menu by right clicking the Servers panel and select **New** → **Server**.

4. Define the server by selecting **JBoss Enterprise Middleware** → **JBoss Enterprise Application Platform 6.1+** and clicking **`Next`**.

5. Set the home directory by clicking the **`Browse`** button. Navigate to and select the installation directory for JBoss EAP 6.1.1 which has JBoss BRMS installed.

6. Provide a name for the server in the **Name** field, ensure that the configuration file is set, and click **Finish**.

9.3. EXPLORING A JBOSS BRMS APPLICATION

Before exploring how to create BRMS projects using JBoss Developer Studio, let us first understand the structure of BRMS projects.

A BRMS project typically comprises the following:

- Facts that are a set of java classes files (POJOs)
- Rules that operate on the facts
- Drools library (jar files) for executing the rules

JBoss Developer Studio helps you generate getter and setter methods for attributes automatically. When you create a BRMS or a BPM Suite project, the following directories are generated:

- **src/main/java** that stores the class files (facts).
- **src/main/resources/rules** that stores the .drl files (rules).
- **src/main/resources/process** that stores the .bpmn files (processes).
- **src/main/resources/Drools Library** that holds the generated .jar files required for rule execution.

9.4. CREATING A JBOSS BRMS PROJECT

Procedure 9.5. Task

To create a new JBoss BRMS project in the Drools perspective:

1. Go to **File** → **New** → **Project**.

   A **New Project** wizard opens.

2. Navigate to **Drools** → **Drools Project**.

   A **New Drools Project** wizard opens.

3. On the **New Drools Project** wizard, click **Next**.

4. Enter a name for your Drools project and click **Next**.

5. Check the required checkboxes with default artifacts you need in your project, and click **Next**.

   The **Drools Runtime** wizard opens.

6. Select a Drools runtime.

   If you have not set up a Drools runtime, click the **Configure Workspace Settings...** link. If you click this link, the workspace preferences showing the currently installed Drools runtimes opens. Add new runtimes there and click **OK**.

7. Select the Drools project version from the **`Select code compatible with:`** option.
8. Provide values for the following:
   - groupid: The id of the project's group or the root of your project's Java package name.
   - artifactid: The id of the artifact (project).
   - version: The version of the artifact under the specified group.
9. Click **`Finish`**.

If you checked the default artifacts checkboxes in the Drools Project wizard, you can see the newly created Drools project in the Package Explorer accordingly containing:

- A sample rule file **`Sample.drl`** in the **`src/main/resources/rules`** folder.
- A sample process file **`Sample.bpmn`** in the **`src/main/resources/process`** folder.
- An example java file **`DroolsTest.java`** in the **`src/main/java`** folder to execute the rules in the Drools engine in the com.sample package.
- All the jar files necessary for execution in the **`src/main/resources/Drools`** Library.

9.5. USING TEXTUAL RULE EDITOR

In the Package Explorer, you can double-click your existing rule file to open it on a textual rule editor or choose **File** → **New** → **Rule Resource** to create a new rule on the textual editor. The textual rule editor has a pattern of a normal text editor and this is where you modify and manage your rules. .

The textual rule editor works on files that have a **`.drl`** (or **`.rule`**) extension. Usually these contain related rules, but it is also possible to have rules in individual files, grouped by being in the same package namespace. These DRL files are plain text files. Even if your rule group is using a domain specific language (DSL), the rules are still stored as plain text. This allows easy management of rules and versions.

Textual editor provides features like:

- Content assistance: The pop-up content assistance helps you quickly create rule attributes such as functions, import statements, and package declarations. You can invoke pop-up content assistance by pressing **`Ctrl+Space`**.
- Code folding: Code Folding allows you to hide and show sections of a file use the icons with minus and plus on the left vertical line of the editor.
- Sysnchronization with outline view: The text editor is in sync with the structure of the rules in the outline view as soon as you save your rules. The outline view provides a quick way of navigating around rules by name, or even in a file containing hundreds of rules. The items are sorted alphabetically by default.

9.6. RED HAT JBOSS BRMS VIEWS

You can alternate between these views when modifying rules:

Working Memory View

Shows all elements in the Red Hat JBoss BRMS working memory.

Agenda View

Shows all elements on the agenda. For each rule on the agenda, the rule name and bound variables are shown.

Global Data View

Shows all global data currently defined in the Red Hat JBoss BRMS working memory.

Audit View

Can be used to display audit logs containing events that were logged during the execution of a rules engine, in tree form.

Rete View

This shows you the current Rete Network for your DRL file. You display it by clicking on the tab "Rete Tree" at the bottom of the DRL Editor window. With the Rete Network visualization being open, you can use drag-and-drop on individual nodes to arrange optimal network overview. You may also select multiple nodes by dragging a rectangle over them so the entire group can be moved around.

NOTE

The Rete view works only in projects where the rule builder is set in the project´s properties. For other projects, you can use a workaround. Set up a JBoss BRMS Project next to your current project and transfer the libraries and the DRLs you want to inspect with the Rete view. Click on the right tab below in the DRL Editor, then click "Generate Rete View".

9.7. DEBUGGING RULES

1. Drools breakpoints are only enabled if you debug your application as a Drools Application. To do this you should perform one of two actions:

   - Select the main class of your application. Right-click on it and select **Debug As → Drools Application** .

   - Alternatively, select **Debug As → Debug Configuration** to open a new dialog window for creating, managing and running debug configurations.

     Select the **`Drools Application`** item in the left tree and click the **`New launch configuration`** button (leftmost icon in the toolbar above the tree). This will create a new configuration with a number of the properties already filled in based on main class you selected in the beginning. All properties shown here are the same as any standard Java program.

NOTE

Remember to change the name of your debug configuration to something meaningful.

2. Click the **`Debug`** button on the bottom to start debugging your application.

3. After enabling the debugging, the application starts executing and will halt if any breakpoint is encountered. This can be a Drools rule breakpoint, or any other standard Java breakpoint. Whenever a Drools rule breakpoint is encountered, the corresponding **`.drl`** file is opened and the active line is highlighted. The **`Variables`** view also contains all rule parameters and their value. You can then use the default Java debug actions to decide what to do next (resume, terminate, step over, etc). The debug views can also be used to determine the contents of the working memory and agenda at that time as well (the current executing working memory is automatically shown).

9.7.1. Creating Breakpoints

Create breakpoints to help monitor rules that have been executed. Instead of waiting for the result to appear at the end of the process, you can inspect the details of the execution at each breakpoint you set. This is useful for debugging and ensuring rules are executed as expected.

1. To create breakpoints in the Package Explorer view or Navigator view of the JBoss BRMS perspective, double-click the selected **`.drl`** file to open it in the editor.

2. You can add and remove rule breakpoints in the **`.drl`** files in two ways:

   - Double-click the rule in the **`Rule editor`** at the line where you want to add a breakpoint. A breakpoint can be removed by double-clicking the rule once more.

     NOTE

     Rule breakpoints can only be created in the consequence of a rule. Double-clicking on a line where no breakpoint is allowed does nothing.

   - Right-click the ruler. Select the **Toggle Breakpoint** action in the context menu. Choosing this action adds a breakpoint at the selected line or remove it if there is one already.

3. The **`Debug perspective`** contains a **`Breakpoints view`** which can be used to see all defined breakpoints, get their properties, enable/disable and remove them. You can switch to it by clicking **Window** → **Perspective** → **Others** → **Debug** .

PART III. ALL ABOUT PROCESSES

CHAPTER 10. GETTING STARTED WITH PROCESSES

JBoss Business Process Management System is a light-weight, open-source, flexible Business Process Management (BPM) Suite that allows you to create, execute, and monitor business processes throughout their life cycle. The business processes allow you to model your business goals. They describe the steps that need to be executed to achieve those goals. It depicts the order of these goals in a flow chart. The business processes greatly improve the visibility and agility of your business logic.

JBoss BPM Suite creates the bridge between business analysts, developers and end users by offering process management features and tools in a way that both business users and developers like. The life cycle of a Business processes includes authoring, deployment, process management and task lists, and dashboards and reporting.

10.1. THE JBOSS BPM SUITE ENGINE

The core of JBoss BPM Suite is a light-weight, extensible workflow engine called the BPM Suite engine in BPMN 2.0 format, written in pure Java that allows you to execute business processes. It can run in any Java environment, embedded in your application or as a service. It has the following features:

- Solid, stable core engine for executing your process instances.
- Native support for the latest BPMN 2.0 specification for modeling and executing business processes.
- Strong focus on performance and scalability.
- Light-weight. You can deploy it on almost any device that supports a simple Java Runtime Environment. It does not require any web container at all.
- Pluggable persistence with a default JPA implementation (Optional).
- Pluggable transaction support with a default JTA implementation.
- Implemented as a generic process engine, so it can be extended to support new node types or other process languages.
- Listeners to be notified of various events.
- Ability to migrate running process instances to a new version of their process definition.

10.2. INTEGRATING BPM SUITE ENGINE WITH OTHER SERVICES

The JBoss BPM Suite engine can be integrated with a few independent core services such as:

- The human task service

  The human task service helps manage human tasks when human actors need to participate in the process. It is fully pluggable and the default implementation is based on the WS-HumanTask specification and manages the life cycle of the tasks, task lists, task forms, and some more advanced features like escalation, delegation, and rule-based assignments.

- The history log

  The history log stores all information about the execution of all the processes in the engine. This is necessary if you need access to historic information as runtime persistence only stores the current state of all active process instances. The history log can be used to store all current and

historic states of active and completed process instances. It can be used to query for any information related to the execution of process instances, for monitoring, and analysis.

CHAPTER 11. WORKING WITH PROCESSES

11.1. BPMN 2.0 NOTATION

11.1.1. Business Process Model and Notation (BPMN) 2.0 Specification

The Business Process Model and Notation (BPMN) 2.0 specification defines a standard for graphically representing a business process; it includes execution semantics for the defined elements and an XML format to store and share process definitions.

The table below shows the supported elements of the BPMN 2.0 specification and includes some additional elements and attributes.

Table 11.1. BPMN 2.0 Supported Elements and Attributes

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
definitions		rootElement BPMNDiagram		
process	processType isExecutable name id	property laneSet flowElement	packageName adHoc version	import global
sequenceFlow	sourceRef targetRef isImmediate name id	conditionExpression	priority	
interface	name id	operation		
operation	name id	inMessageRef		
laneSet		lane		
lane	name id	flowNodeRef		
import*		name		
global*		identifier type		
Events				
startEvent	name id	dataOutput dataOutputAssociation outputSet eventDefinition	x y width height	

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
endEvent	name id	dataInput dataInputAssociati on inputSet eventDefinition	x y width height	
intermediateCatch Event	name id	dataOutput dataOutputAssocia tion outputSet eventDefinition	x y width height	
intermediateThrow Event	name id	dataInput dataInputAssociati on inputSet eventDefinition	x y width height	
boundaryEvent	cancelActivity attachedToRef name id	eventDefinition	x y width height	
terminateEventDef inition				
compensateEvent Definition	activityRef	documentation extensionElements		
conditionalEventD efinition		condition		
errorEventDefinitio n	errorRef			
error	errorCode id			
escalationEventDe finition	escalationRef			
escalation	escalationCode id			
messageEventDefi nition	messageRef			
message	itemRef id			
signalEventDefiniti on	signalRef			

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
timerEventDefinition		timeCycle timeDuration		
Activities				
task	name id	ioSpecification dataInputAssociation dataOutputAssociation	taskName x y width height	
scriptTask	scriptFormat name id	script	x y width height	
script		text[mixed content]		
userTask	name id	ioSpecification dataInputAssociation dataOutputAssociation resourceRole	x y width height	onEntry-script onExit-script
potentialOwner		resourceAssignmentExpression		
resourceAssignmentExpression		expression		
businessRuleTask	name id		x y width height ruleFlowGroup	onEntry-script onExit-script
manualTask	name id		x y width height	onEntry-script onExit-script
sendTask	messageRef name id	ioSpecification dataInputAssociation	x y width height	onEntry-script onExit-script
receiveTask	messageRef name id	ioSpecification dataOutputAssociation	x y width height	onEntry-script onExit-script
serviceTask	operationRef name id	ioSpecification dataInputAssociation dataOutputAssociation	x y width height	onEntry-script onExit-script

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
subProcess	name id	flowElement property loopCharacteristics	x y width height	
adHocSubProcess	cancelRemainingInstances name id	completionCondition flowElement property	x y width height	
callActivity	calledElement name id	ioSpecification dataInputAssociation dataOutputAssociation	x y width height waitForCompletion independent	onEntry-script onExit-script
multiInstanceLoopCharacteristics		loopDataInputRef inputDataItem		
onEntry-script*	scriptFormat		script	
onExit-script*	scriptFormat		script	
Gateways				
parallelGateway	gatewayDirection name id		x y width height	
eventBasedGateway	gatewayDirection name id		x y width height	
exclusiveGateway	default gatewayDirection name id		x y width height	
inclusiveGateway	default gatewayDirection name id		x y width height	
Data				
property	itemSubjectRef id			
dataObject	itemSubjectRef id			
itemDefinition	structureRef id			

Element	Supported attributes	Supported elements	Extension attributes	Extension elements
ioSpecification		dataInput dataOutput inputSet outputSet		
dataInput	name id			
dataInputAssociation		sourceRef targetRef assignment		
dataOutput	name id			
dataOutputAssociation		sourceRef targetRef assignment		
inputSet		dataInputRefs		
outputSet		dataOutputRefs		
assignment		from to		
formalExpression	language	text[mixed content]		
BPMNDI				
BPMNDiagram		BPMNPlane		
BPMNPlane	bpmnElement	BPMNEdge BPMNShape		
BPMNShape	bpmnElement	Bounds		
BPMNEdge	bpmnElement	waypoint		
Bounds	x y width height			
waypoint	x y			

11.1.2. BPMN 2.0 Process Example

Here is a BPMN 2.0 process that prints out a "Hello World" statement when the process is started:

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

<definitions id="Definition"
```

```
            targetNamespace="http://www.example.org/MinimalExample"

            typeLanguage="http://www.java.com/javaTypes"

            expressionLanguage="http://www.mvel.org/2.0"

            xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"

            xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"

xs:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL BPMN20.xsd"

            xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"

            xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"

            xmlns:di="http://www.omg.org/spec/DD/20100524/DI"

            xmlns:tns="http://www.jboss.org/drools">


  <process processType="Private" isExecutable="true"
id="com.sample.HelloWorld" name="Hello World" >


    <!-- nodes -->

    <startEvent id="_1" name="StartProcess" />

    <scriptTask id="_2" name="Hello" >

      <script>System.out.println("Hello World");</script>

    </scriptTask>

    <endEvent id="_3" name="EndProcess" >

        <terminateEventDefinition/>

    </endEvent>


    <!-- connections -->

    <sequenceFlow id="_1-_2" sourceRef="_1" targetRef="_2" />

    <sequenceFlow id="_2-_3" sourceRef="_2" targetRef="_3" />


  </process>
```

```
  <bpmndi:BPMNDiagram>
    <bpmndi:BPMNPlane bpmnElement="Minimal" >
      <bpmndi:BPMNShape bpmnElement="_1" >
        <dc:Bounds x="15" y="91" width="48" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_2" >
        <dc:Bounds x="95" y="88" width="83" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_3" >
        <dc:Bounds x="258" y="86" width="48" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge bpmnElement="_1-_2" >
        <di:waypoint x="39" y="115" />
        <di:waypoint x="75" y="46" />
        <di:waypoint x="136" y="112" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge bpmnElement="_2-_3" >
        <di:waypoint x="136" y="112" />
        <di:waypoint x="240" y="240" />
        <di:waypoint x="282" y="110" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>

</definitions>
```

11.1.3. Supported Elements and Attributes in BPMN 2.0 Specification

JBoss BPM Suite 6 does not implement all elements and attributes as defined in the BPMN 2.0

specification. However, we do support significant node types that you can use inside executable processes. This includes almost all elements and attributes as defined in the Common Executable subclass of the BPMN 2.0 specification, extended with some additional elements and attributes we believe are valuable in that context as well. The full set of elements and attributes that are supported can be found below, but it includes elements like:

- Flow objects
  - Events
    - Start Event (None, Conditional, Signal, Message, Timer)
    - End Event (None, Terminate, Error, Escalation, Signal, Message, Compensation)
    - Intermediate Catch Event (Signal, Timer, Conditional, Message)
    - Intermediate Throw Event (None, Signal, Escalation, Message, Compensation)
    - Non-interrupting Boundary Event (Escalation, Signal, Timer, Conditional, Message)
    - Interrupting Boundary Event (Escalation, Error, Signal, Timer, Conditional, Message, Compensation)
  - Activities
    - Script Task
    - Task
    - Service Task
    - User Task
    - Business Rule Task
    - Manual Task
    - Send Task
    - Receive Task
    - Reusable Sub-Process (Call Activity)
    - Embedded Sub-Process
    - Event Sub-Process
    - Ad-Hoc Sub-Process
    - Data-Object
  - Gateways
    - Diverging
      - Exclusive
      - Inclusive

        - Parallel
        - Event-Based
    - Converging
        - Exclusive
        - Inclusive
        - Parallel
  - Lanes
- Data
  - Java type language
  - Process properties
  - Embedded Sub-Process properties
  - Activity properties
- Connecting objects
  - Sequence flow

11.1.4. Loading and Executing a BPMN2 Process Into Repository

The following example shows how you can load a BPMN2 process into your knowledge base:

```
import org.kie.api.KieServices;
import org.kie.api.builder.KieRepository;
import org.kie.api.builder.KieFileSystem;
import org.kie.api.builder.KieBuilder;
import org.kie.api.runtime.KieContainer;
import org.kie.api.KieBase;
...
KieServices kServices = KieServices.Factory.get();
KieRepository kRepository = kServices.getRepository();
KieFileSystem kFileSystem = kServices.newKieFileSystem();

kFileSystem.write(ResourceFactory.newClassPathResource("MyProcess.bpmn"));

KieBuilder kBuilder = kServices.newKieBuilder(kFileSystem);
kBuilder.buildAll();

KieContainer kContainer =
kServices.newKieContainer(kRepository.getDefaultReleaseId());
KieBase kBase = kContainer.getKieBase();
```

11.2. WHAT COMPRISES A BUSINESS PROCESS

A business process is a graph that describes the order in which a series of steps need to be executed using a flow chart. A process consists of a collection of nodes that are linked to each other using

connections. Each of the nodes represents one step in the overall process, while the connections specify how to transition from one node to the other. A large selection of predefined node types have been defined.

A typical process consists of the following parts:

- The header part that comprises global elements such as the name of the process, imports, and variables.
- The nodes section that contains all the different nodes that are part of the process.
- The connections section that links these nodes to each other to create a flow chart.

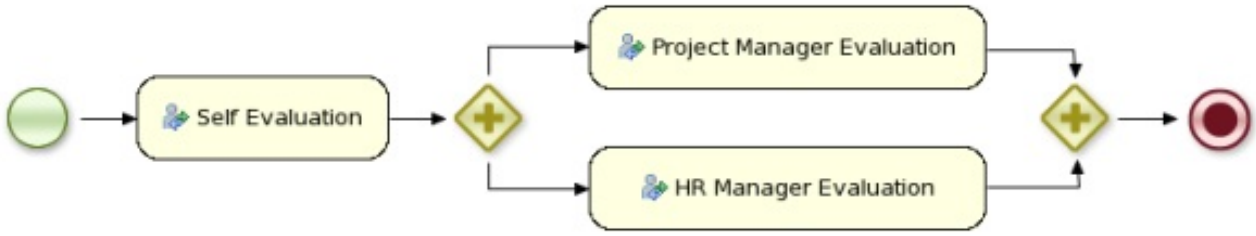
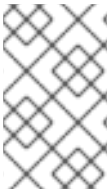


Figure 11.1. A Business Process

Processes can be created with the following methods:

- Using the Business Central or JBoss Developer Studio with BPMN2 modeler
- As an XML file, according to the XML process format as defined in the XML Schema Definition in the BPMN 2.0 specification.
- By directly creating a process using the Process API.



NOTE

The JBoss Developer Studio Process editor has been deprecated in favor of BPMN2 Modeler for process modeling as it is not being developed any more. However, you can still use it for limited number of supported elements.

11.2.1. Process Nodes

Executable processes consist of different types of nodes which are connected to each other. The BPMN 2.0 specification defines three main types of nodes:

Events

Event elements represent a particular event that occurs or can occur during process runtime.

Activities

Activities represent relatively atomic pieces of work that need to be performed as part of the Process execution.

Gateways

Gateways represent forking or merging of workflows during Process execution.

11.2.2. Process Properties

Every process has the following properties:

- ID: The unique ID of the process
- Name: The display name of the process
- Version: The version number of the process
- Package: The package (namespace) the process is defined in
- Variables (optional): Variables to store data during the execution of your process
- Swimlanes: Swimlanes used in the process for assigning human tasks

11.2.3. Defining Processes Using XML

You can create processes directly in XML format using the BPMN 2.0 specifications. The syntax of these XML processes is defined using the BPMN 2.0 XML Schema Definition.

The process XML file consists of:

- The "process" element

  This is the top part of the process XML that contains the definition of the different nodes and their properties. The process XML consist of exactly one <process> element. This element contains parameters related to the process (its type, name, id and package name), and consists of three subsections: a header section (where process-level information like variables, globals, imports and lanes can be defined), a nodes section that defines each of the nodes in the process, and a connections section that contains the connections between all the nodes in the process.

- The "BPMNDiagram" element

  This is the lower part of the process XML that contains all graphical information, like the location of the nodes. In the nodes section, there is a specific element for each node, defining the various parameters and, possibly, sub-elements for that node type.

The following XML fragment shows a simple process that contains a sequence of a Start Event, a Script Task that prints "Hello World" to the console, and an End Event:

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

<definitions id="Definition"

             targetNamespace="http://www.jboss.org/drools"

             typeLanguage="http://www.java.com/javaTypes"

             expressionLanguage="http://www.mvel.org/2.0"

             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"Rule Task

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


xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
BPMN20.xsd"
```

```
            xmlns:g="http://www.jboss.org/drools/flow/gpd"

            xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"

            xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"

            xmlns:di="http://www.omg.org/spec/DD/20100524/DI"

            xmlns:tns="http://www.jboss.org/drools">


  <process processType="Private" isExecutable="true" id="com.sample.hello"
name="Hello Process" >


    <!-- nodes -->

    <startEvent id="_1" name="Start" />

    <scriptTask id="_2" name="Hello" >

      <script>System.out.println("Hello World");</script>

    </scriptTask>

    <endEvent id="_3" name="End" >

        <terminateEventDefinition/>

    </endEvent>


    <!-- connections -->

    <sequenceFlow id="_1-_2" sourceRef="_1" targetRef="_2" />

    <sequenceFlow id="_2-_3" sourceRef="_2" targetRef="_3" />


  </process>


  <bpmndi:BPMNDiagram>

    <bpmndi:BPMNPlane bpmnElement="com.sample.hello" >

      <bpmndi:BPMNShape bpmnElement="_1" >

        <dc:Bounds x="16" y="16" width="48" height="48" />

      </bpmndi:BPMNShape>
```

```
      <bpmndi:BPMNShape bpmnElement="_2" >
        <dc:Bounds x="96" y="16" width="80" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape bpmnElement="_3" >
        <dc:Bounds x="208" y="16" width="48" height="48" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge bpmnElement="_1-_2" >
        <di:waypoint x="40" y="40" />
        <di:waypoint x="136" y="40" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge bpmnElement="_2-_3" >
        <di:waypoint x="136" y="40" />
        <di:waypoint x="232" y="40" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>
```

11.3. ACTIVITIES

An activity is an action performed inside a business process. Activities are classified based on the type of tasks they do:

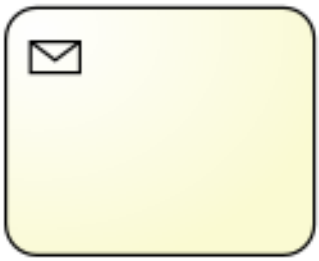
- Task: Use this activity type in your business process to implement a single task which can not be further broken into subtasks.
- Subprocess: Use this activity type in your business process when you have a group of tasks to be processed in a sequential order in order to achieve a single result.

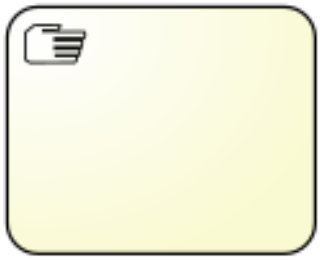
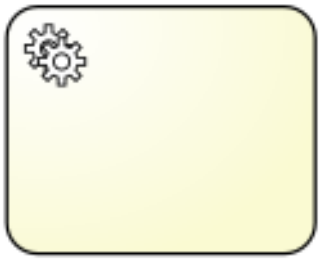
Each activity has one incoming and one outgoing connection.

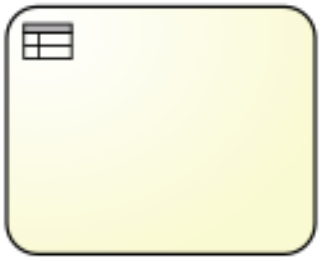
11.3.1. Tasks

A task is an action that is executed inside a business process. Tasks can be of the following types:

Table 11.2. Types of Tasks in the Object Library

Task	Icon	Description
User		Use the User task activity type in your business process when you require a human actor to execute your task. • The User task defines within it, the type of task that needs to be executed. You must pass the data that a human actor may require to execute this task as the content of the task. • The user task has one incoming and one outgoing connection. You can use the user tasks in combination with Swimlanes to assign multiple human tasks to similar human actors.
Send		Use the Send task to send a message. • A send task has a message associated with it. • When a Send task is activated, the message data is assigned to the data input property of the Send task. A Send task completes when this message is sent.
Receive		Use the Receive task in your process when your process is relying on a specific message to continue. • When a Receive task receives the specified message, the data from the message is transferred to the Data Output property of the Receive task and the task completes.

Task	Icon	Description
`Manual`		Use the Manual task when you require a task to be executed by a human actor that need not be managed by your process. • The difference between a manual task and a user task is that a user task is executed in the context of the process, requires system interaction to accomplish the task, and are assigned to specific human actors. The manual tasks on the other hand, execute without the need to interact with the system and not managed by the process.
`Service`		Use the Service task in your business process for specifying the tasks use a service (such as a web service) that must execute outside the process engine. • The Service task may use any service such as email server, message logger, or any other automated service. • You can specify the required input parameters and expected results of this task in its properties. When the associated work is executed and specified result is received, the Service task completes.

Task	Icon	Description
`Business Rule`		Use the Business Rule task when you want a set of rules to be executed as a task in your business process flow. • During the execution of your process flow, when the engine reaches the Business Rule task, all the rules associated with this task are fired and evaluated. • The DataInputSet and DataOutputSet properties define the input to the rule engine and the calculated output received from the rule engine respectively. • The set of rules that this task runs are defined in **`.drl`** format. • All the rules that belong to a Business Rule task must belong to a specific ruleflow group. You can assign a rule its ruleflow group using the ruleflow-group attribute in the header of the rule. So when a Business Rule task executes, all the rules that belong to the ruleflow-group specified in the ruleflow-group property of the task are executed.

Task	Icon	Description
`Script`		Use the Script task in your business process when you want a script to be executed within the task. • A Script task has an associated action that contains the action code and the language that the action is written in. • When a Script Task is reached in the process, it executes the action and then continues to the next node. • Use a Script task in your process to for modeling low level behavior such as manipulating variables. For a complex model, use a Service task. • Ensure that the script associated with a Script task is executed as soon as the task is reached in a business process. If that is not possible, use an asynchronous Service task instead. • Ensure that your script does not contact an external service as the process engine has no visibility of the external services that a script may call. • Ensure that any exception that your script may throw must be caught within the script itself.
`None`		A None task type is an abstract undefined task type.

11.3.2. Subprocesses

A subprocess is a process within another process. When a parent process calls a child process (subprocess), the child process executes in a sequential manner and once complete, the execution control then transfers to the main parent process. Subprocess can be of the following types:

Table 11.3. Types of Subprocesses in the Object Library

Subprocess	Icon	Description
`Reusable`		Use the Reusable subprocess to invoke another process from the parent process. • The Reusable subprocess is independent from its parent process.
`Multiple Instances`		Use the Multiple Instances subprocess when you want to execute the contained subprocess elements multiple number of times. • When the engine reaches a Multiple Instance subprocess in your process flow, the subprocess instances are executed in a sequential manner. • The Multiple Instances subprocess is completed when the condition specified in the MI completion condition property is satisfied.

Subprocess	Icon	Description
Embedded		Use the Embedded subprocess if you want a decomposable activity inside your process flow that encapsulates a part of your main process. • When you expand an Embedded subprocess, you can see a valid BPMN diagram inside it that comprises a Start Event and at least one End Event. • An Embedded subprocess allows you to define local subprocess variables that are accessible to all elements inside this subprocess.
Ad-Hoc		Use an Ad-Hoc subprocess when you want to execute activities inside your process, for which the execution order is irrelevant. An Ad-Hoc subprocess is a group of activities that have no required sequence relationships. • You can define a set of activities for the this subprocess, but the sequence and number of performances for the activities is determined by the performers of the activities. • Use Ad-Hoc subprocesses in cases such as executing a list of tasks that have no dependencies between them and can be executed in any order.

Subprocess	Icon	Description
Event		Use the Event subprocess in your process flow when you want to handle events that occur within the boundary of a subprocess. • The Event subprocess differ from the other subprocess as they are not a part of the regular process flow and occur only in the context of a subprocess. • An Event subprocess becomes active when its start event gets triggered. • An Event subprocess can be interrupting or non-interrupting. An interrupting Event subprocess interrupts the parent process and a non-interrupting Event subprocess does not.

11.4. DATA

Throughout the execution of a process, data can be retrieved, stored, passed on, and used. To store runtime data during the execution of the process, process variables are used. A variable is defined with a name and a data type. A basic data type could include the following: boolean, int, String, or any kind of object subclass.

Variables can be defined inside a variable scope. The top-level scope is the variable scope of the process itself. Sub-scopes can be defined using a sub-process. Variables that are defined in a sub-scope are only accessible for nodes within that scope.

Whenever a variable is accessed, the process will search for the appropriate variable scope that defines the variable. Nesting variable scopes are allowed. A node will always search for a variable in its parent container; if the variable cannot be found, the node will look in the parent's parent container, and so on, until the process instance itself is reached. If the variable cannot be found, a read access yields null, and a write access produces an error message. All of this occurs with the process continuing execution.

