Red Hat Process Automation Manager 7.1

Designing a decision service using guided rule templates

Red Hat Process Automation Manager 7.1 Designing a decision service using guided rule templates

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

Legal Notice

Abstract

This document describes how to design a decision service using guided rule templates in Red Hat Process Automation Manager 7.1.

Table of Contents

PREFACE

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

Prerequisite

The team and project for the guided rule templates 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	- Are tables of rules that you create in a UI-based table designer in Business Central - Are a wizard-led alternative to uploaded decision table spreadsheets - 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	*Designing a decision service using guided decision tables*
Uploaded decision tables	- Are XLS or XLSX decision table spreadsheets that you 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	*Designing a decision service using uploaded decision tables*

Asset	Highlights	Documentation
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	*Designing a decision service using guided rules*
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	*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	*Designing a decision service using DRL rules*

CHAPTER 2. GUIDED RULE TEMPLATES

Guided rule templates are business rule structures with placeholder values (template keys) that are interchanged with actual values defined in separate data tables. Each row of values defined in the corresponding data table for that template results in a rule. Guided rule templates are ideal when many rules have the same conditions, actions, and other attributes but differ in values of facts or constraints. In such cases, instead of creating many similar guided rules and defining values in each rule, you can create a guided rule template with the rule structure that applies to each rule and then define only the differing values in the data table.

The guided rule templates designer provides fields and options for acceptable template input based on the data objects for the rule template being defined, and a corresponding data table where you add template key values. After you create your guided rule template and add values in the corresponding data table, the rules you defined are compiled into Drools Rule Language (DRL) rules as with all other rule assets.

All data objects related to a guided rule template must be in the same project package as the guided rule template. Assets in the same package are imported by default. After you create the necessary data objects and the guided rule template, you can use the **Data Objects** tab of the guided rule templates 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 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.
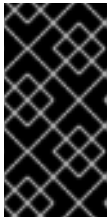
   IMPORTING DATA OBJECTS FROM OTHER PACKAGES

   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

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 RULE TEMPLATES

You can use guided rule templates to define rule structures with placeholder values (template keys) that correspond to actual values defined in a data table. Guided rule templates are an efficient alternative to defining sets of many guided rules individually that use the same structure.

Procedure

1. In Business Central, go to **Menu → Design → Projects** and click the project name.

2. Click **Add Asset → Guided Rule Template**.

3. Enter an informative **Guided Rule Template** 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.
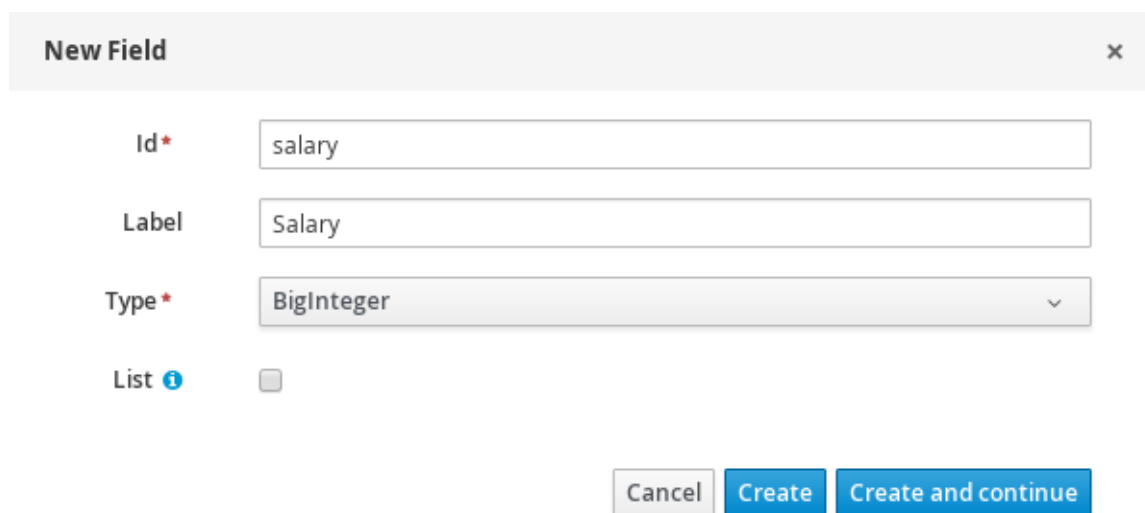
4. Click **Ok** to create the rule template.
   The new guided rule template is now listed in the **Guided Rule Templates** panel of the **Project Explorer**.

