



# Red Hat Process Automation Manager 7.1

Designing a decision service using DRL rules



## Red Hat Process Automation Manager 7.1 Designing a decision service using DRL rules

---

Red Hat Customer Content Services  
brms-docs@redhat.com

## Legal Notice

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

The OpenStack<sup>®</sup> 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

This document describes how to design a decision service using DRL rules in Red Hat Process Automation Manager 7.1.

---

## Table of Contents

<b>PREFACE</b> .....	<b>3</b>
<b>CHAPTER 1. RULE-AUTHORING ASSETS IN RED HAT PROCESS AUTOMATION MANAGER</b> .....	<b>4</b>
<b>CHAPTER 2. DRL (DROOLS RULE LANGUAGE) RULES</b> .....	<b>6</b>
<b>CHAPTER 3. DATA OBJECTS</b> .....	<b>7</b>
3.1. CREATING DATA OBJECTS .....	7
<b>CHAPTER 4. CREATING DRL RULES IN BUSINESS CENTRAL</b> .....	<b>9</b>
4.1. ADDING WHEN CONDITIONS IN DRL RULES .....	12
4.2. ADDING THEN ACTIONS IN DRL RULES .....	15
4.2.1. Rule attributes .....	17
<b>CHAPTER 5. EXECUTING RULES</b> .....	<b>20</b>
<b>CHAPTER 6. OTHER METHODS FOR CREATING AND EXECUTING DRL RULES</b> .....	<b>25</b>
6.1. CREATING AND EXECUTING DRL RULES IN RED HAT JBOSS DEVELOPER STUDIO .....	25
6.2. CREATING AND EXECUTING DRL RULES USING JAVA .....	29
6.3. CREATING AND EXECUTING DRL RULES USING MAVEN .....	32
6.4. EXECUTABLE RULE MODELS .....	37
6.4.1. Embedding an executable rule model in a Maven project .....	38
6.4.2. Embedding an executable rule model in a Java application .....	40
<b>CHAPTER 7. NEXT STEPS</b> .....	<b>42</b>
<b>APPENDIX A. VERSIONING INFORMATION</b> .....	<b>43</b>



## PREFACE

As a business rules developer, you can define business rules using the DRL (Drools Rule Language) designer in Business Central. DRL rules are defined directly in free-form **.drl** text files instead of in a guided or tabular format like other types of rule assets in Business Central. These DRL files form the core of the decision service for your project.

### Prerequisite

The team and project for the DRL rules have been created in Business Central. Each asset is associated with a project assigned to a team. For details, see [Getting started with decision services](#).

# CHAPTER 1. RULE-AUTHORING ASSETS IN RED HAT PROCESS AUTOMATION MANAGER

Red Hat Process Automation Manager provides several assets that you can use to create business rules for your decision service. Each rule-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights each rule-authoring asset in Business Central to help you decide or confirm the best method for creating rules in your decision service.

**Table 1.1. Rule-authoring assets in Business Central**

Asset	Highlights	Documentation
Guided decision tables	<ul style="list-style-type: none"> <li>• Are tables of rules that you create in a UI-based table designer in Business Central</li> <li>• Are a wizard-led alternative to uploaded decision table spreadsheets</li> <li>• Provide fields and options for acceptable input</li> <li>• Support template keys and values for creating rule templates</li> <li>• Support hit policies, real-time validation, and other additional features not supported in other assets</li> <li>• Are optimal for creating rules in a controlled tabular format to minimize compilation errors</li> </ul>	<a href="#">Designing a decision service using guided decision tables</a>
Uploaded decision tables	<ul style="list-style-type: none"> <li>• Are XLS or XLSX decision table spreadsheets that you upload into Business Central</li> <li>• Support template keys and values for creating rule templates</li> <li>• Are optimal for creating rules in decision tables already managed outside of Business Central</li> <li>• Have strict syntax requirements for rules to be compiled properly when uploaded</li> </ul>	<a href="#">Designing a decision service using uploaded decision tables</a>

Asset	Highlights	Documentation
Guided rules	<ul style="list-style-type: none"> <li>• Are individual rules that you create in a UI-based rule designer in Business Central</li> <li>• Provide fields and options for acceptable input</li> <li>• Are optimal for creating single rules in a controlled format to minimize compilation errors</li> </ul>	<a href="#">Designing a decision service using guided rules</a>
Guided rule templates	<ul style="list-style-type: none"> <li>• Are reusable rule structures that you create in a UI-based template designer in Business Central</li> <li>• Provide fields and options for acceptable input</li> <li>• Support template keys and values for creating rule templates (fundamental to the purpose of this asset)</li> <li>• Are optimal for creating many rules with the same rule structure but with different defined field values</li> </ul>	<a href="#">Designing a decision service using guided rule templates</a>
DRL rules	<ul style="list-style-type: none"> <li>• Are individual rules that you define directly in <b>.drl</b> text files</li> <li>• Provide the most flexibility for defining rules and other technicalities of rule behavior</li> <li>• Can be created in certain standalone environments and integrated with Red Hat Process Automation Manager</li> <li>• Are optimal for creating rules that require advanced DRL options</li> <li>• Have strict syntax requirements for rules to be compiled properly</li> </ul>	<a href="#">Designing a decision service using DRL rules</a>

## CHAPTER 2. DRL (DROOLS RULE LANGUAGE) RULES

DRL (Drools Rule Language) rules are business rules that you define directly in **.drl** text files. These DRL files are the source in which all other rule assets in Business Central are ultimately rendered. You can create and manage DRL files within the Business Central interface, or create them externally using Red Hat Developer Studio, Java objects, or Maven archetypes. A DRL file can contain one or more rules that define at minimum the rule conditions (**when**) and actions (**then**). The DRL designer in Business Central provides syntax highlighting for Java, DRL, and XML.

All data objects related to a DRL rule must be in the same project package as the DRL rule in Business Central. Assets in the same package are imported by default. Existing assets in other packages can be imported with the DRL rule.

## CHAPTER 3. DATA OBJECTS

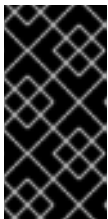
Data objects are the building blocks for the rule assets that you create. Data objects are custom data types implemented as Java objects in specified packages of your project. For example, you might create a **Person** object with data fields **Name**, **Address**, and **DateOfBirth** to specify personal details for loan application rules. These custom data types determine what data your assets and your decision services are based on.

### 3.1. CREATING DATA OBJECTS

The following procedure is a generic overview of creating data objects. It is not specific to a particular business process.

#### Procedure

1. In Business Central, go to **Menu → Design → Projects** and click the project name.
2. Click **Add Asset → Data Object**.
3. Enter a unique **Data Object** name and select the **Package** where you want the data object to be available for other rule assets. Data objects with the same name cannot exist in the same package. The package that you specify must be the same package where the rule assets that require those data objects have been assigned or will be assigned.



#### IMPORTING DATA OBJECTS FROM OTHER PACKAGES

You can also import an existing data object from another package into the package of the rule asset. Select the relevant rule asset within the project and in the asset designer, go to **Data Objects → New item** to select the object to be imported.

