Red Hat Process Automation Manager 7.4

Designing a decision service using guided decision tables

Red Hat Process Automation Manager 7.4 Designing a decision service using guided decision tables

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

Legal Notice

Abstract

This document describes how to design a decision service using guided decision tables in Red Hat Process Automation Manager 7.4.

Table of Contents

PREFACE

As a business analyst or business rules developer, you can use guided decision tables to define business rules in a wizard-led tabular format. These rules are compiled into Drools Rule Language (DRL) and form the core of the decision service for your project.

Prerequisites

- The space and project for the guided decision tables 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. 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 DECISION TABLES

Guided decision tables are a wizard-led alternative to spreadsheet decision tables for defining business rules in a tabular format. With guided decision tables, you are led by a UI-based wizard in Business Central that helps you define rule attributes, metadata, conditions, and actions based on specified data objects in your project. After you create your guided decision tables, 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 decision table must be in the same project package as the guided decision table. Assets in the same package are imported by default. After you create the necessary data objects and the guided decision table, you can use the **Data Objects** tab of the guided decision tables 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.
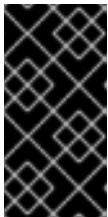
IMPORTING DATA OBJECTS FROM OTHER PACKAGES

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

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 DECISION TABLES

You can use guided decision tables to define rule attributes, metadata, conditions, and actions in a tabular format that can be added to your business rules project.

Procedure

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

2. Click **Add Asset → Guided Decision Table**.

3. Enter an informative **Guided Decision Table** 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. Select **Use Wizard** to finish setting up the table in the wizard, or leave this option unselected to finish creating the table and specify remaining configurations in the guided decision tables designer.

5. Select the hit policy that you want your rows of rules in the table to conform to. For details, see Chapter 5, *Hit policies for guided decision tables* .

6. Specify whether you want the **Extended entry** or **Limited entry** table. For details, see Section 5.1.1, "Types of guided decision tables".

7. Click **Ok** to complete the setup. If you have selected **Use Wizard**, the Guided Decision Table wizard is displayed. If you did not select the **Use Wizard** option, this prompt does not appear and you are taken directly to the table designer.

Figure 4.1. Create guided decision table

8. If you are using the wizard, add any available imports, fact patterns, constraints, and actions, and select whether table columns should expand. Click **Finish** to close the wizard and view the table designer.

Figure 4.2. Guided Decision Table wizard

In the guided decision tables designer, you can add or edit columns and rows, and make other final adjustments.

CHAPTER 5. HIT POLICIES FOR GUIDED DECISION TABLES

Hit policies determine the order in which rules (rows) in a guided decision table are applied, whether top to bottom, per specified priority, or other options.

The following hit policies are available:

- **None:** (Default hit policy) Multiple rows can be executed and the verification warns about rows that conflict. Any decision tables that have been uploaded (using a non-guided decision table spreadsheet) will adopt this hit policy.

- **Resolved Hit:** Only one row at a time can be executed according to specified priority, regardless of list order (you can give row 10 priority over row 5, for example). This means you can keep the order of the rows you want for visual readability, but specify priority exceptions.

- **Unique Hit:** Only one row at a time can be executed, and each row must be unique, with no overlap of conditions being met. If more than one row is executed, then the verification produces a warning at development time.

- **First Hit:** Only one row at a time can be executed in the order listed in the table, top to bottom.

- **Rule Order:** Multiple rows can be executed and verification does not report conflicts between the rows since they are expected to happen.

Figure 5.1. Available hit policies
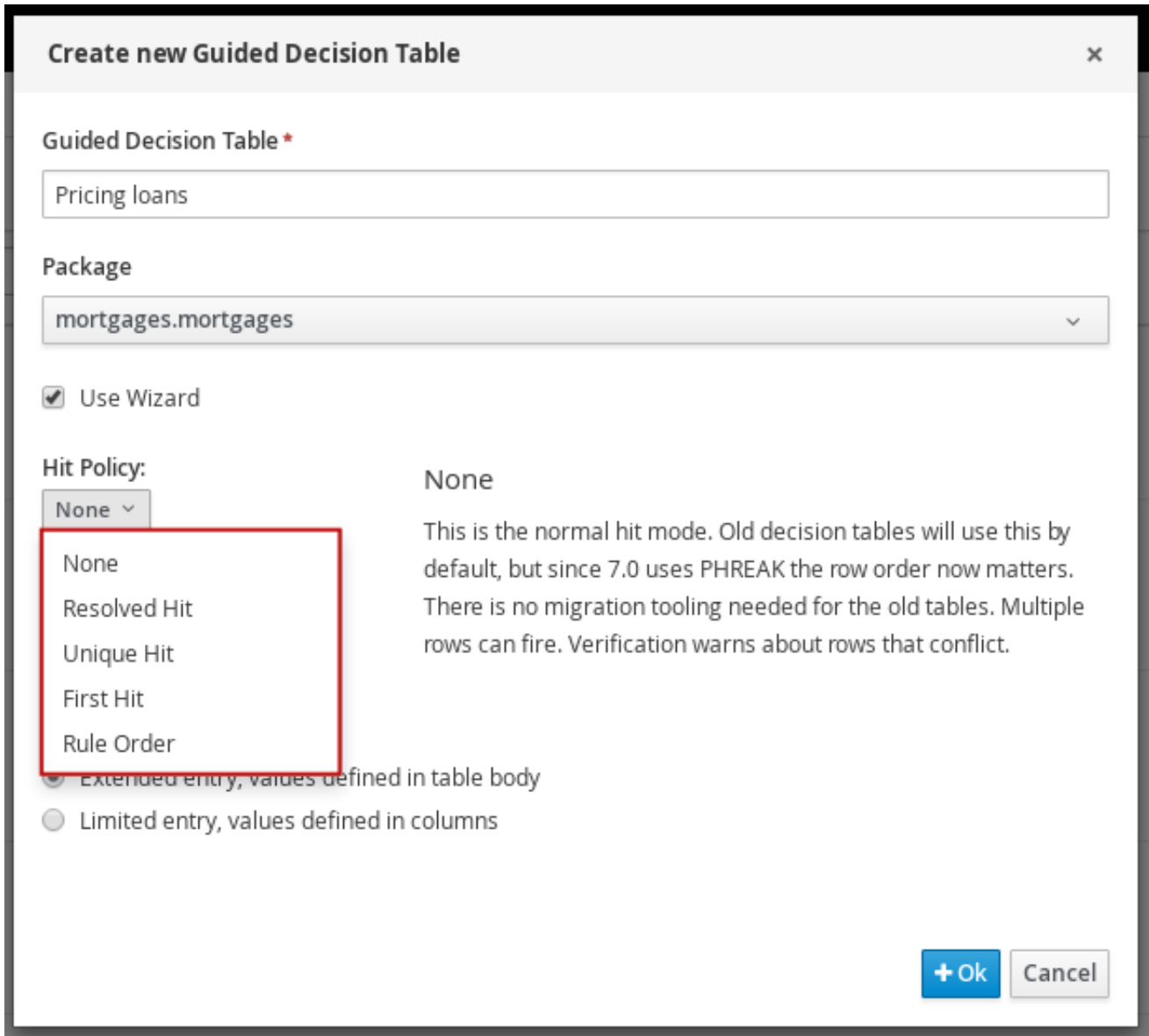
5.1. HIT POLICY EXAMPLES: DECISION TABLE FOR DISCOUNTS ON MOVIE TICKETS

The following is part of a decision table for discounts on movie tickets based on customer age, student status, or military status, or all three.