Variables can be used in the following ways:

- Process-level variables can be set when starting a process by providing a map of parameters to the invocation of the startProcess method. These parameters will be set as variables on the process scope.
- Script actions can access variables directly simply by using the name of the variable as a local parameter in their script. For example, if the process defines a variable of type "org.jbpm.Person" in the process, a script in the process could access this directly:

```
// call method on the process variable "person"

person.setAge(10);
```

Changing the value of a variable in a script can be done through the knowledge context:

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

- Service tasks (and reusable sub-processes) can pass the value of process variables to the outside world (or another process instance) by mapping the variable to an outgoing parameter. For example, the parameter mapping of a service task could define that the value of the process variable x should be mapped to a task parameter y just before the service is invoked. You can also inject the value of the process variable into a hard-coded parameter String using #{expression}. For example, the description of a human task could be defined as the following:

  ```
  You need to contact person #{person.getName()}
  ```

  Where person is a process variable. This will replace this expression with the actual name of the person when the service needs to be invoked. Similarl results of a service (or reusable sub-process) can also be copied back to a variable using result mapping.

- Various other nodes can also access data. Event nodes, for example, can store the data associated to the event in a variable. Check the properties of the different node types for more information.

Finally, processes (and rules) have access to globals, i.e., globally defined variables and data in the Knowledge Session. Globals are directly accessible in actions like variables. Globals need to be defined as part of the process before they can be used. Globals can be set using the following:

```
ksession.setGlobal(name, value)
```

Globals can also be set from inside process scripts using:

```
kcontext.getKieRuntime().setGlobal(name,value);.
```

11.5. EVENTS

Events are triggers, which when occur, impact a business process. Events are classified as start events, end events, and intermediate events. A start event indicates the beginning of a business process. An end event indicates the completion of a business process. And intermediate events drive the flow of a business process. Every event has an event ID and a name. You can implement triggers for each of these event types to identify the conditions under which an event is triggered. If the conditions of the triggers are not met, the events are not initialized, and hence the process flow does not complete.

11.5.1. Start Events

A start event is a flow element in a business process that indicates the beginning of a business process flow. The execution of a business process starts at this node, so a process flow can only have one start event. A start event can have only one outgoing connection which connects to another node to take the process flow ahead. Start events are of the following types:

Table 11.4. Types of Start Events in the Object Library

Event	Icon	Description
None		Use the None start events when your processes do not need a trigger to be initialized. • You can use the start event if your process does not depend on any condition to begin. • The start event is mostly used to initialize a subprocess or a process that needs to trigger by default or the trigger for the process is irrelevant.
Message		Use the Message start event when you require your process to start, on receiving a particular message. • You can have multiple Message start events in your process. • A single message can trigger multiple Message Start Events that instantiates multiple processes.
Timer		Use the Timer start event when you require your process to initialize at a specific time, specific points in time, or after a specific time span. • The Timer start event is mostly used in cases where a waiting state is required, for example, in cases involving a Human Task.

Event	Icon	Description
`Escalation`		Use the Escalation start event in your subprocesses when you require your subprocess to initialize as a response to an escalation. • An escalation is identified by an escalation object in the main process, which is inserted into the main process by an Escalation Intermediate event or/and Escalation end event. An Escalation Intermediate event or/and Escalation end event produce an escalation object, which can be consumed by an Escalation Start event or an Escalation intermediate catch event. • A process flow can have one or more Escalation start events and the process flow does not complete until all the escalation objects are caught and handled in subprocesses.
`Conditional`		Use the Conditional start event to start a process instance based on a business condition. • A condition output is a Boolean value and when a condition is evaluated as True, the process flow is initialized. • You can have one or more Conditional start events in your business process.

Event	Icon	Description
Error		Use the Error start event in a subprocess when you require your subprocess to trigger as a response to a specific error object. • An error object indicates an incorrect process ending and must be handled for the process flow to complete. • An error object is inserted into a business process by an Error end event and can be handled by a Error intermediate catch event, or Error start event depending on the scope of the error in a process flow.
Compensation		Use the Compensation start event in a subprocess when you require to handle a compensation. • A compensation means undoing the results of an already completed action. Note that this is different than an error. An error suspends a process at the location where it occurs, however, a compensation compensates the results of an action the process has already committed and needs to be undone. • A Compensation start event starts a subprocess and is the target Activity of a Compensation intermediate event.

Event	Icon	Description
Signal		Use the Signal start event to start a process instance based on specific signals received from other processes. • A signal is identified by a signal object. A signal object defines a unique reference ID that is unique in a session. • A signal object is inserted in a process by a throw signal intermediate event as an action of an activity.

11.5.2. End Events

An end event marks the end of a business process. Your business process may have more than one end event. An end event has one incoming connection and no outgoing connections. End events are of the following types:

Table 11.5. Types of End Events in the Object Library

Event	Icon	Description
None		Use the None error end event to mark the end of your process or a subprocess flow. Note that this does not influence the workflow of any parallel subprocesses.
Message		Use the Message end event to end your process flow with a message to an element in another process. An intermediate catch message event or a start message event in another process can catch this message to further process the flow.
Escalation		Use the Escalation end event to mark the end of a process as a result of which the case in hand is escalated. This event creates an escalation signal that further triggers the escalation process.

Event	Icon	Description
`Error`		Use the Error end event in your process or subprocess to end the process in an error state and throw a named error, which can be caught by a Catching Intermediate event.
`Cancel`		Use the Cancel end event to end your process as canceled. Note that if your process comprises any compensations, it completes them and then marks the process as canceled.
`Compensation`		Use the Compensation end event to end the current process and trigger compensation as the final step.
`Signal`		Use the Signal end event to end a process with a signal thrown to an element in one or more other processes. Another process can catch this signal using Catch intermediate events.
`Terminate`		Use the Terminate end event to terminate the entire process instance immediately. Note that this terminates all the other parallel execution flows and cancels any running activities.

11.5.3. Intermediate Events

Intermediate events occur during the execution of a process flow, and they drive the flow of the process. Some specific situations in a process may trigger these intermediate events. Intermediate events can occur in a process with one or no incoming flow and an outgoing flow. Intermediate events can further be classified as :

- Catching Intermediate Events
- Throwing Intermediate Events

11.5.3.1. Catching Intermediate Events

Catching intermediate events comprises intermediate events which implement a response to specific indication of a situation from the main process workflow. Catching intermediate events are of the following types:

- **`Message`**: Use the Message catching intermediate events in your process to implement a

reaction to an arriving message. The message that this event is expected to react to, is specified in its properties. It executes the flow only when it receives the specific message.

- **`Timer`**: Use the Timer intermediate event to delay the workflow execution until a specified point or duration. A Timer intermediate event has one incoming flow and one outgoing flow and its execution starts when the incoming flow transfers to the event. When placed on an activity boundary, the execution is triggered at the same time as the activity execution.

- **`Escalation`**: Use the Escalation catching intermediate event in your process to consume an Escalation object. An Escalation catching intermediate event awaits a specific escalation object defined in its properties. Once it receives the object, it triggers execution of its outgoing flow.

- **`Conditional`**: Use the Conditional intermediate event to execute a workflow when a specific business Boolean condition that it defines, evaluates to true. When placed in the Process workflow, a Conditional intermediate event has one incoming flow and one outgoing flow and its execution starts when the incoming flow transfers to the event. When placed on an activity boundary, the execution is triggered at the same time as the activity execution. Note that if the event is non-interrupting, it triggers continuously while the condition is true.

- **`Error`**: Use the Error catching intermediate event in your process to execute a workflow when it received a specific error object defined in its properties.

- **`Compensation`**: Use the Compensation intermediate event to handle compensation in case of partially failed operations. A Compensation intermediate event is a boundary event that is attached to an activity in a transaction subprocess that may finish with a Compensation end event or a Cancel end event. The Compensation intermediate event must have one outgoing flow that connects to an activity that defines the compensation action needed to compensate for the action performed by the activity.

- **`Signal`**: Use the Signal catching intermediate event to execute a workflow once a specified signal object defined in its properties is received from the main process or any other process.

11.5.3.2. Throwing Intermediate Events

Throwing intermediate events comprises events which produce a specified trigger in the form of a message, escalation, or signal, to drive the flow of a process. Throwing intermediate events are of the following types:

- **`Message`**: Use the Message throw intermediate event to produce and send a message to a communication partner (such as an element in another process). Once it sends a message, the process execution continues.

- **`Escalation`**: Use the Escalation throw intermediate event to produce an escalation object.Once it creates an escalation object, the process execution continues. The escalation object can be consumed by an Escalation Start event or an Escalation intermediate catch event, which is looking for this specific escalation object.

- **`Signal`**: Use the Signal throwing intermediate events to produces a signal object. Once it creates a signal object, the process execution continues. The signal object is consumed by a Signal start event or a Signal catching intermediate event, which is looking for this specific signal object.

11.6. GATEWAYS

“Gateways are used to control how Sequence Flows interact as they converge and diverge within a Process.[1]”

Gateways are used to create or synchronize branches in the workflow using a set of conditions which is called the gating mechanism. Gateways are either converging (multiple Flows into one Flow) or diverging (One Flow into multiple Flows).

One Gateway cannot have multiple incoming *and* multiple outgoing Flows.

Depending on the gating mechanism you want to apply, you can use the following types of gateways:

- Parallel (AND): in a converging gateway, waits for all incoming Flows. In a diverging gateway, takes all outgoing Flows simultaneously;
- Inclusive (OR): in a converging gateway, waits for all incoming Flows whose condition evaluates to true. In a diverging gateway takes all outgoing Flows whose condition evaluates to true;
- Exclusive (XOR): in a converging gateway, the next node for each trigger is chosen. In a diverging gateway only one outgoing Flow is chosen.
- Event-based: used only in diverging gateways for reacting to events. See Section 11.6.1.1, “Event-based Gateway”
- Data-based Exclusive: used in both diverging and converging gateways to make decisions based on available data. See Section 11.6.1.4, “Data-based Exclusive Gateway”

11.6.1. Gateway types

11.6.1.1. Event-based Gateway

“The Event-Based Gateway has pass-through semantics for a set of incoming branches (merging behavior). Exactly one of the outgoing branches is activated afterwards (branching behavior), depending on which of Events of the Gateway configuration is first triggered. [2]”

The Gateway is only diverging and allows you to react to possible Events as opposed to the Data-based Exclusive Gateway, which reacts to the process data. It is the Event that actually occurs that decides which outgoing Flow is taken. As it provides the mechanism to react to exactly one of the possible Events, it is exclusive, that is, only one outgoing Flow is taken.

The Gateway might act as a Start Event, where the process is instantiated only if one the Intermediate Events connected to the Event-Based Gateway occurs.

11.6.1.2. Parallel Gateway

“A Parallel Gateway is used to synchronize (combine) parallel flows and to create parallel flows.[3]”

Diverging

Once the incoming Flow is taken, all outgoing Flows are taken simultaneously.

Converging

The Gateway waits untill all incoming Flows have entered and only then triggers the outgoing Flow.

11.6.1.3. Inclusive Gateway

Diverging

Once the incoming Flow is taken, all outgoing Flows whose condition evaluates to true are taken. Connections with lower priority numbers are triggered before triggering higher priority ones; priorities are evaluated but the BPMN2 specification doesn't guarantee this. So for portability reasons it is recommended that you do not depend on this.

IMPORTANT

Make sure that at least one of the outgoing Flow evaluates to true at runtime; otherwise, the process instance terminates with a runtime exception.

Converging

The Gateway merges all incoming Flows previously created by a diverging Inclusive Gateway; that is, it serves as a synchronizing entry point for the Inclusive Gateway branches.

Attributes

Default gate

The outgoing Flow taken by default if no other Flow can be taken

11.6.1.4. Data-based Exclusive Gateway

Diverging

The Gateway triggers exactly one outgoing Flow: the Flow with the constraint evaluated to true and the *lowest* Priority is taken. After evaluating the constraints that are linked to the outgoing Flows: the constraint with the lowest priority number that evaluates to true is selected.

IMPORTANT

Make sure that at least one of the outgoing Flows evaluates to true at runtime: if no Flow can be taken, the execution returns a runtime exception.

Converging

The Gateway allows a workflow branch to continue to its outgoing Flow as soon as it reaches the Gateway; that is, whenever on of the incoming Flows triggers the Gateway, the workflow is sent to the outgoing Flow of the Gateway; if it is triggered from more than one incoming connection, it triggers the next node for each trigger.

Attributes

Default gate

The outgoing Flow taken by default if no other Flow can be taken

11.7. VARIABLES

Variables are elements that serve for storing a particular type of data during runtime. The type of data a variable contains is defined by its data type.

Just like any context data, every variable has its scope that defines its "visibility". An element, such as a Process, Sub-Process, or Task can only access variables in its own and parent contexts: variables defined in the element's child elements cannot be accessed. Therefore, when an elements requires access to a variable on runtime, its own context is searched first. If the variable cannot be found directly in the element's context, the immediate parent context is searched. The search continues to "level up" until the Process context is reached; in case of Globals, the search is performed directly on the Session container. If the variable cannot be found, a read access request returns **`null`** and a write access produces an error message, and the Process continues its execution. Variables are searched for based on their ID.

In Red Hat JBoss BPM Suite, variables can live in the following contexts:

- Session context: **`Globals`** are visible to all Process instances and assets in the given Session and are intended to be used primarily by business rules and by constrains. The are created dynamically by the rules or constrains.
- Process context: **`Process variables`** are defined as properties in the BPMN2 definition file and are visible within the Process instance. They are initialized at Process creation and destroyed on Process finish.
- Element context: **`Local variables`** are available within their Process element, such as an Activity. They are initialized when the element context is initialized, that is, when the execution workflow enters the node and execution of the OnEntry action finished if applicable. They are destroyed when the element context is destroyed, that is, when the execution workflow leaves the element.

  Values of local variables can be mapped to Global or Process variables using the Assignment mechanism (refer to Section 11.8, “Assignment”). This allows you to maintain relative independence of the parent Element that accommodates the local variable. Such isolation may help prevent technical exceptions.

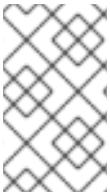
11.8. ASSIGNMENT

The assignment mechanism allows you to assign a value to an object, such as a variable, before or after the particular Element is executed.

When defining assignment on an Activity Element, the value assignment is performed either before or after Activity execution. If the assignment defines mapping to a local variable, the time when the assignment is performed depends on whether the local variable is defined as an DataInput or DataOutput item.

For example, if you need to assign a Task to a user whose ID is a Process variable, use the assignment to map the variable to the parameter ActorId.

Assignment is defined in the Assignments property in case of Activity Elements and in the DataInputAssocations or DataOutputAssociations property in case of non-Activity Elements.



NOTE

As parameters of the type String can make use of the assignment mechanism by applying the respective syntax directly in their value, **`#{userVariable}`**, assignment is rather intended for mapping of properties that are not of type String.

11.9. ACTION SCRIPTS

Action scripts are pieces of code that define the Script property or an Element's interceptor action. They have access to globals, the Process variables, and the predefined variable **`kcontext`**. Accordingly, **`kcontext`** is an instance of ProcessContext class and the interface content can be found at the following location: Interface ProcessContext.

Currently, dialects Java and MVEL are supported for action script definitions. Note that MVEL accepts any valid Java code and additionally provides support for nested access of parameters, for example, the MVEL equivalent of Java call **`person.getName()`** is **`person.name`**. It also provides other improvements over Java and MVEL expressions are generally more convenient for the business user.

Example 11.1. Action script that prints out the name of the person

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

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

11.10. CONSTRAINTS

There are two types of constraints in business processes: code constraints and rule constraints.

- Code constraints are boolean expressions evaluated directly whenever they are reached; these constraints are written in either Java or MVEL. Both Java and MVEL code constraints have direct access to the globals and variables defined in the process.

  Here is an example of a valid Java code constraint, person being a variable in the process:

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

  Here is an example of a valid MVEL code constraint, person being a variable in the process:

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

- Rule constraints are equal to normal Drools rule conditions. They use the Drools Rule Language syntax to express complex constraints. These rules can, like any other rule, refer to data in the working memory. They can also refer to globals directly. Here is an example of a valid rule constraint:

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

  This tests for a person older than 20 in the working memory.

Rule constraints do not have direct access to variables defined inside the process. However, it is possible to refer to the current process instance inside a rule constraint by adding the process instance to the working memory and matching for the process instance in your rule constraint. Logic is included to make sure that a variable processInstance of type WorkflowProcessInstance will only match the current process instance and not other process instances in the working memory. Note, it is necessary to insert the process instance into the session. If it is necessary to update the process instance, use Java code or an on-entry, on-exit, or explicit action in the process. The following example of a rule constraint will search for a person with the same name as the value stored in the variable ***`name`*** of the process:

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

11.11. TIMERS

Timers wait for a predefined amount of time before triggering, once or repeatedly. You can use timers to trigger certain logic after a certain period, or to repeat some action at regular intervals.

Configuring timer with delay and period

A Timer node is set up with a delay and a period. The delay specifies the amount of time to wait after node activation before triggering the timer for the first time. The period defines the time between subsequent trigger activations. A period of *0* results in a one-shot timer. The (period and delay) expression must be of the form *[#d][#h][#m][#s][#[ms]]*. You can specify the amount of days, hours, minutes, seconds, and milliseconds. Milliseconds is the default value. For example, the expression *1h* waits one hour before triggering the timer again.

Configuring timer ISO-8601 date format

You can configure timers version 6 timers with valid *ISO8601* date format that supports both one shot timers and repeatable timers. You can define timers as date and time representation, time duration or repeating intervals. For example:

```
    Date - 2013-12-24T20:00:00.000+02:00 - fires exactly at Christmas Eve
at 8PM
    Duration - PT1S - fires once after 1 second
    Repeatable intervals - R/PT1S - fires every second, no limit.
Alternatively R5/PT1S fires 5 times every second
```

Configuring timer with process variables

In addition to the above mentioned configuration options, you can specify timers using process variable that consists of string representation of either delay and period or ISO8601 date format. By specifying *#{variable}*, the engine dynamically extracts process variable and uses it as timer expression. The timer service is responsible for making sure that timers get triggered at the appropriate times. You can cancel timers so that they are no longer triggered. You can use timers in the following ways inside a process:

- You can add a timer event to a process flow. The process activation starts the timer, and when it triggers, once or repeatedly, it activates the timer node's successor. Subsequently, the outgoing connection of a timer with a positive period is triggered multiple times. Canceling a Timer node also cancels the associated timer, after which no more triggers occur.
- You can associate timer with a sub-process or tasks as a boundary event.

11.12. MULTI-THREADING

11.12.1. Multi-threading

In the following text, we will refer to two types of "multi-threading": *logical* and *technical*. *Technical multi-threading* is what happens when multiple threads or processes are started on a computer, for example by a Java or C program. *Logical multi-threading* is what we see in a BPM process after the process reaches a parallel gateway. From a functional standpoint, the original process will then split into two processes that are executed in a parallel fashion.

The BPM engine supports logical multi-threading; for example, processes that include a parallel gateway are supported. We've chosen to implement logical multi-threading using one thread; accordingly, a BPM process that includes logical multi-threading will only be executed in one technical thread. The main reason for doing this is that multiple (technical) threads need to be be able to communicate state information with each other if they are working on the same process. This requirement brings with it a number of complications. While it might seem that multi-threading would bring performance benefits with it, the extra logic needed to make sure the different threads work together well means that this is not guaranteed. There is also the extra overhead incurred because we need to avoid race conditions and deadlocks.

11.12.2. Engine Execution

In general, the BPM engine executes actions in serial. For example, when the engine encounters a script task in a process, it will synchronously execute that script and wait for it to complete before continuing execution. Similarly, if a process encounters a parallel gateway, it will sequentially trigger each of the outgoing branches, one after the other. This is possible since execution is almost always instantaneous, meaning that it is extremely fast and produces almost no overhead. As a result, the user will usually not even notice this. Similarly, action scripts in a process are also synchronously executed, and the engine will wait for them to finish before continuing the process. For example, doing a **`Thread.sleep(...)`** as part of a script will not make the engine continue execution elsewhere but will block the engine thread during that period.

The same principle applies to service tasks. When a service task is reached in a process, the engine will also invoke the handler of this service synchronously. The engine will wait for the **`completeWorkItem(...)`** method to return before continuing execution. It is important that your service handler executes your service asynchronously if its execution is not instantaneous.

An example of this would be a service task that invokes an external service. Since the delay in invoking this service remotely and waiting for the results might be too long, it might be a good idea to invoke this service asynchronously. This means that the handler will only invoke the service and will notify the engine later when the results are available. In the mean time, the process engine then continues execution of the process.

Human tasks are a typical example of a service that needs to be invoked asynchronously, as we don't want the engine to wait until a human actor has responded to the request. The human task handler will only create a new task (on the task list of the assigned actor) when the human task node is triggered. The engine will then be able to continue execution on the rest of the process (if necessary), and the handler will notify the engine asynchronously when the user has completed the task.

11.13. PROCESS FLUENT API

11.13.1. Using the Process Fluent API to Create Business Process

While it is recommended to define processes using the graphical editor or the underlying XML, you can also create a business process using the Process API directly. The most important process model elements are defined in the packages org.jbpm.workflow.core and org.jbpm.workflow.core.node.

Red Hat JBoss BPM Suite provides you a fluent API that allows you to easily construct processes in a readable manner using factories. You can then validate the process that you were constructing manually.

11.13.2. Process Fluent API Example

Here is an example of a basic process with only a script task:

```
RuleFlowProcessFactory factory =
RuleFlowProcessFactory.createProcess("org.jbpm.HelloWorld");
factory
// Header
.name("HelloWorldProcess")
.version("1.0")
.packageName("org.jbpm")
// Nodes
.startNode(1).name("Start").done()
.actionNode(2).name("Action")
.action("java", "System.out.println(\"Hello World\");").done()
.endNode(3).name("End").done()
// Connections
.connection(1, 2)
.connection(2, 3);
RuleFlowProcess process = factory.validate().getProcess();
KieServices ks = KieServices.Factory.get();
KieFileSystem kfs = ks.newKieFileSystem();
Resource resource = ks.getResources().newByteArrayResource(
XmlBPMNProcessDumper.INSTANCE.dump(process).getBytes());
resource.setSourcePath("helloworld.bpmn2");
kfs.write(resource);
ReleaseId releaseId = ks.newReleaseId("org.jbpm", "helloworld", "1.0");
kfs.generateAndWritePomXML(releaseId);
ks.newKieBuilder(kfs).buildAll();
ks.newKieContainer(releaseId).newKieSession().startProcess("org.jbpm.Hello
World");
```

In this example, we first call the static **`createProcess()`** method from the **`RuleFlowProcessFactory`** class. This method creates a new process and returns the **`RuleFlowProcessFactory`** that can be used to create the process.

A process consists of three parts:

- Header: The header section comprises global elements such as the name of the process, imports, and variables.

  In the above example, the header contains the name and version of the process and the package name.

- Nodes: The nodes section comprises all the different nodes that are part of the process.

  In the above example, nodes are added to the current process by calling the **`startNode()`**, **`actionNode()`** and **`endNode()`** methods. These methods return a specific **`NodeFactory`** that allows you to set the properties of that node. Once you have finished configuring that specific node, the **`done()`** method returns you to the current **`RuleFlowProcessFactory`** so you can add more nodes, if necessary.

- Connections: The connections section links the nodes to create a flow chart.

  In the above example, once you add all the nodes, you must connect them by creating connections between them. This can be done by calling the method **`connection`**, which links the nodes.

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

11.14. TESTING BUSINESS PROCESSES

11.14.1. Unit Testing

You must design business processes at a high level with no implementation details. You must ensure that they are tested as they also have a lifecycle like other development artifacts and can be updated dynamically.

Unit tests are conducted to ensure processes behave as expected in specific use cases, for example, to test the output based on the specific input. The helper class **`JbpmJUnitTestCase`** (in the jbpm-test module) has been included to simplify unit testing. JbpmJUnitTestCase provides the following:

- Helper methods to create a new knowledge base and session for a given set of processes.
- Assert statements to check:
  - The state of a process instance (active, completed, aborted).
  - Which node instances are currently active.
  - Which nodes have been triggered (to check the path that has been followed).
  - The value of variables.

The image below contains a start event, a script task, and an end event. Within the example junit Test, a new session is created, the process is started, and the process instance is verified based on successful completion. It also checks whether these three nodes have been executed.

Figure 11.2. Example Hello World Process

Example 11.2. example junit Test

```
public class ProcessPersistenceTest extends JbpmJUnitBaseTestCase {

    public ProcessPersistenceTest() {
        // setup data source, enable persistence
        super(true, true);
    }

    @Test
    public void testProcess() {
```

```
        // create runtime manager with single process - hello.bpmn

        createRuntimeManager("hello.bpmn");



        // take RuntimeManager to work with process engine

        RuntimeEngine runtimeEngine = getRuntimeEngine();


        // get access to KieSession instance

        KieSession ksession = runtimeEngine.getKieSession();


        // start process

        ProcessInstance processInstance =
ksession.startProcess("com.sample.bpmn.hello");


        // check whether the process instance has completed
successfully

        assertProcessInstanceCompleted(processInstance.getId(),
ksession);


        // check what nodes have been triggered

        assertNodeTriggered(processInstance.getId(), "StartProcess",
"Hello", "EndProcess");

    }

}
}
```

The **`JbpmJUnitBaseTestCase`** method acts as base test case class that you can use for JBoss BPM Suite related tests. It provides four usage areas:

- JUnit life cycle methods:
  - **`setUp`**: This method is executed @Before. It configures data source and EntityManagerFactory and cleans up Singleton's session id
  - **`tearDown`**: This method is executed @After. It clears out history, closes EntityManagerFactory and data source and disposes RuntimeEngines and RuntimeManager
- Knowledge Base and KnowledgeSession management methods:
  - **`createRuntimeManager`**: This method creates RuntimeManager for given set of assets and selected strategy.

  - **`disposeRuntimeManager`**: This method disposes RuntimeManager currently active in the scope of test.
  - **`getRuntimeEngine`**: This method creates new RuntimeEngine for given context.
- Assertions:
  - **`assertProcessInstanceCompleted`**
  - **`assertProcessInstanceAborted`**
  - **`assertProcessInstanceActive`**
  - **`assertNodeActive`**
  - **`assertNodeTriggered`**
  - **`assertProcessVarExists`**
  - **`assertNodeExists`**
  - **`assertVersionEquals`**
  - **`assertProcessNameEquals`**
- Helper methods:
  - **`getDs`**: This method returns currently configured data source.
  - **`getEmf`**: This method returns currently configured EntityManagerFactory.
  - **`getTestWorkItemHandler`**: This method returns test work item handler that might be registered in addition to what is registered by default.
  - **`clearHistory`**: This method clears history log.
  - **`setupPoolingDataSource`**: This method sets up data source.

`JbpmJUnitBaseTestCase` supports all the predefined RuntimeManager strategies as part of the unit testing. It is enough to specify which strategy shall be used whenever creating runtime manager as part of single test. The following example uses PerProcessInstance runtime manager strategy and task service to deal with user tasks:

```
public class ProcessHumanTaskTest extends JbpmJUnitBaseTestCase {



    private static final Logger logger =
LoggerFactory.getLogger(ProcessHumanTaskTest.class);


    public ProcessHumanTaskTest() {

        super(true, false);

    }
```

```
    @Test

    public void testProcessProcessInstanceStrategy() {

        RuntimeManager manager =
createRuntimeManager(Strategy.PROCESS_INSTANCE, "manager",
"humantask.bpmn");

        RuntimeEngine runtimeEngine =
getRuntimeEngine(ProcessInstanceIdContext.get());

        KieSession ksession = runtimeEngine.getKieSession();

        TaskService taskService = runtimeEngine.getTaskService();



        int ksessionID = ksession.getId();

        ProcessInstance processInstance =
ksession.startProcess("com.sample.bpmn.hello");


        assertProcessInstanceActive(processInstance.getId(), ksession);

        assertNodeTriggered(processInstance.getId(), "Start", "Task 1");



        manager.disposeRuntimeEngine(runtimeEngine);

        runtimeEngine =
getRuntimeEngine(ProcessInstanceIdContext.get(processInstance.getId()));



        ksession = runtimeEngine.getKieSession();

        taskService = runtimeEngine.getTaskService();



        assertEquals(ksessionID, ksession.getId());



        // let john execute Task 1

        List<TaskSummary> list =
taskService.getTasksAssignedAsPotentialOwner("john", "en-UK");

        TaskSummary task = list.get(0);
```

```
        logger.info("John is executing task {}", task.getName());

        taskService.start(task.getId(), "john");

        taskService.complete(task.getId(), "john", null);


        assertNodeTriggered(processInstance.getId(), "Task 2");



        // let mary execute Task 2

        list = taskService.getTasksAssignedAsPotentialOwner("mary", "en-
UK");

        task = list.get(0);

        logger.info("Mary is executing task {}", task.getName());

        taskService.start(task.getId(), "mary");

        taskService.complete(task.getId(), "mary", null);


        assertNodeTriggered(processInstance.getId(), "End");

        assertProcessInstanceCompleted(processInstance.getId(), ksession);

    }

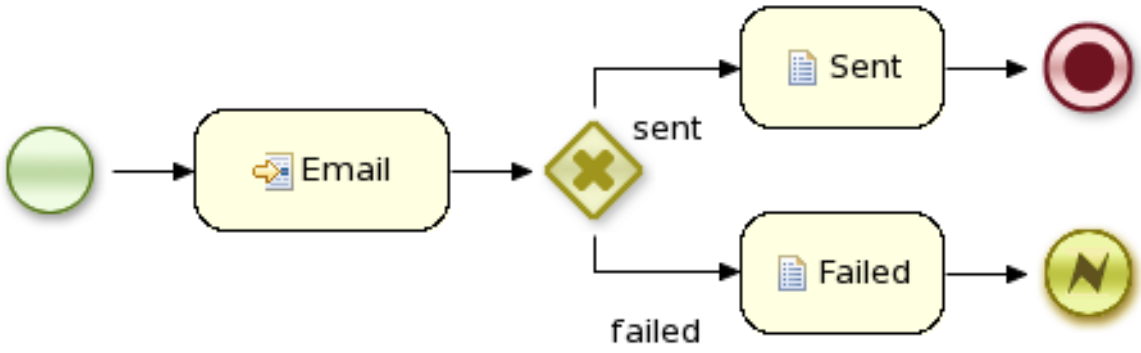
}
```

11.14.2. Testing Integration with External Services

Using domain-specific processes makes it possible to use testing handlers to verify whether or not specific services are requested correctly.

A **`TestWorkItemHandler`** is provided by default that can be registered to collect all work items (each work item represents one unit of work, for example, sending q specific email or invoking q specific service, and it contains all the data related to that task) for a given type. The test handler can be queried during unit testing to check whether specific work was actually requested during the execution of the process and that the data associated with the work was correct.