4. To make your data object persistable, select the **Persistable** checkbox. Persistable data objects are able to be stored in a database according to the JPA specification. The default JPA is Hibernate.
5. Click **Ok**.
6. In the data object designer, click **add field** to add a field to the object with the attributes **Id**, **Label**, and **Type**. Required attributes are marked with an asterisk (\*).
  - **Id**: Enter the unique ID of the field.
  - **Label**: (Optional) Enter a label for the field.
  - **Type**: Enter the data type of the field.
  - **List**: Select this check box to enable the field to hold multiple items for the specified type.

Figure 3.1. Add data fields to a data object

New Field

Id \*


salary

Label

Salary

Type \*

BigInteger

List 

☐

Cancel

Create

Create and continue

- Click **Create** to add the new field, or click **Create and continue** to add the new field and continue adding other fields.

**NOTE**

To edit a field, select the field row and use the **general properties** on the right side of the screen.

## CHAPTER 4. CREATING DRL RULES IN BUSINESS CENTRAL

You can create and manage DRL rules for your project in Business Central. In each DRL rule file, you define rule conditions, actions, and other components related to the rule, based on the data objects you create or import in the package.

### Procedure

1. In Business Central, go to **Menu → Design → Projects** and click the project name.
2. Click **Add Asset → DRL file**.
3. Enter an informative **DRL file** name and select the appropriate **Package**. The package that you specify must be the same package where the required data objects have been assigned or will be assigned.  
You can also select **Show declared DSL sentences** if any domain specific language (DSL) assets have been defined in your project. These DSL assets will then become usable objects for conditions and actions that you define in the DRL designer.
4. Click **Ok** to create the rule asset.  
The new DRL file is now listed in the **DRL** panel of the **Project Explorer**, or in the **DSL** panel if you selected the **Show declared DSL sentences** option. The package to which you assigned this DRL file is listed at the top of the file.
5. In the **Fact types** list in the left panel of the DRL designer, confirm that all data objects and data object fields (expand each) required for your rules are listed. If not, you can either import relevant data objects from other packages by using **import** statements in the DRL file, or [create data objects](#) within your package.
6. After all data objects are in place, return to the **Model** tab of the DRL designer and define the DRL file with any of the following components:

### Components of a DRL file

```
package //automatic

import

function //optional

query //optional

declare //optional

rule

rule

...
```

- **package:** (automatic) This was defined for you when you created the DRL file and selected the package.

- **import:** Use this to identify the data objects from either this package or another package that you want to use in the DRL file. Specify the package and data object in the format **package.name.object.name**, one import per line.

### Importing data objects

```
import mortgages.mortgages.LoanApplication;
```

- **function:** (optional) Use this to include a function to be used by rules in the DRL file. Functions put semantic code in your rule source file. Functions are especially useful if an action (**then**) part of a rule is used repeatedly and only the parameters differ for each rule. Above the rules in the DRL file, you can declare the function or import a static method as a function, and then use the function by name in an action (**then**) part of the rule.

### Declaring and using a function with a rule (option 1)

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

rule "Using a function"
when
    eval( true )
then
    System.out.println( hello( "James" ) );
end
```

### Importing and using the function with a rule (option 2)

```
import function my.package.applicant.hello;

rule "Using a function"
when
    eval( true )
then
    System.out.println( hello( "James" ) );
end
```

- **query:** (optional) Use this to search the process engine for facts related to the rules in the DRL file. Queries search for a set of defined conditions and do not require **when** or **then** specifications. Query names are global to the KIE base and therefore must be unique among all other rule queries in the project. To return the results of a query, construct a traditional **QueryResults** definition using **ksession.getQueryResults("name")**, where **"name"** is the query name. This returns a list of query results, which enable you to retrieve the objects that matched the query. Define the query and query results parameters above the rules in the DRL file.

### Query and query results for people under the age of 21, with a rule

```
query "people under the age of 21"
    person : Person( age < 21 )
end

QueryResults results = ksession.getQueryResults( "people under the age of 21" );
```

```
System.out.println( "we have " + results.size() + " people under the age of 21" );
```

```
rule "Underage"
when
    application : LoanApplication( )
    Applicant( age < 21 )
then
    application.setApproved( false );
    application.setExplanation( "Underage" );
end
```

- **declare:** (optional) Use this to declare a new fact type to be used by rules in the DRL file. The default fact type in the **java.lang** package of Red Hat Process Automation Manager is **Object**, but you can declare other types in DRL files as needed. Declaring fact types in DRL files enables you to define a new fact model directly in the process engine, without creating models in a lower-level language like Java.

### Declaring and using a new fact type

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

rule "Using a declared type"
when
    $p : Person( name == "James" )
then // Insert Mark, who is a customer of James.
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
end
```

- **rule:** Use this to define each rule in the DRL file. Rules consist of a rule name in the format **rule "name"**, followed by optional attributes that define rule behavior (such as **salience** or **no-loop**), followed by **when** and **then** definitions. The same rule name cannot be used more than once in the same package. The **when** part of the rule contains the conditions that must be met to execute an action. For example, if a bank requires loan applicants to have over 21 years of age, then the **when** condition for an **Underage** rule would be **Applicant( age < 21 )**. The **then** part of the rule contains the actions to be performed when the conditional part of the rule has been met. For example, when the loan applicant is under 21 years old, the **then** action would be **setApproved( false )**, declining the loan because the applicant is under age. Conditions (**when**) and actions (**then**) consist of a series of stated fact patterns with optional constraints, bindings, and other supported DRL elements, based on the available data objects in the package. These patterns determine how defined objects are affected by the rule.

### Rule for loan application age limit

```
rule "Underage"
    salience 15
    dialect "mvel"
when
    application : LoanApplication( )
```

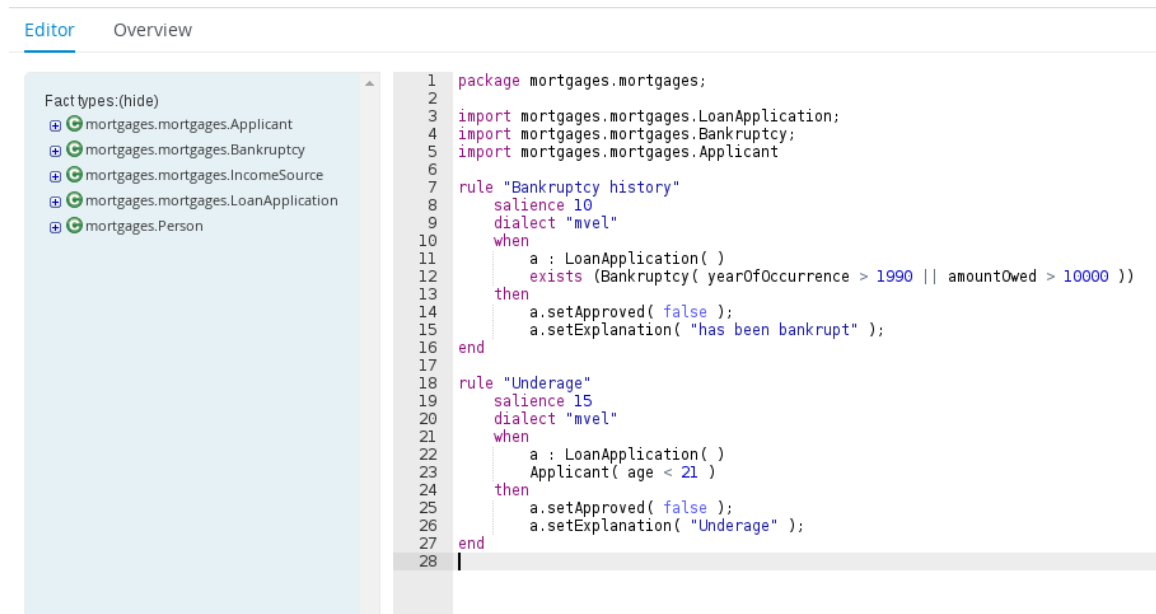
```

Applicant( age < 21 )
then
  application.setApproved( false );
  application.setExplanation( "Underage" );
end

```

At minimum, each DRL file must specify the **package**, **import**, and **rule** components. All other components are optional.