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 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 template, based on the available data objects. For the field values that vary per rule, use template keys in the format **$key** in the rule designer or in the format **@{key}** in free form DRL (if used).

   Figure 4.1. Sample guided rule template

NOTE ON TEMPLATE KEYS

Template keys are fundamental in guided rule templates. Template keys are what enable field values in the templates to be interchanged with actual values that you define in the corresponding data table to generate different rules from the same template. You can use other value types, such as **Literal** or **Formula**, for values that are part of the rule structure of all rules based on that template. However, for any values that differ among the rules, use the **Template key** field type with a specified key. Without template keys in a guided rule template, the corresponding data table is not generated in the template designer and the template essentially functions as an individual guided rule.

The **WHEN** part of the rule template is the condition that must be met to execute an action. For example, if a telecommunications company charges customers based on the services they subscribe to (Internet, phone, and TV), then one of the **WHEN** conditions would be **internetService | equal to | $hasInternetService**. The template key **$hasInternetService** is interchanged with an actual Boolean value (**true** or **false**) defined in the data table for the template.

The **THEN** part of the rule template is the action to be performed when the conditional part of the rule has been met. For example, if a customer subscribes to only Internet service, a **THEN** action for **RecurringPayment** with a template key **$amount** would set the actual monthly amount to the integer value defined for Internet service charges in the data table.

7. After you define all components of the rule, click **Save** in the guided rule templates designer to save your work.

4.1. ADDING WHEN CONDITIONS IN GUIDED RULE TEMPLATES

The **WHEN** part of the rule contains the conditions that must be met to execute an action. For example, if a telecommunications company charges customers based on the services they subscribe to (Internet, phone, and TV), then one of the **WHEN** conditions would be **internetService | equal to | $hasInternetService**. The template key **$hasInternetService** is interchanged with an actual Boolean value (**true** or **false**) defined in the data table for the template.

