Drools

Drools User Guide

The Drools Team

Table of Contents

Chapter 1. Getting Started

1.1. First Rule Project

This guide walks you through the process of creating a simple Drools application project.

1.1.1. Prerequisites

- JDK 8+ with `JAVA_HOME` configured appropriately
- Apache Maven 3.8.1+
- Optionally, an IDE, such as IntelliJ IDEA, VSCode or Eclipse

1.1.2. Creating a project with maven archetype

Create a project with the following command.

```
mvn archetype:generate -DarchetypeGroupId=org.kie -DarchetypeArtifactId=kie-drools
-exec-model-ruleunit-archetype -DarchetypeVersion=8.24.0-SNAPSHOT
```

During the command execution, input property values interactively.

```
Define value for property 'groupId': org.example
Define value for property 'artifactId': my-project
Define value for property 'version' 1.0-SNAPSHOT: :
Define value for property 'package' org.example: :
...
 Y: : Y
...
[INFO] BUILD SUCCESS
```

Now your first rule project is created. Let's look into the project.

Firstly, `pom.xml`.

```xml
        <dependency>
          <groupId>org.drools</groupId>
          <artifactId>drools-engine</artifactId>
        </dependency>
        <dependency>
          <groupId>org.drools</groupId>
          <artifactId>drools-ruleunits-impl</artifactId>
        </dependency>
        <dependency>
          <groupId>org.drools</groupId>
          <artifactId>drools-model-compiler</artifactId>
        </dependency>
```

They are required dependencies for executable model with rule unit use cases.

> ℹ️ You can still use traditional Drools 7 style rules without rule unit. In this case, use `kie-drools-exec-model-archetype`.

The archetype contains one DRL file as an example `src/main/resources/org/example/rules.drl`.

```
package org.example;

unit MeasurementUnit;

rule "will execute per each Measurement having ID color"
when
    /measurements[ id == "color", $colorVal : val ]
then
    controlSet.add($colorVal);
end

query FindColor
    $m: /measurements[ id == "color" ]
end
```

This rule checks incoming `Measurement` data and stores its value in a global variable `controlSet` when it's color information.

`when` part implements the pattern matching and `then` part implements the action when the conditions are met.

Next, find `src/main/java/org/example/MeasurementUnit.java` that is specified as `unit MeasurementUnit` in the rule. It is called `rule unit` that groups data sources, global variables and DRL rules.

```
public class MeasurementUnit implements RuleUnitData {

    private final DataStore<Measurement> measurements;
    private final Set<String> controlSet = new HashSet<>();


    ...
```

`/measurements` in `rules.drl` is bound to the `measurements` field in `MeasurementUnit`. So you know that the inserted data type is `Measurement`. This class also defines a global variable `controlSet`.

Then, `src/main/java/org/example/Measurement.java` is a Java bean class used in the rule. Such an object is called `Fact`.

Finally, `src/test/java/org/example/RuleTest.java` is the test case that executes the rule. You can learn the basic API usage that is used in your own applications.

```
        MeasurementUnit measurementUnit = new MeasurementUnit();

        RuleUnitInstance<MeasurementUnit> instance = RuleUnitInstanceFactory.instance
  (measurementUnit);
```

Create a `MeasurementUnit` instance. Then create a `RuleUnitInstance` with the `MeasurementUnit` instance using `RuleUnitInstanceFactory`.

```
        measurementUnit.getMeasurements().add(new Measurement("color", "red"));
        measurementUnit.getMeasurements().add(new Measurement("color", "green"));
        measurementUnit.getMeasurements().add(new Measurement("color", "blue"));
```

Add `Measurement` facts into `measurementUnit.measurements`. It means the facts are inserted into Drools rule engine.

```
        List<Measurement> queryResult = instance.executeQuery("FindColor").stream()
  .map(tuple -> (Measurement) tuple.get("$m")).collect(toList());
```

Execute a query named `FindColor`. When you execute a query, rules that are matched with inserted facts are automatically fired. If you want to only fire rules without a query, you can call `instance.fire()` instead.

```
        instance.dispose();
```

At the end, call `dispose()` to release resources retained by the `RuleUnitInstance`.

Let's run the test with `mvn clean test`.

