Red Hat Process Automation Manager 7.5

Designing a decision service using guided rules

Last Updated: 2020-05-22

Red Hat Process Automation Manager 7.5 Designing a decision service using guided rules

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

Abstract

This document describes how to design a decision service using guided rules in Red Hat Process Automation Manager 7.5.

Table of Contents

PREFACE

As a business analyst or business rules developer, you can define business rules using the guided rules designer in Business Central. These guided rules are compiled into Drools Rule Language (DRL) and form the core of the decision service for your project.

NOTE

You can also design your decision service using Decision Model and Notation (DMN) models instead of rule-based or table-based assets. For information about DMN support in Red Hat Process Automation Manager 7.5, see the following resources:

- *Getting started with decision services* (step-by-step tutorial with a DMN decision service example)
- *Designing a decision service using DMN models* (overview of DMN support and capabilities in Red Hat Process Automation Manager)

Prerequisites

- The space and project for the guided rules have been created in Business Central. Each asset is associated with a project assigned to a space. For details, see *Getting started with decision services*.

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

Red Hat Process Automation Manager supports several assets that you can use to define business decisions for your decision service. Each decision-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 the main decision-authoring assets supported in Red Hat Process Automation Manager projects to help you decide or confirm the best method for defining decisions in your decision service.

Table 1.1. Decision-authoring assets supported in Red Hat Process Automation Manager

Asset	Highlights	Authoring tools	Documentation
Decision Model and Notation (DMN) models	● Are decision models based on a notation standard defined by the Object Management Group (OMG) ● Use graphical decision requirements diagrams (DRDs) with one or more decision requirements graphs (DRGs) to trace business decision flows ● Use an XML schema that allows the DMN models to be shared between DMN-compliant platforms ● Support Friendly Enough Expression Language (FEEL) to define decision logic in DMN decision tables and other DMN boxed expressions ● Can be integrated efficiently with Business Process Model and Notation (BPMN) process models ● Are optimal for creating comprehensive, illustrative, and stable decision flows	Business Central or other DMN-compliant editor	*Designing a decision service using DMN models*

Asset	Highlights	Authoring tools	Documentation
Guided decision tables	• Are tables of rules that you create in a UI-based table designer in Business Central • Are a wizard-led alternative to spreadsheet decision tables • Provide fields and options for acceptable input • Support template keys and values for creating rule templates • Support hit policies, real-time validation, and other additional features not supported in other assets • Are optimal for creating rules in a controlled tabular format to minimize compilation errors	Business Central	*Designing a decision service using guided decision tables*
Spreadsheet decision tables	• Are XLS or XLSX spreadsheet decision tables that you can upload into Business Central • Support template keys and values for creating rule templates • Are optimal for creating rules in decision tables already managed outside of Business Central • Have strict syntax requirements for rules to be compiled properly when uploaded	Spreadsheet editor	*Designing a decision service using spreadsheet decision tables*
Guided rules	• Are individual rules that you create in a UI-based rule designer in Business Central • Provide fields and options for acceptable input • Are optimal for creating single rules in a controlled format to minimize compilation errors	Business Central	*Designing a decision service using guided rules*

Asset	Highlights	Authoring tools	Documentation
Guided rule templates	• Are reusable rule structures that you create in a UI-based template designer in Business Central • Provide fields and options for acceptable input • Support template keys and values for creating rule templates (fundamental to the purpose of this asset) • Are optimal for creating many rules with the same rule structure but with different defined field values	Business Central	*Designing a decision service using guided rule templates*
DRL rules	• Are individual rules that you define directly in **.drl** text files • Provide the most flexibility for defining rules and other technicalities of rule behavior • Can be created in certain standalone environments and integrated with Red Hat Process Automation Manager • Are optimal for creating rules that require advanced DRL options • Have strict syntax requirements for rules to be compiled properly	Business Central or integrated development environment (IDE)	*Designing a decision service using DRL rules*

Asset	Highlights	Authoring tools	Documentation
Predictive Model Markup Language (PMML) models	• Are predictive data-analytic models based on a notation standard defined by the Data Mining Group (DMG) • Use an XML schema that allows the PMML models to be shared between PMML-compliant platforms • Support Regression, Scorecard, Tree, Mining, and other model types • Can be included with a standalone Red Hat Process Automation Manager project or imported into a project in Business Central • Are optimal for incorporating predictive data into decision services in Red Hat Process Automation Manager	PMML or XML editor	*Designing a decision service using PMML models*

CHAPTER 2. GUIDED RULES

Guided rules are business rules that you create in a UI-based guided rules designer in Business Central that leads you through the rule-creation process. The guided rules designer provides fields and options for acceptable input based on the data objects for the rule being defined. The guided rules that you define are compiled into Drools Rule Language (DRL) rules as with all other rule assets.

All data objects related to a guided rule must be in the same project package as the guided rule. Assets in the same package are imported by default. After you create the necessary data objects and the guided rule, you can use the **Data Objects** tab of the guided rules designer to verify that all required data objects are listed or to import other existing data objects by adding a **New item**.

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 asset.