**Figure 4.1. Sample DRL file with required components and optional rule attributes**



- After you define all components of the rule, click **Validate** in the upper-right toolbar of the DRL designer to validate the DRL file. If the file validation fails, address any problems described in the error message, review all syntax and components in the DRL file, and try again to validate the file until the file passes.

- Click **Save** in the DRL designer to save your work.

For more details about adding conditions to DRL rules, see [Section 4.1, "Adding WHEN conditions in DRL rules"](#).

For more details about adding actions to DRL rules, see [Section 4.2, "Adding THEN actions in DRL rules"](#).

## 4.1. ADDING WHEN CONDITIONS IN DRL RULES

The **when** part of the rule contains the conditions that must be met to execute an action. For example, if a bank requires loan applicants to have over 21 years of age, then the **when** condition of an **Underage** rule would be **Applicant( age < 21 )**. Conditions consist of a series of stated patterns and constraints, with optional bindings and other supported DRL elements, based on the available data objects in the package.

### Prerequisites

- The **package** is defined at the top of the DRL file. This should have been done for you when you created the file.

- The **import** list of data objects used in the rule is defined below the **package** line of the DRL file. Data objects can be from this package or from another package in Business Central.
- The **rule** name is defined in the format **rule "name"** below the **package**, **import**, and other lines that apply to the entire DRL file. The same rule name cannot be used more than once in the same package. Optional rule attributes (such as **salience** or **no-loop**) that define rule behavior are below the rule name, before the **when** section.

## Procedure

1. In the DRL designer, enter **when** within the rule to begin adding condition statements. The **when** section consists of zero or more fact patterns that define conditions for the rule. If the **when** section is empty, then actions in the **then** section are executed every time a **fireAllRules()** call is made in the process engine. This is useful if you want to use rules to set up the process engine state.

### Rule without conditions

```
rule "bootstrap"
  when // empty

  then // actions to be executed once
    insert( new Applicant() );
  end

// The above rule is internally rewritten as:

rule "bootstrap"
  when
    eval( true )
  then
    insert( new Applicant() );
  end
```

2. Enter a pattern for the first condition to be met, with optional constraints, bindings, and other supported DRL elements. A basic pattern format is **patternBinding : patternType ( constraints )**. Patterns are based on the available data objects in the package and define the conditions to be met in order to trigger actions in the **then** section.

- **Simple pattern:** A simple pattern with no constraints matches against a fact of the given type. For example, the following condition is only that the applicant exists.

```
when
  Applicant( )
```

- **Pattern with constraints:** A pattern with constraints matches against a fact of the given type and the additional restrictions in parentheses that are true or false. For example, the following condition is that the applicant is under the age of 21.

```
when
  Applicant( age < 21 )
```

- **Pattern with binding:** A binding on a pattern is a shorthand reference that other components of the rule can use to refer back to the defined pattern. For example, the following binding **a** on **LoanApplication** is used in a related action for underage applicants.

```

when
  a : LoanApplication( )
  Applicant( age < 21 )
then
  a.setApproved( false );
  a.setExplanation( "Underage" )

```

3. Continue defining all condition patterns that apply to this rule. The following are some of the keyword options for defining DRL conditions:

- **and**: Use this to group conditional components into a logical conjunction. Infix and prefix **and** are supported. By default, all listed conditions or actions are combined with **and** when no conjunction is specified.

```

a : LoanApplication( ) and Applicant( age < 21 )

a : LoanApplication( )
and Applicant( age < 21 )

a : LoanApplication( )
Applicant( age < 21 )

// All of the above are the same.

```

- **or**: Use this to group conditional components into a logical disjunction. Infix and prefix **or** are supported.

```

Bankruptcy( amountOwed == 100000 ) or IncomeSource( amount == 20000 )

Bankruptcy( amountOwed == 100000 )
or IncomeSource( amount == 20000 )

```

- **exists**: Use this to specify facts and constraints that must exist. Note that this does not mean that a fact exists, but that a fact must exist. This option is triggered on only the first match, not subsequent matches.

```

exists (Bankruptcy( yearOfOccurrence > 1990 || amountOwed > 10000 ))

```

- **not**: Use this to specify facts and constraints that must not exist.

```

not (Applicant( age < 21 ))

```

- **forall**: Use this to set up a construct where all facts that match the first pattern match all the remaining patterns.

```

forall( app : Applicant( age < 21 )
       Applicant( this == app, status = 'underage' ) )

```

- **from**: Use this to specify a source for data to be matched by the conditional pattern.

```

Applicant( ApplicantAddress : address )
Address( zipcode == "23920W" ) from ApplicantAddress

```

- **entry-point:** Use this to define an **Entry Point** corresponding to a data source for the pattern. Typically used with **from**.

```
Applicant( ) from entry-point "LoanApplication"
```

- **collect:** Use this to define a collection of objects that the construct can use as part of the condition. In the example, all pending applications in the process engine for each given mortgage are grouped in **ArrayLists**. If three or more pending applications are found, the rule is executed.

```
m : Mortgage()
a : ArrayList( size >= 3 )
  from collect( LoanApplication( Mortgage == m, status == 'pending' ) )
```

- **accumulate:** Use this to iterate over a collection of objects, execute custom actions for each of the elements, and return one or more result objects (if the constraints evaluate to **true**). This option is a more flexible and powerful form of **collect**. Use the format **accumulate( <source pattern>; <functions> [<constraints>] )**. In the example, **min**, **max**, and **average** are accumulate functions that calculate the minimum, maximum and average temperature values over all the readings for each sensor. Other supported functions include **count**, **sum**, **variance**, **standardDeviation**, **collectList**, and **collectSet**.

```
s : Sensor()
accumulate( Reading( sensor == s, temp : temperature );
  min : min( temp ),
  max : max( temp ),
  avg : average( temp );
  min < 20, avg > 70 )
```



## ADVANCED DRL OPTIONS

These are examples of basic keyword options and pattern constructs for defining conditions. For more advanced DRL options and syntax supported in the DRL designer, see the [Drools Documentation](#) online.

4. After you define all condition components of the rule, click **Validate** in the upper-right toolbar of the DRL designer to validate the DRL file. If the file validation fails, address any problems described in the error message, review all syntax and components in the DRL file, and try again to validate the file until the file passes.
5. Click **Save** in the DRL designer to save your work.

## 4.2. ADDING THEN ACTIONS IN DRL RULES

The **then** part of the rule contains the actions to be performed when the conditional part of the rule has been met. For example, when a loan applicant is under 21 years old, the **then** action of an **Underage** rule would be **setApproved( false )**, declining the loan because the applicant is under age. Actions execute consequences based on the rule conditions and on available data objects in the package.