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running org.example.RuleTest
2022-06-13 12:49:56,499 [main] INFO  Creating RuleUnit
2022-06-13 12:49:56,696 [main] INFO  Insert data
2022-06-13 12:49:56,700 [main] INFO  Run query. Rules are also fired
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.411 s - in
org.example.RuleTest
```

Now you can add your own rules and facts to this project!

> The rule project requires code generation that is triggered by mvn compile phase.
> If you directly run `RuleTest.java` in IDE, you may need to run `mvn compile` first.

1.2. Getting started with decision services in Drools

As a business rules developer, you can use Drools to design a variety of decision services. This document describes how to create and test an example traffic violation project based on the **Traffic_Violation** sample project available from the [Kogito Examples GitHub repository](#). This sample project uses a Decision Model and Notation (DMN) model to define driver penalty and suspension rules in a traffic violation decision service. You can follow the steps in this document to create the project and the assets it contains, or open and review the existing **Traffic_Violation** sample project. Alternatively, you may directly skip to the project evaluation by checking out locally the ready-made **Traffic_Violation** sample project from the link above, and jumping to the [Evaluating the traffic violations DMN model with Drools](#) section.

For more information about the DMN components and implementation in Drools, see [DMN Engine in Drools section](#).

Prerequisites

- Having identified the target deployment platform, such as using Drools as an embedded Java library or Kogito for cloud-native platform
- Having satisfied prerequisites of the target deployment platform
- Considering deployment procedures of the target deployment platform while following this tutorial

1.2.1. Creating the traffic violations project from scratch

For this example, create a new project called `traffic-violation` in your IDE. A project is a container for assets such as data objects, DMN assets, and test scenarios. This example project that you are creating is similar to the existing **Traffic_Violation** sample project.

Procedure

1. Follow the instruction depending on your IDE to create a new KJAR based Maven project.

Decision Model and Notation (DMN)

Decision Model and Notation (DMN) is a standard established by the Object Management Group (OMG) for describing and modeling operational decisions. DMN defines an XML schema that enables DMN models to be shared between DMN-compliant platforms and across organizations so that business analysts and business rules developers can collaborate in designing and implementing DMN decision services. The DMN standard is similar to and can be used together with the Business Process Model and Notation (BPMN) standard for designing and modeling business processes.

For general information about the background and applications of DMN, see the Drools DMN landing page.

Creating the traffic violations DMN decision requirements diagram (DRD)

A decision requirements diagram (DRD) is a visual representation of your DMN model. Use the KIE DMN Editor to design the DRD for the traffic violations project and to define the decision logic of the DRD components.
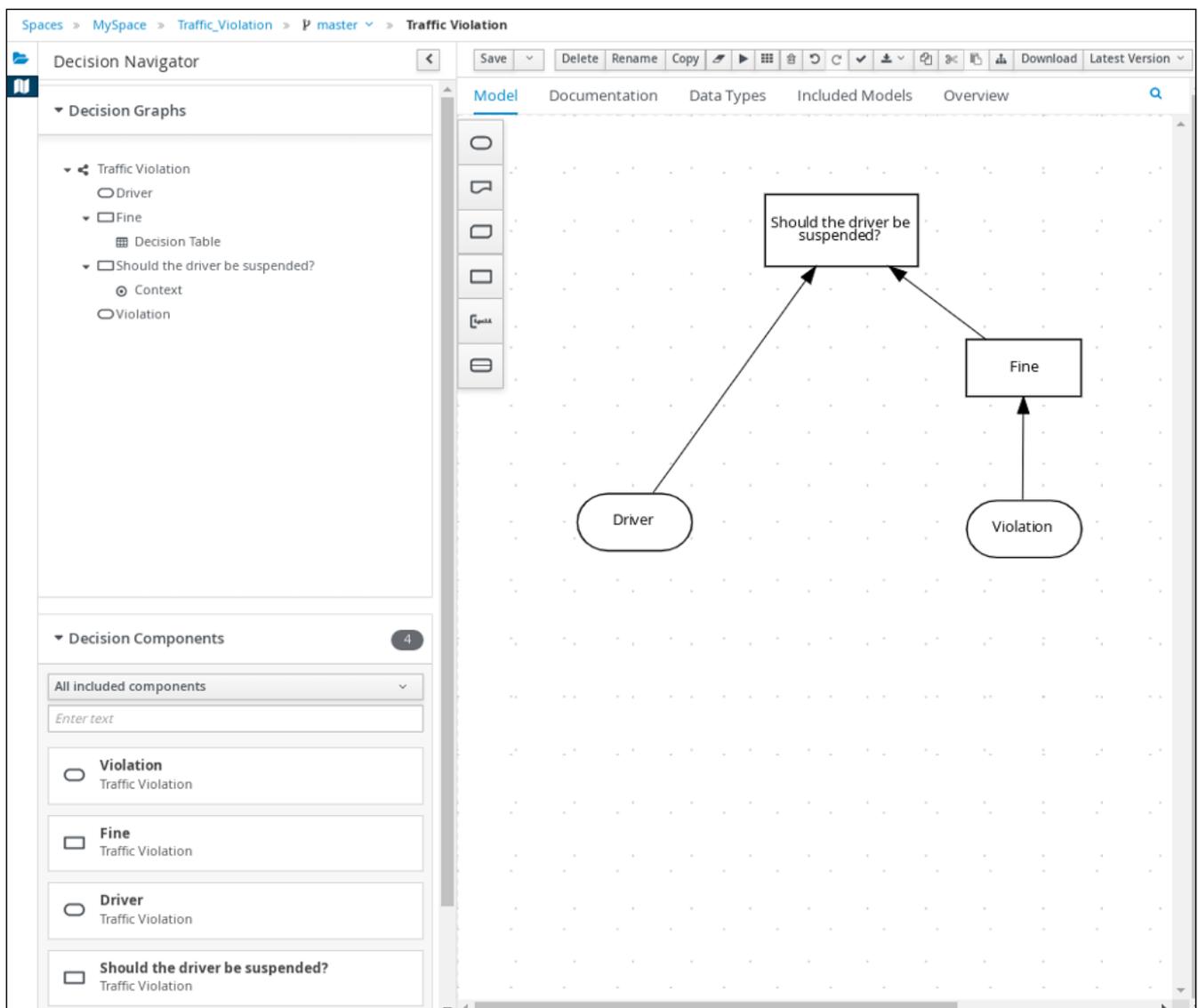
Figure 1. DRD for the Traffic Violations example

Prerequisites

- You have created the traffic violations project in Drools.

Procedure

1. On the **traffic-violation** project's home page, click **Add Asset**.

2. On the **Add Asset** page, click **DMN**. The **Create new DMN** window is opened.

   a. In the **Create new DMN** window, enter `Traffic Violation` in the **DMN** name field.

   b. From the **Package** list, select `com.myspace.traffic_violation`.

   c. Click **Ok**. The DMN asset in the DMN designer is opened.

3. In the DMN designer canvas, drag two **DMN Input Data** input nodes onto the canvas.
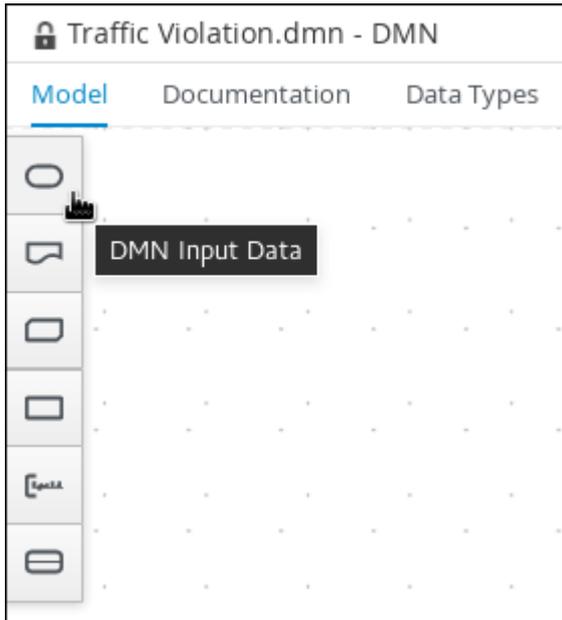
Figure 2. DMN Input Data nodes

4. In the upper-right corner, click the [Properties] icon.

5. Double-click the input nodes and rename one to `Driver` and the other to `Violation`.

6. Drag a **DMN Decision** decision node onto the canvas.

7. Double-click the decision node and rename it to `Fine`.

8. Click the **Violation** input node, select the **Create DMN Information Requirement** icon and click the `Fine` decision node to link the two nodes.
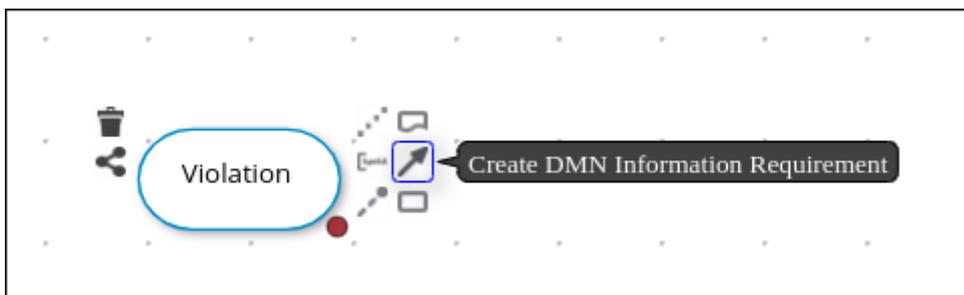
Figure 3. Create DMN Information Requirement icon

9. Drag a **DMN Decision** decision node onto the canvas.

10. Double-click the decision node and rename it to `Should the driver be suspended?`.

11. Click the **Driver** input node, select the **Create DMN Information Requirement** icon and click the **Should the driver be suspended?** decision node to link the two nodes.

12. Click the **Fine** decision node, select the **Create DMN Information Requirement** icon, and select the **Should the driver be suspended?** decision node.

13. Click **Save**.

> ℹ️ As you periodically save a DRD, the DMN designer performs a static validation of the DMN model and might produce error messages until the model is defined completely. After you finish defining the DMN model completely, if any errors remain, troubleshoot the specified problems accordingly.

Creating the traffic violations DMN custom data types

DMN data types determine the structure of the data that you use within a table, column, or field in a DMN boxed expression for defining decision logic. You can use default DMN data types (such as string, number, or boolean) or you can create custom data types to specify additional fields and constraints that you want to implement for the boxed expression values. Use the KIE DMN Editor's **Data Types** tab to define the custom data types for the traffic violations project.
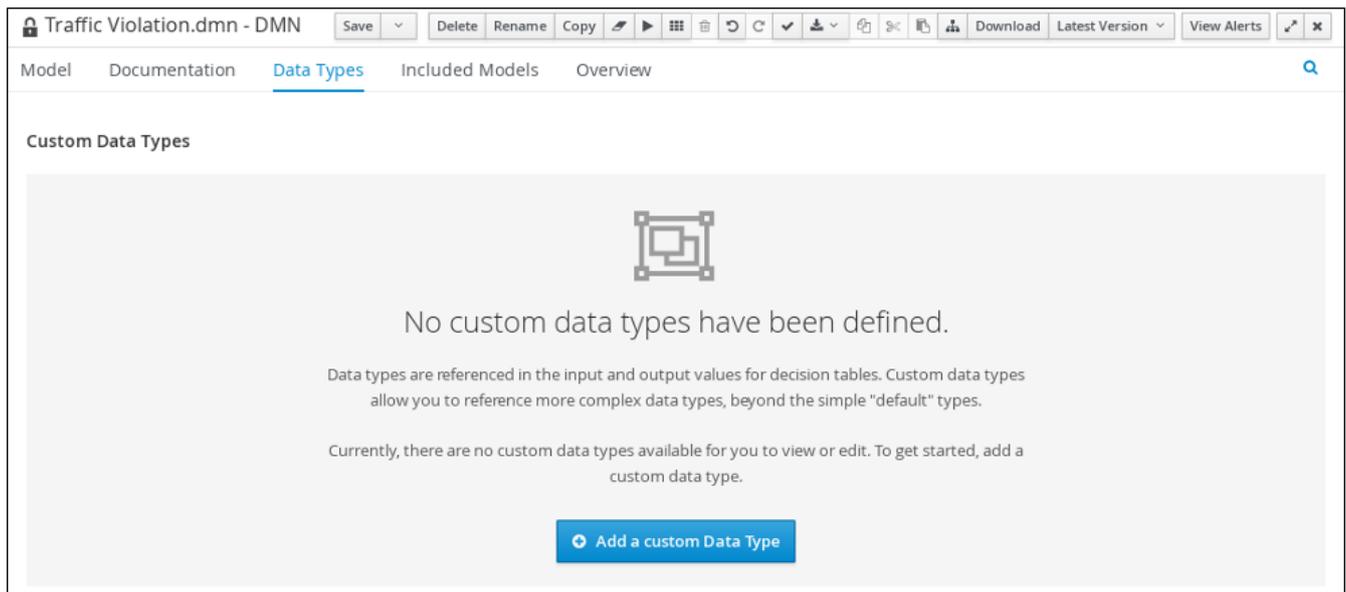
Figure 4. The custom data types tab

The following tables list the `tDriver`, `tViolation`, and `tFine` custom data types that you will create for this project.

Table 1. `tDriver` custom data type

Name	Type
tDriver	Structure
Name	string
Age	number
State	string

Name	Type
City	string
Points	number

Table 2. `tViolation` custom data type

Name	Type
tViolation	Structure
Code	string
Date	date
Type	string
Speed Limit	number
Actual Speed	number

Table 3. `tFine` custom data type

Name	Type
tFine	Structure
Amount	number
Points	number

Prerequisites

- You created the traffic violations DMN decision requirements diagram (DRDs) using KIE DMN Editor.

Procedure

1. To create the `tDriver` custom data type, click **Add a custom Data Type** on the **Data Types** tab, enter `tDriver` in the **Name** field, and select `Structure` from the **Type** list.

2. Click the check mark to the right of the new data type to save your changes.
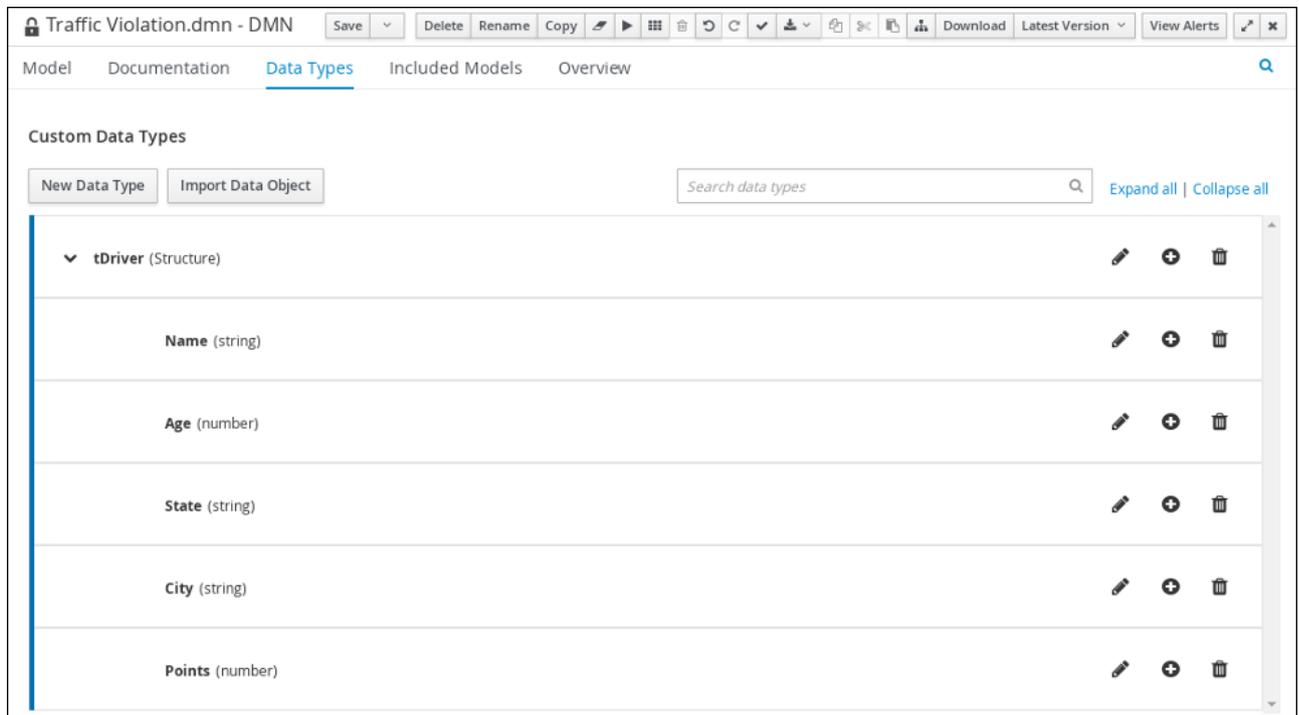
Figure 5. The tDriver custom data type

3. Add each of the following nested data types to the `tDriver` structured data type by clicking the plus sign next to `tDriver` for each new nested data type. Click the check mark to the right of each new data type to save your changes.

   - `Name` (string)

   - `Age` (number)

   - `State` (string)

   - `City` (string)

   - `Points` (number)

4. To create the `tViolation` custom data type, click **New Data Type**, enter `tViolation` in the **Name** field, and select `Structure` from the **Type** list.

5. Click the check mark to the right of the new data type to save your changes.
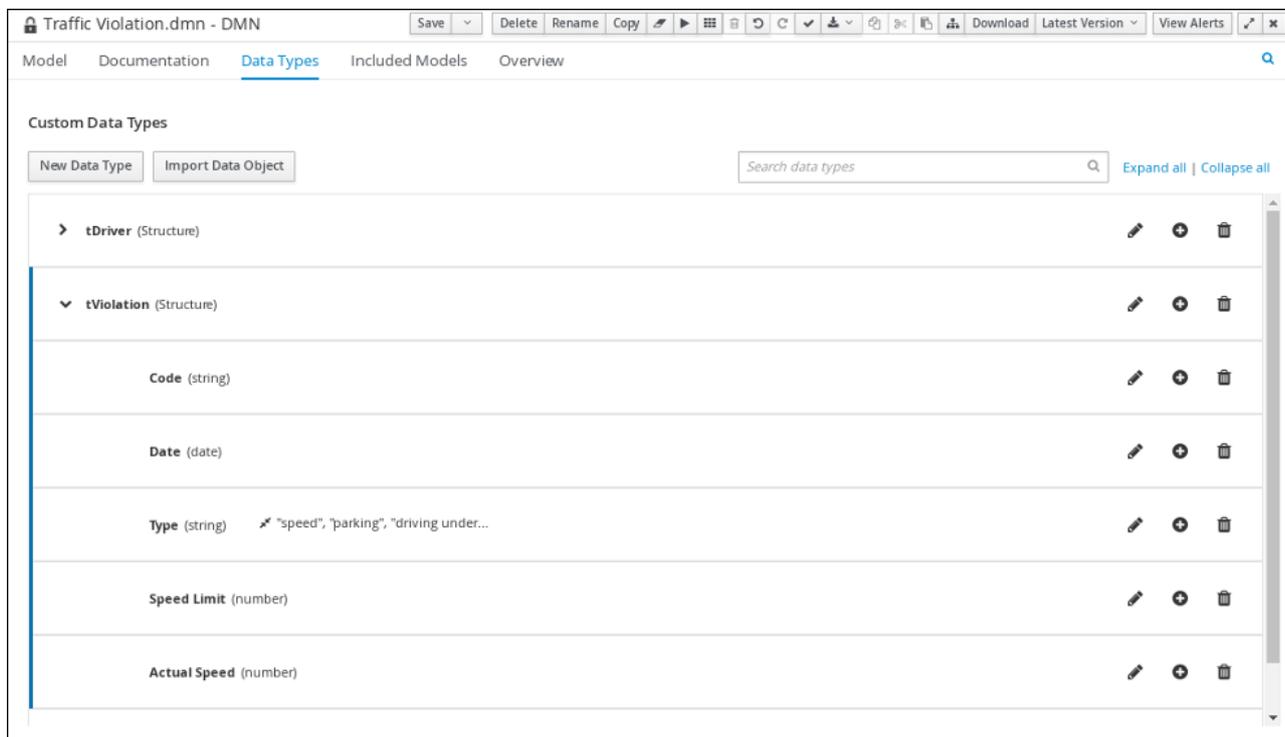
Figure 6. The tViolation custom data type

6. Add each of the following nested data types to the `tViolation` structured data type by clicking the plus sign next to `tViolation` for each new nested data type. Click the check mark to the right of each new data type to save your changes.

   ◦ `Code` (string)

   ◦ `Date` (date)

   ◦ `Type` (string)

   ◦ `Speed Limit` (number)

   ◦ `Actual Speed` (number)

7. To add the following constraints to the `Type` nested data type, click the edit icon, click **Add Constraints**, and select **Enumeration** from the **Select constraint type** drop-down menu.

   ◦ `speed`

   ◦ `parking`

   ◦ `driving under the influence`

8. Click **OK**, then click the check mark to the right of the **Type** data type to save your changes.

9. To create the `tFine` custom data type, click **New Data Type**, enter `tFine` in the **Name** field, select `Structure` from the **Type** list, and click **Save**.
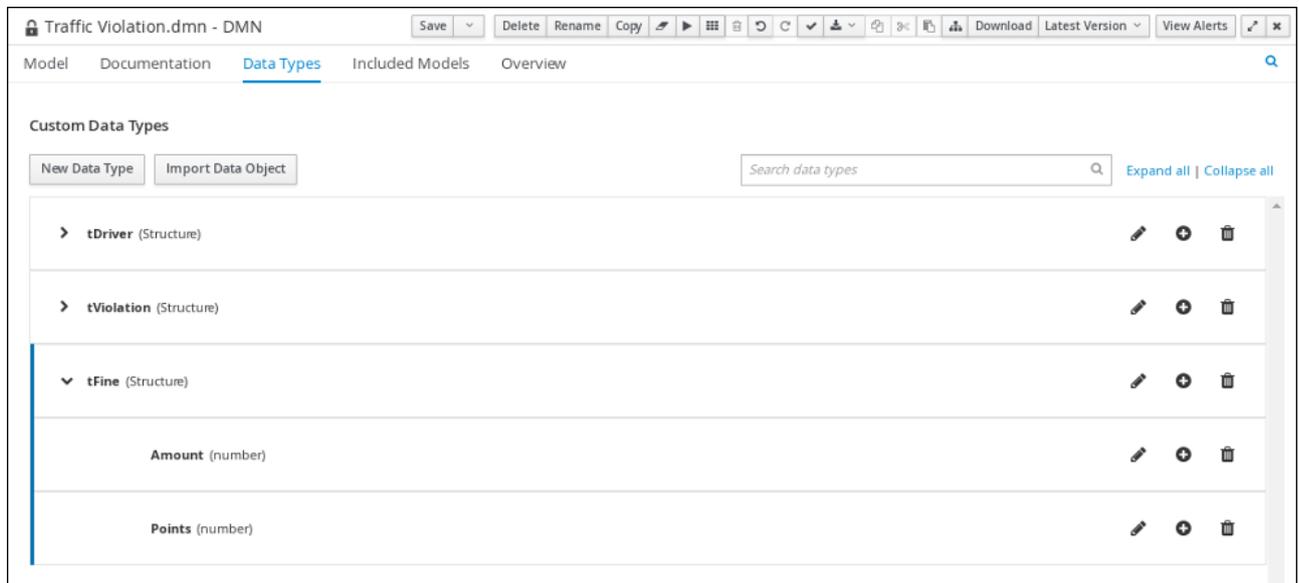
Figure 7. The tFine custom data type

10. Add each of the following nested data types to the `tFine` structured data type by clicking the plus sign next to `tFine` for each new nested data type. Click the check mark to the right of each new data type to save your changes.

    - `Amount` (number)

    - `Points` (number)

11. Click **Save**.

Assigning custom data types to the DRD input and decision nodes

After you create the DMN custom data types, assign them to the appropriate **DMN Input Data** and **DMN Decision** nodes in the traffic violations DRD.

Prerequisites

- You have created the traffic violations DMN custom data types in KIE DMN Editor.

Procedure

1. Click the **Model** tab on the DMN designer and click the **Properties** [diagram properties] icon in the upper-right corner of the DMN designer to expose the DRD properties.

2. In the DRD, select the **Driver** input data node and in the **Properties** panel, select `tDriver` from the **Data type** drop-down menu.

3. Select the **Violation** input data node and select `tViolation` from the **Data type** drop-down menu.

4. Select the **Fine** decision node and select `tFine` from the **Data type** drop-down menu.

5. Select the **Should the driver be suspended?** decision node and set the following properties:

    - **Data type**: `string`

    - **Question**: `Should the driver be suspended due to points on his driver license?`

    - **Allowed Answers**: `Yes,No`

6. Click **Save**.

You have assigned the custom data types to your DRD's input and decision nodes.

Defining the traffic violations DMN decision logic

To calculate the fine and to decide whether the driver is to be suspended or not, you can define the traffic violations DMN decision logic using a DMN decision table and context boxed expression.

Figure 8. Fine expression

Figure 9. Should the driver be suspended expression

Prerequisites

- You have assigned the DMN custom data types to the appropriate decision and input nodes in the traffic violations DRD in KIE DMN Editor.

Procedure

1. To calculate the fine, in the DMN designer canvas, select the **Fine** decision node and click the **Edit** icon to open the DMN boxed expression designer.
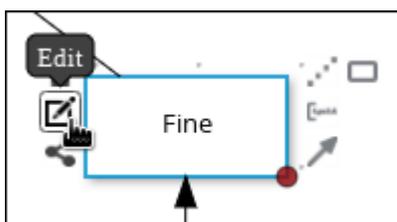
Figure 10. Decision node edit icon

2. Click **Select expression → Decision Table**.
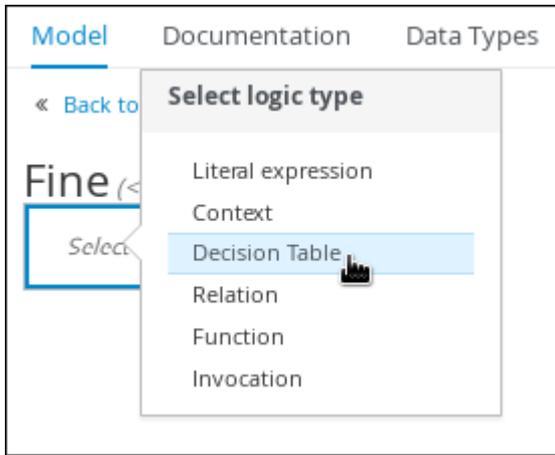
Figure 11. Select Decisiong Table logic type

3. For the **Violation.Date**, **Violation.Code**, and **Violation.Speed Limit** columns, right-click and select **Delete** for each field.

4. Click the **Violation.Actual Speed** column header and enter the expression `Violation.Actual Speed - Violation.Speed Limit` in the **Expression** field."

5. Enter the following values in the first row of the decision table:

   ◦ **Violation.Type**: `"speed"`

   ◦ **Violation.Actual Speed - Violation.Speed Limit**: `[10..30)`

   ◦ **Amount**: `500`

   ◦ **Points**: `3`

   Right-click the first row and select `Insert below` to add another row.

6. Enter the following values in the second row of the decision table:

   ◦ **Violation.Type**: `"speed"`

   ◦ **Violation.Actual Speed - Violation.Speed Limit**: `>= 30`

   ◦ **Amount**: `1000`

   ◦ **Points**: `7`

   Right-click the second row and select `Insert below` to add another row.

7. Enter the following values in the third row of the decision table:

   ◦ **Violation.Type**: `"parking"`

   ◦ **Violation.Actual Speed - Violation.Speed Limit**: `-`

   ◦ **Amount**: `100`

   ◦ **Points**: `1`

   Right-click the third row and select `Insert below` to add another row.

8. Enter the following values in the fourth row of the decision table:

   ◦ **Violation.Type**: `"driving under the influence"`

- **Violation.Actual Speed - Violation.Speed Limit**: -

- **Amount**: 1000

- **Points**: 5

9. Click **Save**.

10. To define the driver suspension rule, return to the DMN designer canvas, select the **Should the driver be suspended?** decision node, and click the **Edit** icon to open the DMN boxed expression designer.

11. Click **Select expression → Context**.

12. Click **ContextEntry-1**, enter `Total Points` as the **Name**, and select `number` from the **Data Type** drop-down menu.

13. Click the cell next to **Total Points**, select `Literal expression` from the context menu, and enter `Driver.Points + Fine.Points` as the expression.

14. In the cell below **Driver.Points** + **Fine.Points**, select `Literal Expression` from the context menu, and enter `if Total Points >= 20 then "Yes" else "No"`.

15. Click **Save**.

You have defined how to calculate the fine and the context for deciding when to suspend the driver. You can navigate to the **traffic-violation** project page and click **Build** to build the example project and address any errors noted in the **Alerts** panel.

Test scenarios

Test scenarios in Drools enable you to validate the functionality of business rules and business rule data (for rules-based test scenarios) or of DMN models (for DMN-based test scenarios) before deploying them into a production environment. With a test scenario, you use data from your project to set given conditions and expected results based on one or more defined business rules. When you run the scenario, the expected results and actual results of the rule instance are compared. If the expected results match the actual results, the test is successful. If the expected results do not match the actual results, then the test fails.

You can run the defined test scenarios in a number of ways, for example, you can run available test scenarios at the project level or inside a specific test scenario asset. Test scenarios are independent and cannot affect or modify other test scenarios. You can run test scenarios at any time during project development in Drools.

You can import data objects from different packages to the same project package as the test scenario. Assets in the same package are imported by default. After you create the necessary data objects and the test scenario, you can use the **Data Objects** tab of the test scenarios designer to verify that all required data objects are listed or to import other existing data objects by adding a **New item**.

Testing the traffic violations using test scenarios

Use the test scenarios designer to test the DMN decision requirements diagrams (DRDs) and define decision logic for the traffic violations project.

Figure 12. Test scenario for the traffic violations example

Prerequisites

- You have successfully built the traffic violations project using Drools.

Procedure

1. Create a new Test Scenario file in your IDE.

   a. Enter `Violation Scenarios` in the **Test Scenario** field.

   b. From the **Package** list, select `com.myspace.traffic_violation`.

   c. Select `DMN` as the **Source type**.

   d. From the **Choose a DMN asset** list, select the path to the DMN asset.

   e. Click **Ok** to open the **Violation Scenarios** test scenario in the **Test Scenarios** designer.

2. Under **Driver** column sub-header, right-click the **State**, **City**, **Age**, and **Name** value cells and select **Delete column** from the context menu options to remove them.

3. Under **Violation** column sub-header, right-click the **Date** and **Code** value cells and select **Delete column** to remove them.

4. Enter the following information in the first row of the test scenarios:

   - **Scenario description**: `Above speed limit: 10km/h and 30 km/h`
   - **Points** (under **Given** column header): `10`
   - **Type**: `"speed"`
   - **Speed Limit**: `100`
   - **Actual Speed**: `120`
   - **Points**: `3`
   - **Amount**: `500`
   - **Should the driver be suspended?**: `"No"`

   Right-click the first row and select **Insert row below** to add another row.

5. Enter the following information in the second row of the test scenarios:

   - **Scenario description**: `Above speed limit: more than 30 km/h`
   - **Points** (under **Given** column header): `10`
   - **Type**: `"speed"`
   - **Speed Limit**: `100`

- **Actual Speed**: 150
- **Points**: 7
- **Amount**: 1000
- **Should the driver be suspended?**: "No"

  Right-click the second row and select **Insert row below** to add another row.

6. Enter the following information in the third row of the test scenarios:

   - **Scenario description**: Parking violation
   - **Points** (under **Given** column header): 10
   - **Type**: "parking"
   - **Speed Limit**: leave blank
   - **Actual Speed**: leave blank
   - **Points**: 1
   - **Amount**: 100
   - **Should the driver be suspended?**: "No"

     Right-click the third row and select **Insert row below** to add another row.

7. Enter the following information in the fourth row of the test scenarios:

   - **Scenario description**: DUI violation
   - **Points** (under **Given** column header): 10
   - **Type**: "driving under the influence"
   - **Speed Limit**: leave blank
   - **Actual Speed**: leave blank
   - **Points**: 5
   - **Amount**: 1000
   - **Should the driver be suspended?**: "No"

     Right-click the fourth row and select **Insert row below** to add another row.

8. Enter the following information in the fifth row of the test scenarios:

   - **Scenario description**: Driver suspended
   - **Points** (under **Given** column header): 15
   - **Type**: "speed"
   - **Speed Limit**: 100
   - **Actual Speed**: 140
   - **Points**: 7
   - **Amount**: 1000

- **Should the driver be suspended?**: `"Yes"`

9. Click **Save**.

10. Click the **Play** icon ▶ to check whether the test scenarios pass or fail.
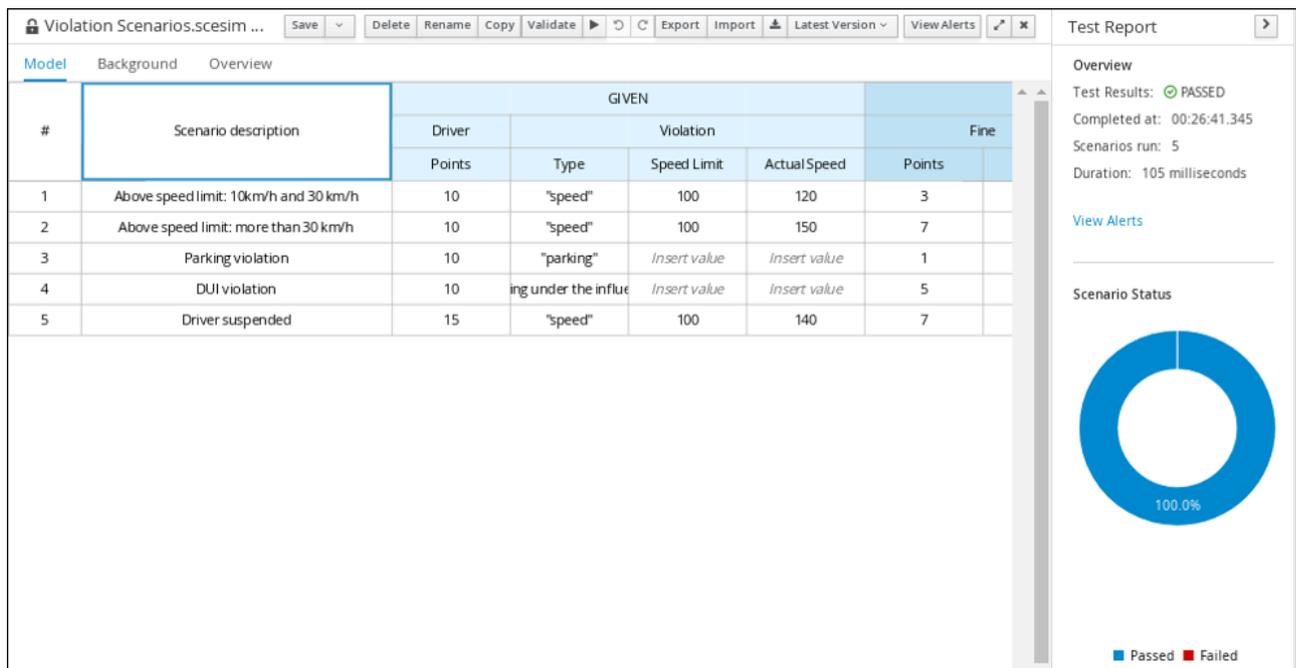
Figure 13. Test scenario execution result for the traffic violations example

In case of failure, correct the errors and run the test scenarios again.

1.2.2. Evaluating the traffic violations DMN model with Drools

The instructions to follow to evaluate the traffic violation DMN model depends on the chosen target deployment platform from the prerequisite.

For more details about available options to evaluate DMN models depending on the target deployment platform, make reference to: DMN model execution.

Example: Kogito as the target deployment platform

The following examples assumes Kogito is the chosen target deployment platform.

The source code and detailed runtime instructions for the **Traffic_Violation** sample project are readily made available for Kogito from the Kogito Examples GitHub repository.

Procedure

1. Determine the base URL for accessing the REST API endpoints. This requires knowing the following values (with the default local deployment values as an example):

   - Host (`localhost`)

   - Port (`8080`)

   - REST path (none specific)

   Example base URL in local deployment for the traffic violations project:

```
http://localhost:8080/Traffic Violation
```

2. Determine user authentication requirements.

   If users and roles are configured on the Kogito on Quarkus application, HTTP Basic authentication may be required with user name and password. Successful requests require that the user have the configured role.

   The following example demonstrates how to add credentials to a curl request:

   ```
   curl -u username:password <request>
   ```

   If the Kogito on Quarkus application is configured with {RH-SSO}, the request must include a bearer token:

   ```
   curl -H "Authorization: bearer $TOKEN" <request>
   ```

3. Execute the DMN model:

   [POST] `Traffic%20Violation`

   Example curl request:

   ```
   curl -L -X POST 'localhost:8080/Traffic Violation' \
   -H 'Content-Type: application/json' \
   -H 'Accept: application/json' \
   --data-raw '{
       "Driver": {
           "Points": 2
       },
       "Violation": {
           "Type": "speed",
           "Actual Speed": 120,
           "Speed Limit": 100
       }
   }'
   ```

   Example JSON request:

```json
{
    "Driver": {
        "Points": 2
    },
    "Violation": {
        "Type": "speed",
        "Actual Speed": 120,
        "Speed Limit": 100
    }
}
```

Example JSON response:

```json
{
    "Driver": {
        "Points": 2
    },
    "Violation": {
        "Type": "speed",
        "Actual Speed": 120,
        "Speed Limit": 100
    },
    "Fine": {
        "Points": 3,
        "Amount": 500
    },
    "Should the driver be suspended?": "No"
}
```

Chapter 2. Drools rule engine

The Drools rule engine stores, processes, and evaluates data to execute the business rules or decision models that you define. The basic function of the Drools rule engine is to match incoming data, or *facts*, to the conditions of rules and determine whether and how to execute the rules.

The Drools rule engine operates using the following basic components:

- **Rules:** Business rules or DMN decisions that you define. All rules must contain at a minimum the conditions that trigger the rule and the actions that the rule dictates.

- **Facts:** Data that enters or changes in the Drools rule engine that the Drools rule engine matches to rule conditions to execute applicable rules.

- **Production memory:** Location where rules are stored in the Drools rule engine.

- **Working memory:** Location where facts are stored in the Drools rule engine.

- **Agenda:** Location where activated rules are registered and sorted (if applicable) in preparation for execution.

When a business user or an automated system adds or updates rule-related information in Drools, that information is inserted into the working memory of the Drools rule engine in the form of one or more facts. The Drools rule engine matches those facts to the conditions of the rules that are stored in the production memory to determine eligible rule executions. (This process of matching facts to rules is often referred to as *pattern matching.*) When rule conditions are met, the Drools rule engine activates and registers rules in the agenda, where the Drools rule engine then sorts prioritized or conflicting rules in preparation for execution.

The following diagram illustrates these basic components of the Drools rule engine:
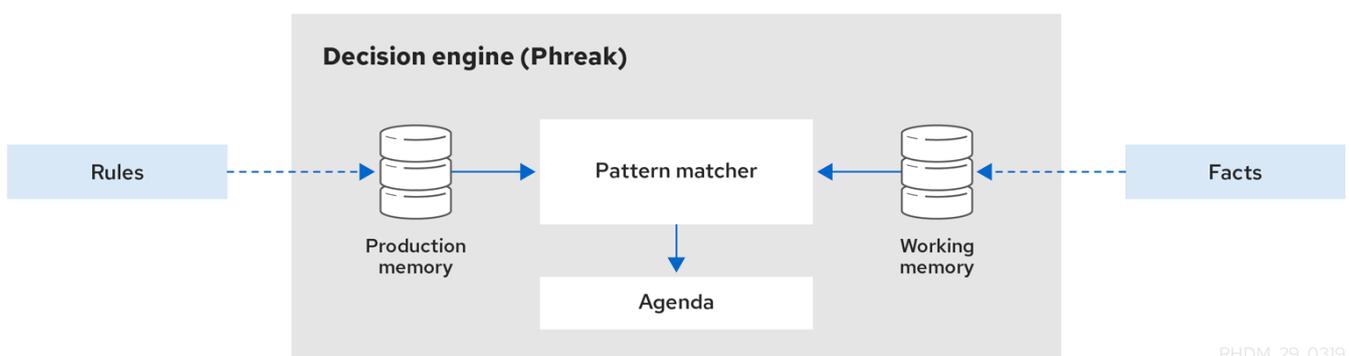
Figure 14. Overview of basic Drools rule engine components

For more details and examples of rule and fact behavior in the Drools rule engine, see Inference and truth maintenance in the Drools rule engine.

These core concepts can help you to better understand other more advanced components, processes, and sub-processes of the Drools rule engine, and as a result, to design more effective business assets in Drools.

2.1. KIE sessions

In Drools, a KIE session stores and executes runtime data. The KIE session is created from a KIE base or directly from a KIE container if you have defined the KIE session in the KIE module descriptor file (`kmodule.xml`) for your project.

Example KIE session configuration in a `kmodule.xml` file

```
<kmodule>
  ...
  <kbase>
    ...
    <ksession name="KSession2_1" type="stateless" default="true" clockType="realtime">
      ...
  </kbase>
  ...
</kmodule>
```

A KIE base is a repository that you define in the KIE module descriptor file (`kmodule.xml`) for your project and contains all in Drools, but does not contain any runtime data.

Example KIE base configuration in a `kmodule.xml` file

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="stream" equalsBehavior=
"equality" declarativeAgenda="enabled" packages="org.domain.pkg2, org.domain.pkg3"
includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

A KIE session can be stateless or stateful. In a stateless KIE session, data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations. In a stateful KIE session, that data is retained. The type of KIE session you use depends on your project requirements and how you want data from different asset invocations to be persisted.

2.1.1. Stateless KIE sessions

A stateless KIE session is a session that does not use inference to make iterative changes to facts over time. In a stateless KIE session, data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations, whereas in a stateful KIE session, that data is retained. A stateless KIE session behaves similarly to a function in that the results that it produces are determined by the contents of the KIE base and by the data that is passed into the KIE session for execution at a specific point in time. The KIE session has no memory of any data that was passed into the KIE session previously.

Stateless KIE sessions are commonly used for the following use cases:

- **Validation**, such as validating that a person is eligible for a mortgage

- **Calculation**, such as computing a mortgage premium

- **Routing and filtering**, such as sorting incoming emails into folders or sending incoming emails to a destination

For example, consider the following driver's license data model and sample DRL rule:

Data model for driver's license application

```
public class Applicant {
    private String name;
    private int age;
    private boolean valid;
    // Getter and setter methods
}
```

Sample DRL rule for driver's license application

```
package com.company.license

rule "Is of valid age"
when
    $a : Applicant(age < 18)
then
    $a.setValid(false);
end
```

The `Is of valid age` rule disqualifies any applicant younger than 18 years old. When the `Applicant` object is inserted into the Drools rule engine, the Drools rule engine evaluates the constraints for each rule and searches for a match. The `"objectType"` constraint is always implied, after which any number of explicit field constraints are evaluated. The variable `$a` is a binding variable that references the matched object in the rule consequence.

> ℹ️ The dollar sign (`$`) is optional and helps to differentiate between variable names and field names.

In this example, the sample rule and all other files in the `~/resources` folder of the Drools project are built with the following code:

Create the KIE container

```
KieServices kieServices = KieServices.Factory.get();

KieContainer kContainer = kieServices.getKieClasspathContainer();
```

This code compiles all the rule files found on the class path and adds the result of this compilation, a `KieModule` object, in the `KieContainer`.

Finally, the `StatelessKieSession` object is instantiated from the `KieContainer` and is executed against specified data:

Instantiate the stateless KIE session and enter data

```
StatelessKieSession kSession = kContainer.newStatelessKieSession();

Applicant applicant = new Applicant("Mr John Smith", 16);

assertTrue(applicant.isValid());

ksession.execute(applicant);

assertFalse(applicant.isValid());
```

In a stateless KIE session configuration, the `execute()` call acts as a combination method that instantiates the `KieSession` object, adds all the user data and executes user commands, calls `fireAllRules()`, and then calls `dispose()`. Therefore, with a stateless KIE session, you do not need to call `fireAllRules()` or call `dispose()` after session invocation as you do with a stateful KIE session.

In this case, the specified applicant is under the age of 18, so the application is declined.

For a more complex use case, see the following example. This example uses a stateless KIE session and executes rules against an iterable list of objects, such as a collection.

Expanded data model for driver's license application

```
public class Applicant {
  private String name;
  private int age;
  // Getter and setter methods
}

public class Application {
  private Date dateApplied;
  private boolean valid;
  // Getter and setter methods
}
```

Expanded DRL rule set for driver's license application

```
package com.company.license

rule "Is of valid age"
when
  Applicant(age < 18)
  $a : Application()
then
  $a.setValid(false);
end

rule "Application was made this year"
when
  $a : Application(dateApplied > "01-jan-2009")
then
  $a.setValid(false);
end
```

Expanded Java source with iterable execution in a stateless KIE session

```
StatelessKieSession ksession = kbase.newStatelessKnowledgeSession();
Applicant applicant = new Applicant("Mr John Smith", 16);
Application application = new Application();

assertTrue(application.isValid());
ksession.execute(Arrays.asList(new Object[] { application, applicant }));   ①
assertFalse(application.isValid());

ksession.execute
  (CommandFactory.newInsertIterable(new Object[] { application, applicant }));   ②

List<Command> cmds = new ArrayList<Command>();   ③
cmds.add(CommandFactory.newInsert(new Person("Mr John Smith"), "mrSmith"));
cmds.add(CommandFactory.newInsert(new Person("Mr John Doe"), "mrDoe"));

BatchExecutionResults results = ksession.execute(CommandFactory.newBatchExecution(
cmds));
assertEquals(new Person("Mr John Smith"), results.getValue("mrSmith"));
```

① Method for executing rules against an iterable collection of objects produced by the `Arrays.asList()` method. Every collection element is inserted before any matched rules are executed. The `execute(Object object)` and `execute(Iterable objects)` methods are wrappers around the `execute(Command command)` method that comes from the `BatchExecutor` interface.

② Execution of the iterable collection of objects using the `CommandFactory` interface.

③ `BatchExecutor` and `CommandFactory` configurations for working with many different commands or result output identifiers. The `CommandFactory` interface supports other commands that you can use in the `BatchExecutor`, such as `StartProcess`, `Query`, and `SetGlobal`.

2.1.1.1. Global variables in stateless KIE sessions

The `StatelessKieSession` object supports global variables (globals) that you can configure to be resolved as session-scoped globals, delegate globals, or execution-scoped globals.

- **Session-scoped globals:** For session-scoped globals, you can use the method `getGlobals()` to return a `Globals` instance that provides access to the KIE session globals. These globals are used for all execution calls. Use caution with mutable globals because execution calls can be executing simultaneously in different threads.

  Session-scoped global

  ```java
  import org.kie.api.runtime.StatelessKieSession;

  StatelessKieSession ksession = kbase.newStatelessKieSession();

  // Set a global `myGlobal` that can be used in the rules.
  ksession.setGlobal("myGlobal", "I am a global");

  // Execute while resolving the `myGlobal` identifier.
  ksession.execute(collection);
  ```

- **Delegate globals:** For delegate globals, you can assign a value to a global (with `setGlobal(String, Object)`) so that the value is stored in an internal collection that maps identifiers to values. Identifiers in this internal collection have priority over any supplied delegate. If an identifier cannot be found in this internal collection, the delegate global (if any) is used.

- **Execution-scoped globals:** For execution-scoped globals, you can use the `Command` object to set a global that is passed to the `CommandExecutor` interface for execution-specific global resolution.

The `CommandExecutor` interface also enables you to export data using out identifiers for globals, inserted facts, and query results:

Out identifiers for globals, inserted facts, and query results

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

// Set up a list of commands.
List cmds = new ArrayList();
cmds.add(CommandFactory.newSetGlobal("list1", new ArrayList(), true));
cmds.add(CommandFactory.newInsert(new Person("jon", 102), "person"));
cmds.add(CommandFactory.newQuery("Get People" "getPeople"));

// Execute the list.
ExecutionResults results = ksession.execute(CommandFactory.newBatchExecution(cmds));

// Retrieve the `ArrayList`.
results.getValue("list1");
// Retrieve the inserted `Person` fact.
results.getValue("person");
// Retrieve the query as a `QueryResults` instance.
results.getValue("Get People");
```

2.1.2. Stateful KIE sessions

A stateful KIE session is a session that uses inference to make iterative changes to facts over time. In a stateful KIE session, data from a previous invocation of the KIE session (the previous session state) is retained between session invocations, whereas in a stateless KIE session, that data is discarded.

> ⚠️ Ensure that you call the `dispose()` method after running a stateful KIE session so that no memory leaks occur between session invocations.

Stateful KIE sessions are commonly used for the following use cases:

- **Monitoring**, such as monitoring a stock market and automating the buying process
- **Diagnostics**, such as running fault-finding processes or medical diagnostic processes
- **Logistics**, such as parcel tracking and delivery provisioning
- **Ensuring compliance**, such as verifying the legality of market trades

For example, consider the following fire alarm data model and sample DRL rules:

Data model for sprinklers and fire alarm

```java
public class Room {
  private String name;
  // Getter and setter methods
}

public class Sprinkler {
  private Room room;
  private boolean on;
  // Getter and setter methods
}

public class Fire {
  private Room room;
  // Getter and setter methods
}

public class Alarm { }
```

Sample DRL rule set for activating sprinklers and alarm

```
rule "When there is a fire turn on the sprinkler"
when
  Fire($room : room)
  $sprinkler : Sprinkler(room == $room, on == false)
then
  modify($sprinkler) { setOn(true) };
  System.out.println("Turn on the sprinkler for room "+$room.getName());
end

rule "Raise the alarm when we have one or more fires"
when
    exists Fire()
then
    insert( new Alarm() );
    System.out.println( "Raise the alarm" );
end

rule "Cancel the alarm when all the fires have gone"
when
    not Fire()
    $alarm : Alarm()
then
    delete( $alarm );
    System.out.println( "Cancel the alarm" );
end


rule "Status output when things are ok"
when
    not Alarm()
    not Sprinkler( on == true )
then
    System.out.println( "Everything is ok" );
end
```

For the `When there is a fire turn on the sprinkler` rule, when a fire occurs, the instances of the `Fire` class are created for that room and inserted into the KIE session. The rule adds a constraint for the specific `room` matched in the `Fire` instance so that only the sprinkler for that room is checked. When this rule is executed, the sprinkler activates. The other sample rules determine when the alarm is activated or deactivated accordingly.

Whereas a stateless KIE session relies on standard Java syntax to modify a field, a stateful KIE session relies on the `modify` statement in rules to notify the Drools rule engine of changes. The Drools rule engine then reasons over the changes and assesses impact on subsequent rule executions. This process is part of the Drools rule engine ability to use *inference* and *truth maintenance* and is essential in stateful KIE sessions.

In this example, the sample rules and all other files in the `~/resources` folder of the Drools project

are built with the following code:

Create the KIE container

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();
```

This code compiles all the rule files found on the class path and adds the result of this compilation, a `KieModule` object, in the `KieContainer`.

Finally, the `KieSession` object is instantiated from the `KieContainer` and is executed against specified data:

Instantiate the stateful KIE session and enter data

```
KieSession ksession = kContainer.newKieSession();

String[] names = new String[]{"kitchen", "bedroom", "office", "livingroom"};
Map<String,Room> name2room = new HashMap<String,Room>();
for( String name: names ){
    Room room = new Room( name );
    name2room.put( name, room );
    ksession.insert( room );
    Sprinkler sprinkler = new Sprinkler( room );
    ksession.insert( sprinkler );
}

ksession.fireAllRules();
```

Console output

```
> Everything is ok
```

With the data added, the Drools rule engine completes all pattern matching but no rules have been executed, so the configured verification message appears. As new data triggers rule conditions, the Drools rule engine executes rules to activate the alarm and later to cancel the alarm that has been activated:

Enter new data to trigger rules

```
Fire kitchenFire = new Fire( name2room.get( "kitchen" ) );
Fire officeFire = new Fire( name2room.get( "office" ) );

FactHandle kitchenFireHandle = ksession.insert( kitchenFire );
FactHandle officeFireHandle = ksession.insert( officeFire );

ksession.fireAllRules();
```

Console output

```
> Raise the alarm
> Turn on the sprinkler for room kitchen
> Turn on the sprinkler for room office
```

```
ksession.delete( kitchenFireHandle );
ksession.delete( officeFireHandle );

ksession.fireAllRules();
```

Console output

```
> Cancel the alarm
> Turn off the sprinkler for room office
> Turn off the sprinkler for room kitchen
> Everything is ok
```

In this case, a reference is kept for the returned `FactHandle` object. A fact handle is an internal engine reference to the inserted instance and enables instances to be retracted or modified later.

As this example illustrates, the data and results from previous stateful KIE sessions (the activated alarm) affect the invocation of subsequent sessions (alarm cancellation).

2.1.3. KIE session pools

In use cases with large amounts of KIE runtime data and high system activity, KIE sessions might be created and disposed very frequently. A high turnover of KIE sessions is not always time consuming, but when the turnover is repeated millions of times, the process can become a bottleneck and require substantial clean-up effort.

For these high-volume cases, you can use KIE session pools instead of many individual KIE sessions. To use a KIE session pool, you obtain a KIE session pool from a KIE container, define the initial number of KIE sessions in the pool, and create the KIE sessions from that pool as usual:

Example KIE session pool

```
// Obtain a KIE session pool from the KIE container
KieContainerSessionsPool pool = kContainer.newKieSessionsPool(10);

// Create KIE sessions from the KIE session pool
KieSession kSession = pool.newKieSession();
```

In this example, the KIE session pool starts with 10 KIE sessions in it, but you can specify the number of KIE sessions that you need. This integer value is the number of KIE sessions that are only initially created in the pool. If required by the running application, the number of KIE sessions in the pool can dynamically grow beyond that value.

After you define a KIE session pool, the next time you use the KIE session as usual and call `dispose()` on it, the KIE session is reset and pushed back into the pool instead of being destroyed.

KIE session pools typically apply to stateful KIE sessions, but KIE session pools can also affect stateless KIE sessions that you reuse with multiple `execute()` calls. When you create a stateless KIE session directly from a KIE container, the KIE session continues to internally create a new KIE session for each `execute()` invocation. Conversely, when you create a stateless KIE session from a KIE session pool, the KIE session internally uses only the specific KIE sessions provided by the pool.

When you finish using a KIE session pool, you can call the `shutdown()` method on it to avoid memory leaks. Alternatively, you can call `dispose()` on the KIE container to shut down all the pools created from the KIE container.

2.2. Inference and truth maintenance in the Drools rule engine

The basic function of the Drools rule engine is to match data to business rules and determine whether and how to execute rules. To ensure that relevant data is applied to the appropriate rules, the Drools rule engine makes *inferences* based on existing knowledge and performs the actions based on the inferred information.

For example, the following DRL rule determines the age requirements for adults, such as in a bus pass policy:

Rule to define age requirement

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

Based on this rule, the Drools rule engine infers whether a person is an adult or a child and performs the specified action (the `then` consequence). Every person who is 18 years old or older has an instance of `IsAdult` inserted for them in the working memory. This inferred relation of age and bus pass can then be invoked in any rule, such as in the following rule segment:

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

In many cases, new data in a rule system is the result of other rule executions, and this new data can affect the execution of other rules. If the Drools rule engine asserts data as a result of executing a rule, the Drools rule engine uses truth maintenance to justify the assertion and enforce truthfulness when applying inferred information to other rules. Truth maintenance also helps to identify inconsistencies and to handle contradictions. For example, if two rules are executed and result in a contradictory action, the Drools rule engine chooses the action based on assumptions

from previously calculated conclusions.

The Drools rule engine inserts facts using either stated or logical insertions:

- **Stated insertions:** Defined with `insert()`. After stated insertions, facts are generally retracted explicitly. (The term *insertion*, when used generically, refers to *stated insertion*.)
- **Logical insertions:** Defined with `insertLogical()`. After logical insertions, the facts that were inserted are automatically retracted when the conditions in the rules that inserted the facts are no longer true. The facts are retracted when no condition supports the logical insertion. A fact that is logically inserted is considered to be *justified* by the Drools rule engine.

For example, the following sample DRL rules use stated fact insertion to determine the age requirements for issuing a child bus pass or an adult bus pass:

Rules to issue bus pass, stated insertion

```
rule "Issue Child Bus Pass"
when
  $p : Person(age < 18)
then
  insert(new ChildBusPass($p));
end

rule "Issue Adult Bus Pass"
when
  $p : Person(age >= 18)
then
  insert(new AdultBusPass($p));
end
```

These rules are not easily maintained in the Drools rule engine as bus riders increase in age and move from child to adult bus pass. As an alternative, these rules can be separated into rules for bus rider age and rules for bus pass type using logical fact insertion. The logical insertion of the fact makes the fact dependent on the truth of the `when` clause.

The following DRL rules use logical insertion to determine the age requirements for children and adults:

Children and adult age requirements, logical insertion

```
rule "Infer Child"
when
  $p : Person(age < 18)
then
  insertLogical(new IsChild($p))
end

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

> ❗ For logical insertions, your fact objects must override the `equals` and `hashCode` methods from the `java.lang.Object` object according to the Java standard. Two objects are equal if their `equals` methods return `true` for each other and if their `hashCode` methods return the same values. For more information, see the Java API documentation for your Java version.