Prerequisite

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

Procedure

1. In the guided rule templates 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
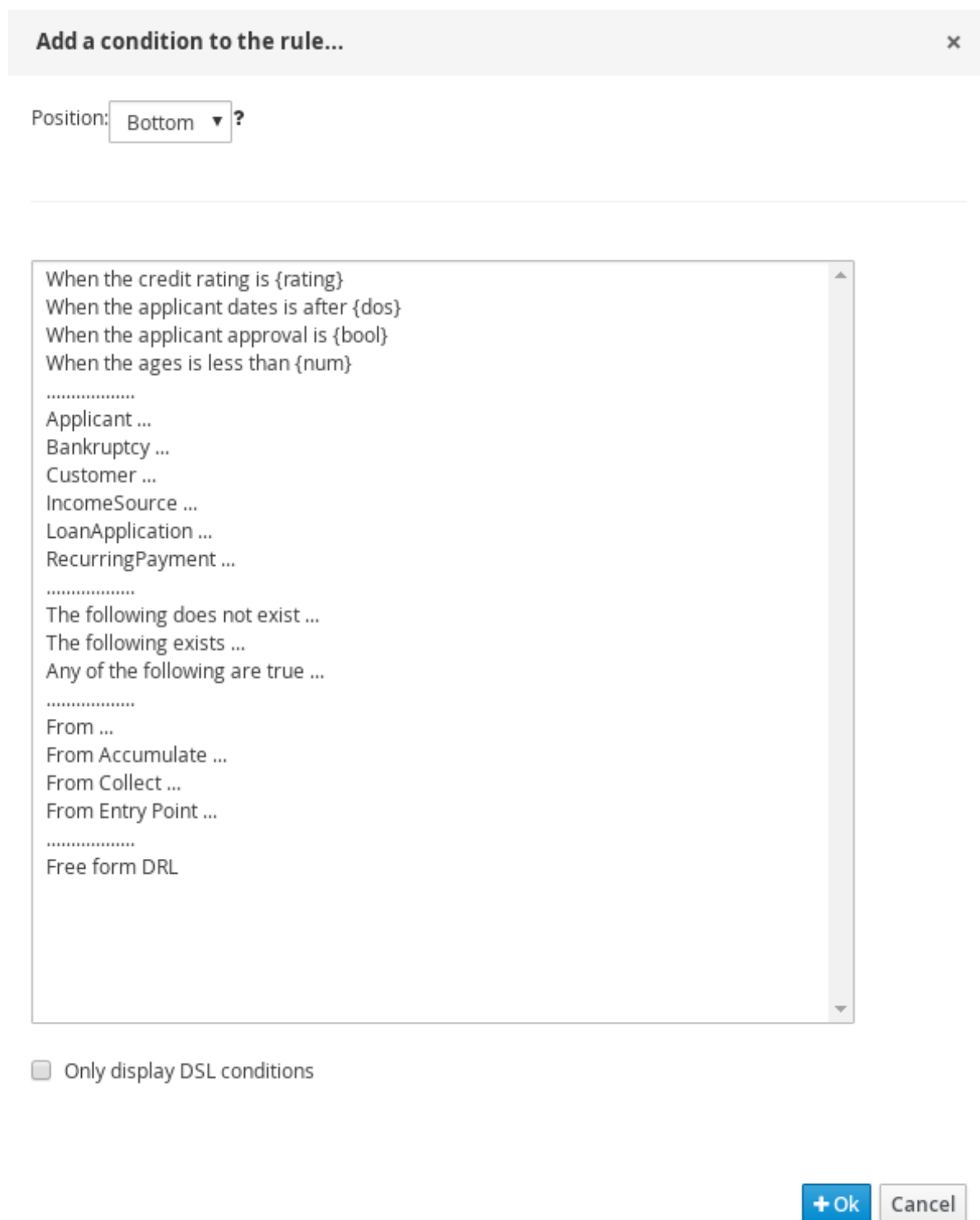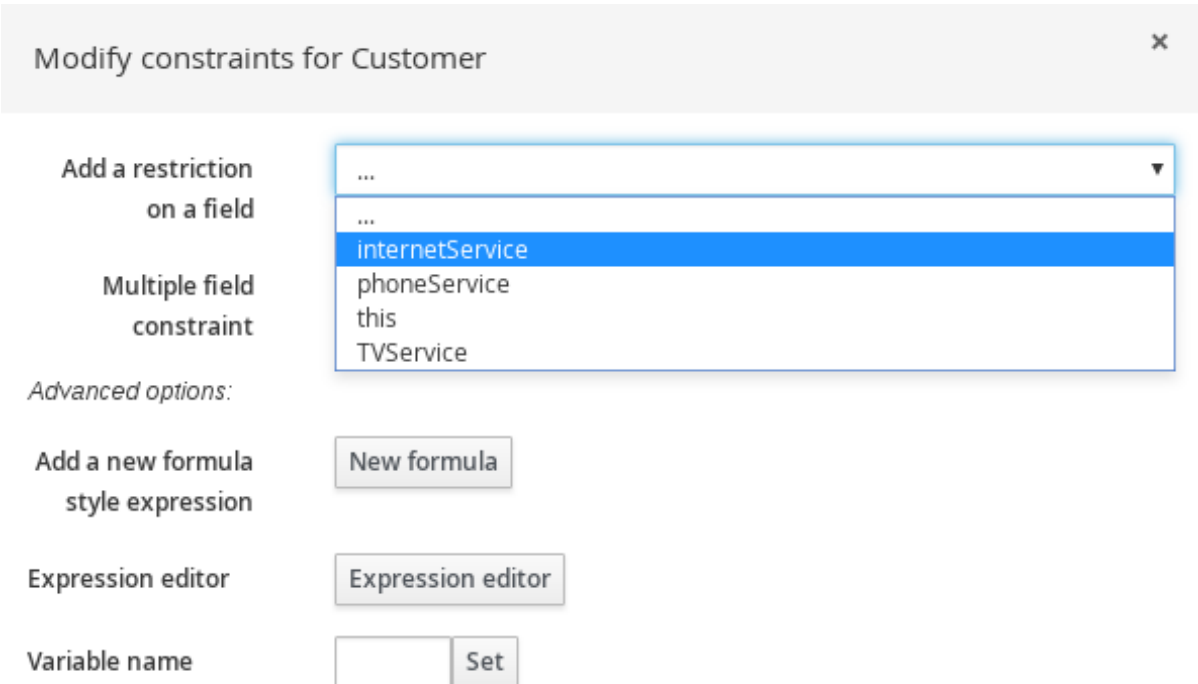
The list includes the data objects from the **Data Objects** tab of the guided rule templates designer, any DSL objects defined for the package, and the following standard options:

- **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. For template keys in free form DRL, use the format **@{key}**.

2. Choose a condition element (for example, **Customer**) and click **Ok**.

3. Click the condition element in the guided rule templates designer and use the **Modify constraints for Customer** 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 **Customer** to **c** and then reference **c** in a separate **Applicant** constraint that specifies that the **Customer** is the **Applicant**.
>
> ```
> c : Customer()
> Applicant( this == c )
> ```
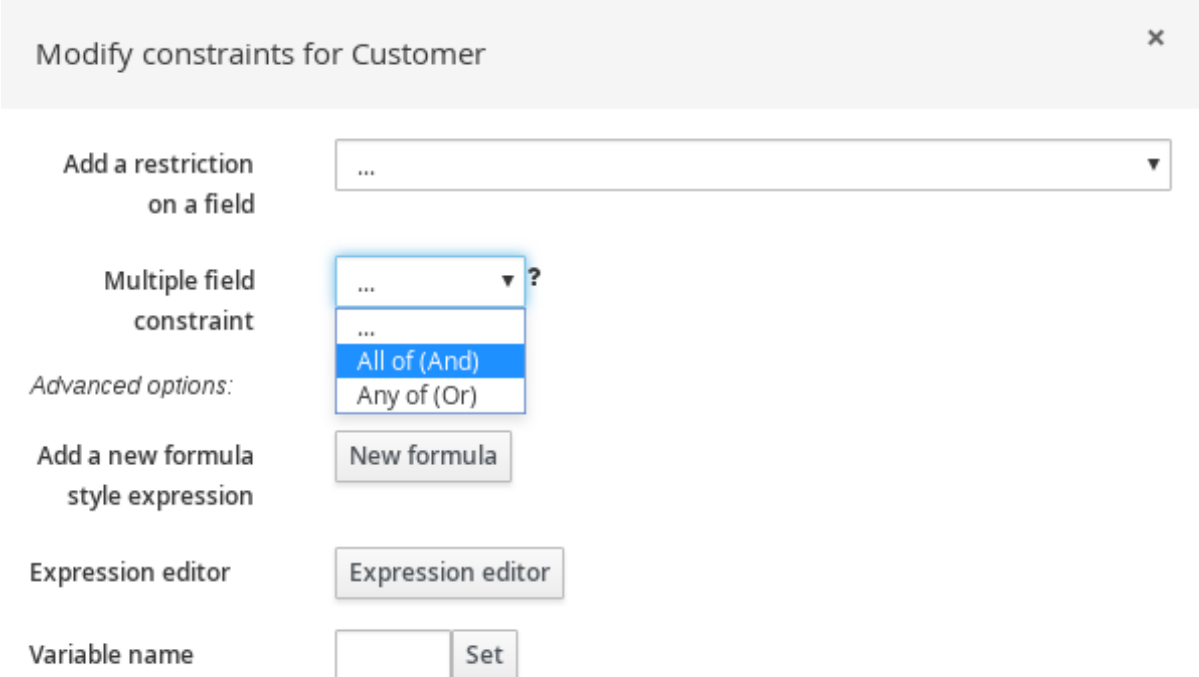
After you select a constraint, the window closes automatically.