### Prerequisites

- The **package** is defined at the top of the DRL file. This should have been done for you when you created the file.

- The **import** list of data objects used in the rule is defined below the **package** line of the DRL file. Data objects can be from this package or from another package in Business Central.
- The **rule** name is defined in the format **rule "name"** below the **package**, **import**, and other lines that apply to the entire DRL file. The same rule name cannot be used more than once in the same package. Optional rule attributes (such as **salience** or **no-loop**) that define rule behavior are below the rule name, before the **when** section.

## Procedure

1. In the DRL designer, enter **then** after the **when** section of the rule to begin adding action statements.
2. Enter one or more actions to be executed on fact patterns based on the conditions for the rule. The following are some of the keyword options for defining DRL actions:
  - **and**: Use this to group action components into a logical conjunction. Infix and prefix **and** are supported. By default, all listed conditions or actions are combined with **and** when no conjunction is specified.

```
application.setApproved ( false ) and application.setExplanation( "has been bankrupt" );

application.setApproved ( false );
and application.setExplanation( "has been bankrupt" );

application.setApproved ( false );
application.setExplanation( "has been bankrupt" );

// All of the above are the same.
```

- **set**: Use this to set the value of a field.

```
application.setApproved ( false );
application.setExplanation( "has been bankrupt" );
```

- **modify**: Use this to specify fields to be modified for a fact and to notify the process engine of the change.

```
modify( LoanApplication ) {
    setAmount( 100 )
}
```

- **update**: Use this to specify fields and the entire related fact to be modified and to notify the process engine of the change. After a fact has changed, you must call **update** before changing another fact that might be affected by the updated values. The **modify** keyword avoids this added step.

```
update( LoanApplication ) {
    setAmount( 100 )
}
```

- **delete**: Use this to remove an object from the process engine. The keyword **retract** is also supported in the DRL designer and executes the same action, but **delete** is preferred for consistency with the keyword **insert**.

–

```
delete( LoanApplication );
```

- **insert:** Use this to insert a **new** fact and define resulting fields and values as needed for the fact.

```
insert( new Applicant() );
```

- **insertLogical:** Use this to insert a **new** fact logically into the process engine and define resulting fields and values as needed for the fact. The Red Hat Process Automation Manager process engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts have to be retracted explicitly. After logical insertions, facts are automatically retracted when the conditions that originally asserted the facts are no longer true.

```
insertLogical( new Applicant() );
```



## ADVANCED DRL OPTIONS

These are examples of basic keyword options and pattern constructs for defining actions. For more advanced DRL options and syntax supported in the DRL designer, see the [Drools Documentation](#) online.

3. After you define all action components of the rule, click **Validate** in the upper-right toolbar of the DRL designer to validate the DRL file. If the file validation fails, address any problems described in the error message, review all syntax and components in the DRL file, and try again to validate the file until the file passes.
4. Click **Save** in the DRL designer to save your work.


### 4.2.1. Rule attributes

Rule attributes are additional specifications that you can add to business rules to modify rule behavior. The following table lists the names and supported values of the attributes that you can assign to rules:

Table 4.1. Rule attributes

Attribute	Value
<b>salience</b>	An integer defining the priority of the rule. Rules with a higher salience value are given higher priority when ordered in the activation queue.  Example: <b>salience 10</b>
<b>enabled</b>	A Boolean value. When the option is selected, the rule is enabled. When the option is not selected, the rule is disabled.  Example: <b>enabled true</b>
<b>date-effective</b>	A string containing a date and time definition. The rule can be activated only if the current date and time is after a <b>date-effective</b> attribute.  Example: <b>date-effective "4-Sep-2018"</b>

Attribute	Value
<b>date-expires</b>	<p>A string containing a date and time definition. The rule cannot be activated if the current date and time is after the <b>date-expires</b> attribute.</p> <p>Example: <b>date-expires "4-Oct-2018"</b></p>
<b>no-loop</b>	<p>A Boolean value. When the option is selected, the rule cannot be reactivated (looped) if a consequence of the rule re-triggers a previously met condition. When the condition is not selected, the rule can be looped in these circumstances.</p> <p>Example: <b>no-loop true</b></p>
<b>agenda-group</b>	<p>A string identifying an agenda group to which you want to assign the rule. Agenda groups allow you to partition the agenda to provide more execution control over groups of rules. Only rules in an agenda group that has acquired a focus are able to be activated.</p> <p>Example: <b>agenda-group "GroupName"</b></p>
<b>activation-group</b>	<p>A string identifying an activation (or XOR) group to which you want to assign the rule. In activation groups, only one rule can be activated. The first rule to fire will cancel all pending activations of all rules in the activation group.</p> <p>Example: <b>activation-group "GroupName"</b></p>
<b>duration</b>	<p>A long integer value defining the duration of time in milliseconds after which the rule can be activated, if the rule conditions are still met.</p> <p>Example: <b>duration 10000</b></p>
<b>timer</b>	<p>A string identifying either <b>int</b> (interval) or <b>cron</b> timer definition for scheduling the rule.</p> <p>Example: <b>timer "**/5 * * * *"</b> (every 5 minutes)</p>
<b>calendar</b>	<p>A Quartz calendar definition for scheduling the rule.</p> <p>Example: <b>calendars "** * 0-7,18-23 ? * *"</b> (exclude non-business hours)</p>
<b>auto-focus</b>	<p>A Boolean value, applicable only to rules within agenda groups. When the option is selected, the next time the rule is activated, a focus is automatically given to the agenda group to which the rule is assigned.</p> <p>Example: <b>auto-focus true</b></p>

Attribute	Value
<b>lock-on-active</b>	<p>A Boolean value, applicable only to rules within rule flow groups or agenda groups. When the option is selected, the next time the ruleflow group for the rule becomes active or the agenda group for the rule receives a focus, the rule cannot be activated again until the ruleflow group is no longer active or the agenda group loses the focus. This is a stronger version of the <b>no-loop</b> attribute, because the activation of a matching rule is discarded regardless of the origin of the update (not only by the rule itself). This attribute 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.</p> <p>Example: <b>lock-on-active true</b></p>
<b>ruleflow-group</b>	<p>A string identifying a rule flow group. In rule flow groups, rules can fire only when the group is activated by the associated rule flow.</p> <p>Example: <b>ruleflow-group "GroupName"</b></p>
<b>dialect</b>	<p>A string identifying either <b>JAVA</b> or <b>MVEL</b> as the language to be used for code expressions in the rule. By default, the rule uses the dialect specified at the package level. Any dialect specified here overrides the package dialect setting for the rule.</p> <p>Example: <b>dialect "JAVA"</b></p> <div>  <div> <p><b>NOTE</b></p> <p>When you use Red Hat Process Automation Manager without the executable model, the <b>dialect "JAVA"</b> rule consequences support only Java 5 syntax. For more information about executable models, see <a href="#">Packaging and deploying a Red Hat Process Automation Manager project</a>.</p> </div> </div>

## CHAPTER 5. EXECUTING RULES

After you create rules in Business Central, you can build and deploy your project and execute rules locally or on Process Server to test your rules.