The following example describes how a process that sends an email could be tested. The test case tests whether an exception is raised when the email could not be sent (which is simulated by notifying the engine that sending the email could not be completed). The test case uses a test handler that simply registers when an email was requested and the data associated with the request. When the engine is notified the email could not be sent (using **`abortWorkItem(..)`**), the unit test verifies that the process handles this case successfully by logging this and generating an error, which aborts the process instance in this case.




```
public void testProcess2() {

    // create runtime manager with single process - hello.bpmn

    createRuntimeManager("sample-process.bpmn");

    // take RuntimeManager to work with process engine

    RuntimeEngine runtimeEngine = getRuntimeEngine();

    // get access to KieSession instance

    KieSession ksession = runtimeEngine.getKieSession();

    // register a test handler for "Email"

    TestWorkItemHandler testHandler = getTestWorkItemHandler();

    ksession.getWorkItemManager().registerWorkItemHandler("Email",
testHandler);

    // start the process

    ProcessInstance processInstance =
ksession.startProcess("com.sample.bpmn.hello2");

    assertProcessInstanceActive(processInstance.getId(), ksession);

    assertNodeTriggered(processInstance.getId(), "StartProcess", "Email");

    // check whether the email has been requested

    WorkItem workItem = testHandler.getWorkItem();

    assertNotNull(workItem);

    assertEquals("Email", workItem.getName());
```

```
    assertEquals("me@mail.com", workItem.getParameter("From"));

    assertEquals("you@mail.com", workItem.getParameter("To"));


    // notify the engine the email has been sent

    ksession.getWorkItemManager().abortWorkItem(workItem.getId());

    assertProcessInstanceAborted(processInstance.getId(), ksession);

    assertNodeTriggered(processInstance.getId(), "Gateway", "Failed",
"Error");


}
```

11.14.3. Configuring Persistence

You can configure whether you want to execute the JUnit tests using persistence or not. By default, the JUnit tests will use persistence, meaning that the state of all process instances will be stored in a (in-memory H2) database (which is started by the JUnit test during setup) and a history log will be used to check assertions related to execution history. When persistence is not used, process instances will only live in memory and an in-memory logger is used for history assertions.

Persistence (and setup of data source) is controlled by the super constructor and allows following:

- **`default`**: This is the no argument constructor and the most simple test case configuration (does NOT initialize data source and does NOT configure session persistence). It is usually used for in memory process management, without human task interaction
- **`super(boolean, boolean)`**: This allows to explicitly configure persistence and data source. This is the most common way of bootstrapping test cases for JBoss BPM Suite.
  - **`super(true, false)`**: To execute with in-memory process management with human tasks persistence.
  - **`super(true, true)`**: To execute with persistent process management with human tasks persistence
- **`super(boolean, boolean, string)`**: This is same as super(boolean, boolean), however it allows use of another persistence unit name than default (**`org.jbpm.persistence.jpa`**).

Here is an example:

```
public class ProcessHumanTaskTest extends JbpmJUnitBaseTestCase {



    private static final Logger logger =
LoggerFactory.getLogger(ProcessHumanTaskTest.class);


    public ProcessHumanTaskTest() {
```

```
        // configure this tests to not use persistence for process engine
but still use it for human tasks

        super(true, false);

    }

}
```

[1] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03 http://www.omg.org/spec/BPMN/2.0

[2] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03 http://www.omg.org/spec/BPMN/2.0

[3] Business Process Model and Notation (BPMN). Version 2.0, OMG Document Number: formal/2011-01-03 http://www.omg.org/spec/BPMN/2.0

CHAPTER 12. HUMAN TASKS MANAGEMENT

12.1. HUMAN TASKS

Human Tasks are tasks within a process that must be carried out by human actors. BRMS Business Process Management supports a human task node inside processes for modeling the interaction with human actors. The human task node allows process designers to define the properties related to the task that the human actor needs to execute; for example, the type of task, the actor, and the data associated with the task can be defined by the human task node. A back-end human task service manages the lifecycle of the tasks at runtime. The implementation of the human task service is based on the WS-HumanTask specification, and the implementation is fully pluggable; this means users can integrate their own human task solution if necessary. Human tasks nodes must be included inside the process model and the end users must interact with a human task client to request their tasks, claim and complete tasks.

12.2. USING USER TASKS IN PROCESSES

JBoss BPM Suite supports the use of human tasks inside processes using a special User Task node defined by the BPMN2 Specification. A User Task node represents an atomic task that is executed by a human actor.

Although JBoss BPM Suite has a special user task node for including human tasks inside a process, human tasks are considered the same as any other kind of external service that is invoked and are therefore implemented as a domain-specific service.

You can edit the values of User Tasks variables in the Properties view of JBoss Developer Studio after selecting the User Task node.

A User Task node contains the following core properties:

- Actors: The actors that are responsible for executing the human task. A list of actor id's can be specified using a comma (',') as separator.
- Group: The group id that is responsible for executing the human task. A list of group id's can be specified using a comma (',') as separator.
- Name: The display name of the node.
- TaskName: The name of the human task. This name is used to link the task to a Form. It also represent the internal name of the Task that can be used for other purposes.
- DataInputSet: all the input variables that the task will receive to work on. Usually you will be interested in copying variables from the scope of the process to the scope of the task.
- DataOutputSet: all the output variables that will be generated by the execution of the task. Here you specify all the name of the variables in the context of the task that you are interested to copy to the context of the process.
- Assignments: here you specify which process variable will be linked to each Data Input and Data Output mapping.

A User Task node contains the following extra properties:

- Comment: A comment associated with the human task. Here you can use expressions.

- Content: The data associated with this task.
- Priority: An integer indicating the priority of the human task.
- Skippable: Specifies whether the human task can be skipped, that is, whether the actor may decide not to execute the task.
- On entry and on exit actions: Action scripts that are executed upon entry and exit of this node, respectively.

Apart from the above mentioned core and extra properties of user tasks, there are some additional generic user properties that are not exposed through the user interface. These properties are:

- ActorId: The performer of the task to whom the task is assigned.
- GroupId: The group to which the task performer belongs.
- TaskStakeholderId : The person who is responsible for the progress and the outcome of a task.
- BusinessAdministratorId: The default business administrator who performs the role of the task stakeholder at task definition level.
- BusinessAdministratorGroupId : The group to which the administrator belongs.
- ExcludedOwnerId: Anybody who has been excluded to perform the task and become an actual or potential owner.
- RecipientId: A person who is the recipient of notifications related to the task. A notification may have more than one recipients.

To override the default values of these generic user properties, you must define a data input with the name of the property, and then set the desired value in the assignment section.

12.3. DATA MAPPING

Human tasks typically present some data related to the task that needs to be performed to the actor that is executing the task. Human tasks usually also request the actor to provide some result data related to the execution of the task. Task forms are typically used to present this data to the actor and request results.

You must specify the data that is used by the Task when you define the User Task in our Process. In order to do that, you need to define which data must be copied from the process context to the task context. Notice that the data is copied, so it can be modified inside the Task context but it will not affect the process variables unless we decide to copy back the value from the task to the process context.

Most of the times Forms are used to display data to the end user. This allows them to generate or create new data to propagate to the process context to be used by future activities. In order to decide how the information flow from the process to a particular task and from the task to the process, you need to define which pieces of information must be automatically copied by the process engine.

12.4. TASK LIFECYCLE

A human task is created when a user task node is encountered during the execution. The process leaves the user task node only when the associated human task is completed or aborted. The human task itself has a complete life cycle as well. The following diagram describes the human task life cycle.

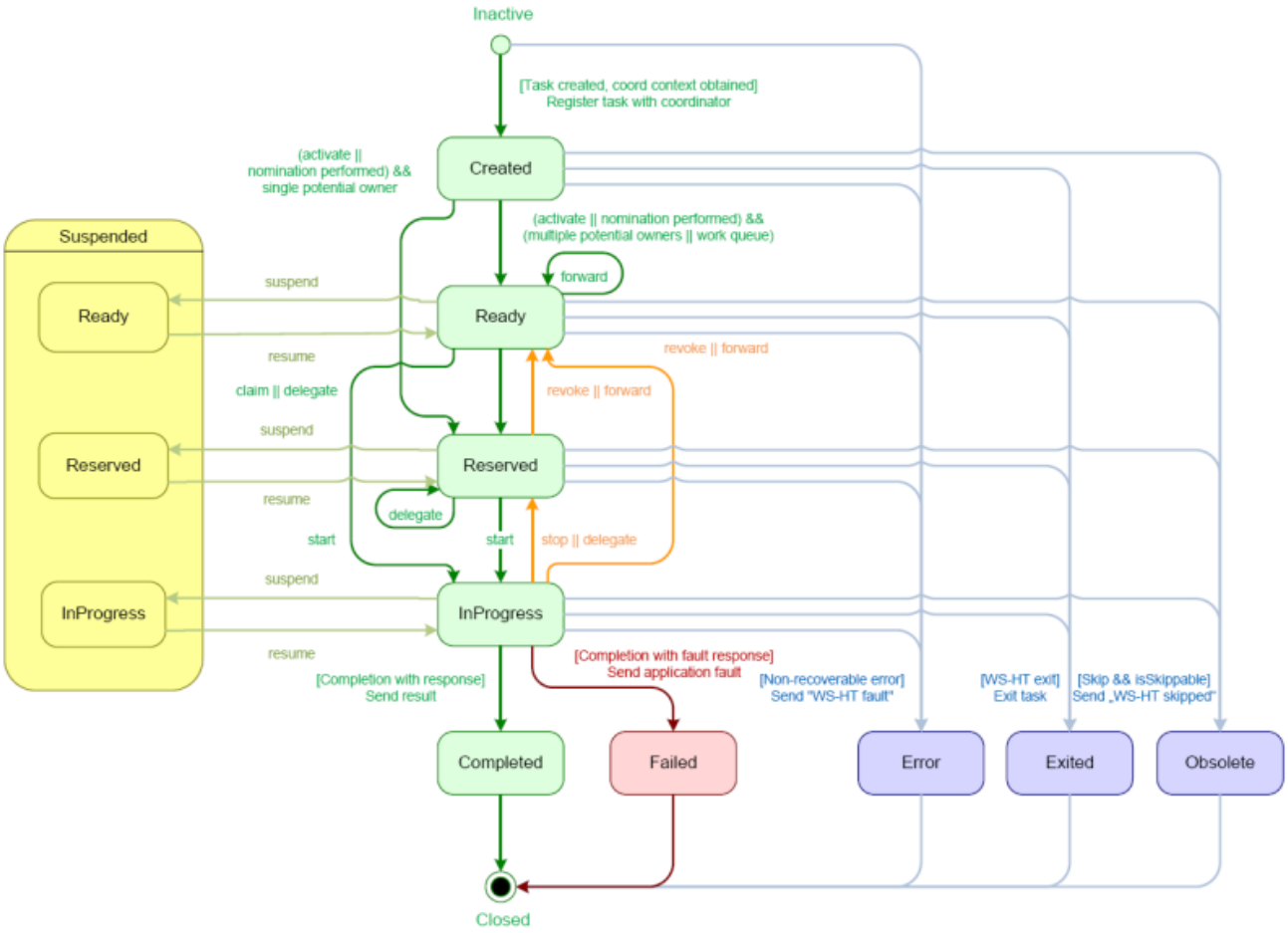


Figure 12.1. Human Task Life Cycle

A newly created task starts in the Created stage. It then automatically comes into the Ready stage. The task then shows up on the task list of all the actors that are allowed to execute the task. The task stays in the Ready stage until one of these actors claims the task. When a user then eventually claims the task, the status changes to Reserved. Note that a task that only has one potential (specific) actor is automatically assigned to that actor upon creation of the task. When the user who has claimed the task starts executing it, the task status changes from Reserved to InProgress.

Once the user has performed and completed the task, the task status changes to Completed. In this step, the user can optionally specify the result data related to the task. If the task could not be completed, the user may indicate this by using a fault response, possibly including fault data, in which case the status changes to Failed.

While this life cycle explained above is the normal life cycle, the specification also describes a number of other life cycle methods, including:

- Delegating or forwarding a task, so that the task is assigned to another actor.
- Revoking a task, so that it is no longer claimed by one specific actor but is (re)available to all actors allowed to take it.
- Temporarily suspending and resuming a task.
- Stopping a task in progress.
- Skipping a task (if the task has been marked as skippable), in which case the task will not be executed.

12.5. TASK PERMISSIONS

Only users associated with a specific task are allowed to modify or retrieve information about the task. This allows users to create a JBoss BPM Suite workflow with multiple tasks and yet still be assured of both the confidentiality and integrity of the task status and information associated with a task.

Some task operations end up throwing a **`org.jbpm.services.task.exception.PermissionDeniedException`** when used with information about an unauthorized user. For example, when a user is trying to directly modify the task (for example, by trying to claim or complete the task), the **`PermissionDeniedException`** is thrown if that user does not have the correct role for that operation. Also, users are not able to view or retrieve tasks in Business Central that they are not involved with.

12.5.1. Task Permissions Matrix

The task permissions matrix below summarizes the actions that specific user roles are allowed to do. The cells of the permissions matrix contain one of three possible characters, each of which indicate the user role permissions for that operation:

- a "+ indicates that the user role can do the specified operation.
- a "-" indicates that the user role may not do the specified operation.
- a "_" indicates that the user role may not do the specified operation, and that it is also not an operation that matches the user's role ("not applicable").

Table 12.1. Task Roles in the Permissions Table

Word	Role	Description
Initiator	Task Initiator	The user who creates the task instance.
Stakeholder	Task Stakeholder	The user involved in the task. This user can influence the progress of a task, by performing administrative actions on the task instance.
Potential	Potential Owner	The user who can claim the task before it has been claimed, or after it has been released or forward. Only tasks that have the status Ready may be claimed. A potential owner becomes the actual owner of a task by claiming the task.
Actual	Actual Owner	The user who has claimed the task and will progress the task to completion or failure.

Word	Role	Description
Administrator	Business Adminstrator	A super user who may modify the status or progress of a task at any point in a task's lifecycle.

User roles are assigned to users by the definition of the task in the JBoss BPM Suite (BPMN2) process definition.

Permissions Matrices

The following matrix describes the authorizations for all operations which modify a task:

Table 12.2. Main Operations Permissions Matrix

Operation/Role	Initiator	Stakeholder	Potential	Actual	Administrator
activate	+	+	_	_	+
claim	-	+	+	_	+
complete	-	+	_	+	+
delegate	+	+	+	+	+
fail	-	+	_	+	+
forward	+	+	+	+	+
nominate	+	+	+	+	+
release	+	+	+	+	+
remove	-	_	_	_	+
resume	+	+	+	+	+
skip	+	+	+	+	+
start	-	+	+	+	+
stop	-	+	_	+	+
suspend	+	+	+	+	+

12.6. TASK PERMISSIONS

12.6.1. Task Service and Process Engine

Human tasks are similar to any other external service that are invoked and implemented as a domain-specific service. As a human task is an example of such a domain-specific service, the process itself only contains a high-level, abstract description of the human task to be executed and a work item handler that is responsible for binding this (abstract) task to a specific implementation.

You can plug in any human task service implementation, such as the one that is provided by JBoss BPM Suite, or may register your own implementation. The JBoss BPM Suite provides a default implementation of a human task service based on the WS-HumanTask specification. If you do not need to integrate JBoss BPM Suite with another existing implementation of a human task service, you can use this service. The JBoss BPM Suite implementation manages the life cycle of the tasks (such as creation, claiming, completion) and stores the state of all the tasks, task lists, and other associated information. It also supports features like internationalization, calendar integration, different types of assignments, delegation, escalation and deadlines. You can find the code for the implementation in the jbpm-human-task module. The JBoss BPM Suite task service implementation is based on the WS-HumanTask (WS-HT) specification. This specification defines (in detail) the model of the tasks, the life cycle, and many other features.

12.6.2. Task Service API

The human task service exposes a Java API for managing the life cycle of tasks. This allows clients to integrate (at a low level) with the human task service. Note that, the end users should probably not interact with this low-level API directly, but use one of the more user-friendly task clients instead. These clients offer a graphical user interface to request task lists, claim and complete tasks, and manage tasks in general. The task clients listed below use the Java API to internally interact with the human task service. Of course, the low-level API is also available so that developers can use it in their code to interact with the human task service directly.

A task service (interface **`org.kie.api.task.TaskService`**) offers the following methods for managing the life cycle of human tasks:

```
                ...

                void start( long taskId, String userId );

                void stop( long taskId, String userId );

                void release( long taskId, String userId );

                void suspend( long taskId, String userId );

                void resume( long taskId, String userId );

                void skip( long taskId, String userId );

                void delegate(long taskId, String userId, String
targetUserId);

                void complete( long taskId, String userId, Map<String,
Object> results );

                ...
```

The common arguments passed to these methods are:

- taskId: The ID of the task that we are working with. This is usually extracted from the currently selected task in the user task list in the user interface.
- userId: The ID of the user that is executing the action. This is usually the id of the user that is logged in into the application.

To make use of the methods provided by the internal interface **InternalTaskService**, you need to manually cast to **InternalTaskService**. One method that can be useful from this interface is **getTaskContent()**:

```
Map<String, Object> getTaskContent( long taskId );
```

This method saves you from the complexity of getting the **ContentMarshallerContext** to unmarshall the serialized version of the task content. If you only want to use the stable or public API's, you can use the following method:

```
              Task taskById =
taskQueryService.getTaskInstanceById(taskId);
              Content contentById =
taskContentService.getContentById(taskById.getTaskData().getDocumentConten
tId());
              ContentMarshallerContext context =
getMarshallerContext(taskById);
              Object unmarshalledObject =
ContentMarshallerHelper.unmarshall(contentById.getContent(),
context.getEnvironment(), context.getClassloader());
              if (!(unmarshalledObject instanceof Map)) {
                  throw new IllegalStateException(" The Task Content
Needs to be a Map in order to use this method and it was:
"+unmarshalledObject.getClass());

              }
              Map<String, Object> content = (Map<String, Object>)
unmarshalledObject;
              return content;
```

12.6.3. Interacting with the Task Service

In order to get access to the Task Service API, it is recommended to let the Runtime Manager ensure that everything is setup correctly. From the API perspective, if you use the following approach, there is no need to register the Task Service with the Process Engine:

```
              ...
              RuntimeEngine engine =
runtimeManager.getRuntimeEngine(EmptyContext.get());
              KieSession kieSession = engine.getKieSession();
              // Start a process
              kieSession.startProcess("CustomersRelationship.customers",
params);
              // Do Task Operations
              TaskService taskService = engine.getTaskService();
              List<TaskSummary> tasksAssignedAsPotentialOwner =
taskService.getTasksAssignedAsPotentialOwner("mary", "en-UK");
```

```
// Claim Task
taskService.claim(taskSummary.getId(), "mary");
// Start Task
taskService.start(taskSummary.getId(), "mary");

...
```

The Runtime Manager registers the Task Service with the Process Engine automatically. If you do not use the Runtime Manager, you have to set the **`LocalHTWorkItemHandler`** in the session to get the Task Service notify the Process Engine once the task completes. In JBoss BPM Suite, the Task Service runs locally to the Process and Rule Engine. This enables you to create multiple light clients for different Process and Rule Engine's instances. All the clients can share the same database.

CHAPTER 13. PERSISTENCE AND TRANSACTIONS

13.1. PROCESS INSTANCE STATE

JBoss BPM Suite allows the persistent storage of certain information.An example of the information stored is the process runtime state. Storing the process runtime state is necessary in order to be able to continue execution of a process instance at any point if something goes wrong. The process definitions and the history information (logs of current and previous process states already) can also be persisted.

13.1.1. Runtime State

Whenever a process is started, a process instance is created, which represents the execution of the process in that specific context. For example, when executing a process that specifies how to process a sales order, one process instance is created for each sales request. The process instance represents the current execution state in that specific context, and contains all the information related to that process instance. Note that it only contains the (minimal) runtime state that is needed to continue the execution of that process instance at some later time. However, it does not include information about the history of that process instance if that information is no longer needed in the process instance.

The runtime state of an executing process can be made persistent, for example, in a database. This allows to restore the state of execution of all running processes in case of unexpected failure, or to temporarily remove running instances from memory and restore them at some later time. JBoss BPM Suite allows you to plug in different persistence strategies. By default, if you do not configure the process engine otherwise, process instances are not made persistent.

If you configure the engine to use persistence, it automatically stores the runtime state into the database. You do not have to trigger persistence yourself, the engine takes care of this when persistence is enabled. Whenever you invoke the engine, it ensures that the changes are stored at the end of that invocation. Whenever something goes wrong and you restore the engine from the database, you must not reload the process instances and trigger them manually to resume execution, as process instances automatically resume execution if they are triggered.

You must not try to access database tables containing runtime persistence data directly and try to modify these, as changing the runtime state of process instances without the engine's knowledge may have unexpected results. In most cases where information about the current execution state of process instances is required, the use of a history log is mostly recommended. In some cases, it may still be useful to for example query the internal database tables directly, but you must only do this if you know what you are doing.

13.1.1.1. Binary Persistence

Binary persistence, also known as marshaling, converts the state of the process instance into a binary dataset. Binary persistence is the mechanism used to store and retrieve information persistently. The same mechanism is also applied to the session state and any work item states.

When the process instance state is persisted, two things happen:

- The process instance information is transformed into binary data. For performance reasons, a custom serialization mechanism is used and not normal Java serialization.
- The binary data is stored alongside other metadata about the process instance. This metadata includes the process instance ID, process ID, and the process start date.

Apart from the process instance state, the session itself can also store other forms of state, such as the state of timer jobs. The session can also store the data that any business rules would be evaluated over.

This session state is stored separately as a binary dataset along with the ID of the session and some metadata. The session state can be restored by reloading the session with the given ID. The session ID can be retrieved using **`ksession.getId()`**.

Note that the process instance binary datasets are usually relatively small, as they only contain the minimal execution state of the process instance. For a simple process instance, the binary datasets usually contain one or a few node instances, i.e., any node that is currently executing and any existing variable values.

As a result of binary persistence, the data model is both simple and small:

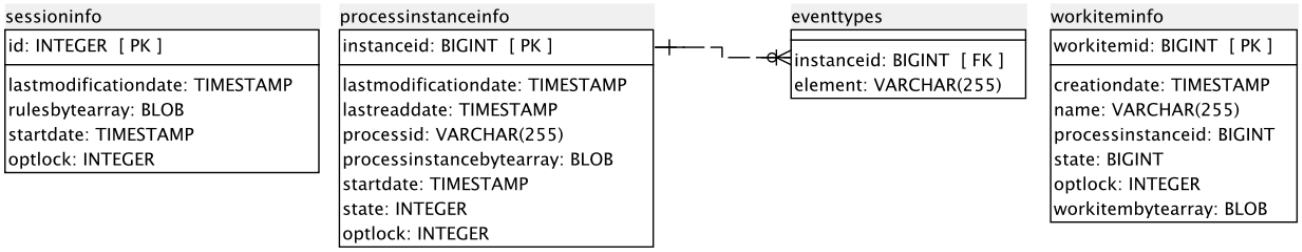


Figure 13.1. Data Model

The **`sessioninfo`** entity contains the state of the (knowledge) session in which the process instance is running.

Table 13.1. SessionInfo

Field	Description	Nullable
`id`	The primary key	NOT NULL
`lastmodificationdate`	The last time that the entity was saved to the database	
`rulesbytearray`	The binary dataset containing the state of the session	NOT NULL
`startdate`	The start time of the session	
`optlock`	The version field that serves as its optimistic lock value	

The **`processinstanceinfo`** entity contains the state of the process instance.

Table 13.2. ProcessInstanceInfo

Field	Description	Nullable
`instanceid`	The primary key	NOT NULL
`lastmodificationdate`	The last time that the entity was saved to the database	

Field	Description	Nullable
lastreaddate	The last time that the entity was retrieved (read) from the database	
processid	The name (ID) of the process	
processinstancebytearray	This is the binary dataset containing the state of the process instance	NOT NULL
startdate	The start time of the process	
state	An integer representing the state of the process instance	NOT NULL
optlock	The version field that serves as its optimistic lock value	

The **eventtypes** entity contains information about events that a process instance will undergo or has undergone.

Table 13.3. EventTypes

Field	Description	Nullable
instanceid	This references the **processinstanceinfo** primary key and there is a foreign key constraint on this column	NOT NULL
element	A text field related to an event that the process has undergone	

The **workiteminfo** entity contains the state of a work item.

Table 13.4. WorkItemInfo

Field	Description	Nullable
workitemid	The primary key	NOT NULL
name	The name of the work item	
processinstanceid	The (primary key) ID of the process: there is no foreign key constraint on this field	NOT NULL

Field	Description	Nullable
`state`	An integer representing the state of the work item	NOT NULL
`optlock`	The version field that serves as its optimistic lock value	
`workitembytearay`	This is the binary dataset containing the state of the work item	NOT NULL

13.2. AUDIT LOG

Storing information about the execution of process instances are useful in many cases. For example, when you need to:

- verify which actions have been executed for a particular process instance.
- monitor and analyze the efficiency of a particular process.

However, storing history information in the runtime database can result in the database rapidly increasing in size. Additionally, monitoring and analysis queries might influence the performance of your runtime engine. This is why process execution history information is stored separately.

This history log of execution information is created based on events that the process engine generates during execution. This is possible because the JBoss BPM Suite runtime engine provides a generic mechanism to listen to events. The necessary information can easily be extracted from these events and then persisted to a database. You can also use filters to limit the scope of the logged information.

The *jbpm-audit* module contains an event listener that stores process-related information in a database using JPA. The data model itself contains three entities, one for process instance information, one for node instance information, and one for (process) variable instance information:

- The *ProcessInstanceLog* table contains the basic log information about a process instance.
- The *NodeInstanceLog* table contains information about which nodes were actually executed inside each process instance. Whenever a node instance is entered from one of its incoming connections or is exited through one of its outgoing connections, that information is stored in this table.
- The *VariableInstanceLog* table contains information about changes in variable instances. The default is to only generate log entries when (after) a variable changes. It is also possible to log entries before the variable (value) changes.

To log process history information in a database, you need to register the logger on your session as shown below:

```
EntityManagerFactory emf = ...;
StatefulKnowledgeSession ksession = ...;
AbstractAuditLogger auditLogger = AuditLoggerFactory.newJPAInstance(emf);
ksession.addProcessEventListener(auditLogger);

// invoke methods one your session here
```

To specify the database where the information must be stored, modify the file persistence.xml file to include the audit log classes as well (**`ProcessInstanceLog`**, **`NodeInstanceLog`** and **`VariableInstanceLog`**), as shown below:

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

<persistence

  version="2.0"

  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd

  http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"

  xmlns="http://java.sun.com/xml/ns/persistence"

  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance>



  <persistence-unit name="org.jbpm.persistence.jpa" transaction-
type="JTA">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <jta-data-source>jdbc/jbpm-ds</jta-data-source>

    <mapping-file>META-INF/JBPMorm.xml</mapping-file>

    <class>org.drools.persistence.info.SessionInfo</class>


<class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>

    <class>org.drools.persistence.info.WorkItemInfo</class>

    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>


<class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>

    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</class>



    <class>org.jbpm.process.audit.ProcessInstanceLog</class>

    <class>org.jbpm.process.audit.NodeInstanceLog</class>

    <class>org.jbpm.process.audit.VariableInstanceLog</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.jta.platform"


value="org.hibernate.service.jta.platform.internal.BitronixJtaPlatform"/>

    </properties>

  </persistence-unit>

</persistence>
```

You can easily query this information and use them in a lot of use cases, ranging from creating a history log for one specific process instance to analyzing the performance of all instances of a specific process.

This audit log must only be considered a default implementation. For performance reasons, it is recommended to only store the relevant data that you require for analysis. Depending on your use cases, define your own data model for storing the information you need, and use the process event listeners to extract that information.

Process events are stored in the database synchronously and within the same transaction as actual process instance execution. This takes some time especially in highly loaded systems and may have some impact on the database when both history log and runtime data are kept in the same database. To provide an alternative option for storing process events, JBoss BPM Suite provides a JMS based logger. You can configure it to submit messages to JMS queue instead of directly persisting them in the database. You can configure it to be transactional as well to avoid issues with inconsistent data in case of process engine transaction is rolled back.