Table 5.1. Decision table for available discounts on movie tickets

Row Number	Discount Type	Discount
1	Senior citizen (age 60+)	10%
2	Student	10%
3	Military	10%

The total discount to be applied in the end will vary depending on the hit policy specified for the table, as follows.

- **None/Rule Order:** With both **None** and **Rule Order** hit policies, all applicable rules are incorporated, in this case allowing discounts to be stacked for each customer.
  Example: A senior citizen who is also a student and a military veteran will receive all three discounts, totaling 30%.

  Key difference: With **None**, warnings are created for multiple rows applied. With **Rule Order**, those warnings are not created.

- **First Hit/Resolved Hit:** With both **First Hit** and **Resolved Hit** policies, only one of the discounts can be applied.
  For **First Hit**, the discount that is satisfied first in the list is applied and the others are ignored.

  Example: A senior citizen who is also a student and a military veteran will receive only the senior citizen discount of 10%, since that is listed first in the table.

  For **Resolved Hit**, a modified table is required. The discount that you assign a priority exception to in the table, regardless of listed order, will be applied first. To assign this exception, include a new column that specifies the priority of one discount (row) over others.

  Example: If military discounts are prioritized higher than age or student discounts, despite the listed order, then a senior citizen who is also a student and a military veteran will receive only the military discount of 10%, regardless of age or student status.

  Consider the following modified decision table that accommodates a **Resolved Hit** policy:

Table 5.2. Modified decision table that accommodates a *Resolved Hit* policy

Row Number	Discount Type	Has Priority over Row	Discount
1	Senior citizen (age 60+)		10%
2	Student		10%
3	Military	1	10%

In this modified table, the military discount is essentially the new row 1 and therefore takes priority over both age and student discounts, and any other discounts added later. You do not need to specify priority over rows "1 and 2", only over row "1". This changes the row hit order to 3 → 1 → 2 → ... and so on as the table grows.

> **NOTE**
>
> The row order would be changed in the same way if you actually moved the military discount to row 1 and applied a **First Hit** policy to the table instead. However, if you want the rules listed in a certain way and applied differently, such as in an alphabetized table, the **Resolved Hit** policy is useful.

Key difference: With **First Hit**, rules are applied strictly in the listed order. With **Resolved Hit**, rules are applied in the listed order unless priority exceptions are specified.

- **Unique Hit:** A modified table is required. With a **Unique Hit** policy, rows must be created in a way that it is impossible to satisfy multiple rules at one time. However, you can still specify row-by-row whether to apply one rule or multiple. In this way, with a **Unique Hit** policy you can make

decision tables more granular and prevent overlap warnings.

Consider the following modified decision table that accommodates a **Unique Hit** policy:

Table 5.3. Modified decision table that accommodates a*Unique Hit* **policy**

Row Number	Is Senior Citizen (age 65+)	Is Student	Is Military	Discount
1	yes	no	no	10%
2	no	yes	no	10%
3	no	no	yes	10%
4	yes	yes	no	20%
5	yes	no	yes	20%
6	no	yes	yes	20%
7	yes	yes	yes	30%

In this modified table, each row is unique, with no allowance of overlap, and any single discount or any combination of discounts is accommodated.

5.1.1. Types of guided decision tables

Two types of decision tables are supported in Red Hat Process Automation Manager: Extended entry and Limited entry tables.

- **Extended entry:** An Extended Entry decision table is one for which the column definitions specify Pattern, Field, and Operator but not value. The values, or states, are themselves held in the body of the decision table.

Pricing loans		application : LoanApplication				ome : IncomeSou	application		
#	Description	amount min	amount max	period	deposit max	income	Loan approved	LMI	rate
1		131000	200000	30	20000	Asset	true	0	2
2		10000	100000	20	2000	Job	true	0	4
3		100001	130000	20	3000	Job	true	10	6

- **Limited entry:** A Limited Entry decision table is one for which the column definitions specify value in addition to Pattern, Field, and Operator. The decision table states, held in the body of the table, are boolean where a positive value (a marked check box) has the effect of meaning the column should apply, or be matched. A negative value (a cleared check box) means the column does not apply.

	Credit rating					
#	Description	CR = AA	CR = OK	CR = Sub prime	Approve	Decline
1		☑	☐	☐	☑	☐
2		☐	☑	☐	☑	☐
3		☐	☐	☑	☐	☑

CHAPTER 6. ADDING COLUMNS TO GUIDED DECISION TABLES

After you have created the guided decision table, you can define and add various types of columns within the guided decision tables designer.

Prerequisites

- Any data objects that will be used for column parameters, such as Facts and Fields, have been created within the same package where the guided decision table is found, or have been imported from another package in **Data Objects → New item** of the guided decision tables designer.

For descriptions of these column parameters, see the "Required column parameters" segments for each column type in Chapter 7, *Types of columns in guided decision tables* .

For details about creating data objects, see Section 3.1, "Creating data objects".