### Procedure

1. In Business Central, go to **Menu → Design → Projects** and click the project name.
2. In the upper-right corner, click **Build** and then **Deploy** to build the project and deploy it to Process Server. If the build fails, address any problems described in the **Alerts** panel at the bottom of the screen. For more information about deploying projects, see [Packaging and deploying a Red Hat Process Automation Manager project](#).
3. Open the **pom.xml** file of your client application and add the following dependencies, if not added already:
  - **kie-ci**: Enables your client application to load Business Central project data locally using **ReleaseId**
  - **kie-server-client**: Enables your client application to interact remotely with assets on Process Server
  - **slf4j**: (Optional) Enables your client application to use Simple Logging Facade for Java (SLF4J) to return debug logging information after you interact with Process Server

Example dependencies for Red Hat Process Automation Manager 7.1 in a client application **pom.xml** file:

```
// For local execution:
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.11.0.Final-redhat-00002</version>
</dependency>

// For remote execution on Process Server:
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.11.0.Final-redhat-00002</version>
</dependency>

// For debug logging (optional):
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.25</version>
</dependency>
```

For available versions of these artifacts, search the group ID and artifact ID in the [Nexus Repository Manager](#) online.

**NOTE**

Instead of specifying a Red Hat Process Automation Manager **<version>** for individual dependencies, consider adding the Red Hat Business Automation bill of materials (BOM) dependency to your project **pom.xml** file. The Red Hat Business Automation BOM applies to both Red Hat Decision Manager and Red Hat Process Automation Manager. When you add the BOM files, the correct versions of transitive dependencies from the provided Maven repositories are included in the project.

Example BOM dependency:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.1.0.GA-redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see [What is the mapping between Red Hat Process Automation Manager and the Maven library version?](#).

4. Ensure that the dependencies for artifacts containing model classes are defined in the client application **pom.xml** file exactly as they appear in the **pom.xml** file of the deployed project. If dependencies for model classes differ between the client application and your projects, execution errors can occur.

To access the project **pom.xml** file in Business Central, select any existing asset in the project and then in the **Project Explorer** menu on the left side of the screen, click the **Customize View** gear icon and select **Repository View → pom.xml**.

For example, the following is a **Person** class dependency as it appears in both the client and deployed project **pom.xml** files:

```
<dependency>
  <groupId>com.sample</groupId>
  <artifactId>Person</artifactId>
  <version>1.0.0</version>
</dependency>
```

5. If you added the **slf4j** dependency to the client application **pom.xml** file for debug logging, create a **simplelogger.properties** file on the relevant classpath (for example, in **src/main/resources/META-INF** in Maven) with the following content:

```
org.slf4j.simpleLogger.defaultLogLevel=debug
```

6. In your client application, create a **.java** main class containing the necessary imports and a **main()** method to load the KIE base, insert facts, and execute the rules.

For example, a **Person** object in a project contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person. The following **Wage** rule in a project calculates the wage and hourly rate values and displays a message based on the result:

```

package com.sample;

import com.sample.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

```

To test this rule locally outside of Process Server (if desired), configure the **.java** class to import KIE services, a KIE container, and a KIE session, and then use the **main()** method to fire all rules against a defined fact model:

### Executing rules locally

```

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

public class RulesTest {

    public static final void main(String[] args) {
        try {
            // Identify the project in the local repository:
            ReleaseId rid = new ReleaseId();
            rid.setGroupId("com.myspace");
            rid.setArtifactId("MyProject");
            rid.setVersion("1.0.0");

            // Load the KIE base:
            KieServices ks = KieServices.Factory.get();
            KieContainer kContainer = ks.newKieContainer(rid);
            KieSession kSession = kContainer.newKieSession();

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

            // Insert the person into the session:
            kSession.insert(p);

            // Fire all rules:
            kSession.fireAllRules();
            kSession.dispose();
        }
    }
}

```

```

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

```

To test this rule on Process Server, configure the **.java** class with the imports and rule execution information similarly to the local example, and additionally specify KIE services configuration and KIE services client details:

### Executing rules on Process Server

```

package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;

import com.sample.Person;

public class RulesTest {

    private static final String containerName = "testProject";
    private static final String sessionName = "myStatelessSession";

    public static final void main(String[] args) {
        try {
            // Define KIE services configuration and client:
            Set<Class<?>> allClasses = new HashSet<Class<?>>();
            allClasses.add(Person.class);
            String serverUrl = "http://$HOST:$PORT/kie-server/services/rest/server";
            String username = "$USERNAME";
            String password = "$PASSWORD";
            KieServicesConfiguration config =
                KieServicesFactory.newRestConfiguration(serverUrl,
                                                         username,
                                                         password);
            config.setMarshallingFormat(MarshallingFormat.JAXB);
            config.addExtraClasses(allClasses);
            KieServicesClient kieServicesClient =
                KieServicesFactory.newKieServicesClient(config);

```

```

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

// Insert Person into the session:
KieCommands kieCommands = KieServices.Factory.get().getCommands();
List<Command> commandList = new ArrayList<Command>();
commandList.add(kieCommands.newInsert(p, "personReturnId"));

// Fire all rules:
commandList.add(kieCommands.newFireAllRules("numberOfFiredRules"));
BatchExecutionCommand batch = kieCommands.newBatchExecution(commandList,
sessionName);

// Use rule services client to send request:
RuleServicesClient ruleClient =
kieServicesClient.getServicesClient(RuleServicesClient.class);
ServiceResponse<ExecutionResults> executeResponse =
ruleClient.executeCommandsWithResults(containerName, batch);
System.out.println("number of fired rules:" +
executeResponse.getResult().getValue("numberOfFiredRules"));
}

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

```

7. Run the configured **.java** class from your project directory. You can run the file in your development platform (such as Red Hat JBoss Developer Studio) or in the command line. Example Maven execution (within project directory):

```
mvn clean install exec:java -Dexec.mainClass="com.sample.app.RulesTest"
```

Example Java execution (within project directory)

```
javac -classpath ".$DEPENDENCIES/*:." RulesTest.java
java -classpath ".$DEPENDENCIES/*:." RulesTest
```

8. Review the rule execution status in the command line and in the server log. If any rules do not execute as expected, review the configured rules in the project and the main class configuration to validate the data provided.

## CHAPTER 6. OTHER METHODS FOR CREATING AND EXECUTING DRL RULES

As an alternative to creating and managing DRL rules within the Business Central interface, you can create DRL rule files in external standalone projects using Red Hat Developer Studio, Java objects, or Maven archetypes. These standalone projects can then be integrated as knowledge JAR (KJAR) dependencies in existing Red Hat Process Automation Manager projects in Business Central. The DRL files in your standalone project must contain at minimum the required **package** specification, **import** lists, and **rule** definitions. Any other DRL components, such as global variables and functions, are optional. All data objects related to a DRL rule must be included with your standalone DRL project or deployment.

You can also use executable rule models in your Maven or Java projects to provide a Java-based representation of a rule set for execution at build time. The executable model is a more efficient alternative to the standard asset packaging in Red Hat Process Automation Manager and enables KIE containers and KIE bases to be created more quickly, especially when you have large lists of DRL (Drools Rule Language) files and other Red Hat Process Automation Manager assets.