```
ConnectionFactory factory = ...;
Queue queue = ...;
StatefulKnowledgeSession ksession = ...;
Map<String, Object> jmsProps = new HashMap<String, Object>();
jmsProps.put("jbpm.audit.jms.transacted", true);
jmsProps.put("jbpm.audit.jms.connection.factory", factory);
jmsProps.put("jbpm.audit.jms.queue", queue);
AbstractAuditLogger auditLogger = AuditLoggerFactory.newInstance(Type.JMS,
session, jmsProps);
ksession.addProcessEventListener(auditLogger);

// invoke methods one your session here
```

13.3. TRANSACTIONS

The JBoss BPM Suite engine supports JTA transactions. If you do not provide transaction boundaries inside your application, the engine automatically executes method invocation on the engine in a separate transaction. You can specify the transaction boundaries which enables you to combine multiple commands into one transaction.

You need to register a transaction manager at the environment before using user-defined transactions. The following sample code uses the Bitronix transaction manager. Next, the Java Transaction API (JTA) is used to specify transaction boundaries, as shown below:

```
// create the entity manager factory and register it in the environment
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
"org.jbpm.persistence.jpa" );
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );
env.set( EnvironmentName.TRANSACTION_MANAGER,
TransactionManagerServices.getTransactionManager() );

// create a new knowledge session that uses JPA to store the runtime state

StatefulKnowledgeSession ksession =
JPAKnowledgeService.newStatefulKnowledgeSession( kbase, null, env );

// start the transaction

UserTransaction ut = (UserTransaction) new InitialContext().lookup(
"java:comp/UserTransaction" );
ut.begin();

// perform multiple commands inside one transaction

ksession.insert( new Person( "John Doe" ) );
ksession.startProcess( "MyProcess" );

// commit the transaction

ut.commit();
```

If you use Bitronix as the transaction manager, you must add a **`jndi.properties`** file in your root classpath to register the Bitronix transaction manager in JNDI. If you are using the jbpm-test module, this is already included by default. If not, create a file called **`jndi.properties`** with the following content:

```
java.naming.factory.initial=bitronix.tm.jndi.BitronixInitialContextFactory
```

In case you need to use a different JTA transaction manager, you can change the **`persistence.xml`** file to use your own transaction manager. You need to modify the transaction manager property in **`persistence.xml`** file as shown below:

```
<property name="hibernate.transaction.jta.platform"
value="org.hibernate.transaction.JBossTransactionManagerLookup" />
```

You can embed JBoss BPM Suite inside an application that executes in Container Managed Transaction (CMT) mode, such as EJB beans. To secure proper execution in CMT environments, a dedicated transaction manager implementation is provided:

```
org.jbpm.persistence.jta.ContainerManagedTransactionManager
```

NOTE

To ensure that container is aware of any exceptions that happened during process instance execution, you must ensure that exceptions thrown by the engine are propagated up to the container to properly rollback transaction.

To configure the transaction manager, perform the following configuration:

1. Insert transaction manager and persistence context manager into environment prior to creating or loading session:

```
Environment env = EnvironmentFactory.newEnvironment();
env.set(EnvironmentName.ENTITY_MANAGER_FACTORY, emf);
env.set(EnvironmentName.TRANSACTION_MANAGER, new
ContainerManagedTransactionManager());
env.set(EnvironmentName.PERSISTENCE_CONTEXT_MANAGER, new
JpaProcessPersistenceContextManager(env));
env.set(EnvironmentName.TASK_PERSISTENCE_CONTEXT_MANAGER, new
JPATaskPersistenceContextManager(env));
```

2. Configure JPA provider (example hibernate and WebSphere):

```
<property name="hibernate.transaction.factory_class"
value="org.hibernate.transaction.CMTTransactionFactory"/>
<property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.WebSphereJtaPlatf
orm"/>
```

13.4. CONFIGURATION

The JBoss BPM Suite engine does not save runtime data persistently by default. This means you can use the engine completely without persistence if necessary. However, if needed, you can configure the engine to use persistence by configuring it to do so. This usually requires adding the necessary dependencies, configuring a datasource and creating the engine with persistence configured.

13.4.1. Adding Dependencies

To use persistence, you must ensure that the necessary dependencies are available in the classpath of your application. Persistence is based on the Java Persistence API (JPA) by default and can thus work with several persistence mechanisms. We are using Hibernate. If you are using the Eclipse IDE and the JBoss BPM Suite plug-in, you must ensure that the necessary JAR files are added to your JBoss BPM Suite runtime directory. However, if you are using the JBoss BPM Suite runtime that is configured by default when using the JBoss BPM Suite installer , you need not do anything.

To manually add the necessary dependencies to your project, you first need the **`jbpm-persistence-jpa.jar`** file, as that contains code for saving the runtime state whenever necessary. Next, you need various other dependencies, depending on the persistence solution and database you are using. For the default combination with Hibernate as the JPA persistence provider and using an H2 in-memory database and Bitronix for JTA-based transaction management, you need the following list of additional dependencies is needed:

- jbpm-persistence-jpa (org.jbpm)

- drools-persistence-jpa (org.drools)
- persistence-api (javax.persistence)
- hibernate-entitymanager (org.hibernate)
- hibernate-annotations (org.hibernate)
- hibernate-commons-annotations (org.hibernate)
- hibernate-core (org.hibernate)
- commons-collections (commons-collections)
- dom4j (dom4j)
- jta (javax.transaction)
- btm (org.codehaus.btm)
- javassist (javassist)
- slf4j-api (org.slf4j)
- slf4j-jdk14 (org.slf4j)
- h2 (com.h2database)

13.4.2. Manually Configuring JBoss BPM Suite Engine to Use Persistence

You can use the **`JPAKnowledgeService`** to create your knowledge session. This gives you full access to the underlying configurations. You can create a new knowledge session using **`JPAKnowledgeService`** based on a knowledge base, a knowledge session configuration (if necessary) and an environment. Ensure that the environment contains a reference to your Entity Manager Factory. For example:

```
// create the entity manager factory and register it in the environment

EntityManagerFactory emf = Persistence.createEntityManagerFactory(
"org.jbpm.persistence.jpa" );
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );

// create a new knowledge session that uses JPA to store the runtime state

StatefulKnowledgeSession ksession =
JPAKnowledgeService.newStatefulKnowledgeSession( kbase, null, env );
int sessionId = ksession.getId();

// invoke methods on your method here

ksession.startProcess( "MyProcess" );
ksession.dispose();
```

Additionally, you can use the **JPAKnowledgeService** to recreate a session based on a specific session id. Here is an example:

```
// recreate the session from database using the sessionId

ksession = JPAKnowledgeService.loadStatefulKnowledgeSession(sessionId,
kbase, null, env );
```

Note that we only save the minimal state that is required to continue execution of the process instance at some later point. It does not contain information about already executed nodes if that information is no longer relevant. This also means that process instances that are completed or aborted are removed from the database. To search for history-related information, you must use the history log.

In order to use Hibernate and the H2 database, add a persistence configuration file **persistence.xml** to your classpath to configure JPA (or your own preference) in the **META-INF directory** as shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistenc version="2.0" xsi:schemaLocation=
   "http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
      http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
      xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance>

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-
type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <class>org.drools.persistence.info.SessionInfo</class>

<class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>

<class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</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.jta.platform"

value="org.hibernate.service.jta.platform.internal.BitronixJtaPlatform"/>
    </properties>
  </persistence-unit>

</persistence>
```

Here, the **`persistence.xml`** file refers to a data source called ***`jdbc/jbpm-ds`***. If you run your application in an application server, these containers typically allow you to easily set up data sources using some configuration (For example, placing a datasource configuration file in the deploy directory). For details, refer your application server documentation.

CHAPTER 14. USING JBOSS DEVELOPER STUDIO TO CREATE AND TEST PROCESSES

The JBoss BPM Suite plug-in provides you with an environment to edit and test processes, and integrate it deeply with your applications. It provides the following features:

- Wizards for creating:
  - a JBoss BPM Suite project
  - a BPMN2 process
- JBoss BPM Suite perspective showing the most commonly used views in a predefined layout

14.1. JBOSS BPM SUITE RUNTIME

14.1.1. JBoss BPM Suite Runtime

A JBoss BPM Suite runtime is a collection of JAR files that represent one specific release of the JBoss BPM Suite project JARs. To create a runtime, download the binary distribution of the version of JBoss BPM Suite you want to use and unzip on your local file system. You must then point the JBoss Developer Studio to the release of your choice by selecting the folder where these JARs are located. If you want to create a new runtime based on the latest jBPM project JARs included in the plugin itself, you can also easily do that. You are required to specify a default JBoss BPM Suite runtime for your JBoss Developer Studio workspace, but each individual project can override the default and select the appropriate runtime for that project specifically.

14.1.2. Setting the JBoss BPM Suite Runtime

In order to use the JBoss BPM Suite plug-in with Red Hat JBoss Developer Studio, it is necessary to set up the runtime.

A runtime is a collection of jar files that represent a specific release of the software.

If you have previously downloaded the JBoss BPM Suite Generic Deployable zip archive from Red Hat Customer Portal, the jar files that make up the runtime are located in the **`jboss-bpms-engine.zip`** archive.

Procedure 14.1. Configure jBPM Runtime

1. From the JBoss Developer Studio menu, select **`Window`** and click **`Preferences`**.
2. Select **jBPM** → **Installed jBPM Runtimes**.
3. Click **`Add...`**; provide a name for the new runtime, and click **`Browse`** to navigate to the directory where the runtime is located.
4. Click **`OK`**, select the new runtime and click **`OK`** again. If you have existing projects, a dialog box will indicate that you have to restart JBoss Developer Studio to update the Runtime.

14.1.3. Configuring the JBoss BPM Suite Server

JBoss Developer Studio can be configured to run the Red Hat JBoss BPM Suite Server.

Procedure 14.2. Configure the JBoss BPM Suite Server

1. Open the jBPM view by selecting **Window** → **Open Perspective** → **Other** and select **jBPM** and click **OK**.
2. Add the server view by selecting **Window** → **Show View** → **Other...** and select **Server** → **Servers**.
3. Open the server menu by right clicking the Servers panel and select **New** → **Server**.
4. Define the server by selecting **JBoss Enterprise Middleware** → **JBoss Enterprise Application Platform 6.1+** and clicking **Next**.
5. Set the home directory by clicking the **Browse** button. Navigate to and select the installation directory for JBoss EAP 6.1.1 which has JBoss BPM Suite installed.
6. Provide a name for the server in the **Name** field, ensure that the configuration file is set, and click **Finish**.

14.2. IMPORTING PROJECTS FROM A GIT REPOSITORY INTO JBOSS DEVELOPER STUDIO

You can configure JBoss Developer Studio to connect to a central Git asset repository. The repository stores rules, models, functions and processes.

You can either clone a remote Git repository or import a local Git repository.

Procedure 14.3. Cloning a Remote Git Repository

1. Start the Red Hat JBoss BRMS/BPM Suite server (whichever is applicable) by selecting the server from the server tab and click the start icon.
2. Simultaneously, start the Secure Shell server, if not running already, by using the following command. The command is Linux and Mac specific only. On these platforms, if sshd has already been started, this command fails. In that case, you may safely ignore this step.

   ```
   /sbin/service sshd start
   ```

3. In JBoss Developer Studio, select **File** → **Import...** and navigate to the Git folder. Open the Git folder to select **Projects from Git** and click **Next**.
4. Select the repository source as **Clone URI** and click **Next**.
5. Enter the details of the Git repository in the next window and click **Next**.

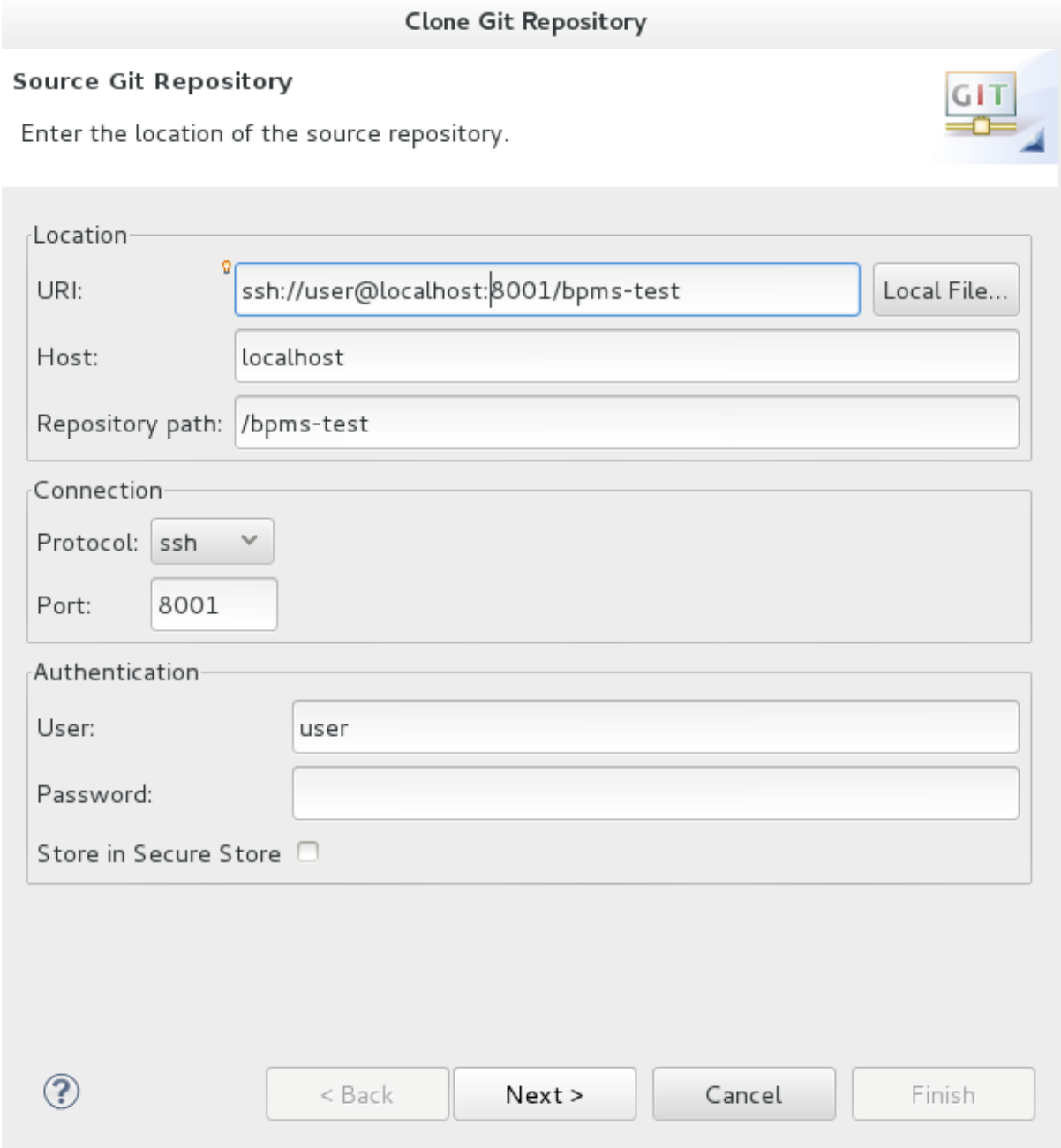


Figure 14.1. Git Repository Details

6. Select the branch you wish to import in the following window and click **`Next`**.

7. To define the local storage for this project, enter (or select) a non-empty directory, make any configuration changes and click **`Next`**.

8. Import the project as a general project in the following window and click **`Next`**. Name the project and click **`Finish`**.

Procedure 14.4. Importing a Local Git Repository

1. Start the Red Hat JBoss BRMS/BPM Suite server (whichever is applicable) by selecting the server from the server tab and click the start icon.

2. In JBoss Developer Studio, select **File** → **Import...** and navigate to the Git folder. Open the Git folder to select **Projects from Git** and click **`Next`**.

3. Select the repository source as **Existing local repository** and click **`Next`**.

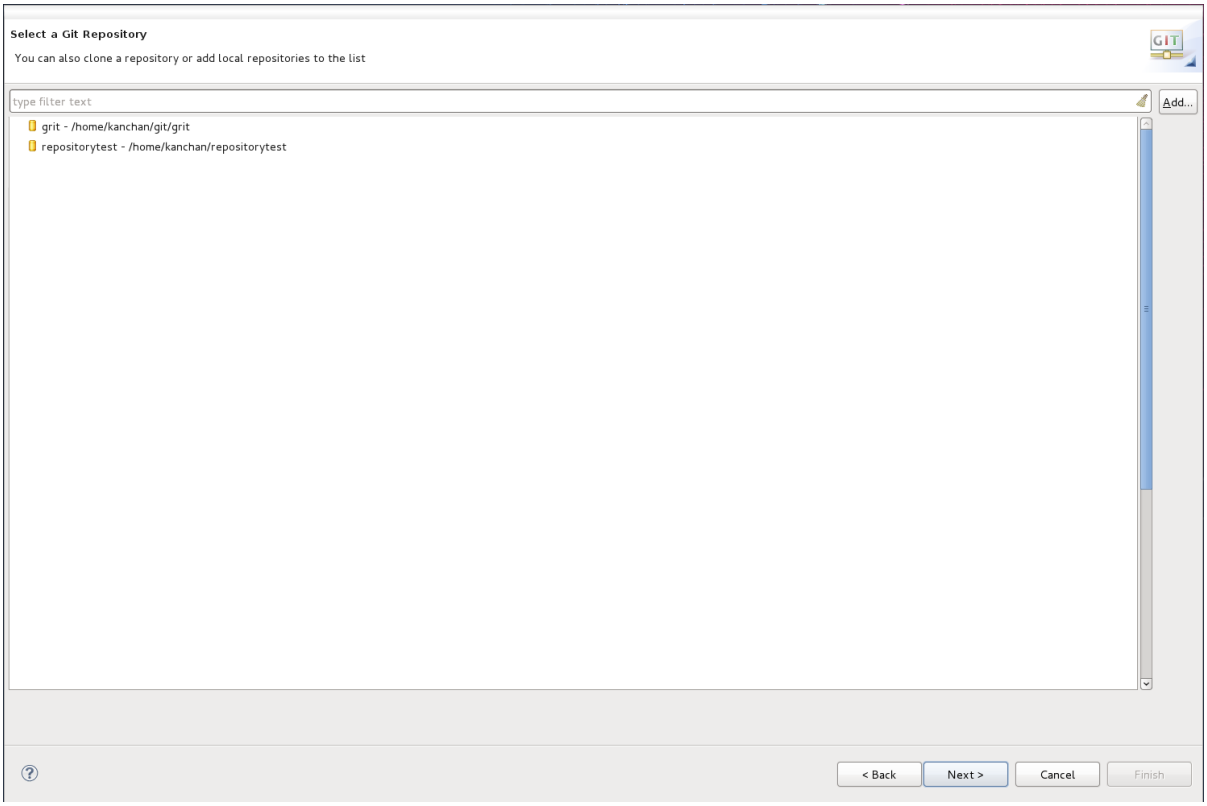


Figure 14.2. Git Repository Details

4. Select the repository that is to be configured from the list of available repositories and click **`Next`**.

5. In the dialog that opens, select the radio button **Import as general project** from the **Wizard for project import group** and click **Next**. Name the project and click **Finish**.

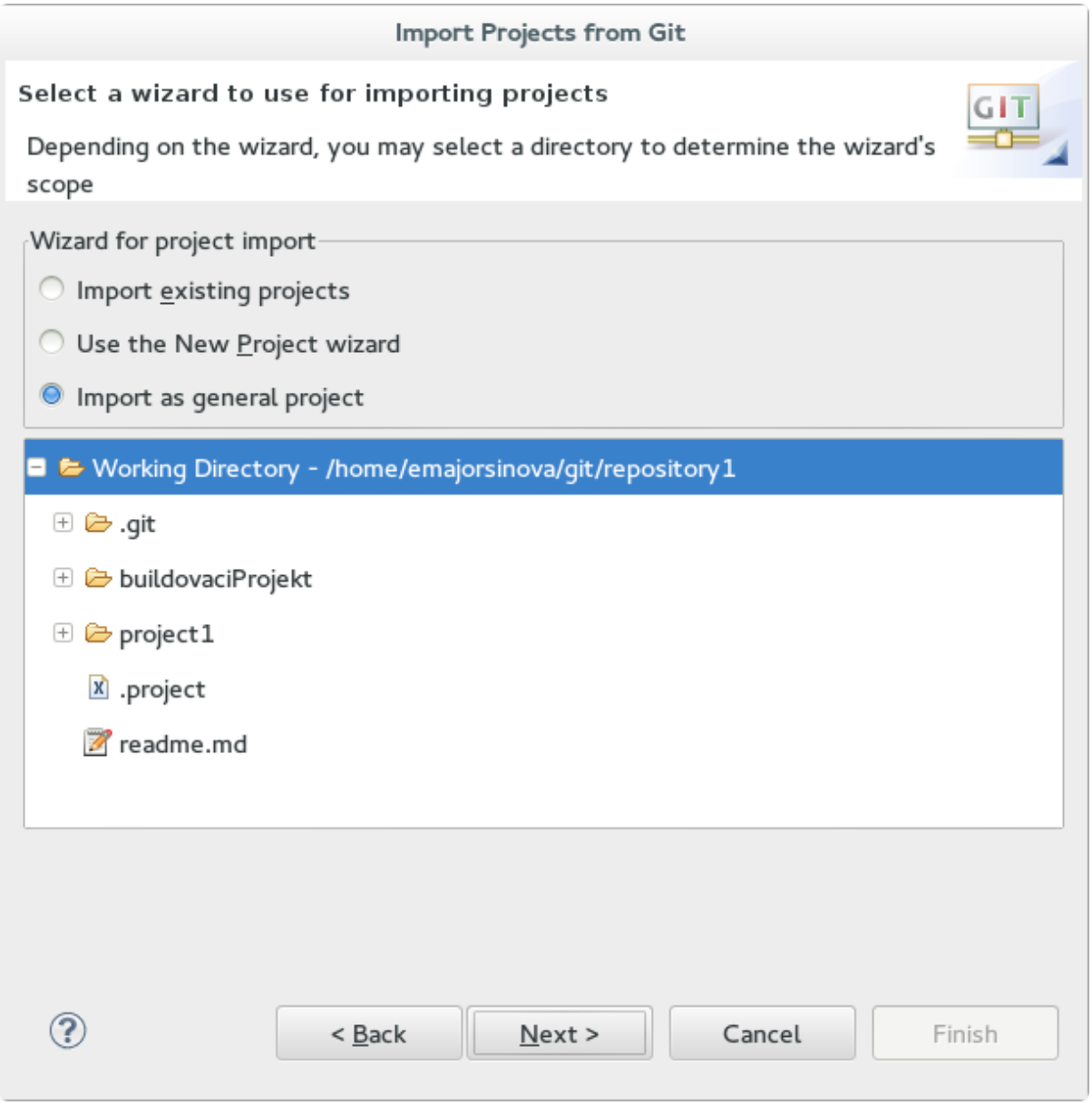


Figure 14.3. Wizard for Project Import

14.3. EXPLORING A JBOSS BPM SUITE APPLICATION

Before exploring how to create JBoss BPM Suite projects using JBoss Developer Studio, let us first understand the structure of JBoss BPM Suite projects. A JBoss BPM Suite application comprises the following components:

- A set of Java classes that will become process variables or facts in rules.
- A set of services accessed from service tasks in the business process model.
- A business process model definition file in BPMN2 format.
- Rules assets (optional).
- Java class that drives the application, including creation of a knowledge session, starting processes, and firing rules.

When you create a BPM Suite project, the following directories are generated:

- **`src/main/java`** that stores the class files (facts).
- **`src/main/resources/rules`** that stores the .drl files (rules).
- **`src/main/resources/process`** that stores the .bpmn files (processes).
- **`src/main/resources/jBPM`** Library that holds the generated .jar files required for rule execution.

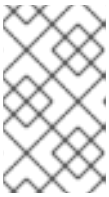
14.4. CREATING A JBOSS BPM SUITE PROJECT

Procedure 14.5. Creating a New JBoss BPM Suite project inRed Hat JBoss Developer Studio

1. From the main menu, select **File** → **New** → **Project**.

   Select **jBPM** → **jBPM Project** and click **`Next`**.

2. Enter a name for the project into the **`Project name:`** text box and click **`Next`**.

   NOTE

   JBoss Developer Studio provides the option to add a sample HelloWorld Rule file to the project. Accept this default by clicking **`Next`** to test the sample project in the following steps.

3. Select the jBPM runtime (or use the default).
4. Select generate code compatible with **`jBPM 6 or above`**, and click **`Finish`**.
5. To test the project, right click the Java file that contains the main method and select **Run** → **run as** → **Java Application**.

   The output will be displayed in the console tab.

14.5. CONVERTING AN EXISTING JAVA PROJECT TO A BPM SUITE PROJECT

To convert an existing Java project to a BPM Suite project:

Procedure 14.6. Task

1. Open the Java project in JBoss Developer Studio.
2. Right-click the project and under the **Configure** category, select **Convert to jBPM Project**.

This converts your Java project to BPM Suite project and adds the jBPM Library to your project's classpath.

14.6. CREATING A PROCESS USING BPMN2 PROCESS WIZARD

Procedure 14.7. Create a New Process

1. To create a new process, select **File** → **New** → **Other** and then select **jBPM** → **BPMN2 Process**.

2. Select the parent folder for the process.

3. Enter a name in the **`File name:`** dialogue box and click **`Finish`**.

This creates your new process containing just one start node. You can then open it in the BPMN2 Process Editor to add more nodes and connections to further build the process.

14.7. BUILDING A PROCESS USING BPMN2 PROCESS EDITOR

Procedure 14.8. Create a New Process

1. Create a new process using the BPMN2 Process Wizard in JBoss Developer Studio.

2. Right click the process .bpmn file, select **`Open With`** and then click the radio button next to **`BPMN2 Process Editor`**.

3. Add nodes to the process by clicking on the required node in the palette and clicking on the canvas where the node should be placed.

4. Connect the nodes with sequence flows. Select **`Sequence Flow`** from the palette, then click the nodes to connect them.

5. To edit a node's properties, click the node, open the properties tab in the bottom panel of the JBoss Developer Studio workspace, and click the values to be edited.

   If the properties tab is not already open, right click the bpmn file in the package panel and select **Show in** → **Properties**.

6. Click the save icon to save the process.

14.8. CREATING A PROCESS USING BPMN MAVEN PROCESS WIZARD

Use can use the JBoss BPM Suite Maven Project Wizard to set up an executable sample project to start using processes immediately by using Maven to define your project's properties and dependencies. This wizard sets up a Maven project using a **`pom.xml`**, and includes a sample process and Java class to execute it.

Procedure 14.9. Create a New Process

1. To create a new project, select **File** → **New** → **Project** and then select **jBPM** → **jBPM Project (Maven)**.

2. Enter a name for your project and click **`Finish`**.

   This creates your maven project with a sample process in the **`src/main/resources`** directory and a Java class that can be used to execute the sample process. In addition to that, the project contains:

   - A **`pom.xml`** file containing the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sample</groupId>
  <artifactId>jbpm-example</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <name>jBPM :: Sample Maven Project</name>
  <description>A sample jBPM Maven project</description>

  <properties>
    <jbpm.version>6.0.0.Final</jbpm.version>
  </properties>

  <repositories>
    <repository>
      <id>redhat-techpreview-all-repository</id>
      <name>Red Hat Tech Preview repository (all)</name>

<url>http://maven.repository.redhat.com/techpreview/all/</url>
      <releases>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>daily</updatePolicy>
      </snapshots>
    </repository>
  </repositories>

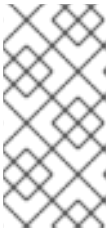
  <dependencies>
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-test</artifactId>
      <version>${jbpm.version}</version>
    </dependency>
  </dependencies>
</project>
```

- A **kmodule.xml** configuration file under the **META-INF** folder. The **kmodule.xml** defines which resources (like processes, rules) are to be loaded as part of your project. In this case, it defines a knowledge base called kbase that loads all the resources in the **com.sample** directory as shown below:

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="kbase" packages="com.sample"/>
</kmodule>
```

3. Update the project properties in the **Properties** tab and specify the JBoss BPM Suite version.

It adds the JBoss Nexus Maven repository (where all the JBoss BPM Suite JARs and their dependencies are located) to your project and configures the dependencies.



NOTE

By default, only the **`jbpm-test`** JAR is specified as a dependency, as this has transitive dependencies to almost all of the core dependencies you will need. You are free to update the dependencies section however to include only the dependencies you need.

14.9. DEBUGGING BUSINESS PROCESSES

JBoss Developer Studio can validate and debug processes.

Validation

To validate a process, right click the .bpmn file and select **`Validate`**.

If validation completes successfully, a dialogue box will appear stating there are no errors or warning.

If validation is unsuccessful, the errors will display in the **`Problems`** tab. Fix the problems and rerun the validation.

Debug

To debug a process, right click the .bpmn file and select **Debug As → Debug Configurations**; make any required changes to the test configuration and click **`Debug`**.

If no errors are found, the process will execute.

If errors are encountered, they will be described in the bottom window of JBoss Developer Studio. Fix the errors and rerun the debug process.

14.9.1. Using the Debug Perspective

In the Red Hat JBoss Developer Studio with Red Hat JBoss BPM Suite plug-in, you can make use of the extended debugging feature (debugging allows you to visualize and inspect the current state of running process instances).

Note that breakpoints on Process elements are currently not supported. However, you can define breakpoints inside any Java code in your Process; that is, your application code that is invoking the engine or invoked by the engine, listeners, etc. or inside rules that are evaluated in the context of a Process.

Procedure 14.10. The Debug Perspective

1. Open the Process Instance view **Window > Show View > Other ...**
2. Select **Process Instances and Process Instance** under the **Drools** category
3. Use a Java breakpoint to stop your application at a specific point (for example, after starting a new process instance).
4. In the Debug perspective, select the ksession you would like to inspect.

5. The *Process Instances* view will show the process instances that are currently active inside that ksession.

6. When double-clicking a process instance, the process instance viewer will graphically show the progress of that process instance.

7. Sometimes, when double-clicking a process instance, the process instance viewer complains that is cannot find the process. This means that the plug-in was not able to find the process definition of the selected process instance in the cache of parsed process definitions. To solve this, simply change the process definition in question and save again.

The screenshot below illustrates the running process instance with an id of "1". This example process instance relies on a human actor to perform "Task 1".

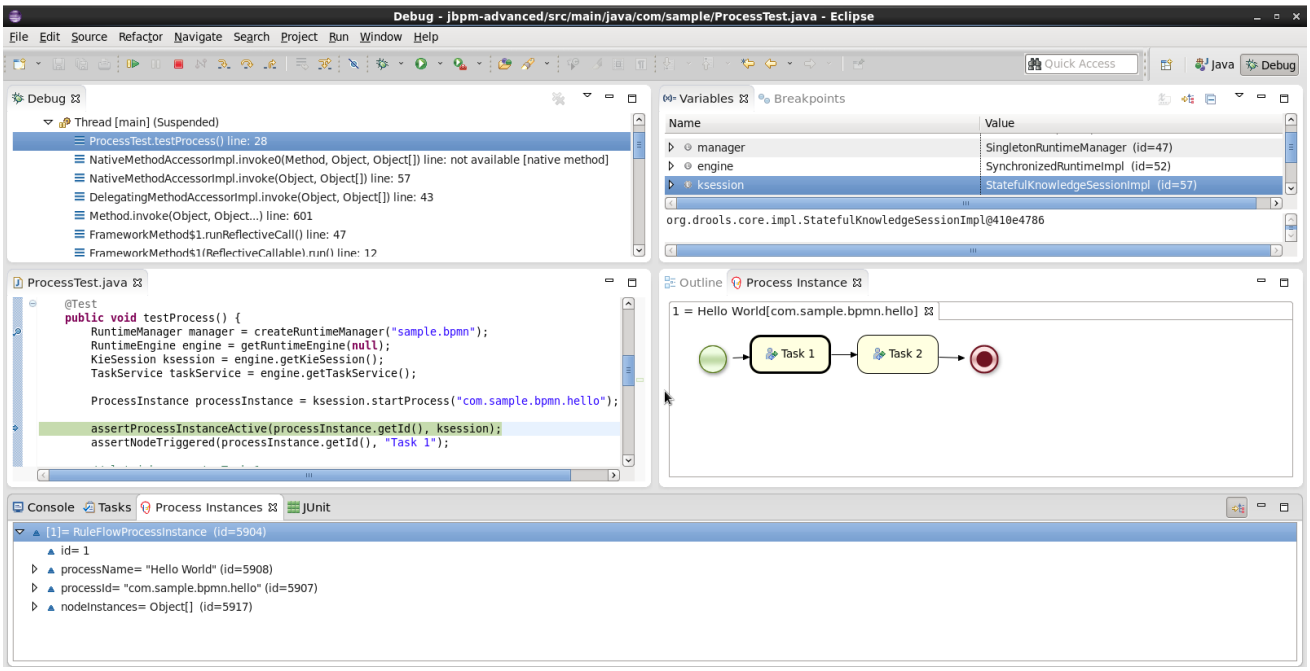
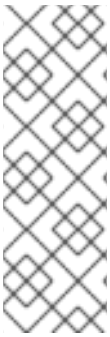


Figure 14.4. Process Instance in the Debugger



NOTE

The process instances view shows the process instances currently active inside the selected ksession. When using persistence, process instances are not kept in memory inside the ksession; that is, they are stored in the database as soon as the command completes. Therefore, you will not be able to use the Process Instances view when using persistence. For example, when executing a JUnit test using the JbpmJUnitBaseTestCase, make sure to call "super(true, false);" in the constructor to create a runtime manager that is not using persistence.