Procedure

1. In the guided decision tables designer, click **Columns → Insert Column**.

2. Click **Include advanced options** to view the full list of column options.

Figure 6.1. Add columns
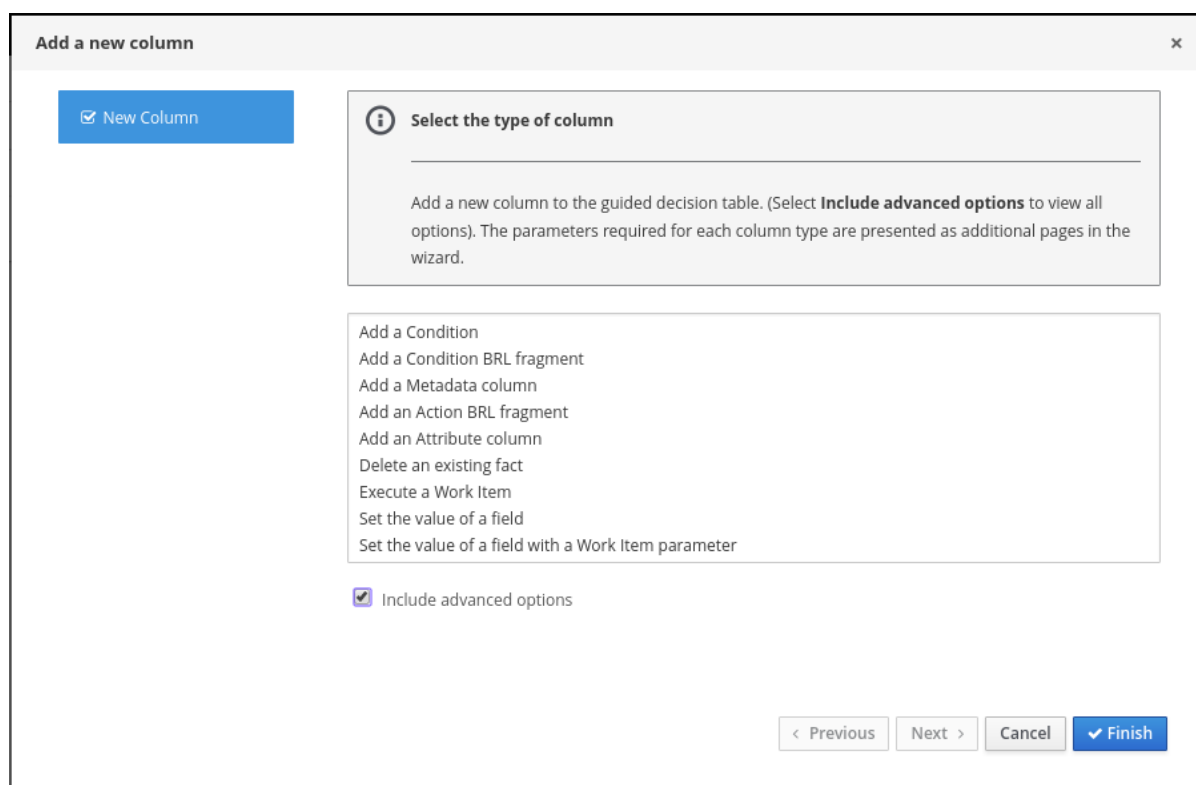
3. Select the column type that you want to add, click **Next**, and follow the steps in the wizard to specify the data required to add the column.
For descriptions of each column type and required parameters for setup, see Chapter 7, *Types of columns in guided decision tables*.

4. Click **Finish** to add the configured column.

After all columns are added, you can begin adding rows of rules correlating to your columns to complete the decision table. For details, see Chapter 9, *Adding rows and defining rules in guided decision tables* .

Figure 6.2. Example of complete guided decision table

Pricing loans									
#	Description	application : LoanApplication				ome : IncomeSou	application		
		amount min	amount max	period	deposit max	income	Loan approved	LMI	rate
1		131000	200000	30	20000	Asset	true	0	2
2		10000	100000	20	2000	Job	true	0	4
3		100001	130000	20	3000	Job	true	10	6

CHAPTER 7. TYPES OF COLUMNS IN GUIDED DECISION TABLES

The **Add a new column** wizard for guided decision tables provides the following column options. (Select **Include advanced options** to view all options.)

- Add a Condition

- Add a Condition BRL fragment

- Add a Metadata column

- Add an Action BRL fragment

- Add an Attribute column

- Delete an existing fact

- Execute a Work Item

- Set the value of a field

- Set the value of a field with a Work Item result

These column types and the parameters required for each in the **Add a new column** wizard are described in the sections that follow.

> **IMPORTANT: REQUIRED DATA OBJECTS FOR COLUMN PARAMETERS**
>
> Some of the column parameters described in this section, such as Fact Patterns and Fields, provide drop-down options consisting only of data objects already defined within the same package where the guided decision table is found. Available data objects for the package are listed in the **Data Objects** panel of the Project Explorer and in the **Data Objects** tab of the guided decision tables designer. You can create additional data objects within the package as needed, or import them from another package in **Data Objects → New item** of the guided decision tables designer. For details about creating data objects, see Section 3.1, "Creating data objects".

7.1. "ADD A CONDITION"

Conditions represent fact patterns defined in the left ("WHEN") portion of a rule. With this column option, you can define one or more condition columns that check for the presence or absence of data objects with certain field values, and that affect the action ("THEN") portion of the rule. You can define a binding for the fact in the condition table, or select one that has previously been defined. You can also choose to negate the pattern.

Example:

```
when
  $i : IncomeSource( type == "Asset" ) // Binds the IncomeSource object to the $i variable
then
  ...
end
```

```
when
  not IncomeSource( type == "Asset" ) // Negates matching pattern
then
  ...
end
```

After a binding is specified, you can define field constraints. If two or more columns are defined using the same fact pattern binding, the field constraints become composite field constraints on the same pattern. If you define multiple bindings for a single model class, each binding becomes a separate model class in the condition ("WHEN") side of the rule.

Required column parameters

The following parameters are required in the **Add a new column** wizard to set up this column type:

- **Pattern:** Select from the list of fact patterns already used in conditions in your table or create a new fact pattern. A fact pattern is a combination of an available data object in the package (see the note on Required data objects for details) and a model class binding that you specify. (Examples: **LoanApplication [application]** or **IncomeSource [income]** where the bracketed portion is the binding to the given fact type)

- **Entry point:** Define the entry point for the fact pattern, if applicable. An entry point is a gate or stream through which facts enter the decision engine, if specified. (Examples: **Application Stream**, **Credit Check Stream**)

- **Calculation type:** Select one of the following calculation types:

  - **Literal value:** The value in the cell will be compared with the field using the operator.

  - **Formula:** The expression in the cell will be evaluated and then compared with the field.

  - **Predicate:** No field is needed; the expression will be evaluated to **true** or **false**.

- **Field:** Select a field from the previously specified fact pattern. The field options are defined in the fact file in the **Data Objects** panel of your project. (Examples: **amount** or **lengthYears** fields within the **LoanApplication** fact type)

- **Binding (optional):** Define a binding for the previously selected field, if needed. (Example: For pattern **LoanApplication [application]**, field **amount**, and operator **equal to**, if binding is set to **$amount**, the end condition will be **application : LoanAppplication($amount : amount == [value])**.)

- **Operator:** Select the operator to be applied to the fact pattern and field previously specified.

- **Value list (optional):** Enter a list of value options, delimited by a comma and space, to limit table input data for the condition ("WHEN") portion of the rule. When this value list is defined, the values will be provided in the table cells for that column as a drop-down list, from which users can select only one option. (Example list: **Monday, Wednesday, Friday** to specify only these three options)

- **Default value (optional):** Select one of the previously defined value options as the default value that will appear in the cell automatically in a new row. If the default value is not specified, the table cell will be blank by default. You can also select a default value from any previously configured data enumerations in the project, listed in the **Enumeration Definitions** panel of the Project Explorer. (You can create enumerations in **Menu → Design → Projects → *[select project]* → Add Asset → Enumeration**.)

- **Header (description):** Add header text for the column.