### 6.1. CREATING AND EXECUTING DRL RULES IN RED HAT JBOSS DEVELOPER STUDIO

You can use Red Hat JBoss Developer Studio to create DRL files with rules and integrate the files with your Red Hat Process Automation Manager decision service. This method of creating DRL rules is helpful if you already use Red Hat Developer Studio for your decision service and want to continue with the same work flow. If you do not already use this method, then the Business Central interface of Red Hat Process Automation Manager is recommended for creating DRL files and other rule assets.

#### Prerequisite

Red Hat JBoss Developer Studio has been installed from the [Red Hat Customer Portal](#).

#### Procedure

1. In the Red Hat JBoss Developer Studio, click **File → New → Project**.
2. In the **New Project** window that opens, select **Drools → Drools Project** and click **Next**.
3. Click the second icon to **Create a project and populate it with some example files to help you get started quickly**. Click **Next**.
4. Enter a **Project name** and select the **Maven** radio button as the project building option. The GAV values are generated automatically. You can update these values as needed for your project:
  - **Group ID: com.sample**
  - **Artifact ID: my-project**
  - **Version: 1.0.0-SNAPSHOT**
5. Click **Finish** to create the project.  
This configuration sets up a basic project structure, class path, and sample rules. The following is an overview of the project structure:

my-project

```

|-- src/main/java
|   |-- com.sample
|       |-- DecisionTableTest.java
|       |-- DroolsTest.java
|       |-- ProcessTest.java
|
|-- src/main/resources
|   |-- dtables
|       |-- Sample.xls
|   |-- process
|       |-- sample.bpmn
|   |-- rules
|       |-- Sample.drl
|       |-- META-INF
|
|-- JRE System Library
|
|-- Maven Dependencies
|
|-- Drools Library
|
|-- src
|
|-- target
|
|-- pom.xml

```

Notice the following elements:

- A **Sample.drl** rule file in the **src/main/resources** directory, containing an example **Hello World** and **GoodBye** rules.
- A **DroolsTest.java** file under the **src/main/java** directory in the **com.sample** package. The **DroolsTest** class can be used to execute the **Sample.drl** rule.
- The **Drools Library** directory, which acts as a custom class path containing JAR files necessary for execution.

You can edit the existing **Sample.drl** file and **DroolsTest.java** files with new configurations as needed, or create new rule and object files. In this procedure, you are creating a new rule and new Java objects.

6. Create a Java object on which the rule or rules will operate.

In this example, a **Person.java** file is created in **my-project/src/main/java/com.sample**. The **Person** class contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person:

```

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;
}
}

```

7. Click **File** → **Save** to save the file.
8. Create a rule file in **.drl** format in **my-project/src/main/resources/rules**. The DRL file must contain at minimum a package specification, an import list of data objects to be used by the rule or rules, and one or more rules with **when** conditions and **then** actions.  
The following **Wage.drl** file contains a **Wage** rule that imports the **Person** class, calculates the wage and hourly rate values, and displays a message based on the result:

```

package com.sample;

import com.sample.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

```

9. Click **File** → **Save** to save the file.

10. Create a main class and save it to the same directory as the Java object that you created. The main class will load the KIE base and execute rules.



#### NOTE

You can also add the **main()** method and **Person** class within a single Java object file, similar to the **DroolsTest.java** sample file.

11. In the main class, add the required **import** statements to import KIE services, a KIE container, and a KIE session. Then load the KIE base, insert facts, and execute the rule from the **main()** method that passes the fact model to the rule.

In this example, a **RulesTest.java** file is created in **my-project/src/main/java/com.sample** with the required imports and **main()** method:

```
package com.sample;

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

public class RulesTest {
    public static final void main(String[] args) {
        try {
            // Load the KIE base:
            KieServices ks = KieServices.Factory.get();
            KieContainer kContainer = ks.getKieClasspathContainer();
            KieSession kSession = kContainer.newKieSession();

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

            // Insert the person into the session:
            kSession.insert(p);

            // Fire all rules:
            kSession.fireAllRules();
            kSession.dispose();
        }

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

12. Click **File → Save** to save the file.
13. After you create and save all DRL assets in your project, right-click your project folder and select **Run As → Java Application** to build the project. If the project build fails, address any problems described in the **Problems** tab of the lower window in Developer Studio, and try again to validate the project until the project builds.



## IF THE RUN AS → JAVA APPLICATION OPTION IS NOT AVAILABLE

If **Java Application** is not an option when you right-click your project and select **Run As**, then go to **Run As → Run Configurations**, right-click **Java Application**, and click **New**. Then in the **Main** tab, browse for and select your **Project** and the associated **Main class**. Click **Apply** and then click **Run** to test the project. The next time you right-click your project folder, the **Java Application** option will appear.

To integrate the new rule assets with an existing project in Red Hat Process Automation Manager, you can compile the new project as a knowledge JAR (KJAR) and add it as a dependency in the **pom.xml** file of the project in Business Central.

## 6.2. CREATING AND EXECUTING DRL RULES USING JAVA

You can use Java objects to create DRL files with rules and integrate the objects with your Red Hat Process Automation Manager decision service. This method of creating DRL rules is helpful if you already use external Java objects for your decision service and want to continue with the same work flow. If you do not already use this method, then the Business Central interface of Red Hat Process Automation Manager is recommended for creating DRL files and other rule assets.

### Procedure

1. Create a Java object on which the rule or rules will operate.

In this example, a **Person.java** file is created in a directory **my-project**. The **Person** class contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person:

```
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 a rule file in **.drl** format under the **my-project** directory. The DRL file must contain at minimum a package specification (if applicable), an import list of data objects to be used by the rule or rules, and one or more rules with **when** conditions and **then** actions.

The following **Wage.drl** file contains a **Wage** rule that calculates the wage and hourly rate values and displays a message based on the result:

```

package com.sample;

import com.sample.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

```

3. Create a main class and save it to the same directory as the Java object that you created. The main class will load the KIE base and execute rules.
4. In the main class, add the required **import** statements to import KIE services, a KIE container, and a KIE session. Then load the KIE base, insert facts, and execute the rule from the **main()** method that passes the fact model to the rule.

In this example, a **RulesTest.java** file is created in **my-project** with the required imports and **main()** method:

```

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

public class RulesTest {
    public static final void main(String[] args) {
        try {
            // Load the KIE base:
            KieServices ks = KieServices.Factory.get();
            KieContainer kContainer = ks.getKieClasspathContainer();
            KieSession kSession = kContainer.newKieSession();

            // Set up the fact model:
            Person p = new Person();

```

```

        p.setWage(12);
        p.setFirstName("Tom");
        p.setLastName("Summers");
        p.setHourlyRate(10);

        // Insert the person into the session:
        kSession.insert(p);

        // Fire all rules:
        kSession.fireAllRules();
        kSession.dispose();
    }

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

```

5. Download the **Red Hat Process Automation Manager 7.1.0 Source Distribution** ZIP file from the [Red Hat Customer Portal](#) and extract it under **my-project/pam-engine-jars/**.
6. In the **my-project/META-INF** directory, create a **kmodule.xml** metadata file with the following content:

```

<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>

```

This **kmodule.xml** file is a KIE module descriptor that selects resources to KIE bases and configures sessions. This file enables you to define and configure one or more KIE bases, and to include DRL files from specific **packages** in a specific KIE base. You can also create one or more KIE sessions from each KIE base.