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

5. Click the edit icon (✏️) to define the field value.

6. Select **Template key** and add a template key in the format **$key** if this value varies among the rules that are based on this template. This allows the field value to be interchanged with actual values that you define in the corresponding data table to generate different rules from the same template. For field values that do not vary among the rules and are part of the rule template, you can use any other value type.

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

   Figure 4.4. Add multiple field constraints

8. Click the constraint in the guided rule templates designer and further define the field values.

9. After you define all condition elements, click **Save** in the guided rule templates designer to save your work.

4.2. ADDING THEN ACTIONS IN GUIDED RULE TEMPLATES

The **THEN** part of the rule template is the action to be performed when the conditional part of the rule has been met. For example, if a customer subscribes to only Internet service, a **THEN** action for **RecurringPayment** with a template key **$amount** would set the actual monthly amount to the integer value defined for Internet service charges in the data table.

Prerequisite

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

Procedure

1. In the guided rule templates 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 ▾ ?

   | Approve the loan |
   | .... |
   | Insert fact Applicant... |
   | Insert fact Bankruptcy... |
   | Insert fact Customer... |
   | Insert fact IncomeSource... |
   | Insert fact LoanApplication... |
   | Insert fact RecurringPayment... |
   | .... |
   | Logically Insert fact Applicant... |
   | Logically Insert fact Bankruptcy... |
   | Logically Insert fact Customer... |
   | Logically Insert fact IncomeSource... |
   | Logically Insert fact LoanApplication... |
   | Logically Insert fact RecurringPayment... |
   | .... |
   | Add free form DRL |

   ☐ Only display DSL actions

   + Ok   Cancel

   The list includes insertion and modification options based on the data objects in the **Data Objects** tab of the guided rule templates designer, and on any DSL objects defined for the package:

   - **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 process engine and define

resulting fields and values 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.

- **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. For template keys in free form DRL, use the format **@{key}**.

2. Choose an action element (for example, **Logically Insert fact RecurringPayment**) and click **Ok**.

3. Click the action element in the guided rule templates designer and use the **Add a field** window to select a field.

Figure 4.6. Add a field
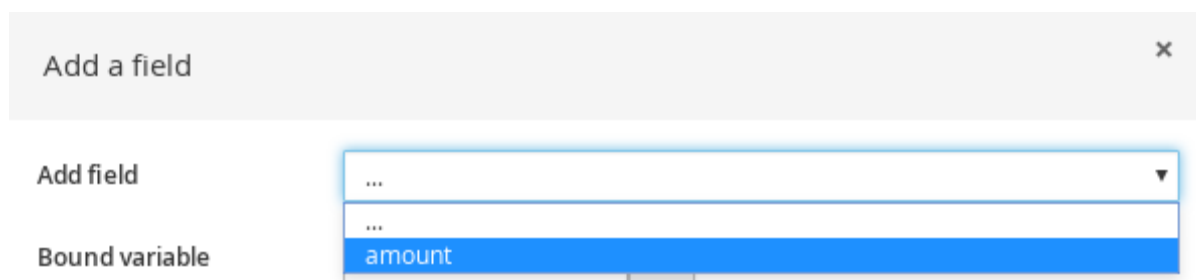
After you select a field, the window closes automatically.

4. Click the edit icon ( ) to define the field value.

5. Select **Template key** and add a template key in the format **$key** if this value varies among the rules that are based on this template. This allows the field value to be interchanged with actual values that you define in the corresponding data table to generate different rules from the same template. For field values that do not vary among the rules and are part of the rule template, you can use any other value type.