Procedure

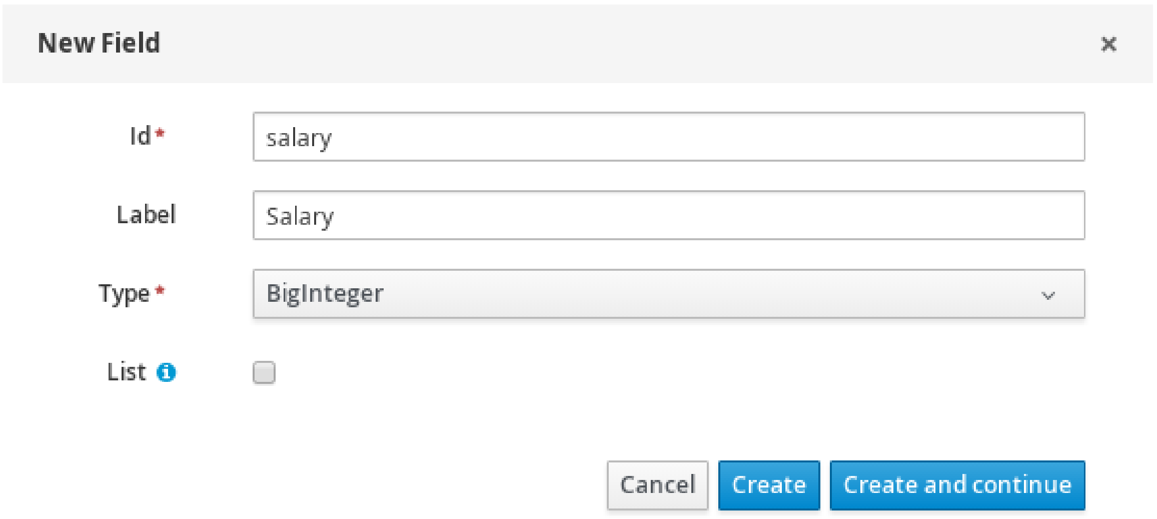
1. In Business Central, go to **Menu → Design → Projects** and click the project name.
2. Click **Add Asset → Data Object**.
3. Enter a unique **Data Object** name and select the **Package** where you want the data object to be available for other rule assets. Data objects with the same name cannot exist in the same package. In the specified DRL file, you can import a data object from any package.

   IMPORTING DATA OBJECTS FROM OTHER PACKAGES

   You can import an existing data object from another package directly into the asset designers like guided rules or guided decision table designers. 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:** (Optional) 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 ☐

7. 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.

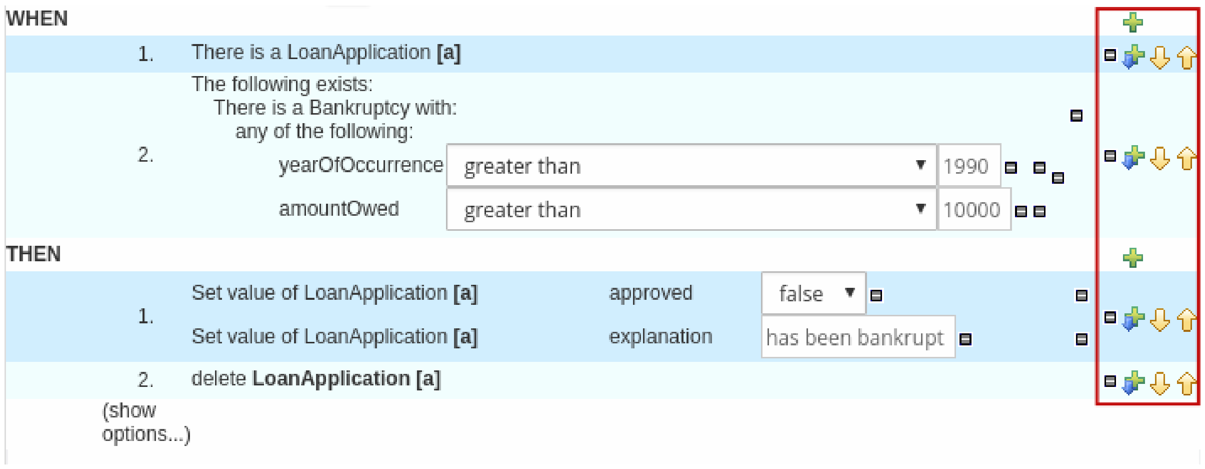
CHAPTER 4. CREATING GUIDED RULES

Guided rules enable you to define business rules in a structured format, based on the data objects associated with the rules. You can create and define guided rules individually for your project.

Procedure

1. In Business Central, go to **Menu** → **Design** → **Projects** and click the project name.
2. Click **Add Asset** → **Guided Rule**.
3. Enter an informative **Guided Rule** 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 guided rules designer.
4. Click **Ok** to create the rule asset.
   The new guided rule is now listed in the **Guided Rules** panel of the **Project Explorer**, or in the **Guided Rules (with DSL)** panel if you selected the **Show declared DSL sentences** option.
5. Click the **Data Objects** tab and confirm that all data objects required for your rules are listed. If not, click **New item** to import data objects from other packages, or create data objects within your package.
6. After all data objects are in place, return to the **Model** tab of the guided rules designer and use the buttons on the right side of the window to add and define the **WHEN** (condition) and **THEN** (action) sections of the rule, based on the available data objects.

Figure 4.1. The guided rules designer



WHEN

1. There is a LoanApplication [a]
2. The following exists:
   There is a Bankruptcy with:
   any of the following:
   yearOfOccurrence greater than 1990
   amountOwed greater than 10000

THEN

1. Set value of LoanApplication [a] approved false
   Set value of LoanApplication [a] explanation has been bankrupt
2. delete LoanApplication [a]

(show options...)

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 **Age | less than | 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 set **approved** to **false**, declining the loan because the applicant is under age.

You can also specify exceptions for more complex rules, such as if a bank may approve of an under-aged applicant when a guarantor is involved. In that case, you would create or import a **guarantor** data object and then add the field to the guided rule.

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

8. Click **Save** in the guided rules designer to save your work.

4.1. ADDING WHEN CONDITIONS IN GUIDED 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 **Age | less than | 21**. You can set simple or complex conditions to determine how and when your rules are applied.