The following example shows a more advanced **kmodule.xml** file:

```

<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
    <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg1">
        <ksession name="KSession1_1" type="stateful" default="true" />
        <ksession name="KSession1_2" type="stateful" default="true" 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="stateless" default="true" clockType="realtime">
            <fileLogger file="debugInfo" 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" />
        </listeners>
    </kbase>
</kmodule>

```

```

    <agendaEventListener type="org.domain.SecondAgendaListener" />
    <processEventListener type="org.domain.ProcessListener" />
  </listeners>
</ksession>
</kbase>
</kmodule>

```

This example defines two KIE bases. Two KIE sessions are instantiated from the **KBase1** KIE base, and one KIE session from **KBase2**. The KIE session from **KBase2** is a **stateless** KIE session, which means that data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations. Specific **packages** of rule assets are included with both KIE bases. When you specify packages in this way, you must organize your DRL files in a folder structure that reflects the specified packages.

- After you create and save all DRL assets in your Java object, navigate to the **my-project** directory in the command line and run the following command to build your Java files. Replace **RulesTest.java** with the name of your Java main class.

```
javac -classpath "./pam-engine-jars/*:." RulesTest.java
```

If the build fails, address any problems described in the command line error messages and try again to validate the Java object until the object passes.

- After your Java files build successfully, run the following command to execute the rules locally. Replace **RulesTest** with the prefix of your Java main class.

```
java -classpath "./pam-engine-jars/*:." RulesTest
```

- Review the rules to ensure that they executed properly, and address any needed changes in the Java files.

To integrate the new rule assets with an existing project in Red Hat Process Automation Manager, you can compile the new Java project as a knowledge JAR (KJAR) and add it as a dependency in the **pom.xml** file of the project in Business Central.

## 6.3. CREATING AND EXECUTING DRL RULES USING MAVEN

You can use Maven archetypes to create DRL files with rules and integrate the archetypes with your Red Hat Process Automation Manager decision service. This method of creating DRL rules is helpful if you already use external Maven archetypes for your decision service and want to continue with the same work flow. If you do not already use this method, then the Business Central interface of Red Hat Process Automation Manager is recommended for creating DRL files and other rule assets.

### Procedure

- Navigate to a directory where you want to create a Maven archetype and run the following command:

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

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

```
my-app
```

```

|-- pom.xml
|-- src
|   |-- main
|   |   |-- java
|   |   |   |-- com
|   |   |   |   |-- sample
|   |   |   |   |   |-- app
|   |   |   |   |   |   |-- App.java
|   |-- test
|   |   |-- java
|   |   |   |-- com
|   |   |   |   |-- sample
|   |   |   |   |   |-- app
|   |   |   |   |   |   |-- AppTest.java

```

The **my-app** directory contains the following key components:

- A **src/main** directory for storing the application sources
  - A **src/test** directory for storing the test sources
  - A **pom.xml** file with the project configuration
2. Create a Java object on which the rule or rules will operate within the Maven archetype. In this example, a **Person.java** file is created in the directory **my-app/src/main/java/com/sample/app**. The **Person** class contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person:

```

package com.sample.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 a rule file in **.drl** format in **my-app/src/main/resources/rules**. The DRL file must contain at minimum a package specification, an import list of data objects to be used by the rule or rules, and one or more rules with **when** conditions and **then** actions.

The following **Wage.drl** file contains a **Wage** rule that imports the **Person** class, calculates the wage and hourly rate values, and displays a message based on the result:

```

package com.sample.app;

import com.sample.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

```

4. In the **my-app/src/main/resources/META-INF** directory, create a **kmodule.xml** metadata file with the following content:

```

<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>

```

This **kmodule.xml** file is a KIE module descriptor that selects resources to KIE bases and configures sessions. This file enables you to define and configure one or more KIE bases, and to include DRL files from specific **packages** in a specific KIE base. You can also create one or more KIE sessions from each KIE base.

The following example shows a more advanced **kmodule.xml** file:

```

<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
    <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg1">
        <ksession name="KSession1_1" type="stateful" default="true" />
    </kbase>
</kmodule>

```

```

    <ksession name="KSession1_2" type="stateful" default="true" beliefSystem="jtsms" />
  </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="stateless" default="true" clockType="realtime">
      <fileLogger file="debugInfo" 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>

```

This example defines two KIE bases. Two KIE sessions are instantiated from the **KBase1** KIE base, and one KIE session from **KBase2**. The KIE session from **KBase2** is a **stateless** KIE session, which means that data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations. Specific **packages** of rule assets are included with both KIE bases. When you specify packages in this way, you must organize your DRL files in a folder structure that reflects the specified packages.

5. In the **my-app/pom.xml** configuration file, specify the libraries that your application requires. Provide the Red Hat Process Automation Manager dependencies as well as the **group ID**, **artifact ID**, and **version** (GAV) of your application.

```

<?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.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0.0</version>
  <repositories>
    <repository>
      <id>jboss-ga-repository</id>
      <url>http://maven.repository.redhat.com/ga/</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>drools-compiler</artifactId>
      <version>VERSION</version>
    </dependency>
    <dependency>
      <groupId>org.kie</groupId>
      <artifactId>kie-api</artifactId>
      <version>VERSION</version>
    </dependency>
  </dependencies>

```

```

</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
</dependencies>
</project>

```

For information about Maven dependencies and the BOM (Bill of Materials) in Red Hat Process Automation Manager, see [What is the mapping between Red Hat Process Automation Manager and Maven library version?](#).

6. Use the **testApp** method in **my-app/src/test/java/com/sample/app/AppTest.java** to test the rule. The **AppTest.java** file is created by Maven by default.
7. In the **AppTest.java** file, add the required **import** statements to import KIE services, a KIE container, and a KIE session. Then load the KIE base, insert facts, and execute the rule from the **testApp()** method that passes the fact model to the rule.

```

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

public void testApp() {

    // Load the KIE base:
    KieServices ks = KieServices.Factory.get();
    KieContainer kContainer = ks.getKieClasspathContainer();
    KieSession kSession = kContainer.newKieSession();

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

    // Insert the person into the session:
    kSession.insert(p);

    // Fire all rules:
    kSession.fireAllRules();
    kSession.dispose();
}

```

8. After you create and save all DRL assets in your Maven archetype, navigate to the **my-app** directory in the command line and run the following command to build your files:

```
mvn clean install
```

If the build fails, address any problems described in the command line error messages and try again to validate the files until the build is successful.

9. After your files build successfully, run the following command to execute the rules locally. Replace **com.sample.app** with your package name.