When the condition in the rule is false, the fact is automatically retracted. This behavior is helpful in this example because the two rules are mutually exclusive. In this example, if the person is younger than 18 years old, the rule logically inserts an `IsChild` fact. After the person is 18 years old or older, the `IsChild` fact is automatically retracted and the `IsAdult` fact is inserted.

The following DRL rules then determine whether to issue a child bus pass or an adult bus pass and logically insert the `ChildBusPass` and `AdultBusPass` facts. This rule configuration is possible because the truth maintenance system in the Drools rule engine supports chaining of logical insertions for a cascading set of retracts.

Rules to issue bus pass, logical insertion

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

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

When a person turns 18 years old, the `IsChild` fact and the person's `ChildBusPass` fact is retracted. To these set of conditions, you can relate another rule that states that a person must return the child pass after turning 18 years old. When the Drools rule engine automatically retracts the `ChildBusPass` object, the following rule is executed to send a request to the person:

Rule to notify bus pass holder of new pass

```
rule "Return ChildBusPass Request"
when
   $p : Person()
     not(ChildBusPass(person == $p))
then
   requestChildBusPass($p);
end
```

The following flowcharts illustrate the life cycle of stated and logical insertions:
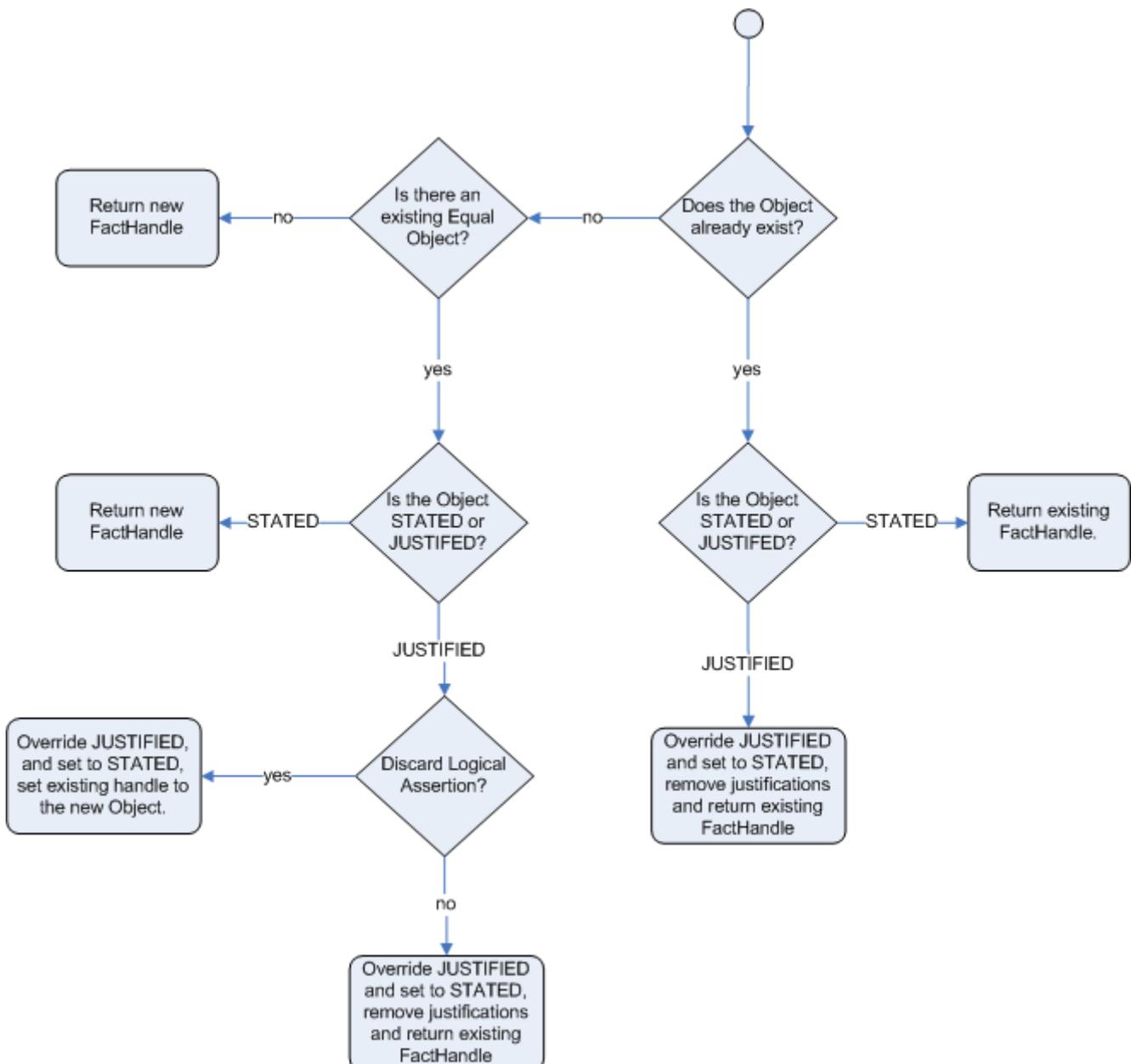
Figure 15. Stated insertion
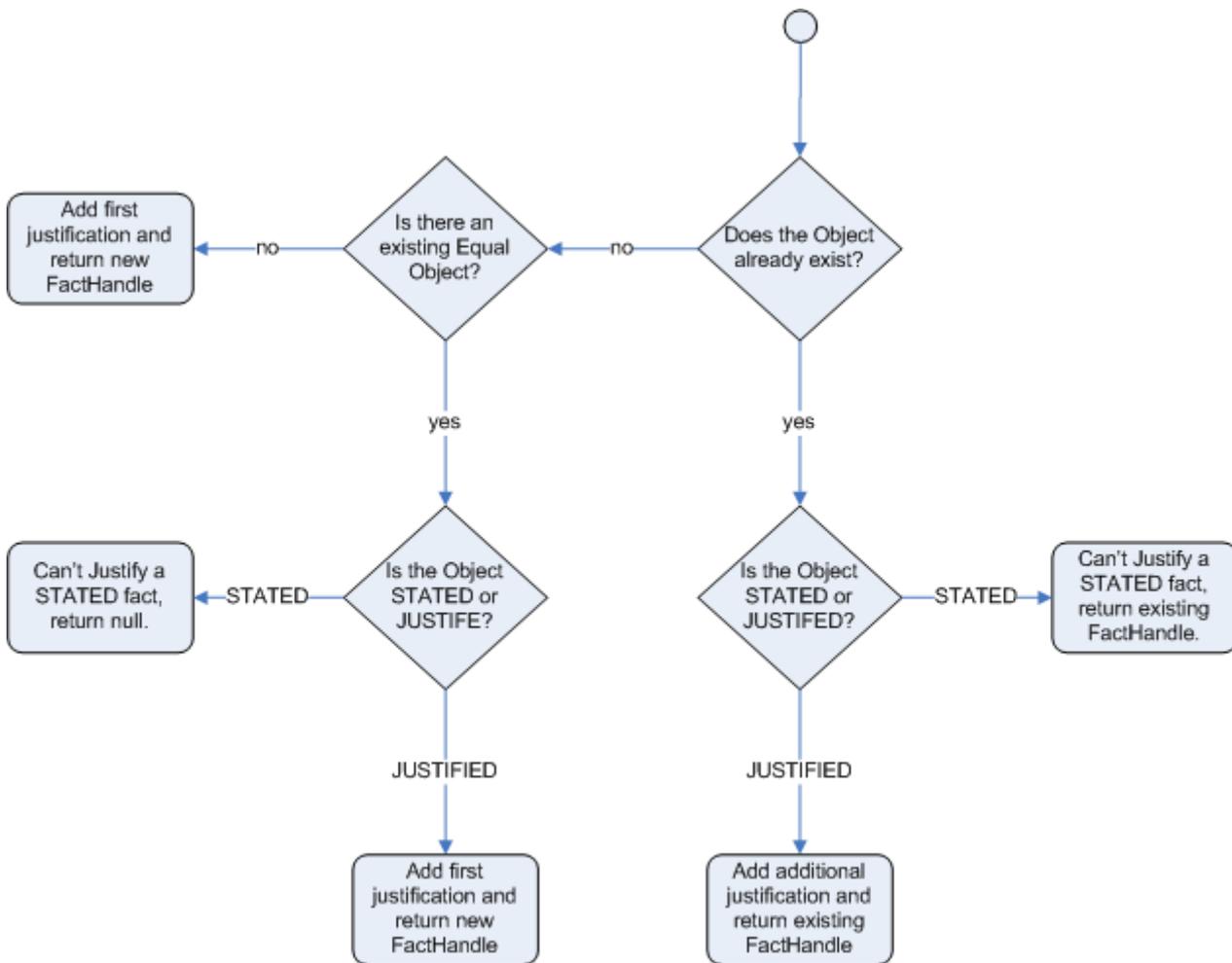
Figure 16. Logical insertion

When the Drools rule engine logically inserts an object during a rule execution, the Drools rule engine *justifies* the object by executing the rule. For each logical insertion, only one equal object can exist, and each subsequent equal logical insertion increases the justification counter for that logical insertion. A justification is removed when the conditions of the rule become untrue. When no more justifications exist, the logical object is automatically retracted.

2.2.1. Government ID example

So now we know what inference is, and have a basic example, how does this facilitate good rule design and maintenance?

Consider a government ID department that is responsible for issuing ID cards when children become adults. They might have a decision table that includes logic like this, which says when an adult living in London is 18 or over, issue the card:

	RuleTable ID Card		
	CONDITION	CONDITION	ACTION
	p : Person		
	location	age >= $1	issueIdCard($1)
	Select Person	Select Adults	Issue ID Card

Issue ID Card to Adults	London	18	p

However the ID department does not set the policy on who an adult is. That's done at a central government level. If the central government were to change that age to 21, this would initiate a change management process. Someone would have to liaise with the ID department and make sure their systems are updated, in time for the law going live.

This change management process and communication between departments is not ideal for an agile environment, and change becomes costly and error prone. Also the card department is managing more information than it needs to be aware of with its "monolithic" approach to rules management which is "leaking" information better placed elsewhere. By this I mean that it doesn't care what explicit "age >= 18" information determines whether someone is an adult, only that they are an adult.

In contrast to this, let's pursue an approach where we split (de-couple) the authoring responsibilities, so that both the central government and the ID department maintain their own rules.

It's the central government's job to determine who is an adult. If they change the law they just update their central repository with the new rules, which others use:

	RuleTable Age Policy	
	CONDITION	ACTION
	p : Person	
	age >= $1	insert($1)
	Adult Age Policy	Add Adult Relation
Infer Adult	18	new IsAdult(p)

The IsAdult fact, as discussed previously, is inferred from the policy rules. It encapsulates the seemingly arbitrary piece of logic "age >= 18" and provides semantic abstractions for its meaning. Now if anyone uses the above rules, they no longer need to be aware of explicit information that determines whether someone is an adult or not. They can just use the inferred fact:

	RuleTable ID Card		
	CONDITION	CONDITION	ACTION
	p : Person	isAdult	
	location	person == $1	issueIdCard($1)
	Select Person	Select Adults	Issue ID Card
Issue ID Card to Adults	London	p	p

While the example is very minimal and trivial it illustrates some important points. We started with a monolithic and leaky approach to our knowledge engineering. We created a single decision table that had all possible information in it and that leaks information from central government that the ID department did not care about and did not want to manage.

We first de-coupled the knowledge process so each department was responsible for only what it needed to know. We then encapsulated this leaky knowledge using an inferred fact IsAdult. The use of the term IsAdult also gave a semantic abstraction to the previously arbitrary logic "age >= 18".

So a general rule of thumb when doing your knowledge engineering is:

- **Bad**
  - Monolithic
  - Leaky
- **Good**
  - De-couple knowledge responsibilities
  - Encapsulate knowledge
  - Provide semantic abstractions for those encapsulations

2.2.2. Fact equality modes in the Drools rule engine

The Drools rule engine supports the following fact equality modes that determine how the Drools rule engine stores and compares inserted facts:

- `identity`: (Default) The Drools rule engine uses an `IdentityHashMap` to store all inserted facts. For every new fact insertion, the Drools rule engine returns a new `FactHandle` object. If a fact is inserted again, the Drools rule engine returns the original `FactHandle` object, ignoring repeated insertions for the same fact. In this mode, two facts are the same for the Drools rule engine only if they are the very same object with the same identity.

- `equality`: The Drools rule engine uses a `HashMap` to store all inserted facts. The Drools rule engine returns a new `FactHandle` object only if the inserted fact is not equal to an existing fact, according to the `equals()` method of the inserted fact. In this mode, two facts are the same for the Drools rule engine if they are composed the same way, regardless of identity. Use this mode when you want objects to be assessed based on feature equality instead of explicit identity.

As an illustration of fact equality modes, consider the following example facts:

Example facts

```
Person p1 = new Person("John", 45);
Person p2 = new Person("John", 45);
```

In `identity` mode, facts `p1` and `p2` are different instances of a `Person` class and are treated as separate objects because they have separate identities. In `equality` mode, facts `p1` and `p2` are treated as the same object because they are composed the same way. This difference in behavior affects how you can interact with fact handles.

For example, assume that you insert facts `p1` and `p2` into the Drools rule engine and later you want to retrieve the fact handle for `p1`. In `identity` mode, you must specify `p1` to return the fact handle for that exact object, whereas in `equality` mode, you can specify `p1`, `p2`, or `new Person("John", 45)` to return the fact handle.

Example code to insert a fact and return the fact handle in `identity` *mode*

```
ksession.insert(p1);

ksession.getFactHandle(p1);
```

Example code to insert a fact and return the fact handle in `equality` *mode*

```
ksession.insert(p1);

ksession.getFactHandle(p1);

// Alternate option:
ksession.getFactHandle(new Person("John", 45));
```

To set the fact equality mode, use one of the following options:

- Set the system property `drools.equalityBehavior` to `identity` (default) or `equality`.

- Set the equality mode while creating the KIE base programmatically:

  ```
  KieServices ks = KieServices.get();
  KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
  kieBaseConf.setOption(EqualityBehaviorOption.EQUALITY);
  KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
  ```

- Set the equality mode in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

  ```
  <kmodule>
    ...
    <kbase name="KBase2" default="false" equalsBehavior="equality" packages=
  "org.domain.pkg2, org.domain.pkg3" includes="KBase1">
      ...
    </kbase>
    ...
  </kmodule>
  ```

2.3. Execution control in the Drools rule engine

When new rule data enters the working memory of the Drools rule engine, rules may become fully matched and eligible for execution. A single working memory action can result in multiple eligible rule executions. When a rule is fully matched, the Drools rule engine creates an activation instance, referencing the rule and the matched facts, and adds the activation onto the Drools rule engine agenda. The agenda controls the execution order of these rule activations using a conflict resolution strategy.

After the first call of `fireAllRules()` in the Java application, the Drools rule engine cycles repeatedly through two phases:

- **Agenda evaluation.** In this phase, the Drools rule engine selects all rules that can be executed. If no executable rules exist, the execution cycle ends. If an executable rule is found, the Drools rule engine registers the activation in the agenda and then moves on to the working memory actions phase to perform rule consequence actions.

- **Working memory actions.** In this phase, the Drools rule engine performs the rule consequence actions (the `then` portion of each rule) for all activated rules previously registered in the agenda. After all the consequence actions are complete or the main Java application process calls `fireAllRules()` again, the Drools rule engine returns to the agenda evaluation phase to reassess rules.
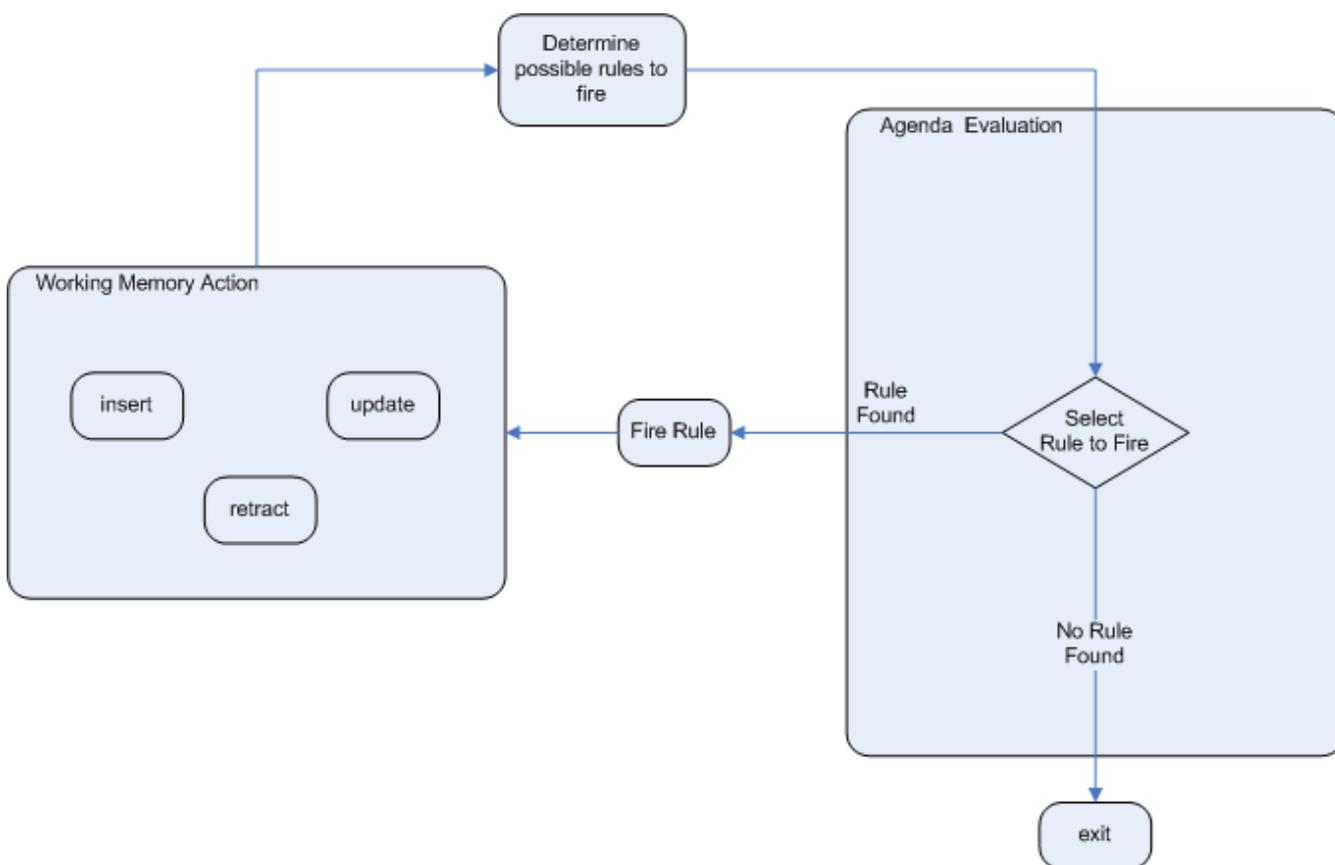
Figure 17. Two-phase execution process in the Drools rule engine

When multiple rules exist on the agenda, the execution of one rule may cause another rule to be removed from the agenda. To avoid this, you can define how and when rules are executed in the Drools rule engine. Some common methods for defining rule execution order are by using rule salience, agenda groups, or activation groups.