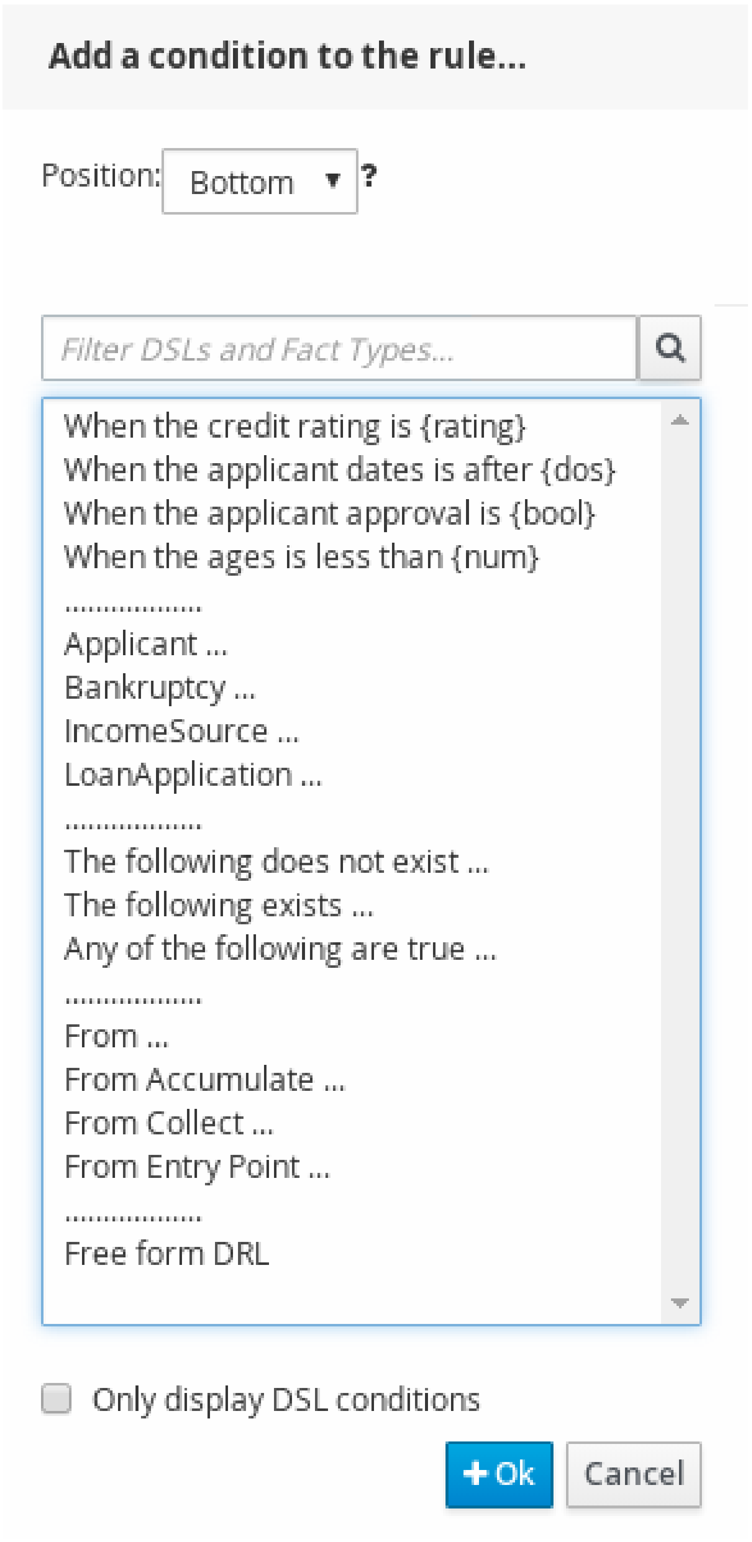
Prerequisites

- All data objects required for your rules have been created or imported and are listed in the **Data Objects** tab of the guided rules designer.

Procedure

1. In the guided rules designer, click the plus icon ( ) on the right side of the **WHEN** section. The **Add a condition to the rule**window with the available condition elements opens.

Figure 4.2. Add a condition to the rule


Add a condition to the rule...
Position: Bottom ?
Filter DSLs and Fact Types...
When the credit rating is {rating}
When the applicant dates is after {dos}
When the applicant approval is {bool}
When the ages is less than {num}
....
Applicant ...
Bankruptcy ...
IncomeSource ...
LoanApplication ...
....
The following does not exist ...
The following exists ...
Any of the following are true ...
....
From ...
From Accumulate ...
From Collect ...
From Entry Point ...
....
Free form DRL
Only display DSL conditions
+ Ok
Cancel

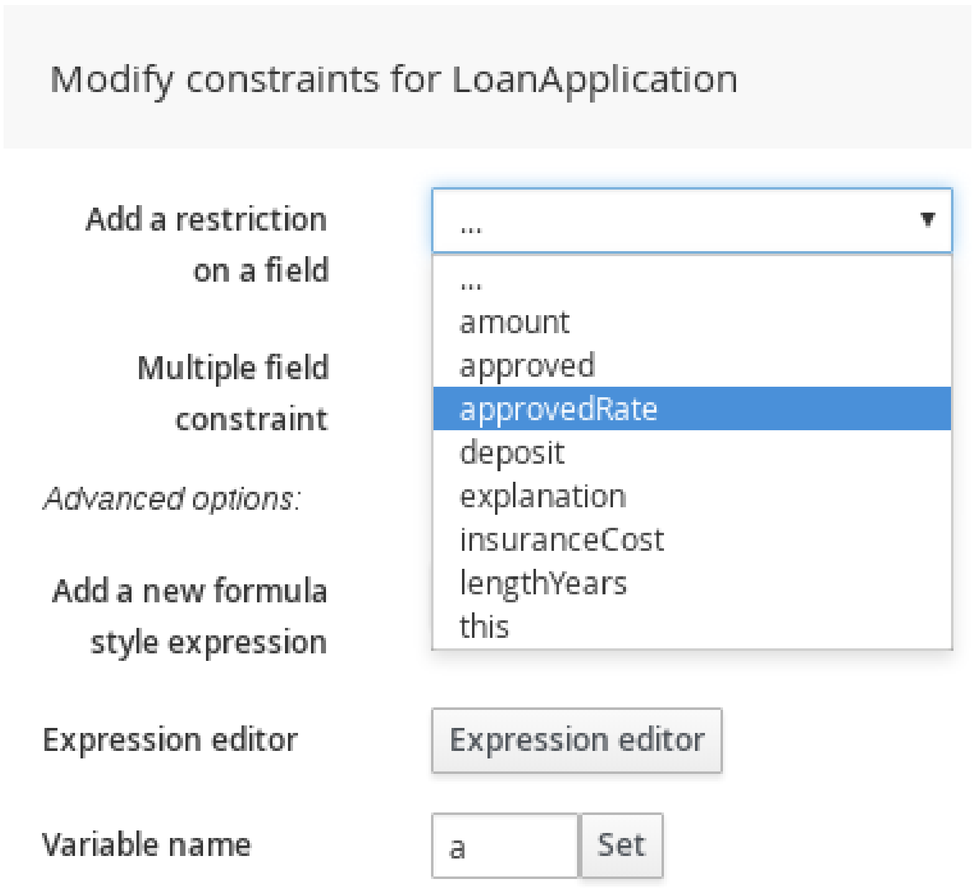
The list includes the data objects from the **Data Objects** tab of the guided rules designer, any DSL objects defined for the package (if you selected **Show declared DSL sentences** when you created this guided rule), and the following standard options:

- **The following does not exist:** Use this to specify facts and constraints that must not exist.
- **The following exists:** Use this to specify facts and constraints that must exist. This option is triggered on only the first match, not subsequent matches.
- **Any of the following are true:** Use this to list any facts or constraints that must be true.