```
mvn exec:java -Dexec.mainClass="com.sample.app"
```

10. Review the rules to ensure that they executed properly, and address any needed changes in the files.

To integrate the new rule assets with an existing project in Red Hat Process Automation Manager, you can compile the new Maven project as a knowledge JAR (KJAR) and add it as a dependency in the **pom.xml** file of the project in Business Central.

## 6.4. EXECUTABLE RULE MODELS

Executable rule models are embeddable models that provide a Java-based representation of a rule set for execution at build time. The executable model is a more efficient alternative to the standard asset packaging in Red Hat Process Automation Manager and enables KIE containers and KIE bases to be created more quickly, especially when you have large lists of DRL (Drools Rule Language) files and other Red Hat Process Automation Manager assets. The model is low level and enables you to provide all necessary execution information, such as the lambda expressions for the index evaluation.

Executable rule models provide the following specific advantages for your projects:

- **Compile time:** Traditionally, a packaged Red Hat Process Automation Manager project (KJAR) contains a list of DRL files and other Red Hat Process Automation Manager artifacts that define the rule base together with some pre-generated classes implementing the constraints and the consequences. Those DRL files must be parsed and compiled when the KJAR is downloaded from the Maven repository and installed in a KIE container. This process can be slow, especially for large rule sets. With an executable model, you can package within the project KJAR the Java classes that implement the executable model of the project rule base and re-create the KIE container and its KIE bases out of it in a much faster way. In Maven projects, you use the **kie-maven-plugin** to automatically generate the executable model sources from the DRL files during the compilation process.
- **Run time:** In an executable model, all constraints are defined as Java lambda expressions. The same lambda expressions are also used for constraints evaluation, so you no longer need to use **mvel** expressions for interpreted evaluation nor the just-in-time (JIT) process to transform the **mvel**-based constraints into bytecode. This creates a quicker and more efficient run time.
- **Development time:** An executable model enables you to develop and experiment with new features of the process engine without needing to encode elements directly in the DRL format or modify the DRL parser to support them.



## NOTE

For query definitions in executable rule models, you can use up to 10 arguments only.

For variables within rule consequences in executable rule models, you can use up to 12 bound variables only (including the built-in **drools** variable). For example, the following rule consequence uses more than 12 bound variables and creates a compilation error:

```
...
then
    $input.setNo13Count(functions.sumOf(new Object[]{$no1Count_1, $no2Count_1,
    $no3Count_1, ..., $no13Count_1}).intValue());
    $input.getFirings().add("fired");
    update($input);
```

### 6.4.1. Embedding an executable rule model in a Maven project

You can embed an executable rule model in your Maven project to compile your rule assets more efficiently at build time.

#### Prerequisite

You have a Mavenized project that contains Red Hat Process Automation Manager business assets.

#### Procedure

1. In the **pom.xml** file of your Maven project, ensure that the packaging type is set to **kjar** and add the **kie-maven-plugin** build component:

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

The **kjar** packaging type activates the **kie-maven-plugin** component to validate and pre-compile artifact resources. The **<version>** is the Maven artifact version for Red Hat Process Automation Manager currently used in your project (for example, 7.11.0.Final-redhat-00002). These settings are required to properly package the Maven project.

**NOTE**

Instead of specifying a Red Hat Process Automation Manager **<version>** for individual dependencies, consider adding the Red Hat Business Automation bill of materials (BOM) dependency to your project **pom.xml** file. The Red Hat Business Automation BOM applies to both Red Hat Decision Manager and Red Hat Process Automation Manager. When you add the BOM files, the correct versions of transitive dependencies from the provided Maven repositories are included in the project.

Example BOM dependency:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.1.0.GA-redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see [What is the mapping between RHPAM product and maven library version?](#).

2. Add the following dependencies to the **pom.xml** file to enable rule assets to be built from an executable model:

- **drools-canonical-model:** Enables an executable canonical representation of a rule set model that is independent from Red Hat Process Automation Manager
- **drools-model-compiler:** Compiles the executable model into Red Hat Process Automation Manager internal data structures so that it can be executed by the process engine

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-canonical-model</artifactId>
  <version>${rhpam.version}</version>
</dependency>

<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-model-compiler</artifactId>
  <version>${rhpam.version}</version>
</dependency>
```

3. In a command terminal, navigate to your Maven project directory and run the following command to build the project from an executable model:

```
mvn clean install -DgenerateModel=<VALUE>
```

The **-DgenerateModel=<VALUE>** property enables the project to be built as a model-based KJAR instead of a DRL-based KJAR.

Replace **<VALUE>** with one of three values:

- **YES:** Generates the executable model corresponding to the DRL files in the original project and excludes the DRL files from the generated KJAR.
- **WITHDRL:** Generates the executable model corresponding to the DRL files in the original project and also adds the DRL files to the generated KJAR for documentation purposes (the KIE base is built from the executable model regardless).
- **NO:** Does not generate the executable model.

Example build command:

```
mvn clean install -DgenerateModel=YES
```

For more information about packaging Maven projects, see [Packaging and deploying a Red Hat Process Automation Manager project](#).

### 6.4.2. Embedding an executable rule model in a Java application

You can embed an executable rule model programmatically within your Java application to compile your rule assets more efficiently at build time.

#### Prerequisite

You have a Java application that contains Red Hat Process Automation Manager business assets.

#### Procedure

1. Add the following dependencies to the relevant classpath for your Java project:
  - **drools-canonical-model:** Enables an executable canonical representation of a rule set model that is independent from Red Hat Process Automation Manager
  - **drools-model-compiler:** Compiles the executable model into Red Hat Process Automation Manager internal data structures so that it can be executed by the process engine

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-canonical-model</artifactId>
  <version>${rhpm.version}</version>
</dependency>

<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-model-compiler</artifactId>
  <version>${rhpm.version}</version>
</dependency>
```

The **<version>** is the Maven artifact version for Red Hat Process Automation Manager currently used in your project (for example, 7.11.0.Final-redhat-00002).



## NOTE

Instead of specifying a Red Hat Process Automation Manager **<version>** for individual dependencies, consider adding the Red Hat Business Automation bill of materials (BOM) dependency to your project **pom.xml** file. The Red Hat Business Automation BOM applies to both Red Hat Decision Manager and Red Hat Process Automation Manager. When you add the BOM files, the correct versions of transitive dependencies from the provided Maven repositories are included in the project.

Example BOM dependency:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.1.0.GA-redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see [What is the mapping between RHPAM product and maven library version?](#).

2. Add rule assets to the KIE virtual file system **KieFileSystem** and use **KieBuilder** with **buildAll(ExecutableModelProject.class)** specified to build the assets from an executable model:

```
import org.kie.api.KieServices;
import org.kie.api.builder.KieFileSystem;
import org.kie.api.builder.KieBuilder;

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

KieBuilder kieBuilder = ks.newKieBuilder( kfs );
// Build from an executable model
kieBuilder.buildAll( ExecutableModelProject.class )
assertEquals(0, kieBuilder.getResults().getMessages(Message.Level.ERROR).size());
```

After **KieFileSystem** is built from the executable model, the resulting **KieSession** uses constraints based on lambda expressions instead of less-efficient **mvel** expressions. If **buildAll()** contains no arguments, the project is built in the standard method without an executable model.

As a more manual alternative to using **KieFileSystem** for creating executable models, you can define a **Model** with a fluent API and create a **KieBase** from it:

```
Model model = new ModelImpl().addRule( rule );
KieBase kieBase = KieBaseBuilder.createKieBaseFromModel( model );
```

For more information about packaging projects programmatically within a Java application, see [Packaging and deploying a Red Hat Process Automation Manager project](#) .

## CHAPTER 7. NEXT STEPS

- *Testing a decision service using test scenarios*
- *Packaging and deploying a Red Hat Process Automation Manager project*

## APPENDIX A. VERSIONING INFORMATION

Documentation last updated on Wednesday, June 10, 2020.