- **Hide column:** Select this to hide the column, or clear this to display the column.

7.1.1. Inserting an *any other* value in condition column cells

For simple condition columns in guided decision tables, you can apply an **any other** value to table cells within the column if the following parameters are set:

- **Calculation type** for the condition column has been set to **Literal value**.

- **Operator** has been set as equality **==** or inequality **!=**.

The **any other** value enables a rule to be defined for any other field values not explicitly defined in the rules already in the table.

Example:

```
when
  IncomeSource( type not in ("Asset", "Job") )
  ...
then
  ...
end
```

Procedure

1. Select a cell of a condition column that uses the **==** or **!=** operator.

2. In the upper-right toolbar of the table designer, click **Edit → Insert "any other" value**.

7.2. "ADD A CONDITION BRL FRAGMENT"

A Business Rule Language (BRL) fragment is a section of a rule created using the guided rules designer. The condition BRL fragment is the "WHEN" portion of the rule, and the action BRL fragment is the "THEN" portion of the rule. With this column option, you can define a condition BRL fragment to be used in the left ("WHEN") side of a rule. Simpler column types can refer to Facts and Fact fields bound in the BRL fragment and vice-versa.

Example condition BRL fragment for a loan application:

Figure 7.1. Add a condition BRL fragment with the embedded guided rules designer

You can also select **Free form DRL** from the list of condition options to define the condition BRL fragment without the embedded guided rules designer.

Figure 7.2. Add a condition BRL fragment with free form DRL
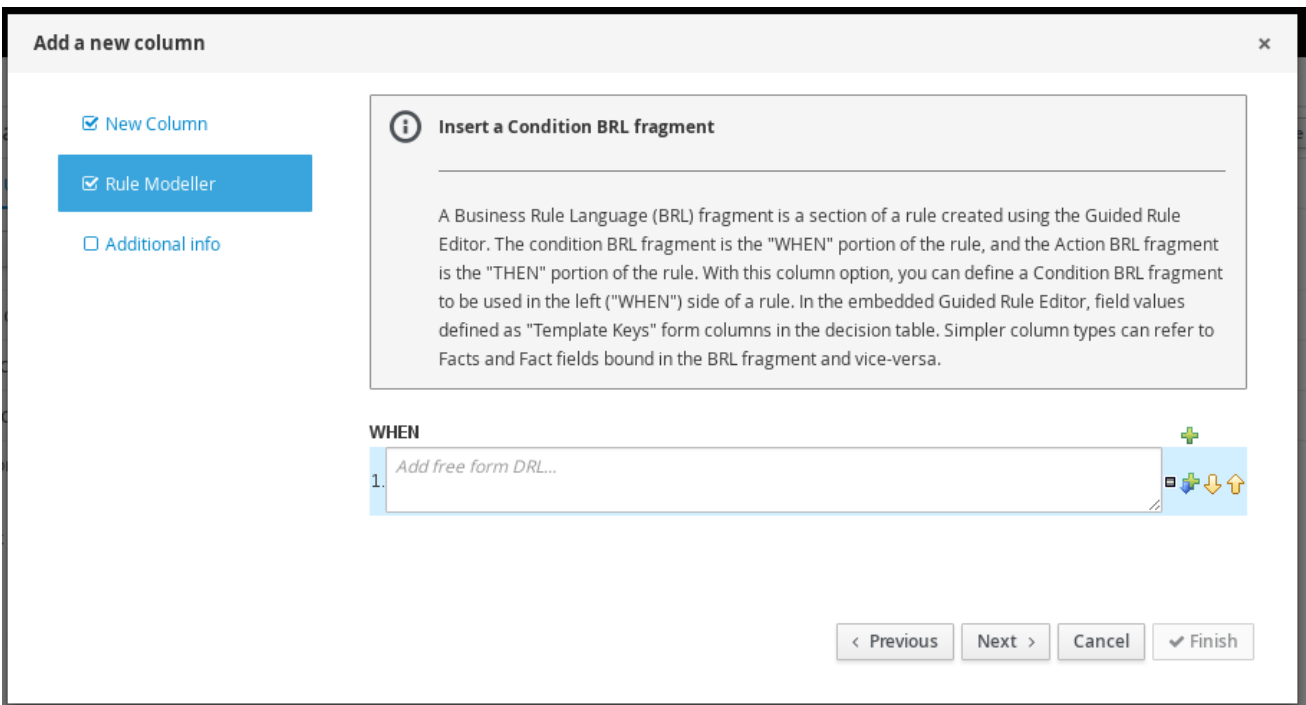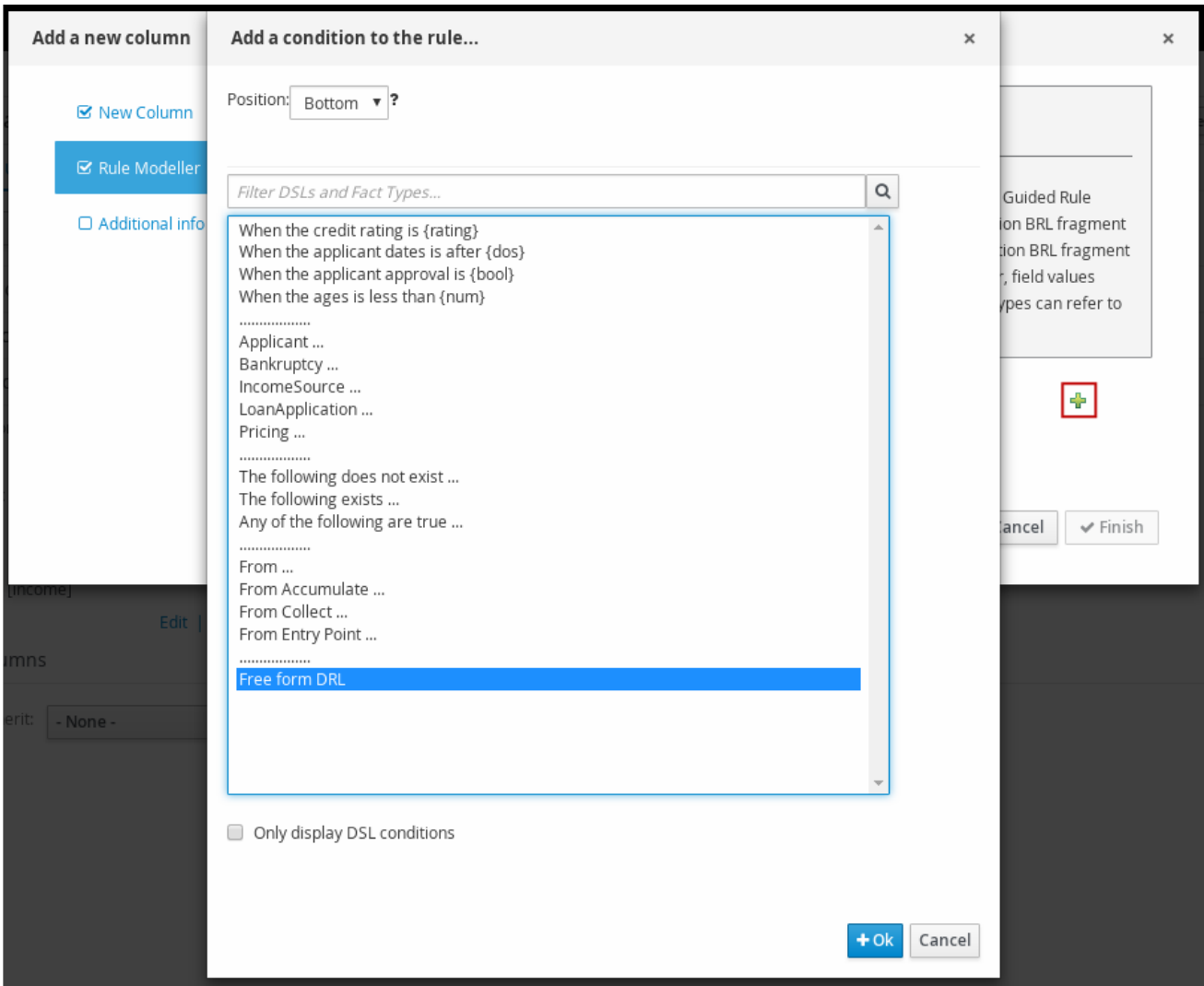
TEMPLATE KEYS