- **From:** Use this to define a **From** conditional element for the rule.
- **From Accumulate:** Use this to define an **Accumulate** conditional element for the rule.
- **From Collect:** Use this to define a **Collect** conditional element for the rule.
- **From Entry Point:** Use this to define an **Entry Point** for the pattern.
- **Free form DRL:** Use this to insert a free-form DRL field where you can define condition elements freely, without the guided rules designer.

2. Choose a condition element (for example, **LoanApplication**) and click **Ok**.
3. Click the condition element in the guided rules designer and use the **Modify constraints for LoanApplication** window to add a restriction on a field, apply multiple field constraints, add a new formula style expression, apply an expression editor, or set a variable name.

Figure 4.3. Modify a condition



NOTE

A variable name enables you to identify a fact or field in other constructs within the guided rule. For example, you could set the variable of **LoanApplication** to **a** and then reference **a** in a separate **Bankruptcy** constraint that specifies which application the bankruptcy is based on.

```
a : LoanApplication()
Bankruptcy( application == a ).
```

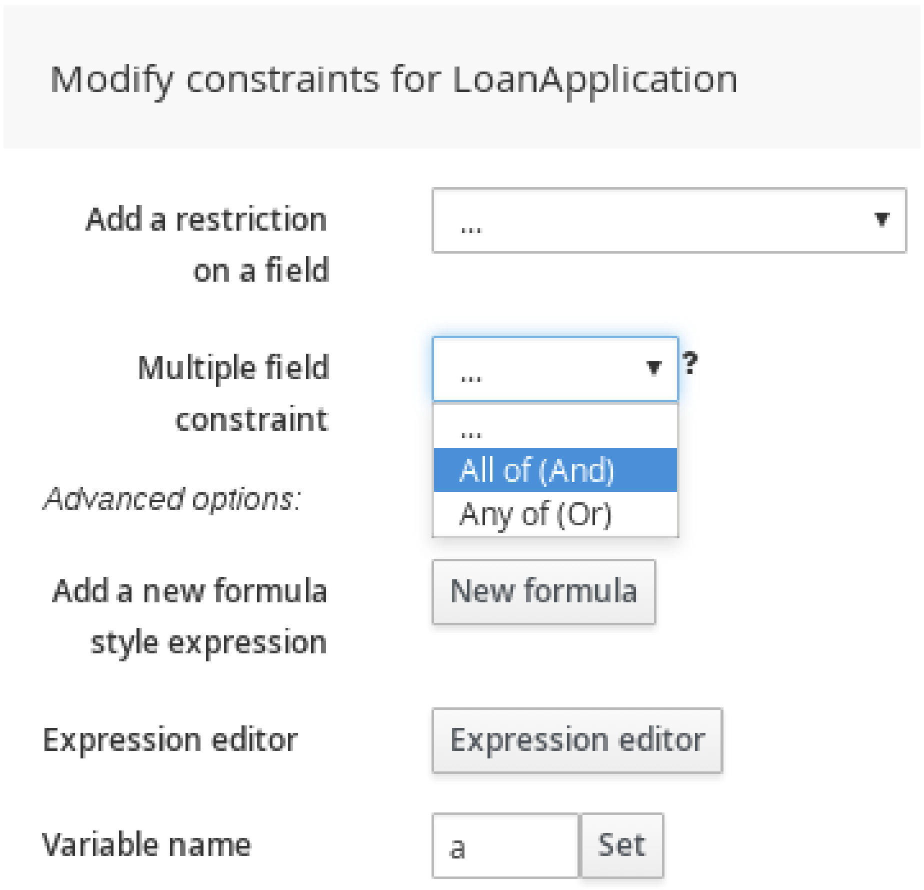
After you select a constraint, the window closes automatically.

4. Choose an operator for the restriction (for example, **greater than**) from the drop-down menu next to the added restriction.

5. Click the edit icon (✎) to define the field value. The field value can be a literal value, a formula, or a full MVEL expression.

6. To apply multiple field constraints, click the condition and in the **Modify constraints for LoanApplication** window, select **All of(And)** or **Any of(Or)** from the **Multiple field constraint** drop-down menu.

Figure 4.4. Add multiple field constraints


Modify constraints for LoanApplication
Add a restriction on a field
...
Multiple field constraint
...
?
...
All of (And)
Any of (Or)
Advanced options:
Add a new formula style expression
New formula
Expression editor
Expression editor
Variable name
a
Set

7. Click the constraint in the guided rules designer and further define the field value.

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

9. Click **Save** in the guided rules designer to save your work.

4.2. ADDING THEN ACTIONS IN GUIDED RULES

The **THEN** part of the rule contains the actions to be performed when the **WHEN** condition of the rule has been met. For example, when a loan applicant is under 21 years old, the **THEN** action might set **approved** to **false**, declining the loan because the applicant is under age. You can set simple or complex actions to determine how and when your rules are applied.