6. After you define all action elements, click **Save** in the guided rule templates designer to save your work.

## 4.3. 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 4.7. Rule options
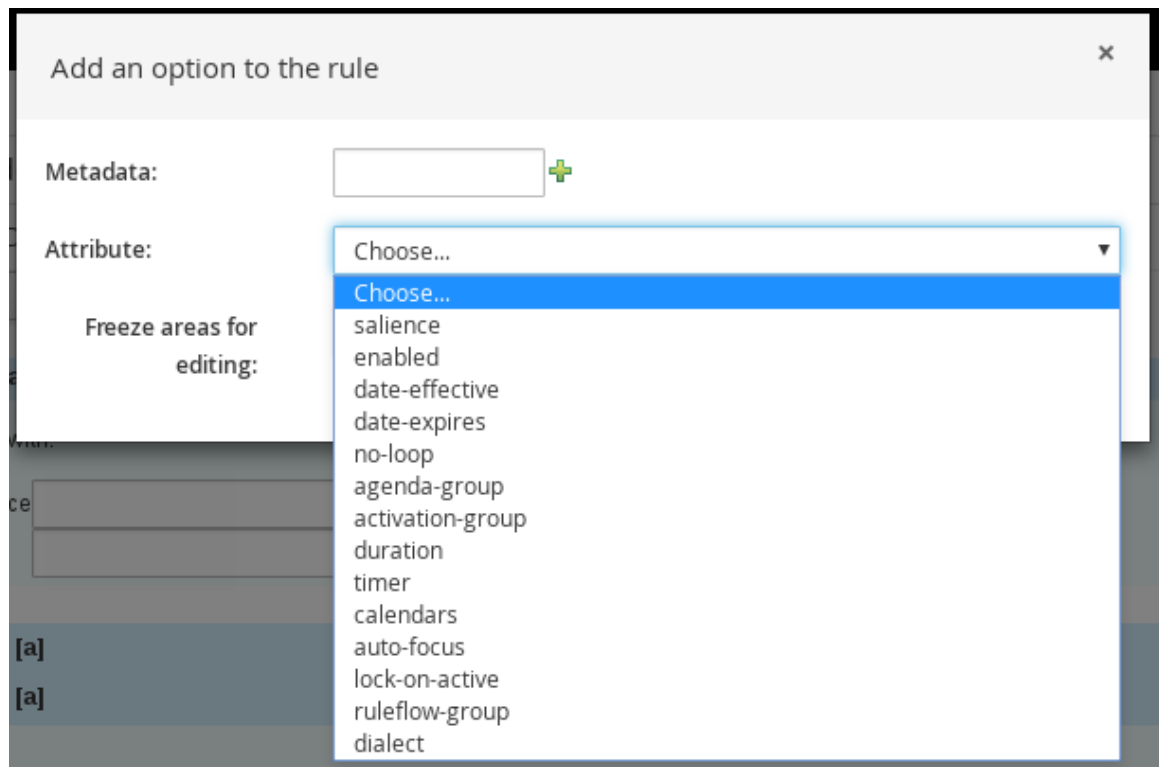
4. Click **Save** in the rule designer to save your work.

4.3.1. 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 definition for scheduling the rule. Example: **timer "*/5 * * * *"** (every 5 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. DEFINING DATA TABLES FOR GUIDED RULE TEMPLATES

After you create a guided rule template and add template keys for field values, a data table is displayed in the **Data** table of the guided rule templates designer. Each column in the data table corresponds to a template key that you added in the guided rule template. Use this table to define values for each template key row by row. Each row of values that you define in the data table for that template results in a rule.

Procedure

1. In the guided rule templates designer, click the **Data** tab to view the data table. Each column in the data table corresponds to a template key that you added in the guided rule template.

   NOTE

   If you did not add any template keys to the rule template, then this data table does not appear and the template does not function as a genuine template but essentially as an individual guided rule. For this reason, template keys are fundamental in creating guided rule templates.

2. Click **Add row** and define the data values for each template key column to generate that rule (row).

3. Continue adding rows and defining data values for each rule that will be generated. You can click **Add row** for each new row, or click the plus icon (✚) or minus icon to add or remove rows.

Figure 5.1. Sample data table for a guided rule template

	$hasInternetService	$hasPhoneService	$hasTVService	$amount
	✔	✔	✔	15
	✔	✔	☐	10
	✔	☐	✔	10
	☐	✔	✔	10
	✔	☐	☐	5
	☐	✔	☐	5
	☐	☐	✔	5

To view the DRL code, click the **Source** tab in the guided rule templates designer.