2.3.1. Salience for rules

Each rule has an integer `salience` attribute that determines the order of execution. Rules with a higher salience value are given higher priority when ordered in the activation queue. The default salience value for rules is zero, but the salience can be negative or positive.

For example, the following sample DRL rules are listed in the Drools rule engine stack in the order shown:

```
rule "RuleA"
salience 95
when
    $fact : MyFact( field1 == true )
then
    System.out.println("Rule2 : " + $fact);
    update($fact);
end

rule "RuleB"
salience 100
when
    $fact : MyFact( field1 == false )
then
    System.out.println("Rule1 : " + $fact);
    $fact.setField1(true);
    update($fact);
end
```

The RuleB rule is listed second, but it has a higher salience value than the RuleA rule and is therefore executed first.

2.3.2. Agenda groups for rules

An agenda group is a set of rules bound together by the same agenda-group rule attribute. Agenda groups partition rules on the Drools rule engine agenda. At any one time, only one group has a *focus* that gives that group of rules priority for execution before rules in other agenda groups. You determine the focus with a setFocus() call for the agenda group. You can also define rules with an auto-focus attribute so that the next time the rule is activated, the focus is automatically given to the entire agenda group to which the rule is assigned.

Each time the setFocus() call is made in a Java application, the Drools rule engine adds the specified agenda group to the top of the rule stack. The default agenda group "MAIN" contains all rules that do not belong to a specified agenda group and is executed first in the stack unless another group has the focus.

For example, the following sample DRL rules belong to specified agenda groups and are listed in the Drools rule engine stack in the order shown:

Sample DRL rules for banking application

```
rule "Increase balance for credits"
  agenda-group "calculation"
when
  ap : AccountPeriod()
  acc : Account( $accountNo : accountNo )
  CashFlow( type == CREDIT,
            accountNo == $accountNo,
            date >= ap.start && <= ap.end,
            $amount : amount )
then
  acc.balance  += $amount;
end
```

```
rule "Print balance for AccountPeriod"
  agenda-group "report"
when
  ap : AccountPeriod()
  acc : Account()
then
  System.out.println( acc.accountNo +
                      " : " + acc.balance );
end
```

For this example, the rules in the `"report"` agenda group must always be executed first and the rules in the `"calculation"` agenda group must always be executed second. Any remaining rules in other agenda groups can then be executed. Therefore, the `"report"` and `"calculation"` groups must receive the focus to be executed in that order, before other rules can be executed:

Set the focus for the order of agenda group execution

```
Agenda agenda = ksession.getAgenda();
agenda.getAgendaGroup( "report" ).setFocus();
agenda.getAgendaGroup( "calculation" ).setFocus();
ksession.fireAllRules();
```

You can also use the `clear()` method to cancel all the activations generated by the rules belonging to a given agenda group before each has had a chance to be executed:

Cancel all other rule activations

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

2.3.3. Activation groups for rules

An activation group is a set of rules bound together by the same `activation-group` rule attribute. In

this group, only one rule can be executed. After conditions are met for a rule in that group to be executed, all other pending rule executions from that activation group are removed from the agenda.

For example, the following sample DRL rules belong to the specified activation group and are listed in the Drools rule engine stack in the order shown:

Sample DRL rules for banking

```
rule "Print balance for AccountPeriod1"
  activation-group "report"
when
  ap : AccountPeriod1()
  acc : Account()
then
  System.out.println( acc.accountNo +
                        " : " + acc.balance );
end
```

```
rule "Print balance for AccountPeriod2"
  activation-group "report"
when
  ap : AccountPeriod2()
  acc : Account()
then
  System.out.println( acc.accountNo +
                        " : " + acc.balance );
end
```

For this example, if the first rule in the `"report"` activation group is executed, the second rule in the group and all other executable rules on the agenda are removed from the agenda.

2.3.4. Rule execution modes and thread safety in the Drools rule engine

The Drools rule engine supports the following rule execution modes that determine how and when the Drools rule engine executes rules:

- **Passive mode**: (Default) The Drools rule engine evaluates rules when a user or an application explicitly calls `fireAllRules()`. Passive mode in the Drools rule engine is best for applications that require direct control over rule evaluation and execution, or for complex event processing (CEP) applications that use the pseudo clock implementation in the Drools rule engine.

Example CEP application code with the Drools rule engine in passive mode

```
KieSessionConfiguration config = KieServices.Factory.get()
.newKieSessionConfiguration();
config.setOption( ClockTypeOption.get("pseudo") );
KieSession session = kbase.newKieSession( conf, null );
SessionPseudoClock clock = session.getSessionClock();

session.insert( tick1 );
session.fireAllRules();

clock.advanceTime(1, TimeUnit.SECONDS);
session.insert( tick2 );
session.fireAllRules();

clock.advanceTime(1, TimeUnit.SECONDS);
session.insert( tick3 );
session.fireAllRules();

session.dispose();
```

- **Active mode**: If a user or application calls `fireUntilHalt()`, the Drools rule engine starts in active mode and evaluates rules continually until the user or application explicitly calls `halt()`. Active mode in the Drools rule engine is best for applications that delegate control of rule evaluation and execution to the Drools rule engine, or for complex event processing (CEP) applications that use the real-time clock implementation in the Drools rule engine. Active mode is also optimal for CEP applications that use active queries.

Example CEP application code with the Drools rule engine in active mode

```java
KieSessionConfiguration config = KieServices.Factory.get()
.newKieSessionConfiguration();
config.setOption( ClockTypeOption.get("realtime") );
KieSession session = kbase.newKieSession( conf, null );

new Thread( new Runnable() {
  @Override
  public void run() {
      session.fireUntilHalt();
  }
} ).start();

session.insert( tick1 );

... Thread.sleep( 1000L ); ...

session.insert( tick2 );

... Thread.sleep( 1000L ); ...

session.insert( tick3 );

session.halt();
session.dispose();
```

This example calls `fireUntilHalt()` from a dedicated execution thread to prevent the current thread from being blocked indefinitely while the Drools rule engine continues evaluating rules. The dedicated thread also enables you to call `halt()` at a later stage in the application code.

Although you should avoid using both `fireAllRules()` and `fireUntilHalt()` calls, especially from different threads, the Drools rule engine can handle such situations safely using thread-safety logic and an internal state machine. If a `fireAllRules()` call is in progress and you call `fireUntilHalt()`, the Drools rule engine continues to run in passive mode until the `fireAllRules()` operation is complete and then starts in active mode in response to the `fireUntilHalt()` call. However, if the Drools rule engine is running in active mode following a `fireUntilHalt()` call and you call `fireAllRules()`, the `fireAllRules()` call is ignored and the Drools rule engine continues to run in active mode until you call `halt()`.

For added thread safety in active mode, the Drools rule engine supports a `submit()` method that you can use to group and perform operations on a KIE session in a thread-safe, atomic action:

Example application code with `submit()` *method to perform atomic operations in active mode*

```
KieSession session = ...;

new Thread( new Runnable() {
  @Override
  public void run() {
      session.fireUntilHalt();
  }
} ).start();

final FactHandle fh = session.insert( fact_a );

... Thread.sleep( 1000L ); ...

session.submit( new KieSession.AtomicAction() {
  @Override
  public void execute( KieSession kieSession ) {
    fact_a.setField("value");
    kieSession.update( fh, fact_a );
    kieSession.insert( fact_1 );
    kieSession.insert( fact_2 );
    kieSession.insert( fact_3 );
  }
} );

... Thread.sleep( 1000L ); ...

session.insert( fact_z );

session.halt();
session.dispose();
```

Thread safety and atomic operations are also helpful from a client-side perspective. For example, you might need to insert more than one fact at a given time, but require the Drools rule engine to consider the insertions as an atomic operation and to wait until all the insertions are complete before evaluating the rules again.

2.3.5. Fact propagation modes in the Drools rule engine

The Drools rule engine supports the following fact propagation modes that determine how the Drools rule engine progresses inserted facts through the engine network in preparation for rule execution:

- **Lazy**: (Default) Facts are propagated in batch collections at rule execution, not in real time as the facts are individually inserted by a user or application. As a result, the order in which the facts are ultimately propagated through the Drools rule engine may be different from the order in which the facts were individually inserted.

- **Immediate**: Facts are propagated immediately in the order that they are inserted by a user or

application.

- **Eager**: Facts are propagated lazily (in batch collections), but before rule execution. The Drools rule engine uses this propagation behavior for rules that have the `no-loop` or `lock-on-active` attribute.

By default, the Phreak rule algorithm in the Drools rule engine uses lazy fact propagation for improved rule evaluation overall. However, in few cases, this lazy propagation behavior can alter the expected result of certain rule executions that may require immediate or eager propagation.

For example, the following rule uses a specified query with a `?` prefix to invoke the query in pull-only or passive fashion:

Example rule with a passive query

```
query Q (Integer i)
    String( this == i.toString() )
end

rule "Rule"
  when
    $i : Integer()
    ?Q( $i; )
  then
    System.out.println( $i );
end
```

For this example, the rule should be executed only when a `String` that satisfies the query is inserted before the `Integer`, such as in the following example commands:

Example commands that should trigger the rule execution

```
KieSession ksession = ...
ksession.insert("1");
ksession.insert(1);
ksession.fireAllRules();
```

However, due to the default lazy propagation behavior in Phreak, the Drools rule engine does not detect the insertion sequence of the two facts in this case, so this rule is executed regardless of `String` and `Integer` insertion order. For this example, immediate propagation is required for the expected rule evaluation.

To alter the Drools rule engine propagation mode to achieve the expected rule evaluation in this case, you can add the `@Propagation(<type>)` tag to your rule and set `<type>` to `LAZY`, `IMMEDIATE`, or `EAGER`.

In the same example rule, the immediate propagation annotation enables the rule to be evaluated only when a `String` that satisfies the query is inserted before the `Integer`, as expected:

Example rule with a passive query and specified propagation mode

```
query Q (Integer i)
    String( this == i.toString() )
end

rule "Rule" @Propagation(IMMEDIATE)
  when
    $i : Integer()
    ?Q( $i; )
  then
    System.out.println( $i );
end
```

2.3.6. Agenda evaluation filters

[AgendaFilter] | *rule-engine/AgendaFilter.png*

Figure 18. AgendaFilters

The Drools rule engine supports an `AgendaFilter` object in the filter interface that you can use to allow or deny the evaluation of specified rules during agenda evaluation. You can specify an agenda filter as part of a `fireAllRules()` call.

The following example code permits only rules ending with the string `"Test"` to be evaluated and executed. All other rules are filtered out of the Drools rule engine agenda.

Example agenda filter definition

```
ksession.fireAllRules( new RuleNameEndsWithAgendaFilter( "Test" ) );
```

2.4. Phreak rule algorithm in the Drools rule engine

The Drools rule engine in Drools uses the Phreak algorithm for rule evaluation. Phreak evolved from the Rete algorithm, including the enhanced Rete algorithm ReteOO that was introduced in previous versions of Drools for object-oriented systems. Overall, Phreak is more scalable than Rete and ReteOO, and is faster in large systems.

While Rete is considered eager (immediate rule evaluation) and data oriented, Phreak is considered lazy (delayed rule evaluation) and goal oriented. The Rete algorithm performs many actions during the insert, update, and delete actions in order to find partial matches for all rules. This eagerness of the Rete algorithm during rule matching requires a lot of time before eventually executing rules, especially in large systems. With Phreak, this partial matching of rules is delayed deliberately to handle large amounts of data more efficiently.

The Phreak algorithm adds the following set of enhancements to previous Rete algorithms:

- Three layers of contextual memory: Node, segment, and rule memory types
- Rule-based, segment-based, and node-based linking

- Lazy (delayed) rule evaluation

- Stack-based evaluations with pause and resume

- Isolated rule evaluation

- Set-oriented propagations

2.4.1. Rule evaluation in Phreak

When the Drools rule engine starts, all rules are considered to be *unlinked* from pattern-matching data that can trigger the rules. At this stage, the Phreak algorithm in the Drools rule engine does not evaluate the rules. The `insert`, `update`, and `delete` actions are queued, and Phreak uses a heuristic, based on the rule most likely to result in execution, to calculate and select the next rule for evaluation. When all the required input values are populated for a rule, the rule is considered to be *linked* to the relevant pattern-matching data. Phreak then creates a goal that represents this rule and places the goal into a priority queue that is ordered by rule salience. Only the rule for which the goal was created is evaluated, and other potential rule evaluations are delayed. While individual rules are evaluated, node sharing is still achieved through the process of segmentation.

Unlike the tuple-oriented Rete, the Phreak propagation is collection oriented. For the rule that is being evaluated, the Drools rule engine accesses the first node and processes all queued insert, update, and delete actions. The results are added to a set, and the set is propagated to the child node. In the child node, all queued insert, update, and delete actions are processed, adding the results to the same set. The set is then propagated to the next child node and the same process repeats until it reaches the terminal node. This cycle creates a batch process effect that can provide performance advantages for certain rule constructs.

The linking and unlinking of rules happens through a layered bit-mask system, based on network segmentation. When the rule network is built, segments are created for rule network nodes that are shared by the same set of rules. A rule is composed of a path of segments. In case a rule does not share any node with any other rule, it becomes a single segment.

A bit-mask offset is assigned to each node in the segment. Another bit mask is assigned to each segment in the path of the rule according to these requirements:

- If at least one input for a node exists, the node bit is set to the on state.

- If each node in a segment has the bit set to the on state, the segment bit is also set to the on state.

- If any node bit is set to the off state, the segment is also set to the off state.

- If each segment in the path of the rule is set to the on state, the rule is considered linked, and a goal is created to schedule the rule for evaluation.

The same bit-mask technique is used to track modified nodes, segments, and rules. This tracking ability enables an already linked rule to be unscheduled from evaluation if it has been modified since the evaluation goal for it was created. As a result, no rules can ever evaluate partial matches.

This process of rule evaluation is possible in Phreak because, as opposed to a single unit of memory in Rete, Phreak has three layers of contextual memory with node, segment, and rule memory types. This layering enables much more contextual understanding during the evaluation of a rule.
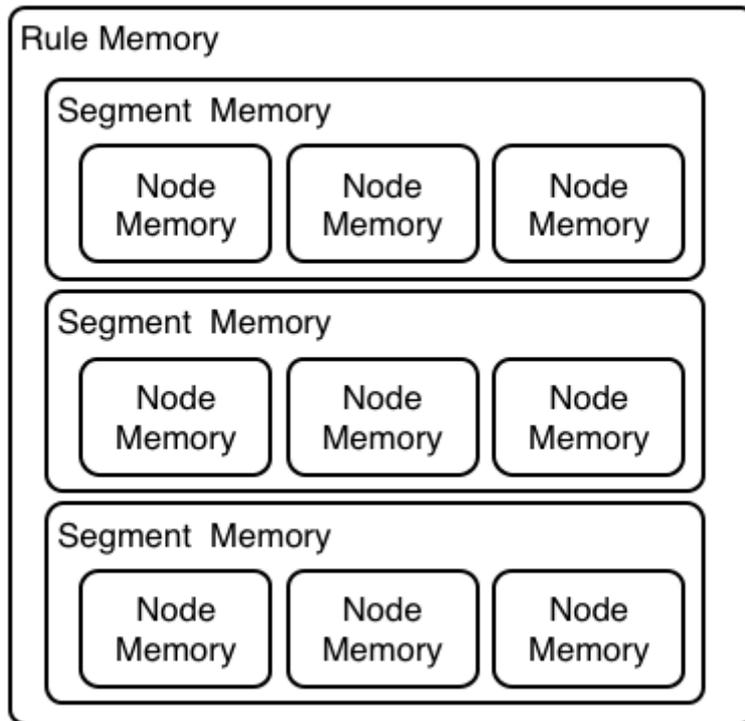
Figure 19. Phreak three-layered memory system

The following examples illustrate how rules are organized and evaluated in this three-layered memory system in Phreak.

Example 1: A single rule (R1) with three patterns: A, B and C. The rule forms a single segment, with bits 1, 2, and 4 for the nodes. The single segment has a bit offset of 1.
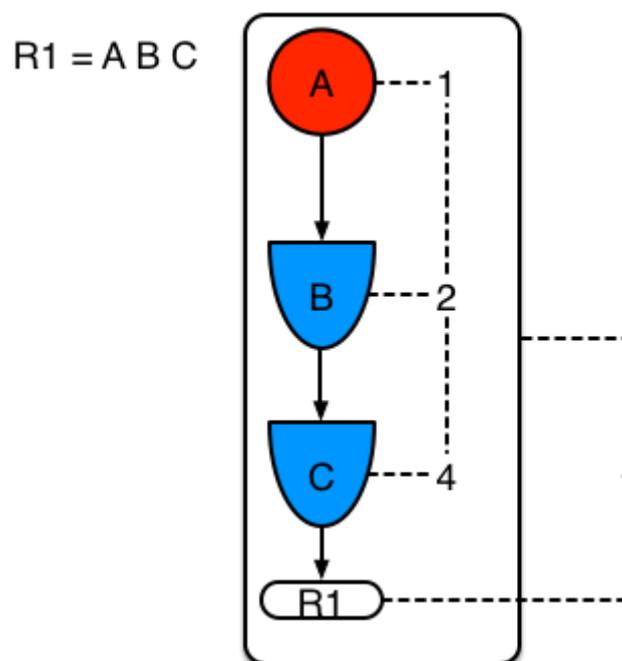

Figure 20. Example 1: Single rule

**Example 2:** Rule R2 is added and shares pattern A.
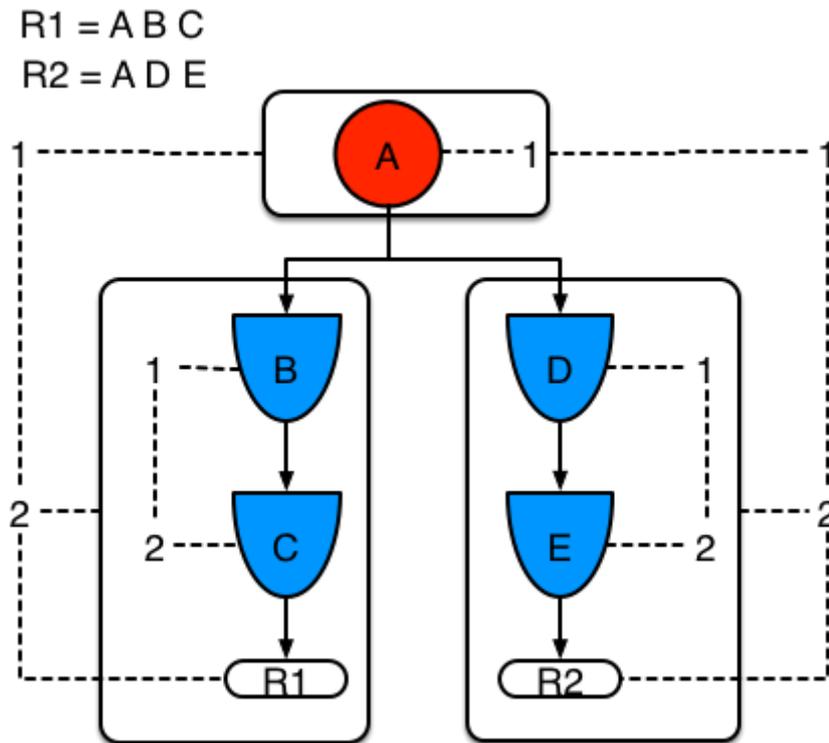
Figure 21. Example 2: Two rules with pattern sharing

Pattern A is placed in its own segment, resulting in two segments for each rule. Those two segments form a path for their respective rules. The first segment is shared by both paths. When pattern A is linked, the segment becomes linked. The segment then iterates over each path that the segment is shared by, setting the bit 1 to on. If patterns B and C are later turned on, the second segment for path R1 is linked, and this causes bit 2 to be turned on for R1. With bit 1 and bit 2 turned on for R1, the rule is now linked and a goal is created to schedule the rule for later evaluation and execution.