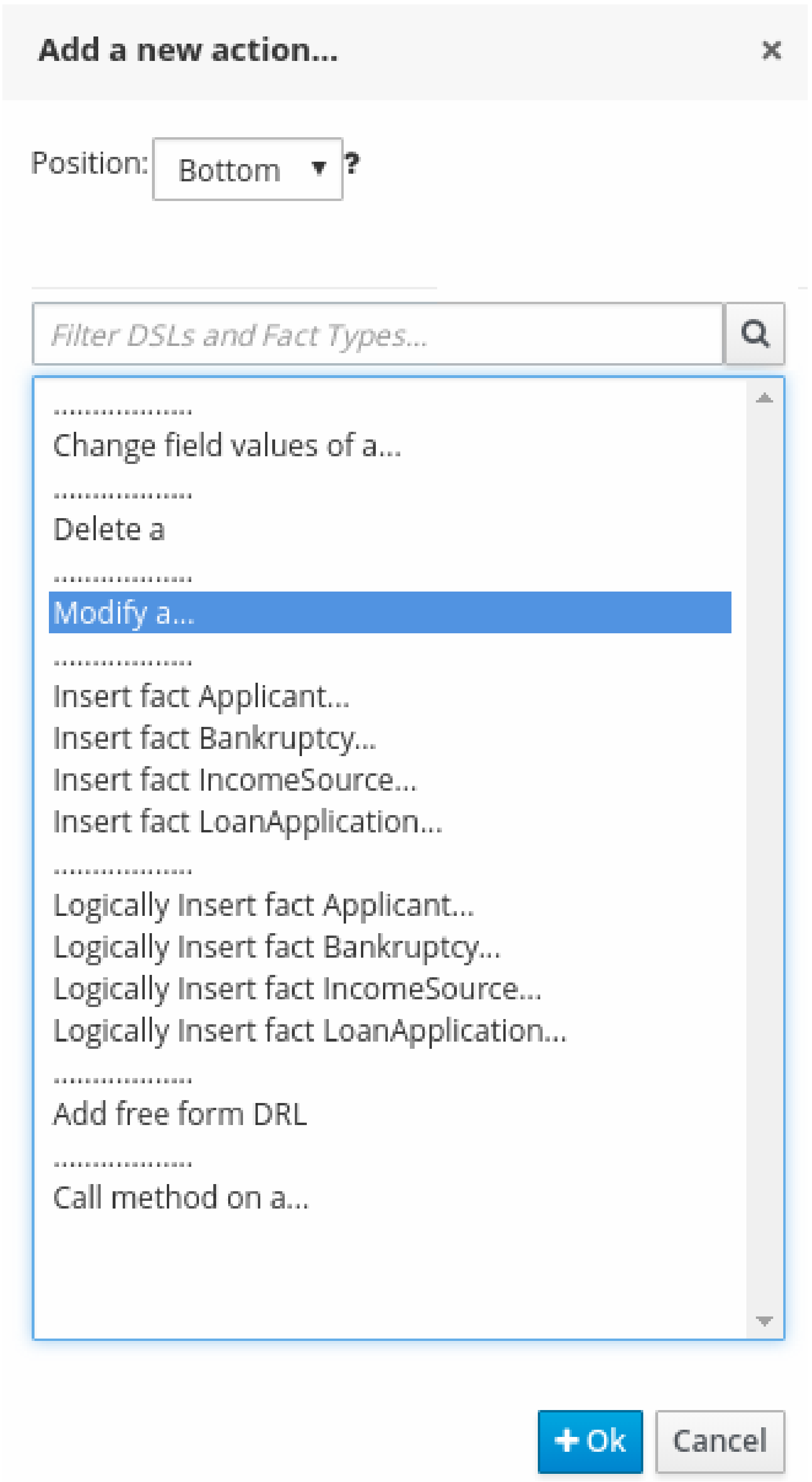
Prerequisites

- All data objects required for your rules have been created or imported and are listed in the **Data Objects** tab of the guided rules designer.

Procedure

1. In the guided rules designer, click the plus icon (✚) on the right side of the **THEN** section.
   The **Add a new action** window with the available action elements opens.

Figure 4.5. Add a new action to the rule



Add a new action...

Position: Bottom

Filter DSLs and Fact Types...

Change field values of a...

Delete a

Modify a...

Insert fact Applicant...
Insert fact Bankruptcy...
Insert fact IncomeSource...
Insert fact LoanApplication...

Logically Insert fact Applicant...
Logically Insert fact Bankruptcy...
Logically Insert fact IncomeSource...
Logically Insert fact LoanApplication...

Add free form DRL

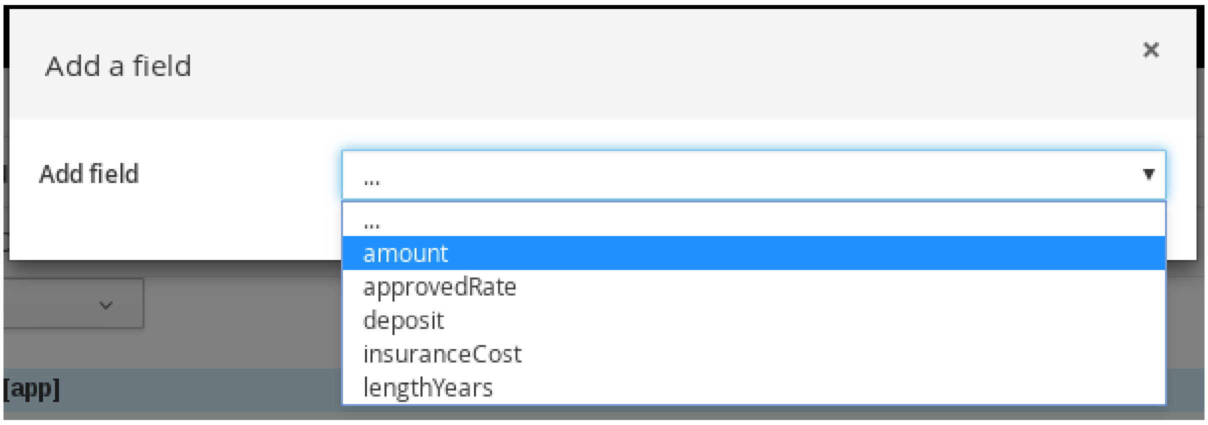
Call method on a...

The list includes insertion and modification options based on the data objects in the **Data Objects** tab of the guided rules designer, and on any DSL objects defined for the package (if you selected **Show declared DSL sentences** when you created this guided rule):

- **Change field values of:** Use this to set the value of fields on a fact (such as **LoanApplication**) without notifying the decision engine of the change.
- **Delete:** Use this to delete a fact.
- **Modify:** Use this to specify fields to be modified for a fact and to notify the decision engine of the change.
- **Insert fact:** Use this to insert a fact and define resulting fields and values for the fact.
- **Logically Insert fact:** Use this to insert a fact logically into the decision engine and define resulting fields and values for the fact. The decision 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.
- **Add free form DRL:**Use this to insert a free-form DRL field where you can define condition elements freely, without the guided rules designer.
- **Call method on:** Use this to invoke a method from another fact.