The environment provides also other views that are related to rule execution like the working memory view, the agenda view, etc. For further information, refer to the Red Hat JBoss BRMS documentation.

14.9.2. Debugging Views in JBoss Developer Studio

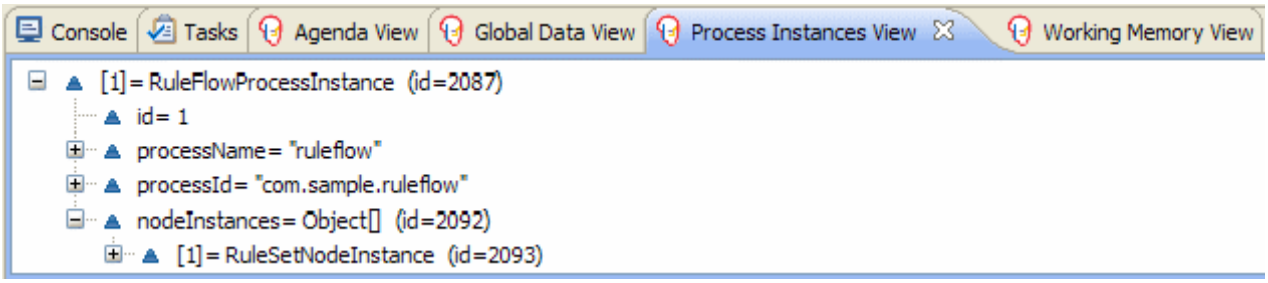
14.9.2.1. The Process Instances View

The process instances view shows the currently running process instances.

To open the process instances viewer, select **Window** → **Show View** → **Other**, then select **Drools** → **Process Instances**.

The Sample Process Instances View below shows that there is currently one running process (instance), currently executing one node instance, i.e. business rule task. When double-clicking a process instance, the process instance viewer will graphically display the progress of the process instance.

Example 14.1. Sample Process Instances View



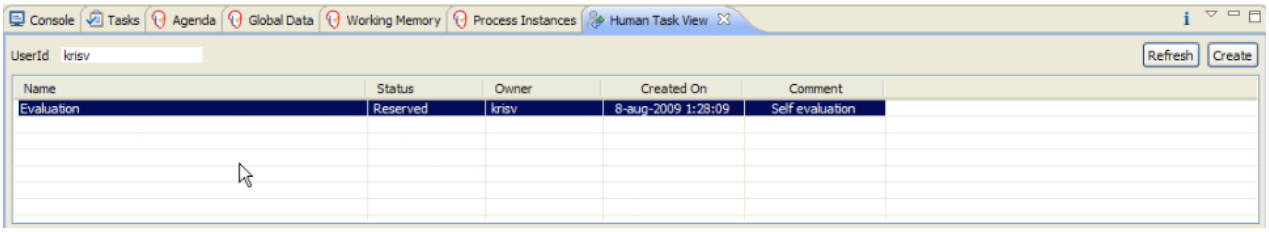
14.9.2.2. The Human Task View

The Human Task View can connect to a running human task service and request the relevant tasks for a particular user (i.e. the tasks where the user is either a potential owner or the tasks that the user already claimed and is executing). The life cycle of these tasks can then be executed, i.e. claiming or releasing a task, starting or stopping the execution of a task, completing a task, etc.

To open the human task viewer, select **Window** → **Show View** → **Other**, then select **jBPM Task** → **Human Task View**.

To configure the task service to connect to, select **Window** → **Preferences** → **Drools Tasks** and enter the IP address, port, and language.

Example 14.2. Sample Human Task View



14.9.2.3. The Audit View

The audit view shows the audit log, which is a log of all events that were logged from the session. To create a logger, use the KnowledgeRuntimeLoggerFactory to create a new logger and attach it to a session. Note that using a threaded file logger will save the audit log to the file system at regular intervals, and the audit viewer will then be able to show the latest state. The Threaded File Logger below shows an example with the audit log file and the interval (in milliseconds) specified.

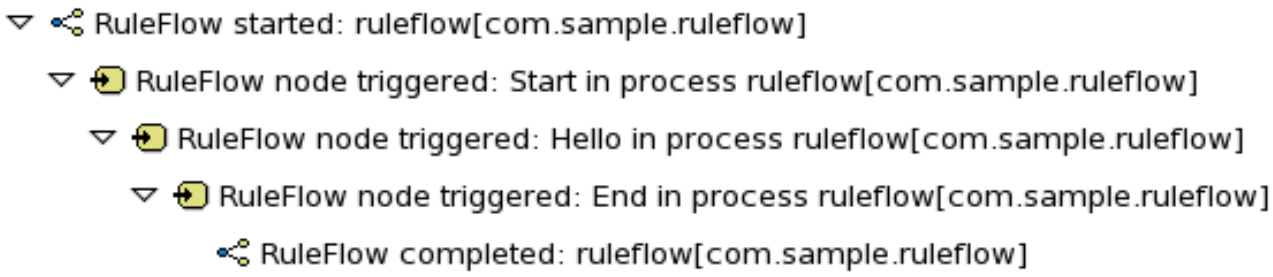
Example 14.3. Threaded File Logger

```
KnowledgeRuntimeLogger logger = KnowledgeRuntimeLoggerFactory
   .newThreadedFileLogger(ksession, "logdir/mylogfile", 1000);
// do something with the session here
```

```
logger.close();
```

To open the audit view, select **Window** → **Show View** → **Audit**.

To open up an audit tree in the audit view, open the selected log file in the audit view or simply drag the file into the audit view. A tree-based view is generated based on the audit log. An event is shown as a sub node of another event if the child event is caused by (a direct consequence of) the parent event:



14.10. SYNCHRONIZING JBOSS DEVELOPER STUDIO WORKSPACE WITH BUSINESS CENTRAL REPOSITORIES

JBoss BPM Suite allows you to synchronize your local workspace with one or more repositories that are managed inside Business Central with the help of Eclipse tooling for Git. Git is a popular distributed source code version control system. You can use any Git tool of your choice.

When you create and execute processes inside JBoss Developer Studio, they get created on your local file system. Alternatively, you can import an existing repository from Business Central, apply changes and push these changes back into the Business Central repositories. This synchronization enables collaboration between developers using JBoss Developer Studio and business analysts or end users using Business Central.

14.10.1. Importing a Business Central Repository using EGit Import Wizard

Procedure 14.11. Task

1. Open JBoss Developer Studio.
2. Navigate to **File** → **Import ...** → **Git** → **Projects from Git** and click **`Next`**.
3. Select **URI** to connect to a repository that is managed by Business Central and click **`Next`**.

   This opens a **Import Project from Git** dialog box.
4. Provide the URI of the repository you would like to import in the **URI** field.

   Provide the following URI to connect to your Business Central repositories:

   ```
   ssh://<hostname>:8001/<repository_name>
   ```

   For example, if you are running the Business Central on your local host by using the jbpm-installer, you would use the following URI to import the jbpm-playground repository:

   ```
   ssh://localhost:8001/jbpm-playground
   ```

You can change the port used by the server to provide ssh access to the git repository if necessary, using the system property org.uberfire.nio.git.ssh.port.

5. Click **`Next`**.

6. Specify where on your local file system you would like this repository to be created in the **Directory** field.

7. Select the master branch in the **Initial branch** field and click **`Next`**.

8. Select **Import as general project** to import the repository you downloaded as a project in your JBoss Developer Studio workspace and click **`Next`**>.

9. Provide a name for the repository and click **`Finish`**.

This adds your repository to your workspace and you can now browse, open and edit the various assets inside it.

14.10.2. Committing Changes to Business Central

To commit and push your local changes back to the Business Central repositories:

Procedure 14.12. Task

1. Open your repository project in JBoss Developer Studio.

2. Right-click on your repository project and select **Team** → **Commit ...**.

   A new dialog box open showing all the changes you have on your local file system.

3. Select the files you want to commit, provide an appropriate commit message, and click **`Commit`**.

   You can double-click each file to get an overview of the changes you did for that file.

4. Right-click your project again, and select **Team** → **Push to Upstream**.

14.10.3. Retrieving the Changes from the Business Central Repository

To retrieve the latest changes from the Business Central repository:

Procedure 14.13. Task

1. Open your repository project in JBoss Developer Studio.

2. Right-click your repository project and select **Team** → **Fetch from Upstream**.

   This action fetches all the changes from the Business Central repository.

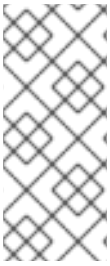
3. Right-click your project again and select **Team** → **Merge**.

   A **Merge 'master'** dialog appears.

4. In the **Merge 'master'** dialog box, select **origin**/**master** branch under **Remote Tracking**.

5. Click **`Merge`**.

This merges all the changes from the original repository in Business Central.



NOTE

It is possible that you have committed and/or conflicting changes in your local version, you might have to resolve these conflicts and commit the merge results before you will be able to complete the merge successfully. It is recommended to update regularly, before you start updating a file locally, to avoid merge conflicts being detected when trying to commit changes.

14.10.4. Importing Individual Projects from Repository

When you import a repository, it downloads all the projects that are inside that repository. It is however useful to mount one specific project as a separate Java project in JBoss Developer Studio. When you do this, JBoss Developer Studio is able to:

- Interpret the information in the project **`pom.xml`** file that you created in Business Central.
- Download and include any dependencies you specified.
- Compile any Java classes you have in your project.

To import a project as a separate Java project:

Procedure 14.14. Task

1. In the JBoss Developer Studio, right-click on one of the projects in your repository project and select **Import ...**.
2. Under the Maven category, select **Existing Maven Projects** and click **`Next`**.

   The Import Maven Project dialog box opens displaying the **`pom.xml`** file of the project you selected.
3. Click **`Finish`** .

14.10.5. Adding JBoss BPM Suite libraries to your Project Classpath

You need to add the JBoss BPM Suite libraries to the classpath of your project to ensure it compiles and executes correctly. To do this:

Procedure 14.15. Task

- Right-click your project and select **Configure** → **Convert to jBPM Project**.

This converts your project into a JBoss BPM Suite project and adds the JBoss BPM Suite library to your project's classpath

PART IV. KIE

CHAPTER 15. KIE API

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

Once you have set up a session, you can use it to start executing processes. Whenever a process is started, a new process instance is created (for that process definition) that maintains the state of that specific instance of the process. For example, imagine you are writing an application to process sales orders. You can then define one or more process definitions that define how the order must be processed. When starting up your application, you first need to create a knowledge base that contains those process definitions. You can then create a session based on this knowledge base so that, whenever a new sales order comes in, a new process instance is started for that sales order. That process instance contains the state of the process for that specific sales request.

A knowledge base can be shared across sessions and usually is only created once, at the start of the application. Knowledge bases can be dynamically changed so that you can add or remove processes at runtime.

You can create sessions based on a knowledge base. These are used to execute processes and interact with the engine. You can create many independent sessions or multiple sessions. For example, if you want all processes from one customer to be completely independent from processes for another customer, you can create an independent session for each customer. At the same time, you may need multiple sessions for scalability reasons. If you do not know what to do, you can start by having one knowledge base that contains all your process definitions and create one session that you then use to execute all your processes.

The JBoss BPM Suite project has a clear separation between the API the users interact with and the actual implementation classes. The public API exposes most of the features that users can safely use. Expert users can still access internal classes but must be aware that the internal API may change in the future.

15.1. KIE

KIE (Knowledge Is Everything) replaces the knowledge keywords used as a knowledge solution for JBoss BRMS and JBoss BPM Suite. KIE is also used for the generic parts of unified API such as building, deploying, and loading. KIE

The following lifecycle stages represent different aspects of working with KIE system:

- Author

  Includes authoring of knowledge using a UI metaphor, such as DRL, BPMN2, decision table, and class models.

- Build

  Includes building the authored knowledge into deployable units. For KIE, this unit is a JAR.

- Test

  Includes testing KIE knowedge before it is deployed to the application.

- Deploy

  Includes deploying the unit to a location where applications may utilize (consume) them. KIE uses Maven style repository.

- Utilize

  Includes loading of a JAR to provide a KIE session (KieSession), for the application to 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

  Includes system interaction with the KieSession, via API.

- Work

  Includes user interaction with the KieSession through command line or UI.

- Manage

  Includes managing any KieSession or KieContainer.

15.2. KIE FRAMEWORK

15.2.1. KIE Systems

The various aspects, or life cycles, of KIE systems in the JBoss BPM Suite 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.

15.2.2. KieBase

The JBoss BPM Suite API allows you to create a knowledge base that includes all your process definitions that may need to be executed by that session. To create a knowledge base, use a **`KieHelper`** to load processes from various resources (for example, from the classpath or from the file system), and then create a new knowledge base from that helper. The following code snippet shows how to create a knowledge base consisting of only one process definition (using in this case a resource from the classpath):

```
KieHelper kHelper = new KieHelper();
  KieBase kBase =
kieHelper.addResource(ResourceFactory.newClassPathResource("MyProcess.bpmn
")).build();
```

This is a manual method that you can use to create simple knowledge base. It uses **`KieHelper`** and **`ResourceFactory`** that are a part of Internal APIs **`org.kie.internal.io.ResourceFactory`** and **`org.kie.internal.utils.KieHelper`**. Using **`RuntimeManager`** is a recommended way to create knowledge base and knowledge session.

NOTE

The classes belonging to the Internal API (**`org.kie.internal`**) are not supported as they are subject to change.

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 15.1. kbase Attributes

Attribute name	Default value	Admitted values	Meaning

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 BRMS 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.
equalsBehavior	identity	identity, equality	Defines the behavior of JBoss BRMS 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.

Attribute name	Default value	Admitted values	Meaning
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.

15.2.3. KieSession

Once you load your knowledge base, you must then create a session to interact with the engine. You can then use this session to start new processes, and signal events. Here is how you create a session:

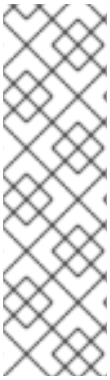
```
KieSession ksession = kbase.newKieSession();

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

Add the following import statement to get access to the process instance:

```
import org.kie.api.runtime.process.ProcessRuntime;
```

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.



NOTE

In case where the JBoss BPM Suite engine is managed within a Container Managed Transaction (CMT) environment and the transactions are out of control of the engine, concurrent access to the same session instance may lead to errors. To handle this situation, an interceptor is provided that locks the KieSession for a single thread until the transaction completes. This enables you to safely use KieSession in a CMT environment. To enable this interceptor, set the system property **`org.kie.tx.lock.enabled`** and the environment entry **`TRANSACTION_LOCK_ENABLED`** to **`true`**. The default value of these properties is **`false`**.

Table 15.2. ksession Attributes

Attribute name	Default value	Admitted values	Meaning

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.
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.

15.2.3.1. The ProcessRuntime Interface

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

```
/**
    * Start a new process instance.  The process (definition) that should
    * be used is referenced by the given process id.
```

```
     *

     * @param processId  The id of the process that should be started

     * @return the ProcessInstance that represents the instance of the
process that was started

     */

    ProcessInstance startProcess(String processId);


    /**

     * Start a new process instance.  The process (definition) that should

     * be used is referenced by the given process id.  Parameters can be
passed

     * to the process instance (as name-value pairs), and these will be
set

     * as variables of the process instance.

     *

     * @param processId  the id of the process that should be started

     * @param parameters  the process variables that should be set when
starting the process instance

     * @return the ProcessInstance that represents the instance of the
process that was started

     */

    ProcessInstance startProcess(String processId,

                                 Map<String, Object> parameters);


    /**

     * Signals the engine that an event has occurred. The type parameter
defines

     * which type of event and the event parameter can contain additional
information

     * related to the event.  All process instances that are listening to
this type

     * of (external) event will be notified.  For performance reasons,
this type of event
```

```
     * signaling should only be used if one process instance should be
able to notify

     * other process instances. For internal event within one process
instance, use the

     * signalEvent method that also include the processInstanceId of the
process instance

     * in question.

     *

     * @param type the type of event

     * @param event the data associated with this event

     */

    void signalEvent(String type,

                     Object event);


    /**

     * Signals the process instance that an event has occurred. The type
parameter defines

     * which type of event and the event parameter can contain additional
information

     * related to the event.  All node instances inside the given process
instance that

     * are listening to this type of (internal) event will be notified.
Note that the event

     * will only be processed inside the given process instance.  All
other process instances

     * waiting for this type of event will not be notified.

     *

     * @param type the type of event

     * @param event the data associated with this event

     * @param processInstanceId the id of the process instance that should
be signaled

     */

    void signalEvent(String type,
```

```
                        Object event,

                        long processInstanceId);


    /**

     * Returns a collection of currently active process instances.  Note
that only process

     * instances that are currently loaded and active inside the engine
will be returned.

     * When using persistence, it is likely not all running process
instances will be loaded

     * as their state will be stored persistently.  It is recommended not
to use this

     * method to collect information about the state of your process
instances but to use

     * a history log for that purpose.

     *

     * @return a collection of process instances currently active in the
session

     */

    Collection<ProcessInstance> getProcessInstances();


    /**

     * Returns the process instance with the given id.  Note that only
active process instances

     * will be returned.  If a process instance has been completed
already, this method will return

     * null.

     *

     * @param id the id of the process instance

     * @return the process instance with the given id or null if it cannot
be found

     */

    ProcessInstance getProcessInstance(long processInstanceId);
```

```
    /**

     * Aborts the process instance with the given id.  If the process
instance has been completed

     * (or aborted), or the process instance cannot be found, this method
will throw an

     * IllegalArgumentException.

     *

     * @param id the id of the process instance

     */

    void abortProcessInstance(long processInstanceId);


    /**

     * Returns the WorkItemManager related to this session.  This can be
used to

     * register new WorkItemHandlers or to complete (or abort) WorkItems.

     *

     * @return the WorkItemManager related to this session

     */

    WorkItemManager getWorkItemManager();
```

15.2.3.2. Event Listeners

The session provides methods for registering and removing listeners. You can use a **`ProcessEventListener`** class to listen to process-related events, such as starting or completing a process and entering and leaving a node. An event object provides access to related information, like the process instance and node instance linked to the event. You can use this API to register your own event listeners. Here is a list of methods of the **`ProcessEventListener`** class:

```
public interface ProcessEventListener {

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

```
  void afterVariableChanged(ProcessVariableChangedEvent event);

}
```

You need to add the following import statement to get access to the **ProcessEventListener** class:

```
import org.kie.api.event.process.ProcessEventListener;
```

15.2.3.3. Before and After Events

The before and after events behave like a stack. It means that any events that occur as a direct result of the previous event, it occurs between the before and the after of that event. For example, if a subsequent node is triggered as result of leaving a node, the node triggered events occur in between the ***beforeNodeLeftEvent*** and the ***afterNodeLeftEvent*** of the node that is left (as the triggering of the second node is a direct result of leaving the first node). This enables you to derive cause relationships between events more easily. Similarly, all node triggered and node left events that are the direct result of starting a process, occur between the ***beforeProcessStarted*** and ***afterProcessStarted*** events. In general, if you just want to be notified when a particular event occurs, you must check for the before events only (as they occur immediately before the event actually occurs). If you are only looking at the after events, you may get an impression of events firing in the wrong order. As the after events are triggered as a stack, they only fire when all events that were triggered as a result of this event have already fired. Use the after events only if you want to ensure that all processing related to this has ended. For example, when you want to be notified when starting of a particular process instance has ended.

Not all nodes always generate node triggered or node left events. Depending on the type of node, some nodes might only generate node left events, and others might only generate node triggered events. Catching intermediate events is like generating left events, as they are not really triggered by another node, rather activated from outside. Similarly, throwing intermediate events are not generating left events. They are only generating triggered events, as they are not really left, as they have no outgoing connection.

15.2.3.4. Correlation Keys

When working with processes, you may require to assign a given process instance some sort of business identifier to reference later without knowing the actual (generated) ID of the process instance. To provide such capabilities, JBoss BPM Suite allows you to use **CorrelationKey** that is composed of **CorrelationProperties**. **CorrelationKey** can have either a single property describing it or can be represented as multi valued properties set. The correlation feature, generally used for long running processes, requires you to enable persistence in order to permanently store correlation information.

Correlation capabilities are provided as part of the **CorrelationAwareProcessRuntime** interface. This interface that exposes following methods:

```
      /**

      * Start a new process instance.  The process (definition) that
should

      * be used is referenced by the given process id.  Parameters can be
passed

      * to the process instance (as name-value pairs), and these will be
```

```
set

      * as variables of the process instance.

      *

      * @param processId  the id of the process that should be started

      * @param correlationKey custom correlation key that can be used to
identify process instance

      * @param parameters  the process variables that should be set when
starting the process instance

      * @return the ProcessInstance that represents the instance of the
process that was started

      */

      ProcessInstance startProcess(String processId, CorrelationKey
correlationKey, Map<String, Object> parameters);


      /**

      * Creates a new process instance (but does not yet start it).  The
process

      * (definition) that should be used is referenced by the given
process id.

      * Parameters can be passed to the process instance (as name-value
pairs),

      * and these will be set as variables of the process instance.  You
should only

      * use this method if you need a reference to the process instance
before actually

      * starting it.  Otherwise, use startProcess.

      *

      * @param processId  the id of the process that should be started

      * @param correlationKey custom correlation key that can be used to
identify process instance

      * @param parameters  the process variables that should be set when
creating the process instance

      * @return the ProcessInstance that represents the instance of the
process that was created (but not yet started)

      */
```

```
      ProcessInstance createProcessInstance(String processId,
CorrelationKey correlationKey, Map<String, Object> parameters);


      /**

      * Returns the process instance with the given correlationKey.  Note
that only active process instances

      * will be returned.  If a process instance has been completed
already, this method will return

      * null.

      *

      * @param correlationKey the custom correlation key assigned when
process instance was created

      * @return the process instance with the given id or null if it
cannot be found

      */

      ProcessInstance getProcessInstance(CorrelationKey correlationKey);
```

Add the following import statement to get access to **`CorrelationAwareProcessRuntime`**:

```
import org.kie.internal.process.CorrelationAwareProcessRuntime;
```

15.2.3.5. Threads

Multi-threading can be classified into technical and logical multi-threading. Technical multi-threading occurs when multiple threads or processes are started on a computer. Logical multi-threading occurs in a BPM process, say after a process reaches a parallel gateway. The original process then splits into two processes that are executed in parallel.

The JBoss BPM Suite engine supports logical multi-threading. The logical multi-threading is implemented using one thread, which is a JBoss BPM Suite process that includes logical multi-threading. This process is executed in only one technical thread. The reason behind this implementation is that multiple technical threads need to be able to communicate state information with each other, if they are working on the same process. While multi-threading provides performance benefits, the extra logic used to ensure the different threads work together well, means that this is not guaranteed. There is an additional overhead of avoiding race conditions and deadlocks.

In general, the JBoss BPM Suite engine executes actions in serial. For example, when the engine encounters a script task in a process, it synchronously executes that script and waits for it to complete before continuing execution. Similarly, if a process encounters a parallel gateway, it sequentially triggers each of the outgoing branches, one after the other. This is possible since execution is almost always instantaneous. As a result, you may not even notice this. Similarly, action scripts in a process are also synchronously executed, and the engine waits for them to finish before continuing the process. For example, doing a **`Thread.sleep(...)`** as part of a script does not make the engine continue execution elsewhere, but blocks the engine thread during that period. The same principle applies to service tasks. When a service task is reached in a process, the engine invokes the handler of this

service synchronously. The engine waits for the **completeWorkItem(...)** method to return before continuing execution. It is important that your service handler executes your service asynchronously if its execution is not instantaneous. An example of this is a service task that invokes an external service. Since the delay in invoking this service remotely and waiting for the results may take too long, invoking this service asynchronously is advised. This means that the handler only invokes the service and notifies the engine later when the results are available. In the mean time, the process engine then continues execution of the process.

Human tasks are a typical example of a service that needs to be invoked asynchronously, as the engine does not have to wait until a human actor responds to the request. The human task handler only creates a new task when the human task node is triggered. The engine then is able to continue execution on the rest of the process (if necessary) and the handler notifies the engine asynchronously when the user completes the task.

15.2.4. 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 15.1. Creating a kmodule.xml programmatically and adding it to a KieFileSystem

```
import org.kie.api.KieServices;
import org.kie.api.builder.model.KieModuleModel;
import org.kie.api.builder.model.KieBaseModel;
import org.kie.api.builder.model.KieSessionModel;
import org.kie.api.builder.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.

15.2.5. KieResources

Example 15.2. Adding Kie artifacts to a KieFileSystem

```
import org.kie.api.builder.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 15.3. Creating and adding a Resource with an explicit type

```
import org.kie.api.builder.KieFileSystem;

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.

15.3. BUILDING WITH MAVEN

15.3.1. The kmodule

JBoss 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 is used for the tooling integration.

Building now uses Maven, and aligns with Maven practices. A KIE project or module is 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 does not provide validation at build time. There is a Maven plug-in which is recommended to use to get build time validation. The plug-in also generates many classes, making the runtime loading faster too.

KIE uses defaults to minimize 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 is used for discovery of the JAR and its contents.

Maven can either **`mvn install`** command to deploy a KieModule to the local machine, where all other applications on the local machine use it. Or it can **`mvn deploy`** command 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 in the following example:

Example 15.4. 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="new
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 two **KieBase**s have been defined and it is possible to instantiate two different types of **KieSession**s from the first one, while only one from the second.

15.3.2. 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 15.5. 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.

15.3.3. 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 15.6. Creating a KieContainer from the classpath

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

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 15.7. Retriving KieBases and KieSessions from the KieContainer

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.KieBase;
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.StatelessKieSession;

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 15.8. Retriving default KieBases and KieSessions from the KieContainer

```
import org.kie.api.runtime.KieContainer;
import org.kie.api.KieBase;
import org.kie.api.runtime.KieSession;

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 15.9. Creating a KieContainer of an existing project by ReleaseId

```
import org.kie.api.KieServices;
import org.kie.api.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;

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

15.3.4. 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.

15.3.5. KIE Plug-in

The KIE plug-in for Maven ensures that artifact resources are validated and pre-compiled. It is recommended that this is used at all times. To use the plug-in, add it to the build section of the Maven **`pom.xml`** as shown below:

Example 15.10. Adding the KIE plug-in 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>
```

NOTE

The kie-maven-plugin requires Maven version 3.1.1 or above due to the migration of sonatype-aether to eclipse-aether. Aether implementation on Sonatype is no longer maintained and supported. As the eclipse-aether requires Maven version 3.1.1 or above, the kie-maven-plugin requires it too.

Building a KIE module without the Maven plug-in copies all the resources, as is, into the resulting JAR. When that JAR is loaded by the runtime, it attempts to build all the resources then. If there are compilation issues, it returns a null **`KieContainer`**. It also pushes the compilation overhead to the runtime. Hence it is recommended that you must use the Maven plug-in.

NOTE

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

15.4. KIE DEPLOYMENT

15.4.1. 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 15.11. Building the contents of a KieFileSystem and creating a KieContainer

```
import org.kie.api.KieServices;
import org.kie.api.KieServices.Factory;
import org.kie.api.builder.KieFileSystem;
import org.kie.api.builder.KieBuilder;
import org.kie.api.runtime.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 15.12. Checking that a compilation didn't produce any error

```
import org.kie.api.builder.KieBuilder;
import org.kie.api.KieServices;

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

15.4.2. 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 15.13. Creating a new KieBase with a custom ClassLoader

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

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.

NOTE

If you are using the WebLogic Server, then you need to be aware of how the WebLogic Server finds and loads application class files at run time. When using a non-exploded WAR deployment, WebLogic packs the contents of **WEB-INF/classes** into **WEB-INF/lib/_wl_cls_gen.jar**. So when you use **KIE-Spring** to create **KieBase** and **KieSession** from resources stored under **WEB-INF/classes**, **KIE-Spring** fails to locate these resources. For this reason, the recommended deployment method in WebLogic, is to use the exploded archives contained within the product's ZIP file.

15.4.3. 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 15.14. Registering and starting a KieScanner on a KieContainer

```
import org.kie.api.KieServices;
import org.kie.api.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;
import org.kie.api.builder.KieScanner;

...

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.

Maven Settings

Since KieScanner relies on Maven, Maven should be configured with the correct updatePolicy of **always** as shown in the following example:

```
<profile>
      <id>guvnor-m2-repo</id>
      <repositories>
        <repository>
          <id>guvnor-m2-repo</id>
          <name>BRMS Repository</name>
          <url>http://10.10.10.10:8080/business-central/maven2/</url>
          <layout>default</layout>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
          </snapshots>
        </repository>
      </repositories>
    </profile>
```

15.5. RUNNING IN KIE

15.5.1. 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".)

15.5.2. 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**.

15.5.3. 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 15.15. Adding an AgendaEventListener

```
import org.kie.api.runtime.process.EventListener;

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

JBoss BRMS 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 15.16. 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

15.5.4. Logger Implementations

JBoss BPM Suite provides a listener for creating an audit log to the console or the a file on the file system. You can use these logs for debugging purposes as it contains all the events occurring at runtime. JBoss BPM Suite provides the following logger implementations:

- Console logger: This logger writes out all the events to the console. The KieServices provides you a KieRuntimeLogger that you can add to your session. When you create a console logger, pass the knowledge session as an argument.
- File logger: This logger writes out all the events to a file using an XML representation. You can use this log file in your IDE to generate a tree-based visualization of the events that occurs during execution. For the file logger, you need to provide name.
- Threaded file logger: As a file logger writes the events to disk only when closing the logger or when the number of events in the logger reaches a predefined level, you can not ise it when debugging processes at runtime. A threaded file logger writes the events to a file after a specified time interval, making it possible to use the logger to visualize the progress in real-time, while debugging processes. For the threaded file logger, you need to provide the interval (in milliseconds) after which the events must be saved. You must always close the logger at the end of your application.

Here is an example of using FileLogger:

Example 15.17. FileLogger

```
import org.kie.api.KieServices;

import org.kie.api.logger.KieRuntimeLogger;
```

```
  ...

  KieRuntimeLogger logger =
KieServices.Factory.get().getLoggers().newFileLogger(ksession, "test");

  // add invocations to the process engine here,

  // e.g. ksession.startProcess(processId);

  ...

  logger.close();
```

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

15.5.5. 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 BRMS.