When a rule is evaluated, the segments enable the results of the matching to be shared. Each segment has a staging memory to queue all inserts, updates, and deletes for that segment. When R1 is evaluated, the rule processes pattern A, and this results in a set of tuples. The algorithm detects a segmentation split, creates peered tuples for each insert, update, and delete in the set, and adds them to the R2 staging memory. Those tuples are then merged with any existing staged tuples and are executed when R2 is eventually evaluated.

**Example 3:** Rules R3 and R4 are added and share patterns A and B.
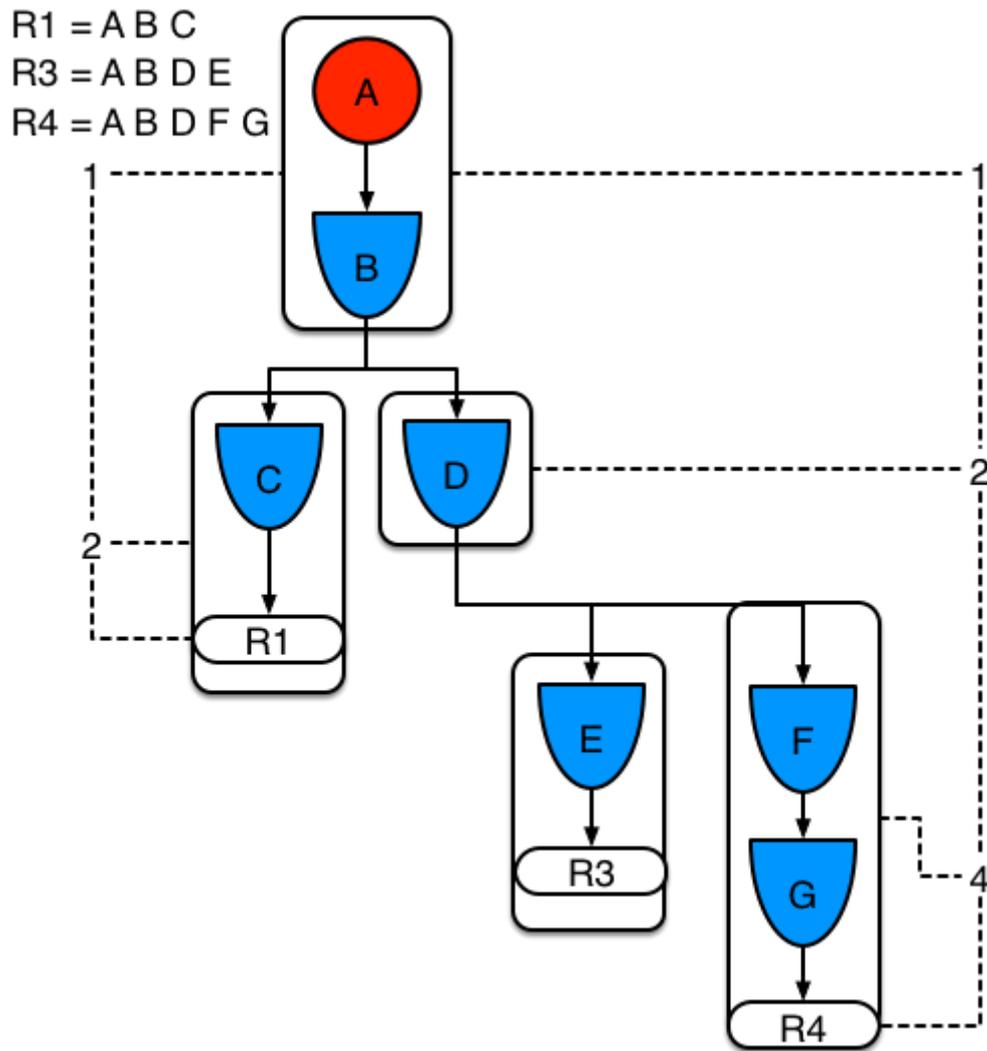
Figure 22. Example 3: Three rules with pattern sharing

Rules R3 and R4 have three segments and R1 has two segments. Patterns A and B are shared by R1, R3, and R4, while pattern D is shared by R3 and R4.

Example 4: A single rule (R1) with a subnetwork and no pattern sharing.

$$R1 = A \text{ not } (B \text{ not } (C)) D$$
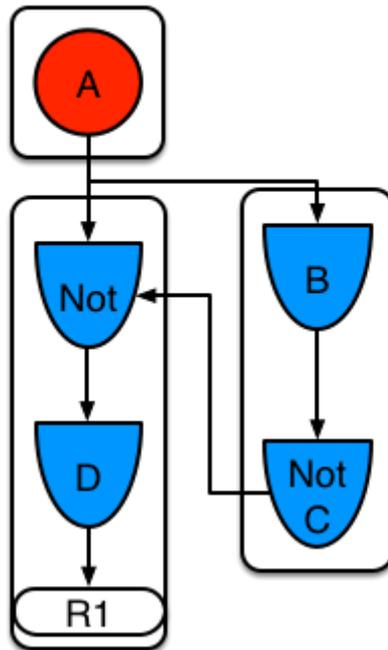
Figure 23. Example 4: Single rule with a subnetwork and no pattern sharing

Subnetworks are formed when a `Not`, `Exists`, or `Accumulate` node contains more than one element. In this example, the element `B not( C )` forms the subnetwork. The element `not( C )` is a single element that does not require a subnetwork and is therefore merged inside of the `Not` node. The subnetwork uses a dedicated segment. Rule R1 still has a path of two segments and the subnetwork forms another inner path. When the subnetwork is linked, it is also linked in the outer segment.

Example 5: Rule R1 with a subnetwork that is shared by rule R2.
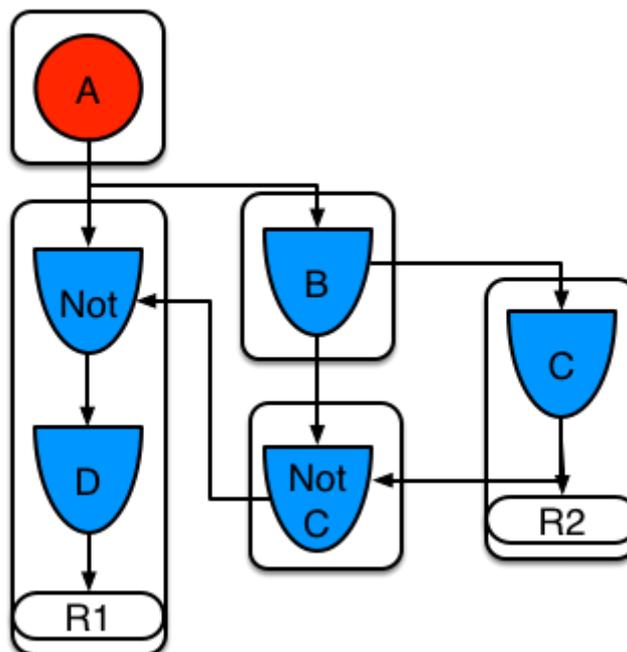
Figure 24. Example 5: Two rules, one with a subnetwork and pattern sharing

The subnetwork nodes in a rule can be shared by another rule that does not have a subnetwork.

This sharing causes the subnetwork segment to be split into two segments.

Constrained `Not` nodes and `Accumulate` nodes can never unlink a segment, and are always considered to have their bits turned on.

The Phreak evaluation algorithm is stack based instead of method-recursion based. Rule evaluation can be paused and resumed at any time when a `StackEntry` is used to represent the node currently being evaluated.

When a rule evaluation reaches a subnetwork, a `StackEntry` object is created for the outer path segment and the subnetwork segment. The subnetwork segment is evaluated first, and when the set reaches the end of the subnetwork path, the segment is merged into a staging list for the outer node that the segment feeds into. The previous `StackEntry` object is then resumed and can now process the results of the subnetwork. This process has the added benefit, especially for `Accumulate` nodes, that all work is completed in a batch, before propagating to the child node.

The same stack system is used for efficient backward chaining. When a rule evaluation reaches a query node, the evaluation is paused and the query is added to the stack. The query is then evaluated to produce a result set, which is saved in a memory location for the resumed `StackEntry` object to pick up and propagate to the child node. If the query itself called other queries, the process repeats, while the current query is paused and a new evaluation is set up for the current query node.

2.4.1.1. Rule evaluation with forward and backward chaining

The Drools rule engine in Drools is a hybrid reasoning system that uses both forward chaining and backward chaining to evaluate rules. A forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the Drools rule engine and reacts to changes to that fact. When objects are inserted into working memory, any rule conditions that become true as a result of the change are scheduled for execution by the agenda.

In contrast, a backward-chaining rule system is a goal-driven system that starts with a conclusion that the Drools rule engine attempts to satisfy, often using recursion. If the system cannot reach the conclusion or goal, it searches for subgoals, which are conclusions that complete part of the current goal. The system continues this process until either the initial conclusion is satisfied or all subgoals are satisfied.

The following diagram illustrates how the Drools rule engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:
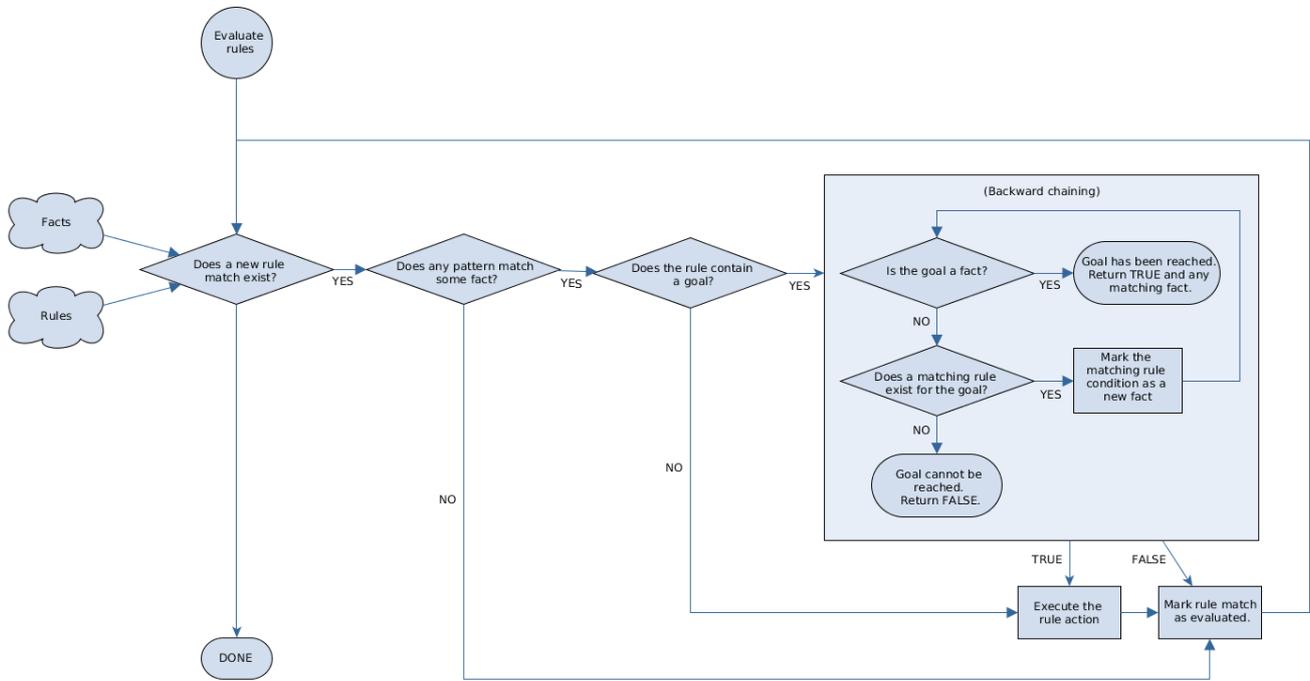
Figure 25. Rule evaluation logic using forward and backward chaining

2.4.2. Rule base configuration

Drools contains a `RuleBaseConfiguration.java` object that you can use to configure exception handler settings, multithreaded execution, and sequential mode in the Drools rule engine.

For the rule base configuration options, see the Drools RuleBaseConfiguration.java page in GitHub.

The following rule base configuration options are available for the Drools rule engine:

drools.consequenceExceptionHandler

When configured, this system property defines the class that manages the exceptions thrown by rule consequences. You can use this property to specify a custom exception handler for rule evaluation in the Drools rule engine.

Default value: `org.drools.core.runtime.rule.impl.DefaultConsequenceExceptionHandler`

You can specify the custom exception handler using one of the following options:

- Specify the exception handler in a system property:

```
drools.consequenceExceptionHandler=org.drools.core.runtime.rule.impl.MyCustomCon
sequenceExceptionHandler
```

- Specify the exception handler while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration(); kieBaseConf
.setOption(ConsequenceExceptionHandlerOption.get(MyCustomConsequenceExceptionHan
dler.class));
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

drools.multithreadEvaluation

When enabled, this system property enables the Drools rule engine to evaluate rules in parallel by dividing the Phreak rule network into independent partitions. You can use this property to increase the speed of rule evaluation for specific rule bases.

Default value: `false`

You can enable multithreaded evaluation using one of the following options:

- Enable the multithreaded evaluation system property:

```
drools.multithreadEvaluation=true
```

- Enable multithreaded evaluation while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(MultithreadEvaluationOption.YES);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

> ⚠️ Rules that use queries, salience, or agenda groups are currently not supported by the parallel Drools rule engine. If these rule elements are present in the KIE base, the compiler emits a warning and automatically switches back to single-threaded evaluation. However, in some cases, the Drools rule engine might not detect the unsupported rule elements and rules might be evaluated incorrectly. For example, the Drools rule engine might not detect when rules rely on implicit salience given by rule ordering inside the DRL file, resulting in incorrect evaluation due to the unsupported salience attribute.

drools.sequential

When enabled, this system property enables sequential mode in the Drools rule engine. In sequential mode, the Drools rule engine evaluates rules one time in the order that they are listed in the Drools rule engine agenda without regard to changes in the working memory. This means that the Drools rule engine ignores any `insert`, `modify`, or `update` statements in rules and executes rules in a single sequence. As a result, rule execution may be faster in sequential mode, but important updates may not be applied to your rules. You can use this property if you use stateless KIE sessions and you do not want the execution of rules to influence subsequent rules in the agenda. Sequential mode applies to stateless KIE sessions only.

Default value: `false`

You can enable sequential mode using one of the following options:

- Enable the sequential mode system property:

```
drools.sequential=true
```

- Enable sequential mode while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(SequentialOption.YES);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

- Enable sequential mode in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" sequential="true" packages=
"org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

2.4.3. Sequential mode in Phreak

Sequential mode is an advanced rule base configuration in the Drools rule engine, supported by Phreak, that enables the Drools rule engine to evaluate rules one time in the order that they are listed in the Drools rule engine agenda without regard to changes in the working memory. In sequential mode, the Drools rule engine ignores any `insert`, `modify`, or `update` statements in rules and executes rules in a single sequence. As a result, rule execution may be faster in sequential mode, but important updates may not be applied to your rules.

Sequential mode applies to only stateless KIE sessions because stateful KIE sessions inherently use data from previously invoked KIE sessions. If you use a stateless KIE session and you want the execution of rules to influence subsequent rules in the agenda, then do not enable sequential mode. Sequential mode is disabled by default in the Drools rule engine.

To enable sequential mode, use one of the following options:

- Set the system property `drools.sequential` to `true`.
- Enable sequential mode while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(SequentialOption.YES);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

- Enable sequential mode in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" sequential="true" packages="org.domain.pkg2,
org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

To configure sequential mode to use a dynamic agenda, use one of the following options:

- Set the system property `drools.sequential.agenda` to `dynamic`.

- Set the sequential agenda option while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(SequentialAgendaOption.DYNAMIC);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

When you enable sequential mode, the Drools rule engine evaluates rules in the following way:

1. Rules are ordered by salience and position in the rule set.

2. An element for each possible rule match is created. The element position indicates the execution order.

3. Node memory is disabled, with the exception of the right-input object memory.

4. The left-input adapter node propagation is disconnected and the object with the node is referenced in a `Command` object. The `Command` object is added to a list in the working memory for later execution.

5. All objects are asserted, and then the list of `Command` objects is checked and executed.

6. All matches that result from executing the list are added to elements based on the sequence number of the rule.

7. The elements that contain matches are executed in a sequence. If you set a maximum number of rule executions, the Drools rule engine activates no more than that number of rules in the agenda for execution.

In sequential mode, the `LeftInputAdapterNode` node creates a `Command` object and adds it to a list in the working memory of the Drools rule engine. This `Command` object contains references to the `LeftInputAdapterNode` node and the propagated object. These references stop any left-input propagations at insertion time so that the right-input propagation never needs to attempt to join the left inputs. The references also avoid the need for the left-input memory.

All nodes have their memory turned off, including the left-input tuple memory, but excluding the right-input object memory. After all the assertions are finished and the right-input memory of all the objects is populated, the Drools rule engine iterates over the list of `LeftInputAdatperNode Command` objects. The objects propagate down the network, attempting to join the right-input objects, but they are not retained in the left input.

The agenda with a priority queue to schedule the tuples is replaced by an element for each rule. The sequence number of the `RuleTerminalNode` node indicates the element where to place the match. After all `Command` objects have finished, the elements are checked and existing matches are executed. To improve performance, the first and the last populated cell in the elements are retained.

When the network is constructed, each `RuleTerminalNode` node receives a sequence number based on its salience number and the order in which it was added to the network.

The right-input node memories are typically hash maps for fast object deletion. Because object deletions are not supported, Phreak uses an object list when the values of the object are not indexed. For a large number of objects, indexed hash maps provide a performance increase. If an object has only a few instances, Phreak uses an object list instead of an index.

2.5. Complex event processing (CEP)

In Drools, an event is a record of a significant change of state in the application domain at a point in time. Depending on how the domain is modeled, the change of state may be represented by a single event, multiple atomic events, or hierarchies of correlated events. From a complex event processing (CEP) perspective, an event is a type of fact or object that occurs at a specific point in time, and a business rule is a definition of how to react to the data from that fact or object. For example, in a stock broker application, a change in security prices, a change in ownership from seller to buyer, or a change in an account holder's balance are all considered to be events because a change has occurred in the state of the application domain at a given time.

The Drools rule engine in Drools uses complex event processing (CEP) to detect and process multiple events within a collection of events, to uncover relationships that exist between events, and to infer new data from the events and their relationships.

CEP use cases share several requirements and goals with business rule use cases.

From a business perspective, business rule definitions are often defined based on the occurrence of scenarios triggered by events. In the following examples, events form the basis of business rules:

- In an algorithmic trading application, a rule performs an action if the security price increases by X percent above the day opening price. The price increases are denoted by events on a stock trading application.
- In a monitoring application, a rule performs an action if the temperature in the server room

increases X degrees in Y minutes. The sensor readings are denoted by events.

From a technical perspective, business rule evaluation and CEP have the following key similarities:

- Both business rule evaluation and CEP require seamless integration with the enterprise infrastructure and applications. This is particularly important with life-cycle management, auditing, and security.
- Both business rule evaluation and CEP have functional requirements such as pattern matching, and non-functional requirements such as response time limits and query-rule explanations.