2. Choose an action element (for example, **Modify**) and click **Ok**.
3. Click the action element in the guided rules designer and use the **Add a field** window to select a field.

Figure 4.6. Add a field



After you select a field, the window closes automatically.

4. Click the edit icon (✎) to define the field value. The field value can be a literal value or a formula.
5. After you define all action components of the rule, click **Validate** in the upper-right toolbar of the guided rules designer to validate the guided rule actions. If the rule validation fails, address any problems described in the error message, review all components in the rule, and try again to validate the rule until the rule passes.
6. Click **Save** in the guided rules designer to save your work.

4.3. DEFINING ENUMERATIONS FOR DROP-DOWN LISTS IN RULE ASSETS

Enumeration definitions in Business Central determine the possible values of fields for conditions or actions in guided rules, guided rule templates, and guided decision tables. An enumeration definition contains a **fact.field** mapping to a list of supported values that are displayed as a drop-down list in the relevant field of a rule asset. When a user selects a field that is based on the same fact and field as the enumeration definition, the drop-down list of defined values is displayed.

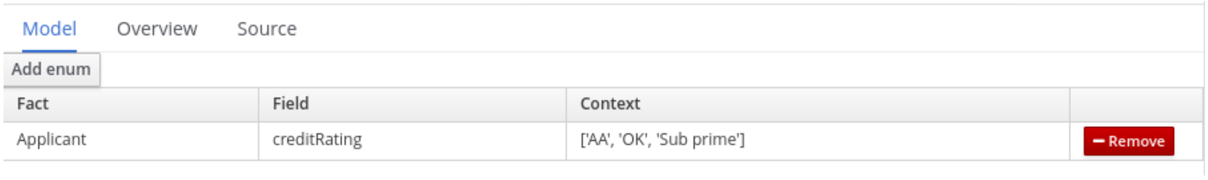
You can define enumerations in Business Central or in the DRL source for your Red Hat Process Automation Manager project.

Procedure

1. In Business Central, go to **Menu** → **Design** → **Projects** and click the project name.
2. Click **Add Asset** → **Enumeration**.
3. Enter an informative **Enumeration** name and select the appropriate **Package**. The package that you specify must be the same package where the required data objects and relevant rule assets have been assigned or will be assigned.
4. Click **Ok** to create the enumeration.
   The new enumeration is now listed in the **Enumeration Definitions** panel of the **Project Explorer**.
5. In the **Model** tab of the enumerations designer, click **Add enum** and define the following values for the enumeration:
   - **Fact**: Specify an existing data object within the same package of your project with which you want to associate this enumeration. Open the **Data Objects** panel in the **Project Explorer** to view the available data objects, or create the relevant data object as a new asset if needed.
   - **Field**: Specify an existing field identifier that you defined as part of the data object that you selected for the **Fact**. Open the **Data Objects** panel in the **Project Explorer** to select the relevant data object and view the list of available **Identifier** options. You can create the relevant identifier for the data object if needed.
   - **Context**: Specify a list of values in the format **['string1','string2','string3']** or **[integer1,integer2,integer3]** that you want to map to the **Fact** and **Field** definitions. These values will be displayed as a drop-down list for the relevant field of the rule asset.

   For example, the following enumeration defines the drop-down values for applicant credit rating in a loan application decision service:

   Figure 4.7. Example enumeration for applicant credit rating in Business Central



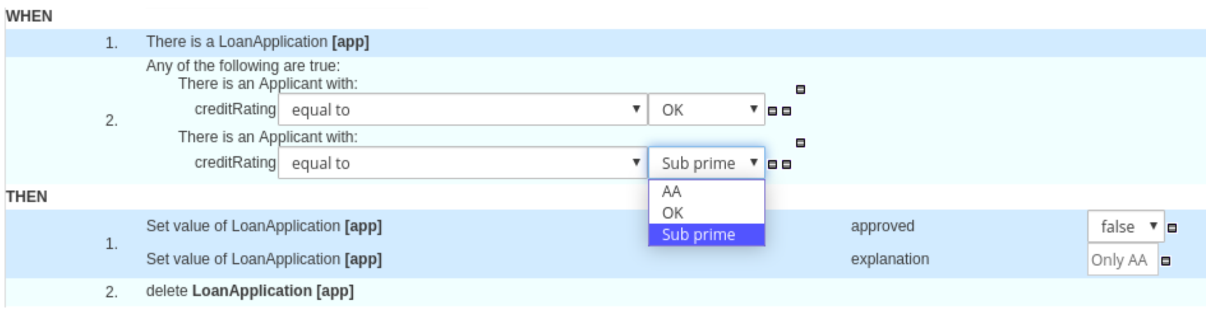
Fact	Field	Context	
Applicant	creditRating	['AA', 'OK', 'Sub prime']	Remove

Example enumeration for applicant credit rating in the DRL source

```
'Applicant.creditRating' : ['AA', 'OK', 'Sub prime']
```

In this example, for any guided rule, guided rule template, or guided decision table that is in the same package of the project and that uses the **Applicant** data object and the **creditRating** field, the configured values are available as drop-down options:

Figure 4.8. Example enumeration drop-down options in a guided rule or guided rule template



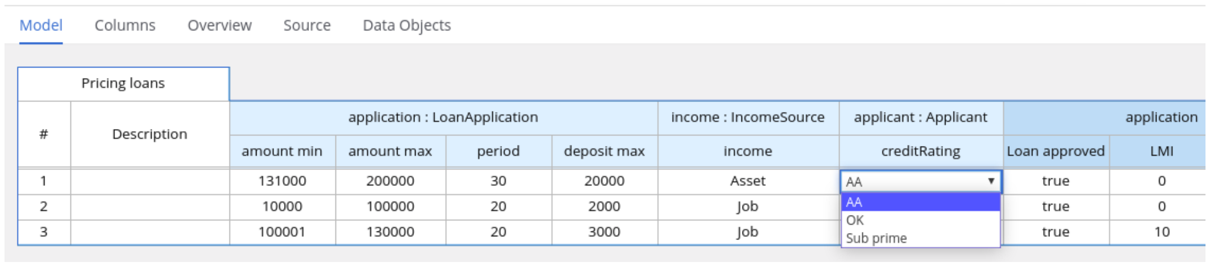
WHEN

1. There is a LoanApplication **[app]**
2. Any of the following are true:
   There is an Applicant with:
   creditRating | equal to | OK
   There is an Applicant with:
   creditRating | equal to | Sub prime
   AA
   OK
   Sub prime