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 15.18. Set Global Command

```
import org.kie.api.runtime.StatelessKieSession;
import org.kie.api.runtime.ExecutionResults;

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 15.19. Get Global Command

```
import org.kie.api.runtime.StatelessKieSession;
import org.kie.api.runtime.ExecutionResults;

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 15.20. BatchExecution Command

```
import org.kie.api.runtime.StatelessKieSession;
import org.kie.api.runtime.ExecutionResults;

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 15.5.6, “Available API”.

15.5.6. Available API

XML marshalling and unmarshalling of the Jboss BRMS Commands requires the use of special classes. This section describes these classes.

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. It is required because it has the commands converters registered. Also ensure that the **`drools-compiler`** library is present on the classpath.

- Marshalling

  `BatchExecutionHelper.newXStreamMarshaller().toXML(command);`

- Unmarshalling

  `BatchExecutionHelperProviderImpl.newXStreamMarshaller().fromXML(xml)`

The fully-qualified class name of the **`BatchExecutionHelper`** class is **`org.kie.internal.runtime.helper.BatchExecutionHelper`**.

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 have a POJO model. In both cases you have to declare your model inside **`JAXBContext`**. In order to do this, 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 KBase that has your XSD model added as a resource.

To do this, add the XSD file as a XSD **`ResourceType`** into the KBase. Finally you can create the **`JAXBContext`** using the KBase (created with the **`KnowledgeBuilder`**). Ensure that the **`drools-compiler`** and **`jaxb-xjc`** libraries are present on the classpath.

```
import org.kie.api.conf.Option;
```

```
import org.kie.api.KieBase;

Options xjcOpts = new Options();
xjcOpts.setSchemaLanguage(Language.XMLSCHEMA);
JaxbConfiguration jaxbConfiguration =
KnowledgeBuilderFactory.newJaxbConfiguration( xjcOpts, "xsd" );
kbuilder.add(ResourceFactory.newClassPathResource("person.xsd",
getClass()), ResourceType.XSD, jaxbConfiguration);
KieBase 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();
```

Ensure that the **`drools-compiler`** and **`jaxb-xjc`** libraries are present on the classpath. The fully-qualified class name of the **`DroolsJaxbHelperProviderImpl`** class is **`org.drools.compiler.runtime.pipeline.impl.DroolsJaxbHelperProviderImpl`**.

15.5.7. Supported JBoss BRMS Commands

JBoss BRMS supports the following list of commands:

- **`BatchExecutionCommand`**
- **`InsertObjectCommand`**
- **`RetractCommand`**
- **`ModifyCommand`**
- **`GetObjectCommand`**
- **`InsertElementsCommand`**
- **`FireAllRulesCommand`**
- **`StartProcessCommand`**

- **`SignalEventCommand`**
- **`CompleteWorkItemCommand`**
- **`AbortWorkItemCommand`**
- **`QueryCommand`**
- **`SetGlobalCommand`**
- **`GetGlobalCommand`**
- **`GetObjectsCommand`**

NOTE

The code snippets provided in the examples for these commands use a POJO **`org.drools.compiler.test.Person`** with the following fields:

- name: String
- age: Integer

15.5.7.1. BatchExecutionCommand

The **`BatchExecutionCommand`** command contains a list of commands that are sent to the Decision Server and executed. It has the following attributes:

Table 15.3. 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

Creating BatchExecutionCommand

```
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>
```

15.5.7.2. InsertObjectCommand

The **`InsertObjectCommand`** command is used to insert an object in the knowledge session. It has the following attributes:

Table 15.4. 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

Creating InsertObjectCommand

```
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>
```

15.5.7.3. RetractCommand

The **`RetractCommand`** command is used to retract an object from the knowledge session. It has the following attributes:

Table 15.5. RetractCommand Attributes

Name	Description	Required

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

Creating RetractCommand

There are two ways to create **`RetractCommand`**. You can either create the Fact Handle from a string, with the same output result as shown below:

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

Or set the Fact Handle that you received when the object was inserted, as shown below:

```
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>
```

15.5.7.4. ModifyCommand

The **`ModifyCommand`** command allows you to modify a previously inserted object in the knowledge session. It has the following attributes:

Table 15.6. ModifyCommand Attributes

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

Creating ModifyCommand

```
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>
```

15.5.7.5. GetObjectCommand

The **`GetObjectCommand`** command is used to get an object from a knowledge session. It has the following attributes:

Table 15.7. BatchExecutionCommand 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

Creating GetObjectCommand

```
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>
```

15.5.7.6. InsertElementsCommand

The **`InsertElementsCommand`** command is used to insert a list of objects. It has the following attributes:

Table 15.8. InsertElementsCommand Attributes

Name	Description	Required
objects	The list of objects to be inserted on the knowledge session	true

Name	Description	Required
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

Creating InsertElementsCommand

```
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":25}},
{"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>
```

15.5.7.7. FireAllRulesCommand

The **`FireAllRulesCommand`** command is used to allow execution of the rules activations created. It has the following attributes:

Table 15.9. 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

Creating FireAllRulesCommand

```
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>
```

15.5.7.8. StartProcessCommand

The **`StartProcessCommand`** command allows you to start a process using the ID. Additionally, you can pass parameters and initial data to be inserted. It has the following attributes:

Table 15.10. 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

Creating StartProcessCommand

```
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>
```

15.5.7.9. SignalEventCommand

The **`SignalEventCommand`** command is used to send a signal event. It has the following attributes:

Table 15.11. SignalEventCommand Attributes

Name	Description	Required
event-type	The type of the incoming event	true
processInstanceId	The ID of the process instance to be started	false
event	The name of the incoming event	false

Creating SignalEventCommand

```
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>
```

15.5.7.10. CompleteWorkItemCommand

The **`CompleteWorkItemCommand`** command allows you to complete a WorkItem. It has the following attributes:

Table 15.12. CompleteWorkItemCommand Attributes

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

Creating CompleteWorkItemCommand

```
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>
```

15.5.7.11. AbortWorkItemCommand

The **`AbortWorkItemCommand`** command allows you abort a WorkItem (same as **`session.getWorkItemManager().abortWorkItem(workItemId)`**) It has the following attributes:

Table 15.13. AbortWorkItemCommand Attributes

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

Creating AbortWorkItemCommand

```
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>
```

15.5.7.12. QueryCommand

The **`QueryCommand`** command executes a query defined in knowledge base. It has the following attributes:

Table 15.14. 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

Creating QueryCommand

```
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>
```

15.5.7.13. SetGlobalCommand

The **`SetGlobalCommand`** command allows you to set an object to global state. It has the following attributes:

Table 15.15. 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

Creating SetGlobalCommand

```
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>
```

15.5.7.14. GetGlobalCommand

The **`GetGlobalCommand`** command allows you to get a previously defined global object. It has the following attributes:

Table 15.16. 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

Creating GetGlobalCommand

```
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>
</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>
```

15.5.7.15. GetObjectsCommand

The **GetObjectsCommand** command returns all the objects from the current session as a Collection. It has the following attributes:

Table 15.17. 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

Creating GetObjectsCommand

```
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>
```

15.6. KIE CONFIGURATION

15.6.1. 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 BRMS, can be done by API calls, system properties or configuration files. As of this version, BRMS 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 15.21. 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>
```

15.6.2. 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 15.22. Simple StatelessKieSession execution with a Collection

```
import org.kie.api.runtime.StatelessKieSession;

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

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

Example 15.23. 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 BRMS 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 15.24. Session scoped global

```
import org.kie.api.runtime.StatelessKieSession;

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 15.25. Out identifiers

```
import org.kie.api.runtime.ExecutionResults;

// 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" );
```

15.6.3. 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 15.26. Simple Marshaller Example

```
import org.kie.api.runtime.KieSession;
import org.kie.api.KieBase;
import org.kie.api.marshalling.Marshaller;

// 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 15.27. IdentityMarshallingStrategy

```
import org.kie.api.marshalling.KieMarshallers;
import org.kie.api.marshalling.ObjectMarshallingStrategy;
import org.kie.api.marshalling.Marshaller;

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 15.28. IdentityMarshallingStrategy with Acceptor

```
import org.kie.api.marshalling.KieMarshallers;
import org.kie.api.marshalling.ObjectMarshallingStrategy;
import org.kie.api.marshalling.Marshaller;

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 15.29. Configuring a trackable timer job factory manager

```
import org.kie.api.runtime.KieSessionConfiguration;
import org.kie.api.KieServices.Factory;
import org.kie.api.runtime.conf.TimerJobFactoryOption;

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

15.6.4. KIE Persistence

Longterm out of the box persistence with Java Persistence API (JPA) is possible with BRMS. 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 15.30. Simple example using transactions

```
import org.kie.api.KieServices;
import org.kie.api.runtime.Environment;
import org.kie.api.runtime.EnvironmentName;
import org.kie.api.runtime.KieSessionConfiguration;

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 15.31. Loading a KieSession

```
import org.kie.api.runtime.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 15.32. 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 15.33. 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 15.34. JNDI properties

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

15.7. KIE SESSIONS

15.7.1. Stateless KIE Sessions

A *stateless KIE session* is a session without inference. A stateless session can be called like a function in that you can use it to pass data and then receive the result back.

Stateless KIE sessions are useful in situations requiring validation, calculation, routing and filtering.

15.7.1.1. Configuring Rules in a Stateless Session

Procedure 15.1. Task

1. Create a data model like the driver's license example below:

```
public class Applicant {
    private String name;
    private int age;
    private boolean valid;
    // getter and setter methods here
}
```

2. Write the first rule. In this example, a rule is added to disqualify any applicant younger than 18:

```
package com.company.license

rule "Is of valid age"
when
    $a : Applicant( age < 18 )
then
    $a.setValid( false );
end
```

3. When the **`Applicant`** object is inserted into the rule engine, each rule's constraints evaluate it and search for a match. (There is always an implied constraint of "object type" after which there can be any number of explicit field constraints.)

   In the **`Is of valid age`** rule there are two constraints:

   - The fact being matched must be of type Applicant
   - The value of Age must be less than eighteen.

   `$a` is a binding variable. It exists to make possible a reference to the matched object in the rule's consequence (from which place the object's properties can be updated).

   NOTE

   Use of the dollar sign (**`$`**) is optional. It helps to differentiate between variable names and field names.

4. To use this rule, save it in a file with .drl extension (for example, **`licenseApplication.drl`**), and store it in a Kie Project. A Kie Project has the structure of a normal Maven project with an additional **`kmodule.xml`** file defining the KieBases and KieSessions. Place this file in the **`resources/META-INF`** folder of the Maven project. Store all the other artifacts, such as the **`licenseApplication.drl`** containing any former rule, in the resources folder or in any other subfolder under it.

5. Create a **`KieContainer`** that reads the files to be built, from the classpath:

```
KieServices kieServices = KieServices.Factory.get();

KieContainer kContainer = kieServices.getKieClasspathContainer();
```

This compiles all the rule files found on the classpath and put the result of this compilation, a **KieModule**, in the **KieContainer**.

6. If there are no errors, you can go ahead and create your session from the **KieContainer** and execute against some data:

```
StatelessKieSession kSession = kContainer.newStatelessKieSession();

Applicant applicant = new Applicant( "Mr John Smith", 16 );

assertTrue( applicant.isValid() );

ksession.execute( applicant );

assertFalse( applicant.isValid() );
```

Here, since the applicant is under the age of eighteen, their application will be marked as "invalid".

Result

The preceding code executes the data against the rules. Since the applicant is under the age of 18, the application is marked as invalid.

15.7.1.2. Configuring Rules with Multiple Objects

Procedure 15.2. Task

1. To execute rules against any object-implementing **iterable** (such as a collection), add another class as shown in the example code below:

```
public class Applicant {
    private String name;
    private int age;
    // getter and setter methods here
}

public class Application {
    private Date dateApplied;
    private boolean valid;
    // getter and setter methods here
}
```

2. In order to check that the application was made within a legitimate time-frame, add this rule:

```
package com.company.license

rule "Is of valid age"
when
    Applicant( age < 18 )
    $a : Application()
then
    $a.setValid( false );
end
```

```
rule "Application was made this year"
when
    $a : Application( dateApplied > "01-jan-2009" )
then
    $a.setValid( false );
end
```

3. Use the JDK converter to implement the iterable interface. (This method commences with the line **`Arrays.asList(...)`**.) The code shown below executes rules against an iterable list. Every collection element is inserted before any matched rules are fired:

```
StatelessKieSession ksession = kbase.newStatelessKnowledgeSession();
Applicant applicant = new Applicant( "Mr John Smith", 16 );
Application application = new Application();
assertTrue( application.isValid() );
ksession.execute( Arrays.asList( new Object[] { application,
applicant } ) );
assertFalse( application.isValid() );
```

NOTE

The **`execute(Object object)`** and **`execute(Iterable objects)`** methods are actually "wrappers" around a further method called **`execute(Command command)`** which comes from the **`BatchExecutor`** interface.

4. Use the **`CommandFactory`** to create instructions, so that the following is equivalent to **`execute( Iterable it )`**:

```
ksession.execute( CommandFactory.newInsertIterable( new Object[] {
application, applicant } ) );
```

5. Use the **`BatchExecutor`** and **`CommandFactory`** when working with many different commands or result output identifiers:

```
List<Command> cmds = new ArrayList<Command>();
cmds.add( CommandFactory.newInsert( new Person( "Mr John Smith" ),
"mrSmith" );
cmds.add( CommandFactory.newInsert( new Person( "Mr John Doe" ),
"mrDoe" );
BatchExecutionResults results = ksession.execute(
CommandFactory.newBatchExecution( cmds ) );
assertEquals( new Person( "Mr John Smith" ), results.getValue(
"mrSmith" ) );
```

NOTE

`CommandFactory` supports many other commands that can be used in the **`BatchExecutor`**. Some of these are **`StartProcess`**, **`Query`** and **`SetGlobal`**.

15.7.2. Stateful KIE Sessions

A *stateful session* allow you to make iterative changes to facts over time. As with the **`StatelessKnowledgeSession`**, the **`StatefulKnowledgeSession`** supports the **`BatchExecutor`** interface. The only difference is the **`FireAllRules`** command is not automatically called at the end.

WARNING

Ensure that the **`dispose()`** method is called after running a stateful session. This is to ensure that there are no memory leaks. This is due to the fact that knowledge bases will obtain references to stateful knowledge sessions when they are created.

15.7.2.1. Common Use Cases for Stateful Sessions

Monitoring

For example, you can monitor a stock market and automate the buying process.

Diagnostics

Stateful sessions can be used to run fault-finding processes. They could also be used for medical diagnostic processes.

Logistical

For example, they could be applied to problems involving parcel tracking and delivery provisioning.

Ensuring compliance

For example, to validate the legality of market trades.

15.7.2.2. Stateful Session Monitoring Example

Procedure 15.3. Task

1. Create a model of what you want to monitor. In this example involving fire alarms, the rooms in a house have been listed. Each has one sprinkler. A fire can start in any of the rooms:

```
public class Room
{
 private String name
  // getter and setter methods here
}

public class Sprinkler
{
 private Room room;
 private boolean on;
 // getter and setter methods here
}

public class Fire
{
```

```
 private Room room;
 // getter and setter methods here
}

public class Alarm
{
}
```

2. The rules must express the relationships between multiple objects (to define things such as the presence of a sprinkler in a certain room). To do this, use a *binding variable* as a constraint in a pattern. This results in a cross-product.

3. Create an instance of the **Fire** class and insert it into the session.

   The rule below adds a binding to **Fire** object's room field to constrain matches. This so that only the sprinkler for that room is checked. When this rule fires and the consequence executes, the sprinkler activates:

```
rule "When there is a fire turn on the sprinkler"
when
    Fire($room : room)
    $sprinkler : Sprinkler( room == $room, on == false )
then
    modify( $sprinkler ) { setOn( true ) };
    System.out.println("Turn on the sprinkler for room
"+$room.getName());
end
```

   Whereas the stateless session employed standard Java syntax to modify a field, the rule above uses the **modify** statement. (It acts much like a "with" statement.)

15.8. RUNTIME MANAGER

15.8.1. The RuntimeManager Interface

The **RuntimeManager** interface simplifies and empowers the usage of knowledge API in context of processes. It provides configurable strategies that control actual runtime execution and by default provides following:

- Singleton: **RuntimeManager** maintains single **KieSession** regardless of number of processes available.
- Per Request: **RuntimeManager** delivers new **KieSession** for every request.
- Per Process Instance: **RuntimeManager** maintains mapping between process instance and **KieSession** and always provides same **KieSession** whenever working with given process instance.

```
package org.kie.api.runtime.manager;
public interface RuntimeManager {


    /**
```

```
     * Returns <code>RuntimeEngine</code> instance that is fully
initialized:



     *  KiseSession is created or loaded depending on the strategy

     *  TaskService is initialized and attached to ksession (via listener)

     *  WorkItemHandlers are initialized and registered on ksession

     *  EventListeners (process, agenda, working memory) are initialized
and added to ksession



     * @param context the concrete implementation of the context that is
supported by given <code>RuntimeManager</code>

     * @return instance of the <code>RuntimeEngine</code>

     */

    RuntimeEngine getRuntimeEngine(Context<?> context);



    /**

     * Unique identifier of the <code>RuntimeManager</code>

     * @return

     */

    String getIdentifier();



    /**

     * Disposes <code>RuntimeEngine</code> and notifies all listeners
about that fact.

     * This method should always be used to dispose
<code>RuntimeEngine</code> that is not needed

     * anymore.

     * ksession.dispose() shall never be used with RuntimeManager as it
will break the internal

     * mechanisms of the manager responsible for clear and efficient
disposal.<br/>

     * Dispose is not needed if <code>RuntimeEngine</code> was obtained
```

```
within active JTA transaction,

     * this means that when getRuntimeEngine method was invoked during
active JTA transaction then dispose of

     * the runtime engine will happen automatically on transaction
completion.

     * @param runtime

     */

    void disposeRuntimeEngine(RuntimeEngine runtime);



    /**

     * Closes <code>RuntimeManager</code> and releases its resources.
Shall always be called when

     * runtime manager is not needed any more. Otherwise it will still be
active and operational.

     */

    void close();



}
```

RuntimeManager is responsible for managing and delivering instances of **RuntimeEngine** to the caller. In turn, **RuntimeEngine** encapsulates two the most important elements of JBoss BPM Suite engine:

- **KieSession**
- **TaskService**

Both these components are already configured to work with each other smoothly without additional configuration from end user. So you do not need to register human task handler or keep track of it's connection to the service. **RuntimeManager** ensures that regardless of the strategy, it will provide same capabilities when it comes to initialization and configuration of the **RuntimeEngine**. This means:

- KieSession will be loaded with same factories (either in memory or JPA based)
- WorkItemHandlers will be registered on every KieSession (either loaded from db or newly created)
- Event listeners (Process, Agenda, WorkingMemory) will be registered on every **KieSession** (either loaded from db or newly created)
- **TaskService** will be configured with:

- JTA transaction manager
- Same entity manager factory as for the **KieSession**
- **UserGroupCallback** from environment

Additionally, **RuntimeManager** maintains the engine disposal by providing dedicated methods to dispose **RuntimeEngine** when it is no more required to release any resources it might have acquired.

15.8.2. The RuntimeEngine Interface

The **RuntimeEngine** interface provides the following methods access the engine components:

```
public interface RuntimeEngine {


    /**

     * Returns <code>KieSession</code> configured for this
<code>RuntimeEngine</code>

     * @return

     */

    KieSession getKieSession();



    /**

     * Returns <code>TaskService</code> configured for this
<code>RuntimeEngine</code>

     * @return

     */

    TaskService getTaskService();

}
```

15.8.3. Strategies

Singleton strategy

This instructs the **RuntimeManager** to maintain single instance of **RuntimeEngine** and in turn single instance of **KieSession** and **TaskService**. Access to the **RuntimeEngine** is synchronized and the thread is safe although it comes with a performance penalty due to synchronization. This strategy is similar to what was available by default in JBoss Enterprise BRMS Platform version 5.x and it is considered the easiest strategy and recommended to start with. It has the following characteristics:

- Small memory footprint, that is a single instance of runtime engine and task service.
- Simple and compact in design and usage.

- Good fit for low to medium load on process engine due to synchronized access.
- Due to single **`KieSession`** instance, all state objects (such as facts) are directly visible to all process instances and vice versa.
- Not contextual, that is when retrieving instances of **`RuntimeEngine`** from singleton **`RuntimeManager`**, Context instance is not important and usually the **`EmptyContext.get()`** method is used, although null argument is acceptable as well.
- Keeps track of the ID of the **`KieSession`** used between **`RuntimeManager`** restarts, to ensure it uses the same session. This ID is stored as serialized file on disc in a temporary location that depends on the environment.

Per request strategy

This instructs the **`RuntimeManager`** to provide new instance of **`RuntimeEngine`** for every request. As the **`RuntimeManager`** request considers one or more invocations within single transaction. It must return same instance of **`RuntimeEngine`** within single transaction to ensure correctness of state as otherwise the operation in one call would not be visible in the other. This is sort of a stateless strategy that provides only request scope state. Once the request is completed, the **`RuntimeEngine`**is permanently destroyed. The **`KieSession`** informationis thene removed from the database in case you used persistence. It has following characteristics:

- Completely isolated process engine and task service operations for every request.
- Completely stateless, storing facts makes sense only for the duration of the request.
- A good fit for high load, stateless processes (no facts or timers involved that shall be preserved between requests).
- **`KieSession`** is only available during life time of request and at the end is destroyed
- Not contextual, that is when retrieving instances of **`RuntimeEngine`** from per request **`RuntimeManager`**, Context instance is not important and usually the **`EmptyContext.get()`** method is used, although null argument is acceptable as well.

Per process instance strategy

This instructs the **`RuntimeManager`** to maintain a strict relationship between **`KieSession`** and **`ProcessInstance`**. That means that the **`KieSession`** will be available as long as the **`ProcessInstance`** that it belongs to is active. This strategy provides the most flexible approach to use advanced capabilities of the engine like rule evaluation in isolation (for given process instance only). It provides maximum performance and reduction of potential bottlenecks introduced by synchronization. Additionally, it reduces number of **`KieSessions`** to the actual number of process instances, rather than number of requests (in contrast to per request strategy). It has the following characteristics:

- Most advanced strategy to provide isolation to given process instance only.
- Maintains strict relationship between **`KieSession`** and **`ProcessInstance`** to ensure it will always deliver same **`KieSession`** for given **`ProcessInstance`**.
- Merges life cycle of **`KieSession`** with **`ProcessInstance`** making both to be disposed on process instance completion (complete or abort).
- Allows to maintain data (such as facts, timers) in scope of process instance, that is, only process instance will have access to that data.

- Introduces a bit of overhead due to need to look up and load **KieSession** for process instance.
- Validates usage of **KieSession**, so it can not be used for other process instances. In such cases, an exception is thrown.
- Is contextual. It accepts ***EmptyContext***, ***ProcessInstanceIdContext***, and ***CorrelationKeyContext*** context instances.

15.8.4. Usage Scenario for RuntimeManager Interface

Regular usage scenario for **RuntimeManager** is:

- At application startup
  - Build the **RuntimeManager** and keep it for entire life time of the application. It is thread safe and you can access it concurrently.
- At request
  - Get **RuntimeEngine** from **RuntimeManager** using proper context instance dedicated to strategy of **RuntimeManager**.
  - Get **KieSession** or **TaskService** from **RuntimeEngine**.
  - Perform operations on **KieSession** or **TaskService** such as **startProcess** and **completeTask**.
  - Once done with processing, dispose **RuntimeEngine** using the **RuntimeManager.disposeRuntimeEngine** method.
- At application shutdown
  - Close **RuntimeManager**.

NOTE

When the **RuntimeEngine** is obtained from **RuntimeManager** within an active JTA transaction, then there is no need to dispose **RuntimeEngine** at the end, as it automatically disposes the **RuntimeEngine** on transaction completion (regardless of the completion status commit or rollback).

15.8.5. Building RuntimeManager

Here is how you can build **RuntimeManager** and get **RuntimeEngine** (that encapsulates **KieSession** and **TaskService**) from it:

```
    // first configure environment that will be used by RuntimeManager

    RuntimeEnvironment environment =
RuntimeEnvironmentBuilder.Factory.get()

    .newDefaultInMemoryBuilder()
```

```
    .addAsset(ResourceFactory.newClassPathResource("BPMN2-
ScriptTask.bpmn2"), ResourceType.BPMN2)

    .get();


    // next create RuntimeManager - in this case singleton strategy is
chosen

    RuntimeManager manager =
RuntimeManagerFactory.Factory.get().newSingletonRuntimeManager(environment
);


    // then get RuntimeEngine out of manager - using empty context as
singleton does not keep track

    // of runtime engine as there is only one

    RuntimeEngine runtime = manager.getRuntimeEngine(EmptyContext.get());


    // get KieSession from runtime runtimeEngine - already initialized
with all handlers, listeners, etc that were configured

    // on the environment

    KieSession ksession = runtimeEngine.getKieSession();


    // add invocations to the process engine here,

    // e.g. ksession.startProcess(processId);


    // and last dispose the runtime engine

    manager.disposeRuntimeEngine(runtimeEngine);
```

15.8.6. RuntimeEnvironment Configuration

The complexity of knowing when to create, dispose, and register handlers is taken away from the end user and moved to the runtime manager that knows when and how to perform such operations. But it still allows to have a fine grained control over this process by providing comprehensive configuration of the **`RuntimeEnvironment`**.

```
  public interface RuntimeEnvironment {


    /**
```

```
     * Returns <code>KieBase</code> that shall be used by the manager

     * @return

     */

    KieBase getKieBase();



    /**

     * KieSession environment that shall be used to create instances of
<code>KieSession</code>

     * @return

     */

    Environment getEnvironment();



    /**

     * KieSession configuration that shall be used to create instances of
<code>KieSession</code>

     * @return

     */

    KieSessionConfiguration getConfiguration();



    /**

     * Indicates if persistence shall be used for the KieSession instances

     * @return

     */

    boolean usePersistence();



    /**

     * Delivers concrete implementation of
<code>RegisterableItemsFactory</code> to obtain handlers and listeners

     * that shall be registered on instances of <code>KieSession</code>

     * @return
```

```
     */

    RegisterableItemsFactory getRegisterableItemsFactory();



    /**

     * Delivers concrete implementation of <code>UserGroupCallback</code>
that shall be registered on instances

     * of <code>TaskService</code> for managing users and groups.

     * @return

     */

    UserGroupCallback getUserGroupCallback();



    /**

     * Delivers custom class loader that shall be used by the process
engine and task service instances

     * @return

     */

    ClassLoader getClassLoader();



    /**

     * Closes the environment allowing to close all depending components
such as ksession factories, etc

     */

    void close();
```

15.8.7. Building RuntimeEnvironment

The **`RuntimeEnvironment`** interface provides access to the data kept as part of the environment. You can use the builder style class that provides fluent API to configure **`RuntimeEnvironment`** with predefined settings:

```
package org.kie.api.runtime.manager;

public interface RuntimeEnvironmentBuilder {
```

```
    public RuntimeEnvironmentBuilder persistence(boolean
persistenceEnabled);


    public RuntimeEnvironmentBuilder entityManagerFactory(Object emf);


    public RuntimeEnvironmentBuilder addAsset(Resource asset, ResourceType
type);


    public RuntimeEnvironmentBuilder addEnvironmentEntry(String name,
Object value);


    public RuntimeEnvironmentBuilder addConfiguration(String name, String
value);


    public RuntimeEnvironmentBuilder knowledgeBase(KieBase kbase);


    public RuntimeEnvironmentBuilder userGroupCallback(UserGroupCallback
callback);


    public RuntimeEnvironmentBuilder
registerableItemsFactory(RegisterableItemsFactory factory);


    public RuntimeEnvironment get();


    public RuntimeEnvironmentBuilder classLoader(ClassLoader cl);

    public RuntimeEnvironmentBuilder schedulerService(Object
globalScheduler);
```

You can obtain instances of the **`RuntimeEnvironmentBuilder`** via **`RuntimeEnvironmentBuilderFactory`** that provides preconfigured sets of builder to simplify and help you build the environment for the **`RuntimeManager`**.

```
public interface RuntimeEnvironmentBuilderFactory {


    /**

     * Provides completely empty <code>RuntimeEnvironmentBuilder</code>
instance that allows to manually

     * set all required components instead of relying on any defaults.

     * @return new instance of <code>RuntimeEnvironmentBuilder</code>

     */
```

```
    public RuntimeEnvironmentBuilder newEmptyBuilder();



    /**

     * Provides default configuration of
<code>RuntimeEnvironmentBuilder</code> that is based on:

     * DefaultRuntimeEnvironment

     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that
is already preconfigured with defaults

     * @see DefaultRuntimeEnvironment

     */

    public RuntimeEnvironmentBuilder newDefaultBuilder();



    /**

     * Provides default configuration of
<code>RuntimeEnvironmentBuilder</code> that is based on:

     * DefaultRuntimeEnvironment

     * but it does not have persistence for process engine configured so
it will only store process instances in memory

     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that
is already preconfigured with defaults

     * @see DefaultRuntimeEnvironment

     */

    public RuntimeEnvironmentBuilder newDefaultInMemoryBuilder();



    /**

     * Provides default configuration of
<code>RuntimeEnvironmentBuilder</code> that is based on:

     *  DefaultRuntimeEnvironment

     * This one is tailored to works smoothly with kjars as the notion of
kbase and ksessions

     * @param groupId group id of kjar

     * @param artifactId artifact id of kjar
```

```
     * @param version version number of kjar

     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that
is already preconfigured with defaults

     * @see DefaultRuntimeEnvironment

     */

    public RuntimeEnvironmentBuilder newDefaultBuilder(String groupId,
String artifactId, String version);



    /**

     * Provides default configuration of
<code>RuntimeEnvironmentBuilder</code> that is based on:

     *  DefaultRuntimeEnvironment


     * This one is tailored to works smoothly with kjars as the notion of
kbase and ksessions

     * @param groupId group id of kjar

     * @param artifactId artifact id of kjar

     * @param version version number of kjar

     * @param kbaseName name of the kbase defined in kmodule.xml stored in
kjar

     * @param ksessionName name of the ksession define in kmodule.xml
stored in kjar

     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that
is already preconfigured with defaults

     * @see DefaultRuntimeEnvironment

     */

    public RuntimeEnvironmentBuilder newDefaultBuilder(String groupId,
String artifactId, String version, String kbaseName, String ksessionName);



    /**

     * Provides default configuration of
<code>RuntimeEnvironmentBuilder</code> that is based on:

     *  DefaultRuntimeEnvironment
```

```
     * This one is tailored to works smoothly with kjars as the notion of
kbase and ksessions

     * @param releaseId <code>ReleaseId</code> that described the kjar

     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that
is already preconfigured with defaults

     * @see DefaultRuntimeEnvironment

     */

    public RuntimeEnvironmentBuilder newDefaultBuilder(ReleaseId
releaseId);



    /**

     * Provides default configuration of
<code>RuntimeEnvironmentBuilder</code> that is based on:

     * DefaultRuntimeEnvironment

     * This one is tailored to works smoothly with kjars as the notion of
kbase and ksessions

     * @param releaseId <code>ReleaseId</code> that described the kjar

     * @param kbaseName name of the kbase defined in kmodule.xml stored in
kjar

     * @param ksessionName name of the ksession define in kmodule.xml
stored in kjar

     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that
is already preconfigured with defaults

     * @see DefaultRuntimeEnvironment

     */

    public RuntimeEnvironmentBuilder newDefaultBuilder(ReleaseId
releaseId, String kbaseName, String ksessionName);



    /**

     * Provides default configuration of
<code>RuntimeEnvironmentBuilder</code> that is based on:

     *  DefaultRuntimeEnvironment

     * It relies on KieClasspathContainer that requires to have
```

```
kmodule.xml present in META-INF folder which

     * defines the kjar itself.

     * Expects to use default kbase and ksession from kmodule.

     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that
is already preconfigured with defaults

     * @see DefaultRuntimeEnvironment

     */

    public RuntimeEnvironmentBuilder newClasspathKmoduleDefaultBuilder();



    /**

     * Provides default configuration of
<code>RuntimeEnvironmentBuilder</code> that is based on:

     *  DefaultRuntimeEnvironment


     * It relies on KieClasspathContainer that requires to have
kmodule.xml present in META-INF folder which

     * defines the kjar itself.

     * @param kbaseName name of the kbase defined in kmodule.xml

     * @param ksessionName name of the ksession define in kmodule.xml

     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that
is already preconfigured with defaults

     *@see DefaultRuntimeEnvironment

     */

    public RuntimeEnvironmentBuilder
newClasspathKmoduleDefaultBuilder(String kbaseName, String ksessionName);
```

Besides **KieSession**, Runtime Manager also provides access to **TaskService**. The default builder comes with predefined set of elements that consists of:

- Persistence unit name: It is set to **org.jbpm.persistence.jpa** (for both process engine and task service).
- Human Task handler: This is automatically registered on the **KieSession**.
- JPA based history log event listener: This is automatically registered on the **KieSession**.
- Event listener to trigger rule task evaluation (fireAllRules): This is automatically registered on the **KieSession**.