When you add a field for a condition BRL fragment, one of the value options is **Template key** (as opposed to **Literal** or **Formula**). Template keys are placeholder variables that are interchanged with a specified value when the guided decision table is generated, and form separate columns in the table for each template key value specified. While Literal and Formula values are static in a decision table, Template key values can be modified as needed.

In the embedded guided rules designer, you can add a template key value to a field by selecting the **Template key** field option and entering the value in the editor in the format **$key**. For example, **$age** creates an **$age** column in the decision table.

In free form DRL, you can add a template key value to facts in the format **@{key}**. For example, **Person(age > @{age})** creates an **$age** column in the decision table.

The data type is String for new columns added using template keys.

Required column parameters

The following parameters are required in the **Add a new column** wizard to set up this column type:

- **Rule Modeller:** Define the condition BRL fragment ("WHEN" portion) for the rule.

- **Header (description):** Add header text for the column.

- **Hide column:** Select this to hide the column, or clear this to display the column.

7.3. "ADD A METADATA COLUMN"

With this column option, you can define a metadata element as a column in your decision table. Each column represents the normal metadata annotation in DRL rules. By default, the metadata column is hidden. To display the column, click **Edit Columns** in the guided decision tables designer and clear the **Hide column** check box.

Required column parameter

The following parameter is required in the **Add a new column** wizard to set up this column type:

- **Metadata:** Enter the name of the metadata item in Java variable form (that is, it cannot start with a number or contain spaces or special characters).

7.4. "ADD AN ACTION BRL FRAGMENT"

A Business Rule Language (BRL) fragment is a section of a rule created using the guided rules designer. The condition BRL fragment is the "WHEN" portion of the rule, and the action BRL fragment is the "THEN" portion of the rule. With this column option you can define an action BRL fragment to be used in the right ("THEN") side of a rule. Simpler column types can refer to Facts and Fact fields bound in the BRL fragment and vice-versa.

Example action BRL fragment for a loan application:

Figure 7.3. Add an action BRL fragment with the embedded guided rules designer
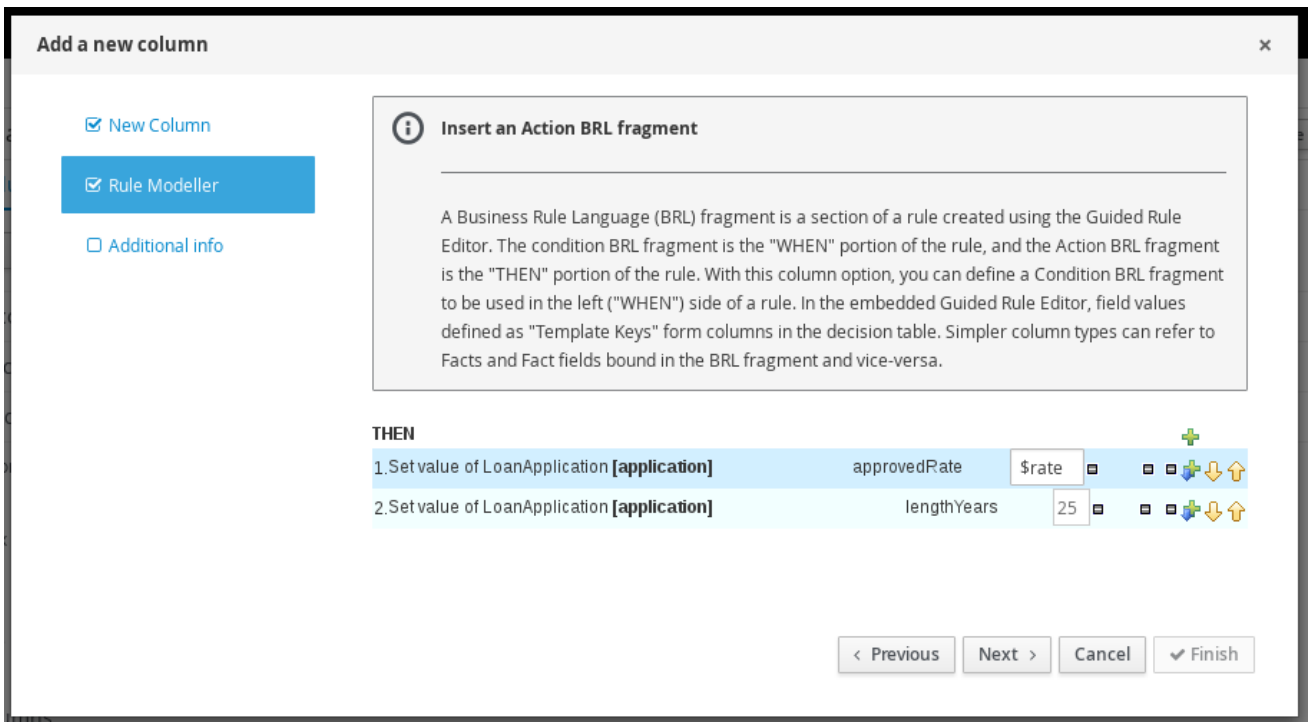
You can also select **Add free form DRL** from the list of action options to define the action BRL fragment without the embedded guided rules designer.