THEN

1. Set value of LoanApplication **[app]** approved false
   Set value of LoanApplication **[app]** explanation Only AA
2. delete **LoanApplication [app]**

Figure 4.9. Example enumeration drop-down options in a guided decision table



Model Columns Overview Source Data Objects

Pricing loans

#	Description	application : LoanApplication				income : IncomeSource	applicant : Applicant	application	
		amount min	amount max	period	deposit max	income	creditRating	Loan approved	LMI
1		131000	200000	30	20000	Asset	AA	true	0
2		10000	100000	20	2000	Job	AA	true	0
3		100001	130000	20	3000	Job	OK Sub prime	true	10

4.3.1. Advanced enumeration options for rule assets

For advanced use cases with enumeration definitions in your Red Hat Process Automation Manager project, consider the following extended options for defining enumerations:

Mapping between DRL values and values in Business Central

If you want the enumeration values to appear differently or more completely in the Business Central interface than they appear in the DRL source, use a mapping in the format **'fact.field' : ['sourceValue1=UIValue1','sourceValue2=UIValue2', ...]** for your enumeration definition values.
For example, in the following enumeration definition for loan status, the options **A** or **D** are used in the DRL file but the options **Approved** or **Declined** are displayed in Business Central:

```
'Loan.status' : ['A=Approved','D=Declined']
```

Enumeration value dependencies

If you want the selected value in one drop-down list to determine the available options in a subsequent drop-down list, use the format **'fact.fieldB[fieldA=value1]' : ['value2', 'value3', ...]** for your enumeration definition.
For example, in the following enumeration definition for insurance policies, the **policyType** field accepts the values **Home** or **Car**. The type of policy that the user selects determines the policy **coverage** field options that are then available:

```
'Insurance.policyType' : ['Home', 'Car']
'Insurance.coverage[policyType=Home]' : ['property', 'liability']
'Insurance.coverage[policyType=Car]' : ['collision', 'fullCoverage']
```

NOTE

Enumeration dependencies are not applied across rule conditions and actions. For example, in this insurance policy use case, the selected policy in the rule condition does not determine the available coverage options in the rule actions, if applicable.

External data sources in enumerations

If you want to retrieve a list of enumeration values from an external data source instead of defining the values directly in the enumeration definition, on the class path of your project, add a helper class that returns a **java.util.List** list of strings. In the enumeration definition, instead of specifying a list of values, identify the helper class that you configured to retrieve the values externally.
For example, in the following enumeration definition for loan applicant region, instead of defining applicant regions explicitly in the format **'Applicant.region' : ['country1', 'country2', ...]**, the enumeration uses a helper class that returns the list of values defined externally:

```
'Applicant.region' : (new com.mycompany.DataHelper()).getListOfRegions()
```

In this example, a **DataHelper** class contains a **getListOfRegions()** method that returns a list of strings. The enumerations are loaded in the drop-down list for the relevant field in the rule asset.

You can also load dependent enumeration definitions dynamically from a helper class by identifying the dependent field as usual and enclosing the call to the helper class within quotation marks:

```
'Applicant.region[countryCode]' : '(new
com.mycompany.DataHelper()).getListOfRegions("@{countryCode}")'
```

If you want to load all enumeration data entirely from an external data source, such as a relational database, you can implement a Java class that returns a **Map<String, List<String>>** map. The key of the map is the **fact.field** mapping and the value is a **java.util.List<String>** list of values.

For example, the following Java class defines loan applicant regions for the related enumeration:

```
public class SampleDataSource {

  public Map<String, List<String>> loadData() {
    Map data = new HashMap();

    List d = new ArrayList();
    d.add("AU");
    d.add("DE");
    d.add("ES");
    d.add("UK");
    d.add("US");
    ...
    data.put("Applicant.region", d);

    return data;
  }

}
```

The following enumeration definition correlates to this example Java class. The enumeration contains no references to fact or field names because they are defined in the Java class:

```
=(new SampleDataSource()).loadData()
```

The **=** operator enables Business Central to load all enumeration data from the helper class. The helper methods are statically evaluated when the enumeration definition is requested for use in an editor.

NOTE

Defining an enumeration without a fact and field definition is currently not supported in Business Central. To define the enumeration for the associated Java class in this way, use the DRL source in your Red Hat Process Automation Manager project.