CEP scenarios have the following key characteristics:

- Scenarios usually process large numbers of events, but only a small percentage of the events are relevant.
- Events are usually immutable and represent a record of change in state.
- Rules and queries run against events and must react to detected event patterns.
- Related events usually have a strong temporal relationship.
- Individual events are not prioritized. The CEP system prioritizes patterns of related events and the relationships between them.
- Events usually need to be composed and aggregated.

Given these common CEP scenario characteristics, the CEP system in Drools supports the following features and functions to optimize event processing:

- Event processing with proper semantics
- Event detection, correlation, aggregation, and composition
- Event stream processing
- Temporal constraints to model the temporal relationships between events
- Sliding windows of significant events
- Session-scoped unified clock
- Required volumes of events for CEP use cases
- Reactive rules
- Adapters for event input into the Drools rule engine (pipeline)

2.5.1. Events in complex event processing

In Drools, an event is a record of a significant change of state in the application domain at a point in time. Depending on how the domain is modeled, the change of state may be represented by a single event, multiple atomic events, or hierarchies of correlated events. From a complex event processing (CEP) perspective, an event is a type of fact or object that occurs at a specific point in time, and a business rule is a definition of how to react to the data from that fact or object. For example, in a stock broker application, a change in security prices, a change in ownership from seller to buyer, or a change in an account holder's balance are all considered to be events because a change has occurred in the state of the application domain at a given time.

Events have the following key characteristics:

- **Are immutable:** An event is a record of change that has occurred at some time in the past and cannot be changed.

  > The Drools rule engine does not enforce immutability on the Java objects that represent events. This behavior makes event data enrichment possible. Your application should be able to populate unpopulated event attributes, and these attributes are used by the Drools rule engine to enrich the event with inferred data. However, you should not change event attributes that have already been populated.

- **Have strong temporal constraints:** Rules involving events usually require the correlation of multiple events that occur at different points in time relative to each other.

- **Have managed life cycles:** Because events are immutable and have temporal constraints, they are usually only relevant for a specified period of time. This means that the Drools rule engine can automatically manage the life cycle of events.

- **Can use sliding windows:** You can define sliding windows of time or length with events. A sliding time window is a specified period of time during which events can be processed. A sliding length window is a specified number of events that can be processed.

2.5.2. Declaring facts as events

You can declare facts as events in your Java class or DRL rule file so that the Drools rule engine handles the facts as events during complex event processing. You can declare the facts as interval-based events or point-in-time events. Interval-based events have a duration time and persist in the working memory of the Drools rule engine until their duration time has lapsed. Point-in-time events have no duration and are essentially interval-based events with a duration of zero.

Procedure

For the relevant fact type in your Java class or DRL rule file, enter the `@role( event )` metadata tag and parameter. The `@role` metadata tag accepts the following two values:

- `fact`: (Default) Declares the type as a regular fact

- `event`: Declares the type as an event

For example, the following snippet declares that the `StockPoint` fact type in a stock broker application must be handled as an event:

Declare fact type as an event

```
import some.package.StockPoint

declare StockPoint
  @role( event )
end
```

If StockPoint is a fact type declared in the DRL rule file instead of in a pre-existing class, you can declare the event in-line in your application code:

Declare fact type in-line and assign it to event role

```
declare StockPoint
  @role( event )

  datetime : java.util.Date
  symbol : String
  price : double
end
```

2.5.3. Event processing modes in the Drools rule engine

The Drools rule engine runs in either cloud mode or stream mode. In cloud mode, the Drools rule engine processes facts as facts with no temporal constraints, independent of time, and in no particular order. In stream mode, the Drools rule engine processes facts as events with strong temporal constraints, in real time or near real time. Stream mode uses synchronization to make event processing possible in Drools.

Cloud mode

Cloud mode is the default operating mode of the Drools rule engine. In cloud mode, the Drools rule engine treats events as an unordered cloud. Events still have time stamps, but the Drools rule engine running in cloud mode cannot draw relevance from the time stamp because cloud mode ignores the present time. This mode uses the rule constraints to find the matching tuples to activate and execute rules.

Cloud mode does not impose any kind of additional requirements on facts. However, because the Drools rule engine in this mode has no concept of time, it cannot use temporal features such as sliding windows or automatic life-cycle management. In cloud mode, events must be explicitly retracted when they are no longer needed.

The following requirements are not imposed in cloud mode:

- No clock synchronization because the Drools rule engine has no notion of time
- No ordering of events because the Drools rule engine processes events as an unordered cloud, against which the Drools rule engine match rules

You can specify cloud mode either by setting the system property in the relevant configuration files or by using the Java client API:

Set cloud mode using system property

```
drools.eventProcessingMode=cloud
```

Set cloud mode using Java client API

```java
import org.kie.api.conf.EventProcessingOption;
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices.Factory;

KieBaseConfiguration config = KieServices.Factory.get().newKieBaseConfiguration();

config.setOption(EventProcessingOption.CLOUD);
```

You can also specify cloud mode using the `eventProcessingMode="<mode>"` KIE base attribute in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

Set cloud mode using project `kmodule.xml` file

```xml
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="cloud" packages=
"org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

Stream mode

Stream mode enables the Drools rule engine to process events chronologically and in real time as they are inserted into the Drools rule engine. In stream mode, the Drools rule engine synchronizes streams of events (so that events in different streams can be processed in chronological order), implements sliding windows of time or length, and enables automatic life-cycle management.

The following requirements apply to stream mode:

- Events in each stream must be ordered chronologically.

- A session clock must be present to synchronize event streams.

> Your application does not need to enforce ordering events between streams, but using event streams that have not been synchronized may cause unexpected results.

You can specify stream mode either by setting the system property in the relevant configuration files or by using the Java client API:

Set stream mode using system property

```
drools.eventProcessingMode=stream
```

Set stream mode using Java client API

```java
import org.kie.api.conf.EventProcessingOption;
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices.Factory;

KieBaseConfiguration config = KieServices.Factory.get().newKieBaseConfiguration();

config.setOption(EventProcessingOption.STREAM);
```

You can also specify stream mode using the `eventProcessingMode="<mode>"` KIE base attribute in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

Set stream mode using project `kmodule.xml` file

```xml
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="stream" packages=
"org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

2.5.3.1. Negative patterns in Drools rule engine stream mode

A negative pattern is a pattern for conditions that are not met. For example, the following DRL rule activates a fire alarm if a fire is detected and the sprinkler is not activated:

Fire alarm rule with a negative pattern

```
rule "Sound the alarm"
when
  $f : FireDetected()
  not(SprinklerActivated())
then
  // Sound the alarm.
end
```

In cloud mode, the Drools rule engine assumes all facts (regular facts and events) are known in advance and evaluates negative patterns immediately. In stream mode, the Drools rule engine can support temporal constraints on facts to wait for a set time before activating a rule.

The same example rule in stream mode activates the fire alarm as usual, but applies a 10-second delay.

Fire alarm rule with a negative pattern and time delay (stream mode only)

```
rule "Sound the alarm"
when
  $f : FireDetected()
  not(SprinklerActivated(this after[0s,10s] $f))
then
  // Sound the alarm.
end
```

The following modified fire alarm rule expects one `Heartbeat` event to occur every 10 seconds. If the expected event does not occur, the rule is executed. This rule uses the same type of object in both the first pattern and in the negative pattern. The negative pattern has the temporal constraint to wait 0 to 10 seconds before executing and excludes the `Heartbeat` event bound to `$h` so that the rule can be executed. The bound event `$h` must be explicitly excluded in order for the rule to be executed because the temporal constraint `[0s, ···]` does not inherently exclude that event from being matched again.

Fire alarm rule excluding a bound event in a negative pattern (stream mode only)

```
rule "Sound the alarm"
when
  $h: Heartbeat() from entry-point "MonitoringStream"
  not(Heartbeat(this != $h, this after[0s,10s] $h) from entry-point
"MonitoringStream")
then
  // Sound the alarm.
end
```

2.5.4. Property-change settings and listeners for fact types

By default, the Drools rule engine does not re-evaluate all fact patterns for fact types each time a rule is triggered, but instead reacts only to modified properties that are constrained or bound inside a given pattern. For example, if a rule calls `modify()` as part of the rule actions but the action does not generate new data in the KIE base, the Drools rule engine does not automatically re-evaluate all fact patterns because no data was modified. This property reactivity behavior prevents unwanted recursions in the KIE base and results in more efficient rule evaluation. This behavior also means that you do not always need to use the `no-loop` rule attribute to avoid infinite recursion.

You can modify or disable this property reactivity behavior with the following `KnowledgeBuilderConfiguration` options, and then use a property-change setting in your Java class or DRL files to fine-tune property reactivity as needed:

- `ALWAYS`: (Default) All types are property reactive, but you can disable property reactivity for a specific type by using the `@classReactive` property-change setting.

- `ALLOWED`: No types are property reactive, but you can enable property reactivity for a specific type by using the `@propertyReactive` property-change setting.

- `DISABLED`: No types are property reactive. All property-change listeners are ignored.

Example property reactivity setting in KnowledgeBuilderConfiguration

```
KnowledgeBuilderConfiguration config =
KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration();
config.setOption(PropertySpecificOption.ALLOWED);
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder(config);
```

Alternatively, you can update the `drools.propertySpecific` system property in the `standalone.xml` file of your Drools distribution:

Example property reactivity setting in system properties

```
<system-properties>
  ...
  <property name="drools.propertySpecific" value="ALLOWED"/>
  ...
</system-properties>
```

The Drools rule engine supports the following property-change settings and listeners for fact classes or declared DRL fact types:

@classReactive

If property reactivity is set to `ALWAYS` in the Drools rule engine (all types are property reactive), this tag disables the default property reactivity behavior for a specific Java class or a declared DRL fact type. You can use this tag if you want the Drools rule engine to re-evaluate all fact patterns for the specified fact type each time the rule is triggered, instead of reacting only to modified properties that are constrained or bound inside a given pattern.

Example: Disable default property reactivity in a DRL type declaration

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

Example: Disable default property reactivity in a Java class

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

@propertyReactive

If property reactivity is set to `ALLOWED` in the Drools rule engine (no types are property reactive unless specified), this tag enables property reactivity for a specific Java class or a declared DRL fact type. You can use this tag if you want the Drools rule engine to react only to modified

properties that are constrained or bound inside a given pattern for the specified fact type, instead of re-evaluating all fact patterns for the fact each time the rule is triggered.

Example: Enable property reactivity in a DRL type declaration (when reactivity is disabled globally)

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

Example: Enable property reactivity in a Java class (when reactivity is disabled globally)

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

@watch

This tag enables property reactivity for additional properties that you specify in-line in fact patterns in DRL rules. This tag is supported only if property reactivity is set to `ALWAYS` in the Drools rule engine, or if property reactivity is set to `ALLOWED` and the relevant fact type uses the `@propertyReactive` tag. You can use this tag in DRL rules to add or exclude specific properties in fact property reactivity logic.

Default parameter: None

Supported parameters: Property name, `*` (all), `!` (not), `!*` (no properties)

```
<factPattern> @watch ( <property> )
```

Example: Enable or disable property reactivity in fact patterns

```
// Listens for changes in both `firstName` (inferred) and `lastName`:
Person(firstName == $expectedFirstName) @watch( lastName )

// Listens for changes in all properties of the `Person` fact:
Person(firstName == $expectedFirstName) @watch( * )

// Listens for changes in `lastName` and explicitly excludes changes in
`firstName`:
Person(firstName == $expectedFirstName) @watch( lastName, !firstName )

// Listens for changes in all properties of the `Person` fact except `age`:
Person(firstName == $expectedFirstName) @watch( *, !age )

// Excludes changes in all properties of the `Person` fact (equivalent to using
`@classReactivity` tag):
Person(firstName == $expectedFirstName) @watch( !* )
```

The Drools rule engine generates a compilation error if you use the `@watch` tag for properties in a fact type that uses the `@classReactive` tag (disables property reactivity) or when property reactivity is set to `ALLOWED` in the Drools rule engine and the relevant fact type does not use the `@propertyReactive` tag. Compilation errors also arise if you duplicate properties in listener annotations, such as `@watch( firstName, ! firstName )`.

@propertyChangeSupport

For facts that implement support for property changes as defined in the JavaBeans Specification, this tag enables the Drools rule engine to monitor changes in the fact properties.

Example: Declare property change support in JavaBeans object

```
declare Person
    @propertyChangeSupport
end
```

2.5.5. Temporal operators for events

In stream mode, the Drools rule engine supports the following temporal operators for events that are inserted into the working memory of the Drools rule engine. You can use these operators to define the temporal reasoning behavior of the events that you declare in your Java class or DRL rule file. Temporal operators are not supported when the Drools rule engine is running in cloud mode.

- after
- before
- coincides
- during

- includes

- finishes

- finished by

- meets

- met by

- overlaps

- overlapped by

- starts

- started by

after

> This operator specifies if the current event occurs after the correlated event. This operator can also define an amount of time after which the current event can follow the correlated event, or a delimiting time range during which the current event can follow the correlated event.
>
> For example, the following pattern matches if $eventA starts between 3 minutes and 30 seconds and 4 minutes after $eventB finishes. If $eventA starts earlier than 3 minutes and 30 seconds after $eventB finishes, or later than 4 minutes after $eventB finishes, then the pattern is not matched.
>
> ```
> $eventA : EventA(this after[3m30s, 4m] $eventB)
> ```
>
> You can also express this operator in the following way:
>
> ```
> 3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
> ```
>
> The after operator supports up to two parameter values:
>
> - If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).
>
> - If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.
>
> - If no value is defined, the interval starts at 1 millisecond and runs indefinitely with no end time.
>
> The after operator also supports negative time ranges:
>
> ```
> $eventA : EventA(this after[-3m30s, -2m] $eventB)
> ```
>
> If the first value is greater than the second value, the Drools rule engine automatically reverses them. For example, the following two patterns are interpreted by the Drools rule

engine in the same way:

```
$eventA : EventA(this after[-3m30s, -2m] $eventB)
$eventA : EventA(this after[-2m, -3m30s] $eventB)
```

before

This operator specifies if the current event occurs before the correlated event. This operator can also define an amount of time before which the current event can precede the correlated event, or a delimiting time range during which the current event can precede the correlated event.

For example, the following pattern matches if $eventA finishes between 3 minutes and 30 seconds and 4 minutes before $eventB starts. If $eventA finishes earlier than 3 minutes and 30 seconds before $eventB starts, or later than 4 minutes before $eventB starts, then the pattern is not matched.

```
$eventA : EventA(this before[3m30s, 4m] $eventB)
```

You can also express this operator in the following way:

```
3m30s <= $eventB.startTimestamp - $eventA.endTimeStamp <= 4m
```

The before operator supports up to two parameter values:

- If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).

- If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.

- If no value is defined, the interval starts at 1 millisecond and runs indefinitely with no end time.

The before operator also supports negative time ranges:

```
$eventA : EventA(this before[-3m30s, -2m] $eventB)
```

If the first value is greater than the second value, the Drools rule engine automatically reverses them. For example, the following two patterns are interpreted by the Drools rule engine in the same way:

```
$eventA : EventA(this before[-3m30s, -2m] $eventB)
$eventA : EventA(this before[-2m, -3m30s] $eventB)
```

coincides

This operator specifies if the two events occur at the same time, with the same start and end times.

For example, the following pattern matches if both the start and end time stamps of $eventA and $eventB are identical:

```
$eventA : EventA(this coincides $eventB)
```

The `coincides` operator supports up to two parameter values for the distance between the event start and end times, if they are not identical:

- If only one parameter is given, the parameter is used to set the threshold for both the start and end times of both events.
- If two parameters are given, the first is used as a threshold for the start time and the second is used as a threshold for the end time.

The following pattern uses start and end time thresholds:

```
$eventA : EventA(this coincides[15s, 10s] $eventB)
```

The pattern matches if the following conditions are met:

```
abs($eventA.startTimestamp - $eventB.startTimestamp) <= 15s
&&
abs($eventA.endTimestamp - $eventB.endTimestamp) <= 10s
```

> ⚠️ The Drools rule engine does not support negative intervals for the `coincides` operator. If you use negative intervals, the Drools rule engine generates an error.

during

This operator specifies if the current event occurs within the time frame of when the correlated event starts and ends. The current event must start after the correlated event starts and must end before the correlated event ends. (With the `coincides` operator, the start and end times are the same or nearly the same.)

For example, the following pattern matches if $eventA starts after $eventB starts and ends before $eventB ends:

```
$eventA : EventA(this during $eventB)
```

You can also express this operator in the following way:

```
$eventB.startTimestamp < $eventA.startTimestamp <= $eventA.endTimestamp <
$eventB.endTimestamp
```

The `during` operator supports one, two, or four optional parameters:

- If one value is defined, this value is the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.

- If two values are defined, these values are a threshold between which the current event start time and end time must occur in relation to the correlated event start and end times.

  For example, if the values are `5s` and `10s`, the current event must start between 5 and 10 seconds after the correlated event starts and must end between 5 and 10 seconds before the correlated event ends.

- If four values are defined, the first and second values are the minimum and maximum distances between the start times of the events, and the third and fourth values are the minimum and maximum distances between the end times of the two events.

includes

This operator specifies if the correlated event occurs within the time frame of when the current event occurs. The correlated event must start after the current event starts and must end before the current event ends. (The behavior of this operator is the reverse of the `during` operator behavior.)

For example, the following pattern matches if `$eventB` starts after `$eventA` starts and ends before `$eventA` ends:

```
$eventA : EventA(this includes $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp < $eventB.startTimestamp <= $eventB.endTimestamp <
$eventA.endTimestamp
```

The `includes` operator supports one, two, or four optional parameters:

- If one value is defined, this value is the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.

- If two values are defined, these values are a threshold between which the correlated event start time and end time must occur in relation to the current event start and end times.

  For example, if the values are `5s` and `10s`, the correlated event must start between 5 and 10 seconds after the current event starts and must end between 5 and 10 seconds before the current event ends.

- If four values are defined, the first and second values are the minimum and maximum distances between the start times of the events, and the third and fourth values are the minimum and maximum distances between the end times of the two events.

finishes

This operator specifies if the current event starts after the correlated event but both events end at the same time.

For example, the following pattern matches if $eventA starts after $eventB starts and ends at the same time when $eventB ends:

```
$eventA : EventA(this finishes $eventB)
```

You can also express this operator in the following way:

```
$eventB.startTimestamp < $eventA.startTimestamp
&&
$eventA.endTimestamp == $eventB.endTimestamp
```

The `finishes` operator supports one optional parameter that sets the maximum time allowed between the end times of the two events:

```
$eventA : EventA(this finishes[5s] $eventB)
```

This pattern matches if these conditions are met:

```
$eventB.startTimestamp < $eventA.startTimestamp
&&
abs($eventA.endTimestamp - $eventB.endTimestamp) <= 5s
```

> ⚠️ The Drools rule engine does not support negative intervals for the `finishes` operator. If you use negative intervals, the Drools rule engine generates an error.

finished by

This operator specifies if the correlated event starts after the current event but both events end at the same time. (The behavior of this operator is the reverse of the `finishes` operator behavior.)

For example, the following pattern matches if $eventB starts after $eventA starts and ends at the same time when $eventA ends:

```
$eventA : EventA(this finishedby $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp < $eventB.startTimestamp
&&
$eventA.endTimestamp == $eventB.endTimestamp
```

The `finished by` operator supports one optional parameter that sets the maximum time allowed between the end times of the two events:

```
$eventA : EventA(this finishedby[5s] $eventB)
```

This pattern matches if these conditions are met:

```
$eventA.startTimestamp < $eventB.startTimestamp
&&
abs($eventA.endTimestamp - $eventB.endTimestamp) <= 5s
```

> ⚠️ The Drools rule engine does not support negative intervals for the `finished by` operator. If you use negative intervals, the Drools rule engine generates an error.

meets

This operator specifies if the current event ends at the same time when the correlated event starts.