Figure 7.4. Add an action BRL fragment with free form DRL
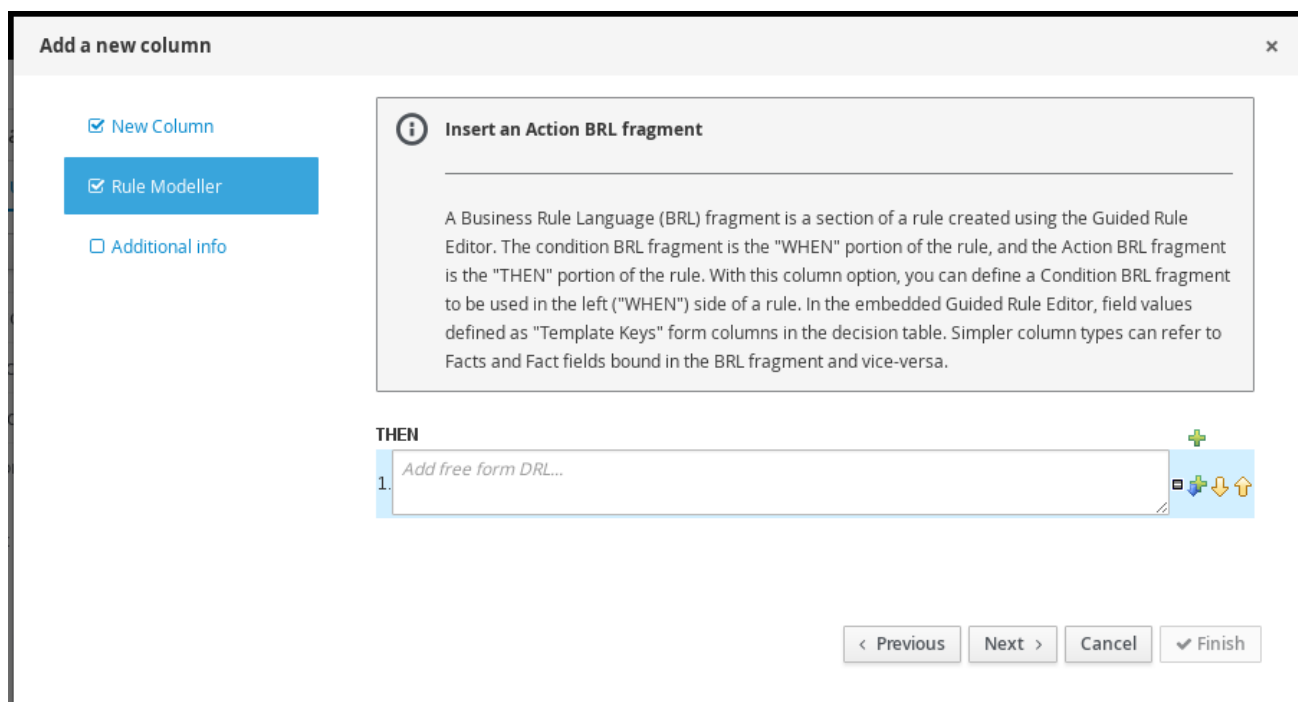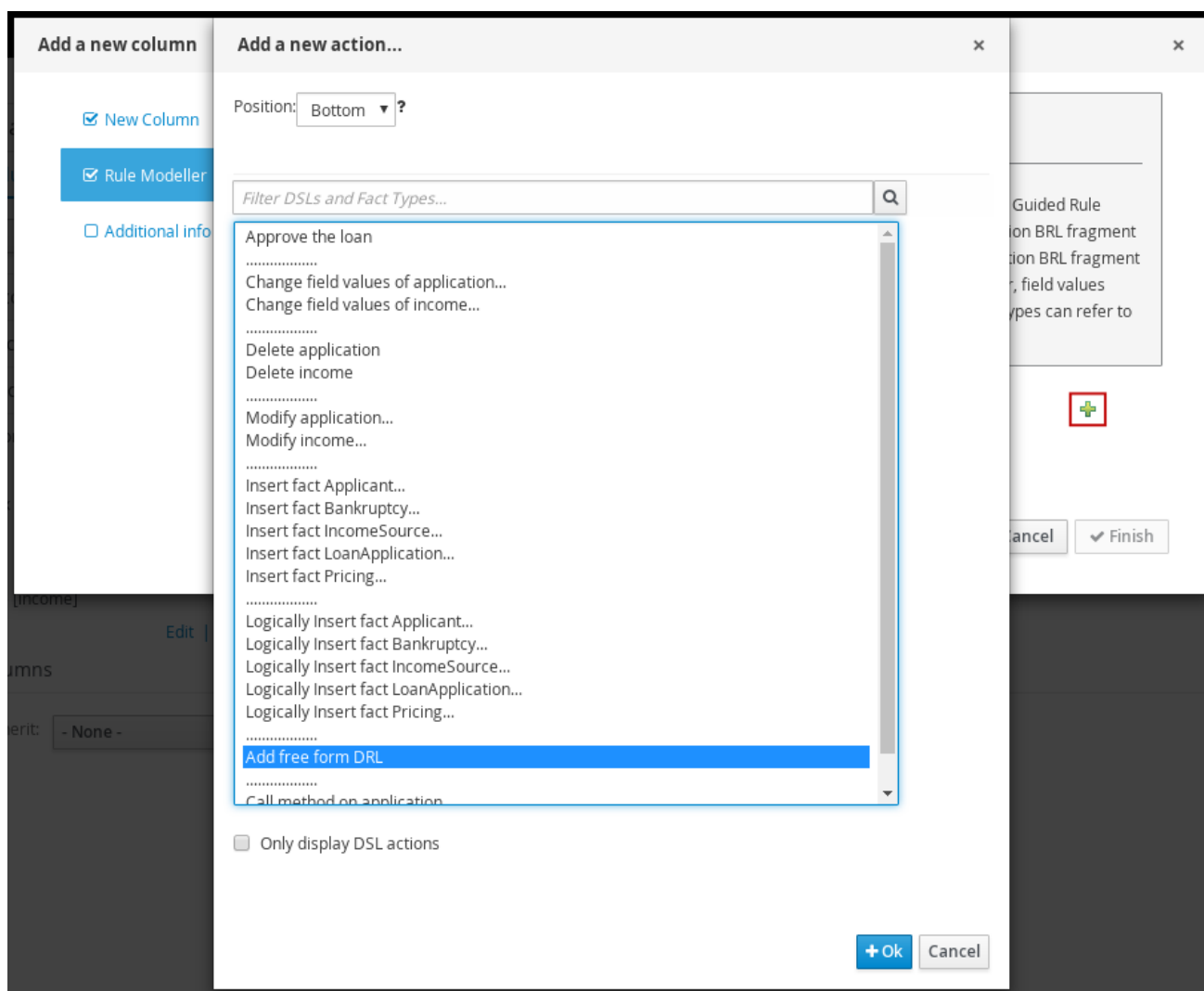
TEMPLATE KEYS

When you add a field for an action BRL fragment, one of the value options is **Template key** (as opposed to **Literal** or **Formula**). Template keys are placeholder variables that are interchanged with a specified value when the guided decision table is generated, and form separate columns in the table for each template key value specified. While Literal and Formula values are static in a decision table, Template key values can be modified as needed.