15.8.8. Registering Handlers and Listeners through RegisterableItemsFactory

The implementation of **`RegisterableItemsFactory`** provides you a dedicated mechanism to create your own handlers or listeners.

```
    /**

     * Returns new instances of <code>WorkItemHandler</code> that will be
registered on <code>RuntimeEngine</code>

     * @param runtime provides <code>RuntimeEngine</code> in case handler
need to make use of it internally

     * @return map of handlers to be registered - in case of no handlers
empty map shall be returned.

     */

    Map<String, WorkItemHandler> getWorkItemHandlers(RuntimeEngine
runtime);



    /**

     * Returns new instances of <code>ProcessEventListener</code> that
will be registered on <code>RuntimeEngine</code>

     * @param runtime provides <code>RuntimeEngine</code> in case
listeners need to make use of it internally

     * @return list of listeners to be registered - in case of no
listeners empty list shall be returned.

     */

    List<ProcessEventListener> getProcessEventListeners(RuntimeEngine
runtime);



    /**

     * Returns new instances of <code>AgendaEventListener</code> that will
be registered on <code>RuntimeEngine</code>

     * @param runtime provides <code>RuntimeEngine</code> in case
listeners need to make use of it internally

     * @return list of listeners to be registered - in case of no
listeners empty list shall be returned.

     */

    List<AgendaEventListener> getAgendaEventListeners(RuntimeEngine
runtime);
```

```
    /**

     * Returns new instances of <code>WorkingMemoryEventListener</code>
that will be registered on <code>RuntimeEngine</code>

     * @param runtime provides <code>RuntimeEngine</code> in case
listeners need to make use of it internally

     * @return list of listeners to be registered - in case of no
listeners empty list shall be returned.

     */

    List<WorkingMemoryEventListener>
getWorkingMemoryEventListeners(RuntimeEngine runtime);
```

Extending out-of-the-box implementation and adding your own is a good practice. You may not always need extensions, as the default implementations of **`RegisterableItemsFactory`** provides a mechanism to define custom handlers and listeners. Following is a list of available implementations ordered in the hierarchy of inheritance:

- **`org.jbpm.runtime.manager.impl.SimpleRegisterableItemsFactory`**: This is the simplest possible implementation that comes empty and is based on a reflection to produce instances of handlers and listeners based on given class names.
- **`org.jbpm.runtime.manager.impl.DefaultRegisterableItemsFactory`**: This is an extension of the simple implementation(**`org.jbpm.runtime.manager.impl.SimpleRegisterableItemsFactory`**) that introduces defaults described above and still provides same capabilities as the **`org.jbpm.runtime.manager.impl.SimpleRegisterableItemsFactory`** implementation.
- **`org.jbpm.runtime.manager.impl.KModuleRegisterableItemsFactory`**: This is an extension of the default implementation (**`org.jbpm.runtime.manager.impl.DefaultRegisterableItemsFactory`**) that provides specific capabilities for kmodule and still provides same capabilities as the Simple implementation (**`org.jbpm.runtime.manager.impl.SimpleRegisterableItemsFactory`**).
- **`org.jbpm.runtime.manager.impl.cdi.InjectableRegisterableItemsFactory`**: This is an extension of the default implementation (**`org.jbpm.runtime.manager.impl.DefaultRegisterableItemsFactory`**) that is tailored for CDI environments and provides CDI style approach to finding handlers and listeners through producers.

15.8.9. Registering Handlers through Configuration Files

Alternatively, you may also register simple (stateless or requiring only **`KieSession`**) work item handlers by defining them as part of **`CustomWorkItem.conf`** file and update the class path. To use this approach do the following:

1. Create a file called **drools.session.conf** inside **META-INF** of the root of the class path (**WEB-INF/classes/META-INF** for web applications).

2. Add the following line to the **drools.session.conf file**:

```
drools.workItemHandlers = CustomWorkItemHandlers.conf
```

3. Create a file called **CustomWorkItemHandlers.conf** inside **META-INF** of the root of the class path (**WEB-INF/classes/META-INF** for web applications).

4. Define custom work item handlers in MVEL format inside the **CustomWorkItemHandlers.conf** file:

```
[
  "Log": new
org.jbpm.process.instance.impl.demo.SystemOutWorkItemHandler(),
  "WebService": new
org.jbpm.process.workitem.webservice.WebServiceWorkItemHandler(ksess
ion),
  "Rest": new org.jbpm.process.workitem.rest.RESTWorkItemHandler(),
  "Service Task" : new
org.jbpm.process.workitem.bpmn2.ServiceTaskHandler(ksession)
]
```

These steps register the work item handlers for any **KieSession** created by the application, regardless of it using the **RuntimeManager** or not.

15.8.10. Registering Handlers and Listeners in CDI Environment

When you are using **RuntimeManager** in a CDI environment, you can use the dedicated interfaces to provide custom **WorkItemHandlers** and **EventListeners** to the **RuntimeEngine**.

```
public interface WorkItemHandlerProducer {


    /**

     * Returns map of (key = work item name, value work item handler
instance) of work items

     * to be registered on KieSession

     * Parameters that might be given are as follows:

     * ksessiontaskService

     * runtimeManager

     * @param identifier - identifier of the owner - usually
RuntimeManager that allows the producer to filter out

     * and provide valid instances for given owner

     * @param params - owner might provide some parameters, usually
```

```
KieSession, TaskService, RuntimeManager instances

     * @return map of work item handler instances (recommendation is to
always return new instances when this method is invoked)

     */

    Map<String, WorkItemHandler> getWorkItemHandlers(String identifier,
Map<String, Object> params);

}
```

The Event listener producer is annotated with proper qualifier to indicate what type of listeners they provide. You can select one of following to indicate the type:

- @Process for **`ProcessEventListener`**
- @Agenda for **`AgendaEventListener`**
- @WorkingMemory for **`WorkingMemoryEventListener`**

```
public interface EventListenerProducer<T> {


    /**

     * Returns list of instances for given (T) type of listeners

     * <br/>

     * Parameters that might be given are as follows:

     * ksession

     *  taskServiceruntimeManager

     * @param identifier - identifier of the owner - usually
RuntimeManager that allows the producer to filter out

     * and provide valid instances for given owner

     * @param params - owner might provide some parameters, usually
KieSession, TaskService, RuntimeManager instances

     * @return list of listener instances (recommendation is to always
return new instances when this method is invoked)

     */

    List<T> getEventListeners(String identifier, Map<String, Object>
params);

}
```

Package these interface implementations as bean archive that includes **`beans.xml`** inside **`META-INF`** folder and update the application classpath (for example, **`WEB-INF/lib`** for web application). This

enables the CDI based **RuntimeManager** to discover them and register on every **KieSession** that is created or loaded from the data store.

All the components (**KieSession**, **TaskService**, and **RuntimeManager**) are provided to the producers to allow handlers or listeners to be more stateful and be able to do more advanced things with the engine. You can also apply filtering based on the identifier (that is given as argument to the methods) to decide if the given **RuntimeManager** can receive handlers or listeners or not.

NOTE

Whenever there is a need to interact with the process engine or task service from within handler or listener, recommended approach is to use **RuntimeManager** and retrieve **RuntimeEngine** (and then **KieSession** or **TaskService**) from it as that ensures a proper state.

15.8.11. Control Parameters to Alter Default Engine Behavior

The following control parameters available to alter engine default behavior:

Table 15.18. Table Title

Name	Possible Values	Default Value	Description
jbpm.ut.jndi.lookup	String		Alternative JNDI name to be used when there is no access to the default one (java:comp/UserTransaction).
jbpm.enable.multi.con	true|false	false	Enables multiple incoming/outgoing sequence flows support for activities.
jbpm.business.calendar.properties	String	**/jbpm.business.calendar.properties**	Allows to provide alternative classpath location of business calendar configuration file.
jbpm.overdue.timer.delay	Long	2000	Specifies delay for overdue timers to allow proper initialization, in milliseconds.

Name	Possible Values	Default Value	Description
jbpm.process.name.comparator	String		Allows to provide alternative comparator class to empower start process by name feature. If not set, NumberVersionComparator is used.
jbpm.loop.level.disabled	true|false	true	Allows to enable or disable loop iteration tracking, to allow advanced loop support when using XOR gateways.
org.kie.mail.session	String	**mail/jbpmMailSession**	Allows to provide alternative JNDI name for mail session used by Task Deadlines.
jbpm.usergroup.callback.properties	String	**/jbpm.usergroup.callback.properties**	Allows to provide alternative classpath location for user group callback implementation (LDAP, DB).
jbpm.user.group.mapping	String	**${jboss.server.config.dir}/roles.properties**	Allows to provide alternative classpath location of user info configuration (used by LDAPUserInfoImpl).
jbpm.user.info.properties	String	**/jbpm.user.info.properties**	Allows to provide alternative classpath location for user group callback implementation (LDAP, DB).
org.jbpm.ht.user.separator	String	,	Allows to provide alternative separator of actors and groups for user tasks, default is comma (,).
org.quartz.properties	String		Allows to provide location of the quartz config file to activate quartz based timer service.

Name	Possible Values	Default Value	Description
jbpm.data.dir	String	**${jboss.server.data.dir}** is available otherwise **${java.io.tmpdir}**	Allows to provide location where data files produced by JBoss BPM Suite must be stored.
org.kie.executor.pool.size	Integer	1	Allows to provide thread pool size for JBoss BPM Suite executor.
org.kie.executor.retry.count	Integer	3	Allows to provide number of retries attempted in case of error by JBoss BPM Suite executor.
org.kie.executor.interval	Integer	3	Allows to provide frequency used to check for pending jobs by JBoss BPM Suite executor, in seconds.
org.kie.executor.disabled	true|false	true	Enables or disable JBoss BPM Suite executor.

These allows you to fine tune the execution for the environment needs and actual requirements. All of these parameters are set as JVM system properties, usually with -D when starting a program such as an application server.

15.8.12. Storing Process Variables Without Serialization

Objects in JBoss BPM Suite that are used as process variables must be serializable. That is, they must implement the java.io.Serializable interface. Objects that are not serializable can be used as process variables but for these you must implement and use a marshaling strategy and register it. The default strategy will not convert these variables into bytes. By default all objects need to be serializable.

For internal objects, which are modified only by the engine, it is sufficient if java.io.Serializable is implemented. The variable will be transformed into a byte stream and stored in a database.

For external data that can be modified by external systems and people (like documents from a CMS, or other database entities), other strategies need to be implemented.

JBoss BPM Suite uses what is known as the pluggable **`Variable Persistence Strategy`** - that is, it uses serialization for objects that do implement the java.io.Serializable interface but uses the Java Persistence Architecture (JPA) based **`JPAPlaceholderResolverStrategy`** class to work on objects that are entities (not implementing the java.io.Serializable interface).

Configuring Variable Persistence Strategy

To use this strategy, configure it by placing it in your Runtime Environment used for creating your Knowledge Sessions. This strategy should be set as the first one and the serialization based strategy as the last, default one. An example on how to set this is shown here:

```
// create entity manager factory
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("com.redhat.sample");

RuntimeEnvironment environment =
RuntimeEnvironmentBuilder.Factory.get().newDefaultBuilder()
  .entityManagerFactory(emf)
  .addEnvironmentEntry(EnvironmentName.OBJECT_MARSHALLING_STRATEGIES,
    new ObjectMarshallingStrategy[] {
// set the entity manager factory to JPA strategy so it knows how to store
and read entities
      new JPAPlaceholderResolverStrategy(emf),
// set the serialization based strategy as last one to deal with non entity
classes
      new
SerializablePlaceholderResolverStrategy(ClassObjectMarshallingStrategyAcce
ptor.DEFAULT)})
     .addAsset(ResourceFactory.newClassPathResource("example.bpmn"),
ResourceType.BPMN2)
     .get();

// now create the runtime manager and start using entities as part of your
process
RuntimeManager manager =
RuntimeManagerFactory.Factory.get().newSingletonRuntimeManager(environment
);
```

NOTE

Make sure to add your entity classes into **`persistence.xml`** configuration file that will be used by the JPA strategy.

How Does the JPA Strategy Work?

At runtime, process variables that need persisting are evaluated using the available strategy. It is up to the strategy to accept or reject the variable. If the variable is rejected by the first strategy, it is passed on till it reaches the default strategy.

A JPA based strategy will only accept classes that declare a field with the ***`@Id`*** annotation (javax.persistence.Id) This is the unique id that is used to retrieve the variable. On the other hand, a serialization based strategy simply accepts all variables by default.

Once the variable has been accepted, a JPA marshalling operation to store the variable is performed by the **`marshal()`** method, while the **`unmarshal()`** method will retrieve the variable from the storage.

Creating Your Own Persistence Strategy

The previous section alluded to the two methods that are used to **`marshal()`** and **`unmarshal()`** objects. These methods are part of the org.kie.api.marshalling.ObjectMarshallingStrategy interface and you can implement this interface to create a custom persistence strategy.

```
public interface ObjectMarshallingStrategy {
```

```
  public boolean accept(Object object);

  public void write(ObjectOutputStream os,
                 Object object) throws IOException;

  public Object read(ObjectInputStream os) throws IOException,
ClassNotFoundException;

  public byte[] marshal(Context context,
                       ObjectOutputStream os,
                       Object object ) throws IOException;

  public Object unmarshal(Context context,
                         ObjectInputStream is,
                         byte[] object,
                         ClassLoader classloader ) throws IOException,
ClassNotFoundException;

  public Context createContext();
}
```

The methods **`read()`** and **`write()`** are for backwards compatibility. Use the methods **`accept()`**, **`marshal()`** and **`unmarshal()`** to create your strategy.

CHAPTER 16. REMOTE API

16.1. 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 and Red Hat JBoss BRMS provide 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.

  The following two API's are only available in Red Hat JBoss BPM Suite.

- 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 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.

16.1.1. 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. All **`POST`** and **`DELETE`** return details of the request as a well as a job id that can be used to request the job status and verify whether the job finished successfully. The **`GET`** operations return information about repositories, projects and organizational units.

Parameters and results of these calls are provided in the form of JSON entities.

16.1.1.1. 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 16.1. 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]

16.1.1.2. 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 16.2. 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"}]
```

[GET] /repositories/{repositoryName}

This returns information on a specific repository - [GET]

[DELETE] /repositories/{repositoryName}

This deletes the repository - [DELETE]

[POST] /repositories/

This creates or clones the repository defined by the JSON entity - [POST]

Example 16.3. JSON entity with repository details of a repository to be cloned

```
{"name":"myClonedRepository", "description":"", "userName":"",
"password":"", "requestType":"clone",
"gitURL":"git://localhost/example-repository"}
```

[GET] /repositories/{repositoryName}/projects/

This returns a list of the projects in a specific repository as a JSON entity - [POST]

Example 16.4. JSON entity with details of existing projects

```
[ {
  "name" : "my-project-name",
  "description" : "Project to illustrate REST output",
  "groupId" : "com.acme",
  "version" : "1.0"
}, {
  "name" : "yet-another-project-name",
  "description" : "Yet Another Project to illustrate REST output",
  "groupId" : "com.acme",
  "version" : "2.2.1"
} ]
```

[POST] /repositories/{repositoryName}/projects/

This creates a project in the repository - [POST]

Example 16.5. 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 16.6. Request body that defines the project to be deleted

```
"{"name":"myProject","description": "my project"}"
```

16.1.1.3. 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].

Example 16.7. Organizational unit list in JSON

```
[ {
  "name" : "EmployeeWage",
  "description" : null,
  "owner" : "Employee",
  "defaultGroupId" : "org.bpms",
  "repositories" : [ "EmployeeRepo", "OtherRepo" ]
}, {
  "name" : "OrgUnitName",
  "description" : null,
  "owner" : "OrgUnitOwner",
  "defaultGroupId" : "org.group.id",
  "repositories" : [ "repository-name-1", "repository-name-2" ]
} ]
```

[GET] /organizationalunits/{organizationalUnitName}

This returns a JSON entity with info about a specific organizational unit - [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 16.8. Organizational unit in JSON

```
{
  "name":"testgroup",
  "description":"",
  "owner":"tester",
  "repositories":["testGroupRepository"]
}
```

[POST] /organizationalunits/{organizationalUnitName}

This *updates* the details of an existing organizational unit - [POST].

Both the **`name`** and **`owner`** fields in the consumed **`UpdateOrganizationalUnit`** instance can be left empty. Both the **`description`** field and the repository association can *not* be updated via this operation.

Example 16.9. Update organizational unit input in JSON

```
{
  "owner" : "NewOwner",
  "defaultGroupId" : "org.new.default.group.id"
}
```

[DELETE] /organizationalunits/{organizationalUnitName}

This deletes an organizational unit - [GET].

[POST] /organizationalunits/{organizationalUnitName}/repositories/{repositoryName}

This adds the repository to the organizational unit - [POST].

[DELETE] /organizationalunits/{organizationalUnitName}/repositories/{repositoryName}

This removes a repository from the organizational unit - [POST].

16.1.1.4. 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.

16.1.2. 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.

Additionally, starting with the 6.1 release, the **`activate`** and **`deactivate`** operations are available. When a deployment is deployed, it is "activated" by default: that means that new process instances can be started using the process definitions and other information in the deployment. However, at later point in time, users may want to make sure that a deployment is no longer used without necessarily aborting or stopping the existing (running) process instances. In order to do this, the deployment can first be *deactivated* before it is removed at a later date.

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)

16.1.2.1. 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).

16.1.2.2. Deployment calls

The following **`deployment`** calls are provided:

[GET] /deployment/

returns a list of all available deployed instances [GET]

[GET] /deployment/{deploymentId}

Returns a JaxbDeploymentUnit instance containing the information (including the configuration) of the deployment unit [GET]

[POST] /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]

[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.

[POST] /deployment/{deploymentId}/activate

Activates a deployment [POST]

[POST] /deployment/{deploymentId}/deactivate

Deactivates a deployment [POST]

NOTE

How to use the **`activate`** and **`deactivate`** operations:

- The **`deactivate`** operation ensures that no new process instances can be started with the existing deployment.
- If users decide that a deactivated deployment should be reactivated (instead of deleted), they can then use the **`activate`** operation to reactivate the deployment. A deployment is always "activated" by default when it is initially deployed.

16.1.3. 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 URL and the parameter with the parameter value; for example, **`http://localhost:8080/business-central/rest/task/query?workItemId=393`** returns a TaskSummary list of all tasks based on the work item with ID 393. Note that parameters and their values are case-sensitive.

Some runtime REST API calls can use a Map parameter, which means that 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") }
```

Example 16.10. 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 16.11. A GET call that returns a task details to a locally running application in Java with the direct tasks/TASKID request

```
package rest;

import java.io.InputStream;
import java.net.URL;

import javax.xml.bind.JAXBContext;
import javax.xml.transform.stream.StreamSource;

import org.apache.http.auth.AuthScope;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.impl.client.DefaultHttpClient;
import org.jboss.resteasy.client.ClientExecutor;
import org.jboss.resteasy.client.ClientRequest;
import org.jboss.resteasy.client.ClientRequestFactory;
import org.jboss.resteasy.client.ClientResponse;
import
org.jboss.resteasy.client.core.executors.ApacheHttpClient4Executor;
import org.jboss.resteasy.spi.ResteasyProviderFactory;
import org.kie.api.task.model.Task;
import
org.kie.services.client.serialization.jaxb.impl.task.JaxbTaskResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
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 16.1.5, “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.

16.1.3.1. Usage Information

16.1.3.1.1. 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/instances
/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 16.12. REST request body with the pagination parameter

```
/history/instances?page=3&pageSize=20
/history/instances?p=3&s=20
```

16.1.3.1.2. 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 16.13. REST request body with the Integer mySignal parameter

```
/rest/runtime/business-central/process/org.jbpm.test/start?
map_var1=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 16.1.5, “Execute operations”).

16.1.3.2. 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 16.1.5, “Execute operations”).

16.1.3.2.1. Process calls

The REST **`/runtime/{deploymentId}/process/`** calls are send to the Process Execution Engine.

The following **`process`** calls are provided:

/runtime/{deploymentId}/process/{processDefId}/start

creates and starts a Process instance of the provided Process definition [POST]

/runtime/{deploymentId}/process/{processDefId}/startform

Checks to see if the process defined by the ***processDefId*** exists, and if it does, returns a URL to show the form as a ***JaxbProcessInstanceFormResponse*** on a remote application [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 16.14. 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.

16.1.3.2.2. 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 16.15. Signal Call Example

```
/runtime/{deploymentId}/signal?signal={signalCode}
```

This call is equivalent to the **`ksession.signal("signalName", eventValue)`** method.

16.1.3.2.3. 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 16.16. 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]

16.1.3.2.4. 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 JBoss BPM Suite 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 16.17. 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]

16.1.3.2.5. 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).

16.1.3.3. 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 16.1.3.3.1, “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 16.1.3.3.3, “Query operations”).

/task/content/{contentId: \\d+}

returns the task content in the JAXB format [GET]

For further information, refer to Section 16.1.3.3.2, “Content operations”)

16.1.3.3.1. 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 16.1. 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 to which the task is delegated.)
`exit`	exit task [POST] **NOTE** The **`exit`** operation can be performed by any user or group specified as the administrator of a human task. If the task does not specify any values, the system automatically adds user *Administrator* and group *Administrators* to a task.
`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]

Task	Action
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.)

16.1.3.3.2. 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 .

16.1.3.3.3. 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 16.18. 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.

16.1.4. The REST Query API

The REST Query API allows developers to *richly* query tasks, variables and process instances.

The results for both operations are organized around the process instance. For example, when querying tasks, the results will be organized so that the variables and tasks returned are grouped by which process instance they belong to.

16.1.4.1. URL Layout

The rich query operations can be reached via the following URLs:

```
http://server.address:port/{application-id}/rest/query/
  runtime
    task                     * [GET] rich query for task summaries and
process variables
    process                  * [GET] rich query for process instances and
process variables
```

Both URL's take a number of different query parameters. See the next section for a description of these.

16.1.4.2. Query Parameters

In the documentation below,

- "query parameters" are strings like **processInstanceId**, **taskId** and **tid**. The case (lowercase or uppercase) of these parameters does not matter, except when the query parameter also specifies the name of a user-defined variable.
- "parameters" are the values that are passed with some query parameters. These are values like **org.process.frombulator**, **29** and **harry**.

When you submit a REST call to the query operation, your URL will look something like this:

```
http://localhost:8080/business-central/rest/query/runtime/process?
processId=org.process.frombulator&piid=29
```

A query containing multiple different query parameters will search for the intersection of the given parameters.

However, many of the query parameters described below can be entered multiple times: when multiple values are given for the same query parameter, the query will then search for any results that match one or more of the values entered.

Example 16.19. Repeated query parameters

The following process instance query:

```
processId=org.example.process&processInstanceId=27&processInstanceId=29
```

will return a result that

- only contains information about process instances with the org.example.process process definition
- only contains information about process instances that have an id of 27 *or* 29

16.1.4.2.1. Range and Regular Expression Parameters

Some query criteria can be given in ranges while for others, a simple regular expression language can be used to describe the value.

Query parameters that

- can be given in ranges have an **X** in the **min**/**max** column in the table below.
- use regular expressions have an **X** in the **regex** column below.

16.1.4.2.2. Range Query Parameters

In order to pass the lower end or start of a range, add **_min** to end of the parameter name. In order to pass the upper end or end of a range, add **_max** to end of the parameter name.

Range ends are inclusive.

Only passing one end of the range (the lower **or** upper end), results in querying on an open ended range.

Example 16.20. Range parameters

The following task query:

```
processId=org.example.process&taskId_min=50&taskId_max=53
```

will return a result that

- only contains information about tasks associated with the org.example.process process definition
- only contains information about tasks that have a task id between 50 and 53, inclusive.

While the following task query:

> **`processId=org.example.process&taskId_min=52`**
>
> will return a result that
>
> - only contains information about tasks associated with the org.example.process process definition
> - only contains information about tasks that have a task id that is *larger than or equal* to 52

16.1.4.2.3. Regular Expression Query Parameters

In order to apply regular expressions to a query parameter, add **`_re`** to the end of the parameter name.

The regular expression language contains 2 special characters:

- **`*`** means 0 or more characters
- **`.`** means 1 character

The slash character (`\`) is not interpreted.

> **Example 16.21. Regular expression parameters**
>
> The following process instance query
>
> **`processId_re=org.example.*&processVersion=2.0`**
>
> will return a result that
>
> - only contains information about process instances associated with a process definition whose name matches the regular expression org.example.*. This includes:
> - **`org.example.process`**
> - **`org.example.process.definition.example.long.name`**
> - **`orgXexampleX`**
> - only contains information about process instances that have a process (definition) version of 2.0

16.1.4.3. Parameter Table

The **`task or process`** column describes whether or not a query parameter can be used with the task and/or the process instance query operations.

Table 16.2. Query Parameters

parameter	short form	description	regex	min / max	task or process
`processinstanceid`	**`piid`**	Process instance id		X	T,P

parameter	short form	description	regex	min / max	task or process
processid	**pid**	Process id	X		T,P
workitemid	**wid**	Work item id			T,P
deploymentid	**did**	Deployment id	X		T,P
taskid	**tid**	Task id		X	T
initiator	**init**	Task initiator/creator	X		T
stakeholder	**stho**	Task stakeholder	X		T
potentialowner	**po**	Task potential owner	X		T
taskowner	**to**	Task owner	X		T
businessadmin	**ba**	Task business admin	X		T
taskstatus	**tst**	Task status			T
processinstancestatus	**pist**	Process instance status			T,P
processversion	**pv**	Process version	X		T,P
startdate	**stdt**	Process instance start date[1]		X	T,P
enddate	**edt**	Process instance end date[1]		X	T,P
varid	**vid**	Variable id	X		T,P
varvalue	**vv**	Variable value	X		T,P
var	**var**	Variable id and value [2]			T,P
varregex	**vr**	Variable id and value [3]	X		T,P
all	**all**	Which variable history logs [4]			T,P

[1] The date operations take strings with a specific date format as their values: **`yy-MM-dd_HH:mm:ss`**. However, users can also submit only part of the date:

- Submitting only the date (**`yy-MM-dd`**) means that a time of 00:00:00 is used (the beginning of the day).
- Submitting only the time (**`HH:mm:ss`**) means that the current date is used.

Table 16.3. Example date strings

Date str ing	Actual meaning
`15-05-29_13:40:12`	May 29th, 2015, 13:40:12 (1:40:12 PM)
`14-11-20`	November 20th, 2014, 00:00:00
`9:30:00`	Today, 9:30:00 (AM)

For the format used, see the SimpleDateFormat documentation.

[2] The **`var`** query parameter is used differently than other parameters. If you want to specify **both** the variable id and value of a variable (as opposed to just the variable id), then you can do it by using the **`var`** query parameter. The syntax is **`var_<variable-id>=<variable-value>`**

> **Example 16.22. `var_X=Y` example**
>
> The query parameter and parameter pair **`var_myVar=value3`** queries for process instances with variables[4] that are called **`myVar`** and that have the value **`value3`**

[3] The **`varreggex`** (or shortened version **`vr`**) parameter works similarly to the **`var`** query parameter. However, the value part of the query parameter can be a regular expression.

[4] By default, only the information from most recent (last) variable instance logs is retrieved. However, users can also retrieve all variable instance logs (that match the given criteria) by using this parameter.

16.1.4.4. Parameter Examples

Table 16.4. Query parameters examples

parameter	short form	example
`processinstanceid`	**`piid`**	**`piid=23`**
`processid`	**`pid`**	**`processid=com.acme.example`**
`workitemid`	**`wid`**	**`wid_max=11`**
`deploymentid`	**`did`**	**`did_re=com.willy.loompa.*`**

parameter	short form	example
taskid	tid	taskid=4
initiator	init	init_re=Davi*
stakeholder	stho	stho=theBoss&stho=theBossesAssistant
potentialowner	po	potentialowner=sara
taskowner	to	taskowner_re=*anderson
businessadmin	ba	ba=admin
taskstatus	tst	tst=Reserved
processinstancestatus	pist	pist=3&pist=4
processversion	pv	processVersion_re=4.2*
startdate	stdt	stdt_min=00:00:00
enddate	edt	edt_max=15-01-01
varid	vid	varid=numCars
varvalue	vv	vv=abracadabra
var	var	var_numCars=10
varregex	vr	vr_nameCar=chitty*
all	all	all

16.1.4.5. Query Output Format

The process instance query returns a JaxbQueryProcessInstanceResult instance.

The task query returns a JaxbQueryTaskResult instance.