Example:

```
rule "PaymentRules_6"
 dialect "mvel"
  when
```

```
  Customer( internetService == false ,
   phoneService == false ,
   TVService == true )
  then
   RecurringPayment fact0 = new RecurringPayment();
   fact0.setAmount( 5 );
   insertLogical( fact0 );
  end

rule "PaymentRules_5"
 dialect "mvel"
 when
  Customer( internetService == false ,
   phoneService == true ,
   TVService == false )
  then
   RecurringPayment fact0 = new RecurringPayment();
   fact0.setAmount( 5 );
   insertLogical( fact0 );
  end
  ...
  //Other rules omitted for brevity.
```
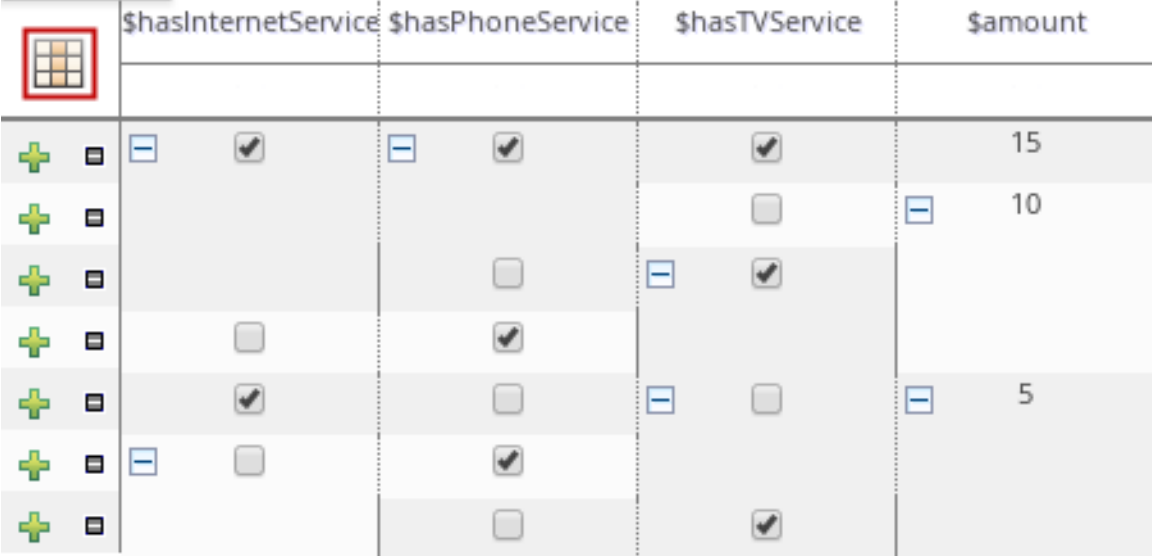
4. As a visual aid, click the grid icon in the upper-left corner of the data table to toggle cell merging on and off, if needed. Cells in the same column with identical values are merged into a single cell.

Figure 5.2. Merge cells in a data table

You can then use the expand/collapse icon [+/-] in the upper-left corner of each newly merged cell to collapse the rows corresponding to the merged cell, or to re-expand the collapsed rows.

Figure 5.3. Collapse merged cells

5. After you define the template key data for all rules and adjust the table display as needed, click **Validate** in the upper-right toolbar of the guided rule templates designer to validate the guided rule template. If the rule template validation fails, address any problems described in the error message, review all components in the rule template and data defined in the data table, and try again to validate the rule template until the rule template passes.

6. Click **Save** in the guided rule templates designer to save your work.

CHAPTER 6. 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

```java
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);
```

```java
        // 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.

6.1. 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 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);
> ```

6.1.1. 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.

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*.

6.1.2. 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.

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>
```
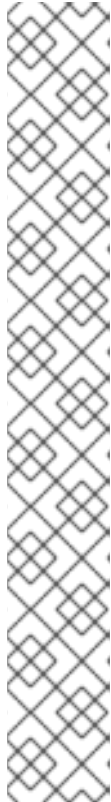
```
    <version>${rhpam.version}</version>
  </dependency>

  <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, 7.11.0.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 Friday, May 22, 2020.