In the embedded guided rules designer, you can add a template key value to a field by selecting the **Template key** field option and entering the value in the editor in the format **$key**. For example, **$age** creates an **$age** column in the decision table.

In free form DRL, you can add a template key value to facts in the format **@{key}**. For example, **Person(age > @{age})** creates an **$age** column in the decision table.

The data type is String for new columns added using template keys.

Required column parameters

The following parameters are required in the **Add a new column** wizard to set up this column type:

- **Rule Modeller:** Define the action BRL fragment ("THEN" portion) for the rule.

- **Header (description):** Add header text for the column.

- **Hide column:** Select this to hide the column, or clear this to display the column.

7.5. "ADD AN ATTRIBUTE COLUMN"

With this column option, you can add one or more attribute columns representing any of the DRL rule attributes, such as Saliance, Enabled, Date-Effective, and others.

Example:

```
rule "Rule1"
salience 100 // This rule has the highest priority
when
  $i : IncomeSource( type == "Asset" )
then
  ...
end
```

For descriptions of each attribute, select the attribute from the list in the wizard.

HIT POLICIES AND ATTRIBUTES

Note that depending on the hit policy that you have defined for the decision table, some attributes may be disabled because they are internally used by the hit policy. For example, if you have assigned the **Resolved Hit** policy to this table so that rows (rules) are applied according to a priority order specified in the table, then the Salience attribute would be obsolete. The reason is that the Salience attribute escalates rule priority according to a defined salience value, and that value would be overridden by the **Resolved Hit** policy in the table.

Required Column Parameter

The following parameter is required in the **Add a new column** wizard to set up this column type:

- **Attribute:** Select the attribute to be applied to the column.

7.6. "DELETE AN EXISTING FACT"

With this column option, you can implement an action to delete a fact that was added previously as a fact pattern in the table. When this column is created, the fact types are provided in the table cells for that column as a drop-down list, from which users can select only one option.

Required column parameters

The following parameters are required in the **Add a new column** wizard to set up this column type:

- **Header (description):** Add header text for the column.

- **Hide column:** Select this to hide the column, or clear this to display the column.

7.7. "EXECUTE A WORK ITEM"

With this column option, you can execute a work item handler, based on your predefined work item definitions in Business Central. (You can create work items in **Menu → Design → Projects → [select project] → Add Asset → Work Item definition**.)

Required column parameters

The following parameters are required in the **Add a new column** wizard to set up this column type:

- **Work Item:** Select from the list of your predefined work items.

- **Header (description):** Add header text for the column.

- **Hide column:** Select this to hide the column, or clear this to display the column.

7.8. "SET THE VALUE OF A FIELD"

With this column option, you can implement an action to set the value of a field on a previously bound fact for the "THEN" portion of the rule. You have the option to notify the decision engine of the modified values which could lead to other rules being reactivated.

Required column parameters

The following parameters are required in the **Add a new column** wizard to set up this column type:

- **Pattern:** Select from the list of fact patterns already used in conditions or condition BRL fragments in your table or create a new fact pattern. A fact pattern is a combination of an available data object in the package (see the note on Required data objects for details) and a model class binding that you specify. (Examples: **LoanApplication [application]** or **IncomeSource [income]** where the bracketed portion is the binding to the given fact type)

- **Field:** Select a field from the previously specified fact pattern. The field options are defined in the fact file in the **Data Objects** panel of your project. (Examples: **amount** or **lengthYears** fields within the **LoanApplication** fact type)

- **Value list (optional):** Enter a list of value options, delimited by a comma and space, to limit table

input data for the action ("THEN") portion of the rule. When this value list is defined, the values will be provided in the table cells for that column as a drop-down list, from which users can select only one option. (Example list: **Accepted, Declined, Pending**)

- **Default value (optional):** Select one of the previously defined value options as the default value that will appear in the cell automatically in a new row. If the default value is not specified, the table cell will be blank by default. You can also select a default value from any previously configured data enumerations in the project, listed in the **Enumeration Definitions** panel of the Project Explorer. (You can create enumerations in **Menu → Design → Projects → *[select project]* → Add Asset → Enumeration**.)

- **Header (description):** Add header text for the column.

- **Hide column:** Select this to hide the column, or clear this to display the column.

- **Logically insert:** This option appears when the selected Fact Pattern is not currently used in another column in the guided decision table (see the next field description). Select this to insert the fact pattern logically into the decision engine, or clear this to insert it regularly. 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 asserted the facts in the first place are no longer true.

- **Update engine with changes:** This option appears when the selected Fact Pattern is already used in another column in the guided decision table. Select this to update the decision engine with the modified field values, or clear this to not update the decision engine.

7.9. "SET THE VALUE OF A FIELD WITH A WORK ITEM RESULT"

With this column option, you can implement an action to set the value of a previously defined fact field to the value of a result of a work item handler for the "THEN" portion of the rule. The work item must define a result parameter of the same data type as a field on a bound fact in order for you to set the field to the return parameter. (You can create work items in **Menu → Design → Projects → *[select project]* → Add Asset → Work Item definition**.)

An *Execute a Work Item* column must already be created in the table for this column option to be created.

Required column parameters

The following parameters are required in the **Add a new column** wizard to set up this column type:

- **Pattern:** Select from the list of fact patterns already used in your table or create a new fact pattern. A fact pattern is a combination of an available data object in the package (see the note on Required data objects for details) and a model class binding that you specify. (Examples: **LoanApplication [application]** or **IncomeSource [income]** where the bracketed portion is the binding to the given fact type)

- **Field:** Select a field from the previously specified fact pattern. The field options are defined in the fact file in the **Data Objects** panel of your project. (Examples: **amount** or **lengthYears** fields within the **LoanApplication** fact type)

- **Work Item:** Select from the list of your predefined work items. (The work item must define a result parameter of the same data type as a field on a bound fact in order for you to set the field to the return parameter.)

- **Header (description):** Add header text for the column.

- **Hide column:** Select this to hide the column, or clear this to display the column.

- **Logically insert:** This option appears when the selected Fact Pattern is not currently used in another column in the guided decision table (see the next field description). Select this to insert the fact pattern logically into the decision engine, or clear this to insert it regularly. 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 asserted the facts in the first place are no longer true.

- **Update engine with changes:** This option appears when the selected Fact Pattern is already used in another column in the guided decision table. Select this to update the decision engine with the modified field values, or clear this to not update the decision engine.

CHAPTER 8. EDITING OR DELETING COLUMNS IN GUIDED DECISION TABLES

You can edit or delete the columns you have created at any time in the guided decision tables designer.