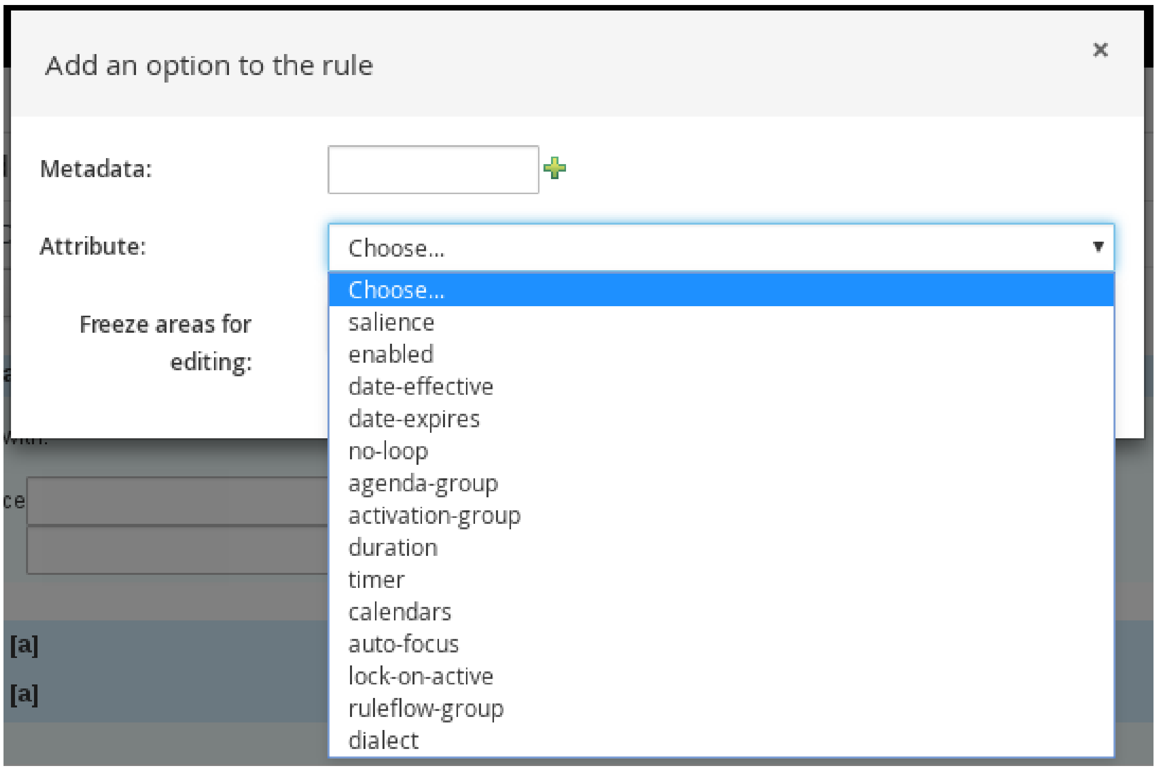
4.4. ADDING OTHER RULE OPTIONS

You can also use the rule designer to add metadata within a rule, define additional rule attributes (such as **salience** and **no-loop**), and freeze areas of the rule to restrict modifications to conditions or actions.

Procedure

1. In the rule designer, click **(show options...)** under the **THEN** section.
2. Click the plus icon (✚) on the right side of the window to add options.
3. Select an option to be added to the rule:
    - **Metadata:** Enter a metadata label and click the plus icon (✚). Then enter any needed data in the field provided in the rule designer.
    - **Attribute:** Select from the list of rule attributes. Then further define the value in the field or option displayed in the rule designer.
    - **Freeze areas for editing:**Select **Conditions** or **Actions** to restrict the area from being modified in the rule designer.

Figure 4.10. Rule options



4. Click **Save** in the rule designer to save your work.

4.4.1. 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
salience	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: **salience 10**
enabled	A Boolean value. When the option is selected, the rule is enabled. When the option is not selected, the rule is disabled. Example: **enabled true**
date-effective	A string containing a date and time definition. The rule can be activated only if the current date and time is after a **date-effective** attribute. Example: **date-effective "4-Sep-2018"**

Attribute	Value
date-expires	A string containing a date and time definition. The rule cannot be activated if the current date and time is after the **date-expires** attribute. Example: **date-expires "4-Oct-2018"**
no-loop	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. Example: **no-loop true**
agenda-group	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. Example: **agenda-group "GroupName"**
activation-group	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. Example: **activation-group "GroupName"**
duration	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. Example: **duration 10000**
timer	A string identifying either **int** (interval) or **cron** timer definitions for scheduling the rule. Example: **timer (cron:* 0/15 * * * ?)** (every 15 minutes)
calendar	A Quartz calendar definition for scheduling the rule. Example: **calendars "* * 0-7,18-23 ? * *"** (exclude non-business hours)
auto-focus	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. Example: **auto-focus true**

Attribute	Value
lock-on-active	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 **no-loop** 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. Example: **lock-on-active true**
ruleflow-group	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. Example: **ruleflow-group "GroupName"**
dialect	A string identifying either **JAVA** or **MVEL** 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. Example: **dialect "JAVA"**

CHAPTER 5. EXECUTING RULES

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

Prerequisites

- Business Central and Process Server are installed and running. For installation options, see *Planning a Red Hat Process Automation Manager installation* .

Procedure

1. In Business Central, go to **Menu** → **Design** → **Projects** and click the project name.
2. In the upper-right corner of the project **Assets** page, click **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 project deployment options, see *Packaging and deploying a Red Hat Process Automation Manager project*.
3. Create a Maven or Java project outside of Business Central, if not created already, that you can use for executing rules locally or that you can use as a client application for executing rules on Process Server. The project must contain a **pom.xml** file and any other required components for executing the project resources.
   For example test projects, see "Other methods for creating and executing DRL rules".
4. Open the **pom.xml** file of your test project or 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 7.5 in a client application **pom.xml** file:

```
<!-- For local execution -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.26.0.Final-redhat-00005</version>
</dependency>

<!-- For remote execution on Process Server -->
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.26.0.Final-redhat-00005</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>7.5.1.redhat-00001</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?.

5. 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 **Person** class dependency 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>
```

6. 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
```

7. 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 needed), 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.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.drools.compiler.kproject.ReleaseIdImpl;

public class RulesTest {

  public static final void main(String[] args) {
    try {
      // Identify the project in the local repository:
      ReleaseId rid = new ReleaseIdImpl("com.myspace", "MyProject", "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();
      }
    }
}
```

8. Run the configured **.java** class from your project directory. You can run the file in your development platform (such as Red Hat CodeReady 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
   ```

9. 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.

5.1. 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 decision 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 13 bound variables only (including the built-in **drools** variable). For example, the following rule consequence uses more than 13 bound variables and creates a compilation error:

```
...
then
  $input.setNo14Count(functions.sumOf(new Object[]{$no1Count_1, $no2Count_1,
$no3Count_1, ..., $no14Count_1}).intValue());
  $input.getFirings().add("fired");
  update($input);
```

5.1.1. 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.

Prerequisites

- 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, 7.26.0.Final-redhat-00005). 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>7.5.1.redhat-00001</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 decision 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*.

5.1.2. 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.

Prerequisites

- 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 decision 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>
```

The **<version>** is the Maven artifact version for Red Hat Process Automation Manager currently used in your project (for example, 7.26.0.Final-redhat-00005).

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>7.5.1.redhat-00001</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 6. 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 Thursday, October 31, 2019.