Results are structured as follows:

- a list of process instance info (JaxbQueryProcessInstanceInfo) objects
- or a list of task instance info (JaxbQueryTaskInfo) objects

A JaxbQueryProcessInstanceInfo object contains:

- a process instance object

- a list of 0 or more variable objects

A JaxbQueryTaskInfo info object contains:

- the process instance id
- a list of 0 or more task summary obejcts
- a list of 0 or more variable objects

16.1.5. 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.

```
package rest;

import java.io.InputStream;
import java.net.URL;

import javax.xml.bind.JAXBContext;
import javax.xml.transform.stream.StreamSource;

import org.apache.http.auth.AuthScope;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.impl.client.DefaultHttpClient;
import org.jboss.resteasy.client.ClientExecutor;
import org.jboss.resteasy.client.ClientRequest;
import org.jboss.resteasy.client.ClientRequestFactory;
import org.jboss.resteasy.client.ClientResponse;
import org.jboss.resteasy.client.core.executors.ApacheHttpClient4Executor;
import org.jboss.resteasy.spi.ResteasyProviderFactory;
import org.kie.api.task.model.Task;
import
```

```
org.kie.services.client.serialization.jaxb.impl.task.JaxbTaskResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
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 16.1. 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	

16.1.6. REST summary

The URL templates in the table below are relative to the following URL:

- **`http://server:port/business-central/rest`**

Table 16.5. 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 16.6. 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 16.7. 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}/showTaskForm	GET	Generates a URL to show the task form on a remote application as a ***JaxbTaskFormResponse*** instance.
/task/{taskID}/content	GET	returns the content of a task

Table 16.8. history REST calls

URL Template	Type	Description
/history/clear/	POST	delete all process, node and history records
/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 16.9. 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
/deployment/{deploymentId}/deactivate	POST	deactivate a deployment
/deployment/{deploymentId}/activate	POST	activate a deployment

Table 16.10. query REST calls

URL Template	Type	Description
/query/runtime/process	GET	query process instances and process variables
/query/runtime/task	GET	query tasks and process variables
/query/task	POST	query tasks

16.2. 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 through the JMS API exposed by Business Central. However, it is 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.

16.2.1. JMS Queue Setup

When you deploy Business Central on the WebSphere application server 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.

16.2.2. 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

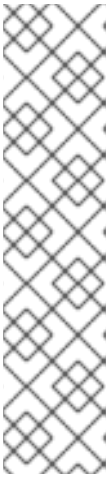
- All fields or getter/setter methods must be annotated with a **`javax.xml.bind.annotation.XmlElement`** or **`javax.xml.bind.annotation.XmlAttribute`** annotations.

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 requirements 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.



NOTE

If you create your class definitions from an XSD schema, you may end up creating classes that inconsistently (among classes) refer to a namespace. This inconsistent use of a namespace can end up preventing a these class instance from being correctly deserialized when received as a parameter in a command on the server side.

For example, you may create a class that is used in a BPMN2 process, and add an instance of this class as a parameter when starting the process. While sending the command/operation request (via the Remote (client) Java API) will succeed, the parameter will not be correctly deserialized on the server side, leading the process to eventually throw an exception about an unexpected type for the class.

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.

16.2.3. 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 JBoss BPM Suite Remote Java API.

The JBoss BPM Suite 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;
// 1
import
org.kie.services.client.api.command.exception.RemoteCommunicationException
;
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;

    //6

    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());
      }

   //5

      if (response.getIndex() == oompaProcessingResultIndex) {
        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
, JaxbSerializationProvider.JMS_SERIALIZATION_TYPE);

      //3

        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);

      //3

      } 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;

      //4

      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**.

16.3. EJB INTERFACE

Starting with version 6.1, the BPM Suite execution engine supports an EJB interface for accessing **KieSession** and **TaskService** remotely. This allows for close transaction integration between the execution engine and remote customer applications.

The implementation of this interface is a single, framework independent and container agnostic API that can be used with framework specific code. The services are exposed using the jbpm.services.api and the org.jbpm.services.ejb package and described in the next section. There is also support for CDI via the org.jbpm.services.cdi package.

The implementation doesn't support a **RuleService** at this time, but the **ProcessService** class exposes an **execute** method that allows you to use various rule related commands, like **InsertCommand** and **FireAllRulesCommand**.

Deployment of EJB Client

The EJB interface is currently only supported on Red Hat JBoss EAP. An implementation client in the form of a WAR file is available and can be extracted from the Maven Repository for Red Hat JBoss BPM Suite from the customer portal. This EJB client is present in the Maven Repository in the form of a JAR file: **jbpm-services-ejb-client-VERSION-redhat-MINOR.jar**.

Note that the inclusion of EJB doesn't mean that CDI based services will be replaced. CDI And EJB can be used together but this is not recommended. Since EJB's are not available by default in Business Central, the package kie-services must always be present in the classpath. The EJB services are suitable for embedded use cases.

16.3.1. EJB Interface Methods

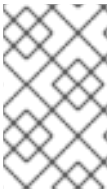
There are 5 main service interfaces that can be used by remote EJB clients. These methods are present in the following packages:

1. org.jbpm.services.ejb.api: the extension to the Services API for EJB needs.
2. org.jbpm.services.ejb.impl: EJB wrappers on top of the core service implementation.
3. org.jbpm.services.ejb.client: The EJB remote client implementation that works on JBoss EAP only.

- **`DefinitionService`**: Use this interface to gather information about processes (id, name and version), process variables (name and type), defined reusable subprocesses, domain specific service and user tasks and user task input and outputs.
- **`DeploymentService`**: Use this interface to initiate deployments and un-deployments. Methods include **`deploy`**, **`undeploy`**, **`getRuntimeManager`**, **`getDeployedUnits`**, **`isDeployed`**, **`activate`**, **`deactivate`** and **`getDeployedUnit`**. Calling the **`deploy`** method with an instance of **`DeploymentUnit`** deploys it into the runtime engine by building **`RuntimeManager`** instance for the deployed unit. Upon successful deployment an instance of **`DeployedUnit`** instance is created and cached for further usage.

  These methods only work if the artifact/project is already installed in a Maven repository.
- **`ProcessService`**: Use this interface to control the lifecycle of one or more Processes and Work Items.
- **`RuntimeDataService`**: Use this interface to retrieve data about the runtime: process instances, process definitions, node instance information and variable information. It includes several convenience methods for gathering task information based on owner, status and time.
- **`UserTaskService`**: Use this interface to control the lifecycle of a user task. Methods include all the usual ones: **`activate`**, **`start`**, **`stop`**, **`execute`** amongst others.

A synchronization service that syncs information between Business Central and the EJBs is also available. The synchronization interval can be set with the org.jbpm.deploy.sync.int system property.



NOTE

You must wait for the synchronization service to finish its synchronization of information before trrying to access this updated information via REST. Until this synchronization is finished the EJBs will not see changes done via REST.

16.3.2. Generating the EJB Services WAR

This example shows you how to create an EJB Services WAR using the new EJB Interface.

- Update the **`persistence.xml`** file in Business Central. Edit the property hibernate.hbm2ddl.auto and set its value to **`update`** (instead of **`create`**).
- Register the Human Task CallBack using a startup class:

```
@Singleton
@Startup
public class StartupBean {

@PostConstruct
public void init()
{ System.setProperty("org.jbpm.ht.callback", "jaas"); }

}
```

- Generate the WAR file: **`mvn assembly:assembly`**

- Deploy the generated war file (**sample-war-ejb-app.war**) in the JBoss EAP instance that JBoss BPM Suite 6.1 is running in.

NOTE

If deploying on a JBoss EAP container separate from the one where JBoss BPM Suite is running, you need to:

1. You need to configure your application/app server to invoke a remote EJB.
2. You need to configure your application/app server to propagate the security context.

WARNING

When you deploy your EJB WAR on the same instance of JBoss EAP, avoid using the **Singleton** strategy for your runtime sessions. If you use the **Singleton** strategy, both applications will load the same **ksession** instance from the underlying file system and cause optimistic lock exceptions.

- To test, create a simple web application and inject the EJB Services:

```
@EJB(lookup = "ejb:/sample-war-ejb-
app/ProcessServiceEJBImpl!org.jbpm.services.ejb.api.ProcessServiceEJ
BRemote")
private ProcessServiceEJBRemote processService;

@EJB(lookup = "ejb:/sample-war-ejb-
app/UserTaskServiceEJBImpl!org.jbpm.services.ejb.api.UserTaskService
EJBRemote")
private UserTaskServiceEJBRemote userTaskService;

@EJB(lookup = "ejb:/sample-war-ejb-
app/RuntimeDataServiceEJBImpl!org.jbpm.services.ejb.api.RuntimeDataS
erviceEJBRemote")
private RuntimeDataServiceEJBRemote runtimeDataService;
```

16.4. 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 allow you to interact with Business Central through 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.

16.4.1. 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. This class creates builder instances of REST using the **`newBuilder()`** method. These builder instances are then used to either directly create a **`RuntimeEngine`** instance that act as a client to the remote REST API, or to create an instance of this factory.The **`RemoteRestRuntimeEngineBuilder`** method exposes the following properties for configuration:

Table 16.11. RemoteRestRuntimeEngineBuilder Methods

Method Name	Parameter Type	Description
`Url`	**`java.net.URL`**	The URL of the deployed jbpm-console, kie-wb, or BPMS instance. For example: http://localhost:8080/jbpm-console/
`UserName`	**`java.lang.String`**	The user name to access the REST API.
`Password`	**`java.lang.String`**	The password to access the REST API.
`DeploymentId`	**`java.lang.String`**	The name (id) of the deployment the **`RuntimeEngine`** must interact with. This can be an empty String in case you are only interested in task operations.
`Timeout`	**`int`**	The maximum number of seconds the engine must wait for a response from the server.
`ProcessInstanceId`	**`long`**	The method that adds the process instance id, which may be necessary when interacting with deployments that employ the per process instance runtime strategy.

Method Name	Parameter Type	Description
ExtraJaxbClasses	**class**	The method that 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 is important to have added the classes via this method first so that the class instances can be serialized correctly.

Once you have configured all necessary properties, call build() to get access to the **RuntimeEngine**.

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 startProcessAndHandleTaskViaRestRemoteJavaAPI(URL instanceUrl,
String deploymentId, String user, String password) {

  // the serverRestUrl should contain a URL similar to
"http://localhost:8080/jbpm-console/"



  // Setup the factory class with the necessary information to communicate
with the REST services

 RuntimeEngine engine = RemoteRestRuntimeEngineFactory.newRestBuilder()
    .addUrl(new URL(instanceUrl))
    .addUserName(user).addPassword(password)
    .addDeploymentId(deploymentId)
      .build();

  KieSession ksession = engine.getKieSession();

  TaskService taskService = engine.getTaskService();


  // Each operation on a KieSession, TaskService or AuditService (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);

}
```

NOTE

To guarantee high performance, the **`TaskSummary`** method **`getPotentialOwners()`** does not return the list of potential owners of a task.

Instead, you should retrieve information about owners on an individual task basis.

```
Task task = taskService.getTaskById(sum.getId());

List<OrganizationalEntity> org =
task.getPeopleAssignments().getPotentialOwners();

  for (OrganizationalEntity ent: org) {
    System.out.println("org: " + ent.getId());
  }

}
```

Further, actual owners and created by users can be retrieved using the **`getActualOwnerId()`** and **`getCreateById()`** methods.

16.4.2. Calling tasks without Deployment Id

The **`addDeploymentId()`** method of the **`RemoteRestRuntimeEngineFactory`** class requires the calling application to pass ***`deploymentId`*** parameter to connect to the KIE-WB. The ***`deploymentId`*** parameter is the name or ID of the deployment with which the RuntimeEngine interacts. However, there may be applications that require working with a Human Task node and dealing with processes across multiple deployments. In such cases, where providing ***`deploymentId`*** parameters for multiple deployments to connect to the KIE-WB is not feasible, you can use the fluent factory builder API for working with Human Tasks.

This API does not require the calling application to pass the ***`deploymentId`*** parameter. In case there are requests that require the ***`deploymentId`*** parameter but do not have it configured, an exception is thrown.

Here is an example to show this in action:

```
RemoteRestRuntimeEngineFactory factory =
RemoteRestRuntimeEngineFactory.newBuilder()
      .addUserName("user")
      .addPassword("pass")
      .addUrl(new URL("http://localhost:8080/business-central"))
      .build();

RuntimeEngine runtimeEngine = factory.newRuntimeEngine();

// this call doesn't require the deployment id and will return
successfully

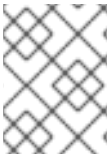
runtimeEngine.getTaskService().claim(23l, "user");

// this will throw a "MissingRequiredInfoException" because the deployment
id is required here

runtimeEngine.getKieSession().startProcess("org.test.process");
```

16.4.3. Custom Model Objects and Remote API

Usage of custom model objects from a client application using the Remote API is supported in JBoss BPM Suite. 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 16.1. Accessing custom model objects using the Remote API

1. Ensure 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 JBoss BPM Suite 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);
```

Ensure that your client application has imported the correct JBoss BPM Suite libraries for the example to work.

16.4.4. 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 16.12. 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.
`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.

Method Name	Parameter Type	Description
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 and Websphere.
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.
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);
    }
}
```

16.4.5. Supported Methods

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 throw an **`UnsupportedOperationException`** explaining that the called method is not available.

Table 16.13. 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

Returns	Method signature	Description
`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 16.14. 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 16.15. Available `WorkItemManager` methods

Returns	Method signature	Description
`void`	`abortWorkItem(long id)`	Abort the work item
`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 16.16. 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
`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

Returns	Method signature	Description
`void`	`stop(long taskId, String userId)`	Stop a task
`void`	`suspend(long taskId, String userId)`	Suspend a task

Table 16.17. 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)`
`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)`

NOTE

The ***`language`*** parameter is not used for task retrieval and query **`TaskService`** methods anymore. However, the method signatures still contain it to support backward compatibility. This parameter will be removed in future releases.

Table 16.18. 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)`
`List<VariableInstanceLog>`	`findVariableInstancesByName(String variableId, boolean onlyActiveProcesses)`
`List<VariableInstanceLog>`	`findVariableInstancesByNameAndValue( String variableId, String value, boolean onlyActiveProcesses)`
`void`	`clear()`

CHAPTER 17. CDI INTEGRATION

17.1. JBOSS BPM SUITE WITH CDI INTEGRATION

Apart from the API based approach, JBoss BPM Suite 6 also provides the Context and Dependency Injection (CDI) to build your custom applications. As the service modules in JBoss BPM Suite 6 are now grouped with their framework dependencies, you can build systems that can be consumed irrespective of the framework used. This grouping of individual modules with a framework gives you the freedom to choose a suitable option for your application.

The **`jbpm-services-cdi`** module is designed with CDI framework for CDI containers. It provides CDI wrappers on top of the core BPM Suite services.

JBoss BPM Suite 6 provides the following set of services, available for injection in any other CDI bean:

- **`DeploymentService`**
- **`ProcessService`**
- **`UserTaskService`**
- **`RuntimeDataService`**
- **`DefinitionService`**

17.2. DEPLOYMENT SERVICE

The **`DeploymentService`** service is responsible for deploying and undeploying deployment units into the runtime environment. Deployment units includes resources such as rule, processes, and forms. The **`DeploymentService`** can be used to retrieve:

- a RuntimeManager instance for given deployment id
- a deployed unit that represents complete deployment process for given deployment id
- list of all deployed units known to the deployment **`DeploymentService`** service

`DeploymentService` service fires CDI events in case of deployment or undeployment of deployment units. This allows application components to react real time to the CDI events and store or remove deployment details from the memory. The deployment event with qualifier **`@Deploy`** is fired on deployment and the deployment event with qualifier **`@Undeploy`** is fired on undeployment. You can use CDI observer mechanism to get a notification on these events.

17.2.1. Saving and Removing Deployments from Database

The deployment service stores the deployed units in memory by default. Here is how you can save deployments in the data store of your choice:

```
public void saveDeployment(@Observes @Deploy DeploymentEvent event) {

  // store deployed unit info for further needs
```

```
  DeployedUnit deployedUnit = event.getDeployedUnit();

    }
```

Here is how you can remove a saved deployment when undeployed:

```
public void removeDeployment(@Observes @Undeploy DeploymentEvent event) {

 // remove deployment with id event.getDeploymentId()

    }
```

NOTE

Deployment service comes with deployment synchronization mechanism that allows to persist deployed units into data base that is by default enabled.

17.2.2. Available Deployment Services

You can use qualifiers to instruct the CDI container which deployment service it must use. JBoss BPM Suite comes with the following Deployment Services out of the box:

- **`@Kjar`**: This Kmodule deployment service is tailored to work with KmoduleDeploymentUnits that is a small descriptor on top of a kjar.
- **`@Vfs`**: This VFS deployment service allows you to deploy assets directly from VFS (Virtual File System).

Note that every implementation of deployment service must have with a dedicated implementation of deployment unit as the services mentioned above.

17.2.3. FormProviderService Service

The **`FormProviderService`** service provides access to form representations for the user and process forms. It is built on the concept of isolated **`FormProviders`**.

Implementations of **`FormProvider`** interface must define a priority, as this is the main driver for the **`FormProviderService`** service to ask for the content of the form of a given provider. **`FormProviderService`** service collects all available providers and iterates over them asking for the form content in the order of the specified priority. The lower the priority number, the higher priority it gets during evaluation. For example, a provider with priority 5 is evaluated before provider with priority 10. **`FormProviderService`** service iterates over available providers as long as one delivers the content. In a worse case scenario, it returns simple text based forms.

The **`FormProvider`** interface shown below describes contract for the implementations:

```
public interface FormProvider {
    int getPriority();
    String render(String name, ProcessDesc process, Map<String, Object>
renderContext);
    String render(String name, Task task, ProcessDesc process, Map<String,
Object> renderContext);
}
```

JBoss BPM Suite comes with following **`FormProvidersService`** out of the box:

- Additional **`FormProviderService`** available with the form modeler. The priority number of this **`FormProviderService`** is 2.
- Fremarker based implementation to support process and task forms. The priority number of this **`FormProvideServicer`** is 3.
- Default forms provider. This is has the lowest priority and considered as a last resort if none of the other providers deliver content. This provider provides simplest possible forms.

17.2.4. RuntimeDataService Service

The **`RuntimeDataService`** service provides access to actual data that is available on runtime such as:

- Available processes to be executed
- Active process instances
- Process instance history
- Process instance variables
- Active and completed nodes of process instance

In a default implementation, the **`RuntimeDataService`** service observes deployment events and indexes all deployed processes to expose them to the calling components.

17.2.5. DefinitionService Service

A **`DefinitionService`** is a service that provides access to process details stored as part of BPMN2 XML. Before using any method that provides information, you must invoke the **`buildProcessDefinition`** method to populate repository with process information taken from BPMN2 content.

The BPMN2 Data Service provides access to following data:

- Overall description of process for given process definition
- Collection of all user tasks found in the process definition
- Information about defined inputs for user task node
- Information about defined outputs for user task node
- IDs of reusable processes (call activity) defined within the given process definition
- Information about process variables defined within given process definition

Information about all organizational entities (users and groups) included in the process definition. Depending on the actual process definition the returned values for users and groups can contain actual user or group name or process variable that is used to get actual user or group name on runtime.

17.3. CONFIGURING CDI INTEGRATION

In order to use the **`jbpm-services-cdi`** API in your system, you need to provide some JavaBeans for

the out of the box services to satisfy all dependencies, such as:

- Entity manager and entity manager factory
- User group callback for human tasks
- Identity provider to pass authenticated user information to the services

Here is an example of a producer bean, that satisfy all the requirements of the **jbpm-services-cdi** API in a JEE environment like the JBoss Application Server:

```
public class EnvironmentProducer {

    @PersistenceUnit(unitName = "org.jbpm.domain")

    private EntityManagerFactory emf;


    @Inject

    @Selectable

    private UserGroupInfoProducer userGroupInfoProducer;


    @Inject

    @Kjar

    private DeploymentService deploymentService;


    @Produces

    public EntityManagerFactory getEntityManagerFactory() {

        return this.emf;

    }


    @Produces

    public org.kie.api.task.UserGroupCallback
produceSelectedUserGroupCalback() {

        return userGroupInfoProducer.produceCallback();

    }


    @Produces

    public UserInfo produceUserInfo() {

        return userGroupInfoProducer.produceUserInfo();
```

```
    }

    @Produces
    @Named("Logs")
    public TaskLifeCycleEventListener produceTaskAuditListener() {
        return new JPATaskLifeCycleEventListener(true);
    }

    @Produces
    public DeploymentService getDeploymentService() {
        return this.deploymentService;
    }

    @Produces
    public IdentityProvider produceIdentityProvider {
        return new IdentityProvider() {
             // implement IdentityProvider
        };
    }
}
```

Provide an alternative for user group callback in the configuration file called the **`beans.xml`**. For example, the **`org.jbpm.kie.services.cdi.producer.JAASUserGroupInfoProducer`** class allows JBoss Application Server to reuse security settings on application server regardless of what it actually is (such as LDAP and DB):

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://docs.jboss.org/cdi/beans_1_0.xsd">

 <alternatives>


<class>org.jbpm.kie.services.cdi.producer.JAASUserGroupInfoProducer</class
>
```

```
  </alternatives>

</beans>
```

Optionally, you can use several other producers provided to deliver components like Process, Agenda, **WorkingMemory** event listeners, and **WorkItemHandlers**. To provide these components, you need to implement the following interfaces:

- ```
  /**

   * Allows to provide custom implementations to deliver WorkItem name and WorkItemHandler instance pairs

   * for the runtime.

   * <br/>

   * It will be invoked by RegisterableItemsFactory implementation (especially InjectableRegisterableItemsFactory

   * in CDI world) for every KieSession. Recommendation is to always produce new instances to avoid unexpected

   * results.

   *

   */

  public interface WorkItemHandlerProducer {


      /**

       * Returns map of (key = work item name, value work item handler instance) of work items

       * to be registered on KieSession

       * <br/>

       * Parameters that might be given are as follows:

       * <ul>

       *  <li>ksession</li>

       *  <li>taskService</li>

       *  <li>runtimeManager</li>

       * </ul>

       *

       * @param identifier - identifier of the owner - usually
  ```
  ```

```
RuntimeManager that allows the producer to filter out

     * and provide valid instances for given owner

     * @param params - owner might provide some parameters, usually
KieSession, TaskService, RuntimeManager instances

     * @return map of work item handler instances (recommendation is
to always return new instances when this method is invoked)

     */

    Map<String, WorkItemHandler> getWorkItemHandlers(String
identifier, Map<String, Object> params);

}
```

-
```
/**

 * Allows do define custom producers for know EventListeners.
Intention of this is that there might be several

 * implementations that might provide different listener instance
based on the context they are executed in.

 * <br/>

 * It will be invoked by RegisterableItemsFactory implementation
(especially InjectableRegisterableItemsFactory

 * in CDI world) for every KieSession. Recommendation is to always
produce new instances to avoid unexpected

 * results.

 *

 * @param <T> type of the event listener - ProcessEventListener,
AgendaEventListener, WorkingMemoryEventListener

 */

public interface EventListenerProducer<T> {


    /**

     * Returns list of instances for given (T) type of listeners

     * <br/>

     * Parameters that might be given are as follows:

     * <ul>

     *  <li>ksession</li>
```

```
     *  <li>taskService</li>

     *  <li>runtimeManager</li>

     * </ul>

     * @param identifier - identifier of the owner - usually
RuntimeManager that allows the producer to filter out

     * and provide valid instances for given owner

     * @param params - owner might provide some parameters, usually
KieSession, TaskService, RuntimeManager instances

     * @return list of listener instances (recommendation is to
always return new instances when this method is invoked)

     */

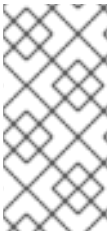
    List<T> getEventListeners(String identifier, Map<String, Object>
params);}
```

JavaBeans implementing the above mentioned interfaces are collected on runtime and consulted when building **KieSession** by **RuntimeManager**.

17.4. RUNTIMEMANAGER AS CDI BEAN

You can inject **RuntimeManager** as CDI bean into any other CDI bean within your application. **RuntimeManager** comes with the following predefined strategies and each of them have CDI qualifiers:

- @Singleton
- @PerRequest
- @PerProcessInstance



NOTE

Though you can directly inject **RuntimeManager** as CDI bean, it is recommended to utilize JBoss BPM Suite services when frameworks like CDI, ejb or Spring are used. JBoss BPM Suite services provide significant amount of features that encapsulate best practices when using **RuntimeManager**.

Here is an example of a producer method implementation that provides **RuntimeEnvironment**:

```
public class EnvironmentProducer {

    //add same producers as for services


    @Produces

    @Singleton
```

```
    @PerRequest

    @PerProcessInstance

    public RuntimeEnvironment produceEnvironment(EntityManagerFactory emf)
{


        RuntimeEnvironment environment =
RuntimeEnvironmentBuilder.Factory.get()

                .newDefaultBuilder()

                .entityManagerFactory(emf)

                .userGroupCallback(getUserGroupCallback())


.registerableItemsFactory(InjectableRegisterableItemsFactory.getFactory(be
anManager, null))

                .addAsset(ResourceFactory.newClassPathResource("BPMN2-
ScriptTask.bpmn2"), ResourceType.BPMN2)

                .addAsset(ResourceFactory.newClassPathResource("BPMN2-
UserTask.bpmn2"), ResourceType.BPMN2)

                .get();

        return environment;

    }

}
```

In the example above, a single producer method is capable of providing **`RuntimeEnvironment`** for all strategies of **`RuntimeManager`** by specifying all qualifiers on the method level. Once a complete producer is available, you can inject **`RuntimeManager`** into the application's CDI bean as shown below:

```
public class ProcessEngine {

    @Inject

    @Singleton

    private RuntimeManager singletonManager;


    public void startProcess() {


        RuntimeEngine runtime =
singletonManager.getRuntimeEngine(EmptyContext.get());

        KieSession ksession = runtime.getKieSession();
```

```
        ProcessInstance processInstance =
ksession.startProcess("UserTask");

        singletonManager.disposeRuntimeEngine(runtime);

    }

}
```

NOTE

Although there is an option available for having a single **RuntimeManager** in the application, but it is recommended to make use of **DeploymentService** whenever you need to have many **RuntimeManagers** active within your application.

As an alternative to **DeploymentService**, the application can inject **RuntimeManagerFactory** and then create **RuntimeManager** instance manually. In such cases, **EnvironmentProducer** remains the same as the **DeploymentService**. Here is an example of a simple **ProcessEngine** bean:

```
public class ProcessEngine {


    @Inject

    private RuntimeManagerFactory managerFactory;



    @Inject

    private EntityManagerFactory emf;



    @Inject

    private BeanManager beanManager;


    public void startProcess() {

        RuntimeEnvironment environment =
RuntimeEnvironmentBuilder.Factory.get()

                .newDefaultBuilder()

                .entityManagerFactory(emf)

                .addAsset(ResourceFactory.newClassPathResource("BPMN2-
ScriptTask.bpmn2"), ResourceType.BPMN2)

                .addAsset(ResourceFactory.newClassPathResource("BPMN2-
UserTask.bpmn2"), ResourceType.BPMN2)
```

```
.registerableItemsFactory(InjectableRegisterableItemsFactory.getFactory(be
anManager, null))

                .get();



        RuntimeManager manager =
managerFactory.newSingletonRuntimeManager(environment);

        RuntimeEngine runtime =
manager.getRuntimeEngine(EmptyContext.get());

        KieSession ksession = runtime.getKieSession();



        ProcessInstance processInstance =
ksession.startProcess("UserTask");



        manager.disposeRuntimeEngine(runtime);

        manager.close();

    }

}
```

CHAPTER 18. SOAP INTERFACE

18.1. SOAP API

Simple Object Access Protocol (SOAP) is a type of distribution architecture used for exchanging information. This protocol is lightweight; that is, it requires a minimal amount of overhead on the system. SOAP is used as a protocol for communication, and it is versatile enough to allow the use of different transport protocols. Like REST, SOAP allows client-server communication: clients can initiate requests to servers of a particular URL with parameters if necessary. The servers then process the requests and return a response based on the particular URL.

Red Hat JBoss BPM Suite provides one SOAP service in the form of the CommandWebService

18.2. CLIENT-SIDE JAVA WEBSERVICE CLIENT

The execution server that is part of JBoss BPM Suite web tooling comes with a Web Service interface. In addition, JBoss BPM Suite incorporates existing REST and JMS interfaces, as well as client-side Java clients to deal with REST and JMS.

Classes generated by the **`kie-remote-client`** module function as a client-side interface for SOAP. The **`CommandWebServiceClient`** class referenced in the test code below is generated by the Web Service Description Language (WSDL) in the kie-remote-client jar.

```
import org.kie.remote.client.api.RemoteRuntimeEngineFactory;
import org.kie.remote.client.jaxb.JaxbCommandsRequest;
import org.kie.remote.client.jaxb.JaxbCommandsResponse;
import org.kie.remote.jaxb.gen.StartProcessCommand;
import org.kie.remote.services.ws.command.generated.CommandWebService;
import
org.kie.services.client.serialization.jaxb.impl.JaxbCommandResponse;
```

```
    public void runCommandWebService(String user, String password, String
processId, String deploymentId, String applicationUrl) throws Exception {

        CommandWebService client =
RemoteRuntimeEngineFactory.newCommandWebServiceClientBuilder()
                .addDeploymentId(deploymentId)
                .addUserName(user)
                .addPassword(password)
                .addServerUrl(applicationUrl)
                .buildBasicAuthClient();

        // Get a response from the WebService
        StartProcessCommand cmd = new StartProcessCommand();
        cmd.setProcessId(processId);
        JaxbCommandsRequest req = new JaxbCommandsRequest(deploymentId,
cmd);
        final JaxbCommandsResponse response = client.execute(req);

        JaxbCommandResponse<?> cmdResp = response.getResponses().get(0);
        JaxbProcessInstanceResponse procInstResp =
(JaxbProcessInstanceResponse) cmdResp;
        long procInstId = procInstResp.getId();
    }
```

-

The SOAP interface for the JBoss BPM Suite Execution Server is currently available for EAP, EWS, WAS, and AS. This client-side interface incorporates the **`CommandWebService`** class and includes the execute operation as depicted below:

```
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;
import org.kie.remote.client.jaxb.JaxbCommandsRequest;
import org.kie.remote.client.jaxb.JaxbCommandsResponse

@WebService(name = "CommandServicePortType", targetNamespace =
"http://services.remote.kie.org/6.3.0.1/command")
public interface CommandWebService {

    /**
     *
     * @param request
     * @return
     *     returns org.kie.remote.client.jaxb.JaxbCommandsResponse
     * @throws CommandWebServiceException
     */
    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "execute", targetNamespace =
"http://services.remote.kie.org/6.3.0.1/command", className =
"org.kie.remote.services.ws.command.generated.Execute")
    @ResponseWrapper(localName = "executeResponse", targetNamespace =
"http://services.remote.kie.org/6.3.0.1/command", className =
"org.kie.remote.services.ws.command.generated.ExecuteResponse")
    public JaxbCommandsResponse execute(@WebParam(name = "request",
targetNamespace = "") JaxbCommandsRequest request) throws
CommandWebServiceException;

}
```

APPENDIX A. REVISION HISTORY

Revision 1.0.0-72 **Thu Dec 17 2015** **Vidya Iyengar**

Build includes various enhancements and fixes