Procedure

1. In the guided decision tables designer, click **Columns**.

2. Expand the appropriate section and click **Edit** or **Delete** next to the column name.

Figure 8.1. Edit or delete columns

NOTE

A condition column cannot be deleted if an existing action column uses the same pattern-matching parameters as the condition column.

3. After any column changes, click **Finish** in the wizard to save.

CHAPTER 9. ADDING ROWS AND DEFINING RULES IN GUIDED DECISION TABLES

After you have created your columns in the guided decision table, you can add rows and define rules within the guided decision tables designer.

Prerequisites

- Columns for the guided decision table have been added as described in Chapter 6, *Adding columns to guided decision tables*.

Procedure

1. In the guided decision tables designer, click **Insert → Append row** or one of the **Insert row** options. (You can also click **Insert column** to open the column wizard and define a new column.)

   Figure 9.1. Add Rows

2. Double-click each cell and enter data. For cells with specified values, select from the cell drop-down options.

   Figure 9.2. Enter input data in each cell

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

   > **NOTE**
   >
   > Although guided decision tables have real-time verification and validation, you should still manually validate the completed decision table to ensure optimal results.

4. Click **Save** in the table designer to save your changes.

CHAPTER 10. REAL-TIME VERIFICATION AND VALIDATION OF GUIDED DECISION TABLES

Business Central provides a real-time verification and validation feature for guided decision tables to ensure that your tables are complete and error free. Guided decision tables are validated after each cell change. If a problem in logic is detected, an error notification appears and describes the problem.

10.1. TYPES OF PROBLEMS IN GUIDED DECISION TABLES

The validation and verification feature detects the following types of problems:

Redundancy

> Redundancy occurs when two rows in a decision table execute the same consequences for the same set of facts. For example, two rows checking a client's birthday and providing a birthday discount may result in double discount.

Subsumption

> Subsumption is similar to redundancy and occurs when two rules execute the same consequences, but one executes on a subset of facts of the other. For example, consider these two rules:
>
> - *when Person age > 10 then Increase Counter*
>
> - *when Person age > 20 then Increase Counter*
>
> In this case, if a person is 15 years old, only one rule fires and if a person is 20 years old, both rules fire. Such cases cause similar trouble during runtime as redundancy.

Conflicts

> A conflicting situation occurs when two similar conditions have different consequences. Conflicts can occur between two rows (rules) or two cells in a decision table.
> The following example illustrates conflict between two rows in a decision table:
>
> - *when Deposit > 20000 then Approve Loan*
>
> - *when Deposit > 20000 then Refuse Loan*
>
> In this case, there is no way to know if the loan will be approved or not.
>
> The following example illustrates conflict between two cells in a decision table:
>
> - *when Age > 25*
>
> - *when Age < 25*
>
> A row with conflicting cells never executes.

Broken *Unique Hit* Policy

> When the **Unique Hit** policy is applied to a decision table, only one row at a time can be executed and each row must be unique, with no overlap of conditions being met. If more than one row is executed, then the verification report identifies the broken hit policy. For example, consider the following conditions in a table that determines eligibility for a price discount:
>
> - *when Is Student = true*

- *when Is Military = true*

If a customer is both a student and in the military, both conditions apply and break the **Unique Hit** policy. Rows in this type of table must therefore be created in a way that does not allow multiple rules to fire at one time. For details about hit policies, see Chapter 5, *Hit policies for guided decision tables*.

Deficiency

Deficiency is similar to a conflict and occurs the logic of a rule in a decision table is incomplete. For example, consider the following two deficient rules:

- *when Age > 20 then Approve Loan*

- *when Deposit < 20000 then Refuse Loan*

These two rules may lead to confusion for a person who is over 20 years old and has deposited less than 20000. You can add more constraints to avoid the conflict.

Missing Columns

When deleted columns result in incomplete or incorrect logic, rules cannot fire properly. This is detected so that you can address the missing columns, or adjust the logic to not rely on intentionally deleted conditions or actions.

Incomplete Ranges

Ranges of field values are incomplete if a table contains constraints against possible field values but does not define all possible values. The verification report identifies any incomplete ranges provided. For example, if your table has a check for if an application is approved, the verification report reminds you to make sure you also handle situations where the application was not approved.

10.2. TYPES OF NOTIFICATIONS

The verification and validation feature uses three types of notifications:

- Error: A serious problem that may lead to the guided decision table failing to work as designed at run time. Conflicts, for example, are reported as errors.

- Warning: Likely a serious problem that may not prevent the guided decision table from working but requires attention. Subsumptions, for example, are reported as warnings.

- Information: A moderate or minor problem that may not prevent the guided decision table from working but requires attention. Missing columns, for example, are reported as information.

> **NOTE**
>
> Business Central verification and validation does not prevent you from saving an incorrect change. The feature only reports issues while editing and you can still continue to overlook those and save your changes.

10.3. DISABLING VERIFICATION AND VALIDATION OF GUIDED DECISION TABLES

The decision table verification and validation feature of Business Central is enabled by default. You can disable it by setting the **org.kie.verification.disable-dtable-realtime-verification** system property value to **true** in your Red Hat JBoss EAP directory.

Procedure

Navigate to **$EAP_HOME/standalone/configuration/standalone-full.xml** and add the following system property:

```
<property name="org.kie.verification.disable-dtable-realtime-verification" value="true"/>
```

CHAPTER 11. 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 7.4 in a client application **pom.xml** file:

   ```
   <!-- For local execution -->
   <dependency>
     <groupId>org.kie</groupId>
     <artifactId>kie-ci</artifactId>
     <version>7.23.0.Final-redhat-00002</version>
   </dependency>

   <!-- For remote execution on Process Server -->
   <dependency>
     <groupId>org.kie.server</groupId>
     <artifactId>kie-server-client</artifactId>
     <version>7.23.0.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>7.4.0.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?.

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 desired), configure the **.java** class to import KIE services, a KIE container, and a KIE session, and then use the **main()** method to fire all rules against a defined fact model:

Executing rules locally

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

public class RulesTest {

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

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

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

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

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

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

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

Executing rules on Process Server

```
package com.sample;

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

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

import com.sample.Person;

public class RulesTest {

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

  public static final void main(String[] args) {
    try {
      // Define KIE services configuration and client:
      Set<Class<?>> allClasses = new HashSet<Class<?>>();
      allClasses.add(Person.class);
      String serverUrl = "http://$HOST:$PORT/kie-server/services/rest/server";
      String username = "$USERNAME";
      String password = "$PASSWORD";
```

```java
        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 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
```

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.

11.1. 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 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);
> ```

11.1.1. 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, 7.23.0.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>7.4.0.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 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*.

11.1.2. 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

```xml
<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, 7.23.0.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:
>
> ```xml
> <dependency>
>   <groupId>com.redhat.ba</groupId>
>   <artifactId>ba-platform-bom</artifactId>
>   <version>7.4.0.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:

```java
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 12. 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.