For example, the following pattern matches if $eventA ends at the same time when $eventB starts:

```
$eventA : EventA(this meets $eventB)
```

You can also express this operator in the following way:

```
abs($eventB.startTimestamp - $eventA.endTimestamp) == 0
```

The `meets` operator supports one optional parameter that sets the maximum time allowed between the end time of the current event and the start time of the correlated event:

```
$eventA : EventA(this meets[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs($eventB.startTimestamp - $eventA.endTimestamp) <= 5s
```

> ⚠️ The Drools rule engine does not support negative intervals for the `meets` operator. If you use negative intervals, the Drools rule engine generates an error.

met by

This operator specifies if the correlated event ends at the same time when the current event starts. (The behavior of this operator is the reverse of the `meets` operator behavior.)

For example, the following pattern matches if $eventB ends at the same time when $eventA starts:

```
$eventA : EventA(this metby $eventB)
```

You can also express this operator in the following way:

```
abs($eventA.startTimestamp - $eventB.endTimestamp) == 0
```

The `met by` operator supports one optional parameter that sets the maximum distance between the end time of the correlated event and the start time of the current event:

```
$eventA : EventA(this metby[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs($eventA.startTimestamp - $eventB.endTimestamp) <= 5s
```

> ⚠️ The Drools rule engine does not support negative intervals for the `met by` operator. If you use negative intervals, the Drools rule engine generates an error.

overlaps

This operator specifies if the current event starts before the correlated event starts and it ends during the time frame that the correlated event occurs. The current event must end between the start and end times of the correlated event.

For example, the following pattern matches if $eventA starts before $eventB starts and then ends while $eventB occurs, before $eventB ends:

```
$eventA : EventA(this overlaps $eventB)
```

The `overlaps` operator supports up to two parameters:

- ◦ If one parameter is defined, the value is the maximum distance between the start time of the correlated event and the end time of the current event.

- ◦ If two parameters are defined, the values are the minimum distance (first value) and the maximum distance (second value) between the start time of the correlated event and the end time of the current event.

overlapped by

This operator specifies if the correlated event starts before the current event starts and it ends during the time frame that the current event occurs. The correlated event must end between the start and end times of the current event. (The behavior of this operator is the reverse of the `overlaps` operator behavior.)

For example, the following pattern matches if $eventB starts before $eventA starts and then ends while $eventA occurs, before $eventA ends:

```
$eventA : EventA(this overlappedby $eventB)
```

The `overlapped by` operator supports up to two parameters:

- ◦ If one parameter is defined, the value is the maximum distance between the start time of the current event and the end time of the correlated event.

- ◦ If two parameters are defined, the values are the minimum distance (first value) and the maximum distance (second value) between the start time of the current event and the end time of the correlated event.

starts

This operator specifies if the two events start at the same time but the current event ends before the correlated event ends.

For example, the following pattern matches if $eventA and $eventB start at the same time, and $eventA ends before $eventB ends:

```
$eventA : EventA(this starts $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp == $eventB.startTimestamp
&&
$eventA.endTimestamp < $eventB.endTimestamp
```

The `starts` operator supports one optional parameter that sets the maximum distance between the start times of the two events:

```
$eventA : EventA(this starts[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs($eventA.startTimestamp - $eventB.startTimestamp) <= 5s
&&
$eventA.endTimestamp < $eventB.endTimestamp
```

⚠ The Drools rule engine does not support negative intervals for the `starts` operator. If you use negative intervals, the Drools rule engine generates an error.

started by

This operator specifies if the two events start at the same time but the correlated event ends before the current event ends. (The behavior of this operator is the reverse of the `starts` operator behavior.)

For example, the following pattern matches if $eventA and $eventB start at the same time, and $eventB ends before $eventA ends:

```
$eventA : EventA(this startedby $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp == $eventB.startTimestamp
&&
$eventA.endTimestamp > $eventB.endTimestamp
```

The `started by` operator supports one optional parameter that sets the maximum distance between the start times of the two events:

```
$eventA : EventA( this starts[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s
&&
$eventA.endTimestamp > $eventB.endTimestamp
```

⚠ The Drools rule engine does not support negative intervals for the `started by` operator. If you use negative intervals, the Drools rule engine generates an error.

2.5.6. Session clock implementations in the Drools rule engine

During complex event processing, events in the Drools rule engine may have temporal constraints and therefore require a session clock that provides the current time. For example, if a rule needs to determine the average price of a given stock over the last 60 minutes, the Drools rule engine must be able to compare the stock price event time stamp with the current time in the session clock.

The Drools rule engine supports a real-time clock and a pseudo clock. You can use one or both clock types depending on the scenario:

- **Rules testing:** Testing requires a controlled environment, and when the tests include rules with temporal constraints, you must be able to control the input rules and facts and the flow of time.

- **Regular execution:** The Drools rule engine reacts to events in real time and therefore requires a real-time clock.

- **Special environments:** Specific environments may have specific time control requirements. For example, clustered environments may require clock synchronization or Java Enterprise Edition (JEE) environments may require a clock provided by the application server.

- **Rules replay or simulation:** In order to replay or simulate scenarios, the application must be able to control the flow of time.

Consider your environment requirements as you decide whether to use a real-time clock or pseudo clock in the Drools rule engine.

Real-time clock

> The real-time clock is the default clock implementation in the Drools rule engine and uses the system clock to determine the current time for time stamps. To configure the Drools rule engine to use the real-time clock, set the KIE session configuration parameter to `realtime`:

> *Configure real-time clock in KIE session*

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

KieSessionConfiguration config = KieServices.Factory.get()
.newKieSessionConfiguration();

config.setOption(ClockTypeOption.get("realtime"));
```

Pseudo clock

> The pseudo clock implementation in the Drools rule engine is helpful for testing temporal rules and it can be controlled by the application. To configure the Drools rule engine to use the pseudo clock, set the KIE session configuration parameter to `pseudo`:

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

KieSessionConfiguration config = KieServices.Factory.get()
.newKieSessionConfiguration();

config.setOption(ClockTypeOption.get("pseudo"));
```

You can also use additional configurations and fact handlers to control the pseudo clock:

Control pseudo clock behavior in KIE session

```
import java.util.concurrent.TimeUnit;

import org.kie.api.runtime.KieSessionConfiguration;
import org.kie.api.KieServices.Factory;
import org.kie.api.runtime.KieSession;
import org.drools.core.time.SessionPseudoClock;
import org.kie.api.runtime.rule.FactHandle;
import org.kie.api.runtime.conf.ClockTypeOption;

KieSessionConfiguration conf = KieServices.Factory.get().
newKieSessionConfiguration();

conf.setOption( ClockTypeOption.get("pseudo"));
KieSession session = kbase.newKieSession(conf, null);

SessionPseudoClock clock = session.getSessionClock();

// While inserting facts, advance the clock as necessary.
FactHandle handle1 = session.insert(tick1);
clock.advanceTime(10, TimeUnit.SECONDS);

FactHandle handle2 = session.insert(tick2);
clock.advanceTime(30, TimeUnit.SECONDS);

FactHandle handle3 = session.insert(tick3);
```

2.5.7. Event streams and entry points

The Drools rule engine can process high volumes of events in the form of event streams. In DRL rule declarations, a stream is also known as an *entry point*. When you declare an entry point in a DRL rule or Java application, the Drools rule engine, at compile time, identifies and creates the proper internal structures to use data from only that entry point to evaluate that rule.

Facts from one entry point, or stream, can join facts from any other entry point in addition to facts

already in the working memory of the Drools rule engine. Facts always remain associated with the entry point through which they entered the Drools rule engine. Facts of the same type can enter the Drools rule engine through several entry points, but facts that enter the Drools rule engine through entry point A can never match a pattern from entry point B.

Event streams have the following characteristics:

- Events in the stream are ordered by time stamp. The time stamps may have different semantics for different streams, but they are always ordered internally.

- Event streams usually have a high volume of events.

- Atomic events in streams are usually not useful individually, only collectively in a stream.

- Event streams can be homogeneous and contain a single type of event, or heterogeneous and contain events of different types.

2.5.7.1. Declaring entry points for rule data

You can declare an entry point (event stream) for events so that the Drools rule engine uses data from only that entry point to evaluate the rules. You can declare an entry point either implicitly by referencing it in DRL rules or explicitly in your Java application.

Procedure

Use one of the following methods to declare the entry point:

- In the DRL rule file, specify `from entry-point "<name>"` for the inserted fact:

  Authorize withdrawal rule with "ATM Stream" entry point

  ```
  rule "Authorize withdrawal"
  when
    WithdrawRequest($ai : accountId, $am : amount) from entry-point "ATM Stream"
    CheckingAccount(accountId == $ai, balance > $am)
  then
    // Authorize withdrawal.
  end
  ```

  Apply fee rule with "Branch Stream" entry point

  ```
  rule "Apply fee on withdraws on branches"
  when
    WithdrawRequest($ai : accountId, processed == true) from entry-point "Branch
  Stream"
    CheckingAccount(accountId == $ai)
  then
    // Apply a $2 fee on the account.
  end
  ```

  Both example DRL rules from a banking application insert the event `WithdrawalRequest` with the fact `CheckingAccount`, but from different entry points. At run time, the Drools rule engine

evaluates the `Authorize withdrawal` rule using data from only the "ATM Stream" entry point, and evaluates the `Apply fee` rule using data from only the "Branch Stream" entry point. Any events inserted into the "ATM Stream" can never match patterns for the "Apply fee" rule, and any events inserted into the "Branch Stream" can never match patterns for the "Authorize withdrawal rule".

- In the Java application code, use the `getEntryPoint()` method to specify and obtain an `EntryPoint` object and insert facts into that entry point accordingly:

Java application code with EntryPoint object and inserted facts

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

// Create your KIE base and KIE session as usual.
KieSession session = ...

// Create a reference to the entry point.
EntryPoint atmStream = session.getEntryPoint("ATM Stream");

// Start inserting your facts into the entry point.
atmStream.insert(aWithdrawRequest);
```

Any DRL rules that specify `from entry-point "ATM Stream"` are then evaluated based on the data in this entry point only.

2.5.8. Sliding windows of time or length

In stream mode, the Drools rule engine can process events from a specified sliding window of time or length. A sliding time window is a specified period of time during which events can be processed. A sliding length window is a specified number of events that can be processed. When you declare a sliding window in a DRL rule or Java application, the Drools rule engine, at compile time, identifies and creates the proper internal structures to use data from only that sliding window to evaluate that rule.

For example, the following DRL rule snippets instruct the Drools rule engine to process only the stock points from the last 2 minutes (sliding time window) or to process only the last 10 stock points (sliding length window):

Process stock points from the last 2 minutes (sliding time window)

```
StockPoint() over window:time(2m)
```

Process the last 10 stock points (sliding length window)

```
StockPoint() over window:length(10)
```

2.5.8.1. Declaring sliding windows for rule data

You can declare a sliding window of time (flow of time) or length (number of occurrences) for events so that the Drools rule engine uses data from only that window to evaluate the rules.

Procedure

In the DRL rule file, specify `over window:<time_or_length>(<value>)` for the inserted fact.

For example, the following two DRL rules activate a fire alarm based on an average temperature. However, the first rule uses a sliding time window to calculate the average over the last 10 minutes while the second rule uses a sliding length window to calculate the average over the last one hundred temperature readings.

Average temperature over sliding time window

```
rule "Sound the alarm if temperature rises above threshold"
when
  TemperatureThreshold($max : max)
  Number(doubleValue > $max) from accumulate(
    SensorReading($temp : temperature) over window:time(10m),
    average($temp))
then
  // Sound the alarm.
end
```

Average temperature over sliding length window

```
rule "Sound the alarm if temperature rises above threshold"
when
  TemperatureThreshold($max : max)
  Number(doubleValue > $max) from accumulate(
    SensorReading($temp : temperature) over window:length(100),
    average($temp))
then
  // Sound the alarm.
end
```

The Drools rule engine discards any `SensorReading` events that are more than 10 minutes old or that are not part of the last one hundred readings, and continues recalculating the average as the minutes or readings "slide" forward in real time.

The Drools rule engine does not automatically remove outdated events from the KIE session because other rules without sliding window declarations might depend on those events. The Drools rule engine stores events in the KIE session until the events expire either by explicit rule declarations or by implicit reasoning within the Drools rule engine based on inferred data in the KIE base.

2.5.9. Memory management for events

In stream mode, the Drools rule engine uses automatic memory management to maintain events that are stored in KIE sessions. The Drools rule engine can retract from a KIE session any events that no longer match any rule due to their temporal constraints and release any resources held by the retracted events.

The Drools rule engine uses either explicit or inferred expiration to retract outdated events:

- **Explicit expiration:** The Drools rule engine removes events that are explicitly set to expire in rules that declare the `@expires` tag:

DRL rule snippet with explicit expiration

```
declare StockPoint
  @expires( 30m )
end
```

This example rule sets any `StockPoint` events to expire after 30 minutes and to be removed from the KIE session if no other rules use the events.

- **Inferred expiration:** The Drools rule engine can calculate the expiration offset for a given event implicitly by analyzing the temporal constraints in the rules:

DRL rule with temporal constraints

```
rule "Correlate orders"
when
  $bo : BuyOrder($id : id)
  $ae : AckOrder(id == $id, this after[0,10s] $bo)
then
  // Perform an action.
end
```

For this example rule, the Drools rule engine automatically calculates that whenever a `BuyOrder` event occurs, the Drools rule engine needs to store the event for up to 10 seconds and wait for the matching `AckOrder` event. After 10 seconds, the Drools rule engine infers the expiration and removes the event from the KIE session. An `AckOrder` event can only match an existing `BuyOrder` event, so the Drools rule engine infers the expiration if no match occurs and removes the event immediately.

The Drools rule engine analyzes the entire KIE base to find the offset for every event type and to ensure that no other rules use the events that are pending removal. Whenever an implicit expiration clashes with an explicit expiration value, the Drools rule engine uses the greater time frame of the two to store the event longer.

2.6. Drools rule engine queries and live queries

You can use queries with the Drools rule engine to retrieve fact sets based on fact patterns as they are used in rules. The patterns might also use optional parameters.

To use queries with the Drools rule engine, you add the query definitions in DRL files and then obtain the matching results in your application code. While a query iterates over a result collection, you can use any identifier that is bound to the query to access the corresponding fact or fact field by calling the `get()` method with the binding variable name as the argument. If the binding refers to a fact object, you can retrieve the fact handle by calling `getFactHandle()` with the variable name as the parameter.

[QueryResults] | *rule-engine/QueryResults.png*

Figure 26. QueryResults

[QueryResultsRow] | *rule-engine/QueryResultsRow.png*

Figure 27. QueryResultsRow

Example query definition in a DRL file

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

Example application code to obtain and iterate over query results

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

System.out.println( "These people are under the age of 21:" );

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

Invoking queries and processing the results by iterating over the returned set can be difficult when you are monitoring changes over time. To alleviate this difficulty with ongoing queries, Drools provides *live queries*, which use an attached listener for change events instead of returning an iterable result set. Live queries remain open by creating a view and publishing change events for the contents of this view.

To activate a live query, start your query with parameters and monitor changes in the resulting view. You can use the `dispose()` method to terminate the query and discontinue this reactive scenario.

Example query definition in a DRL file

```
query colors(String $color1, String $color2)
    TShirt(mainColor = $color1, secondColor = $color2, $price: manufactureCost)
end
```

Example application code with an event listener and a live query

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

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

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

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

// Open the live query:
LiveQuery query = ksession.openLiveQuery( "colors",
                                          new Object[] { "red", "blue" },
                                          listener );
...
...

// Terminate the live query:
query.dispose()
```

For more live query examples, see Glazed Lists examples for Drools Live Queries.

2.7. Drools rule engine event listeners and debug logging

The Drools rule engine generates events when performing activities such as fact insertions and rule executions. If you register event listeners, the Drools rule engine calls every listener when an activity is performed.

Event listeners have methods that correspond to different types of activities. The Drools rule engine passes an event object to each method; this object contains information about the specific activity.

Your code can implement custom event listeners and you can also add and remove registered event listeners. In this way, your code can be notified of Drools rule engine activity, and you can separate logging and auditing work from the core of your application.

The Drools rule engine supports the following event listeners with the following methods:

Agenda event listener

```
public interface AgendaEventListener
    extends
    EventListener {
    void matchCreated(MatchCreatedEvent event);
    void matchCancelled(MatchCancelledEvent event);
    void beforeMatchFired(BeforeMatchFiredEvent event);
    void afterMatchFired(AfterMatchFiredEvent event);
    void agendaGroupPopped(AgendaGroupPoppedEvent event);
    void agendaGroupPushed(AgendaGroupPushedEvent event);
    void beforeRuleFlowGroupActivated(RuleFlowGroupActivatedEvent event);
    void afterRuleFlowGroupActivated(RuleFlowGroupActivatedEvent event);
    void beforeRuleFlowGroupDeactivated(RuleFlowGroupDeactivatedEvent event);
    void afterRuleFlowGroupDeactivated(RuleFlowGroupDeactivatedEvent event);
}
```

Rule runtime event listener

```
public interface RuleRuntimeEventListener extends EventListener {
    void objectInserted(ObjectInsertedEvent event);
    void objectUpdated(ObjectUpdatedEvent event);
    void objectDeleted(ObjectDeletedEvent event);
}
```

For the definitions of event classes, see the [GitHub repository](#).

Drools includes default implementations of these listeners: `DefaultAgendaEventListener` and `DefaultRuleRuntimeEventListener`. You can extend each of these implementations to monitor specific events.

For example, the following code extends `DefaultAgendaEventListener` to monitor the `AfterMatchFiredEvent` event and attaches this listener to a KIE session. The code prints pattern matches when rules are executed (fired):

Example code to monitor and print `AfterMatchFiredEvent` events in the agenda

```
ksession.addEventListener( new DefaultAgendaEventListener() {
    public void afterMatchFired(AfterMatchFiredEvent event) {
        super.afterMatchFired( event );
        System.out.println( event );
    }
});
```

Drools also includes the following Drools rule engine agenda and rule runtime event listeners for debug logging:

- `DebugAgendaEventListener`
- `DebugRuleRuntimeEventListener`

These event listeners implement the same supported event-listener methods and include a debug print statement by default. You can add additional monitoring code for a specific supported event.

For example, the following code uses the `DebugRuleRuntimeEventListener` event listener to monitor and print all working memory (rule runtime) events:

Example code to monitor and print all working memory events

```
ksession.addEventListener( new DebugRuleRuntimeEventListener() );
```

2.7.1. Practices for development of event listeners

The Drools rule engine calls event listeners during rule processing. The calls block the execution of the Drools rule engine. Therefore, the event listener can affect the performance of the Drools rule engine.

To ensure minimal disruption, follow the following guidelines:

- Any action must be as short as possible.
- A listener class must not have a state. The Drools rule engine can destroy and re-create a listener class at any time.
- Do not use logic that relies on the order of execution of different event listeners.
- Do not include interactions with different entities outside the Drools rule engine within a listener. For example, do not include REST calls for notification of events. An exception is the output of logging information; however, a logging listener must be as simple as possible.
- You can use a listener to modify the state of the Drools rule engine, for example, to change the values of variables.

2.7.2. Configuring a logging utility in the Drools rule engine

The Drools rule engine uses the Java logging API SLF4J for system logging. You can use one of the following logging utilities with the Drools rule engine to investigate Drools rule engine activity, such as for troubleshooting or data gathering:

- Logback
- Apache Commons Logging
- Apache Log4j
- `java.util.logging` package

Procedure

For the logging utility that you want to use, add the relevant dependency to your Maven project or save the relevant XML configuration file in the `org.drools` package of your Drools distribution:

Example Maven dependency for Logback

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>${logback.version}</version>
</dependency>
```

Example logback.xml configuration file in org.drools package

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

Example log4j.xml configuration file in org.drools package

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <category name="org.drools">
    <priority value="debug" />
  </category>
  ...
</log4j:configuration>
```

> ℹ️ If you are developing for an ultra light environment, use the `slf4j-nop` or `slf4j-simple` logger.

2.8. Performance tuning considerations with the Drools rule engine

The following key concepts or suggested practices can help you optimize Drools rule engine performance. These concepts are summarized in this section as a convenience and are explained in more detail in the cross-referenced documentation, where applicable. This section will expand or change as needed with new releases of Drools.

Use sequential mode for stateless KIE sessions that do not require important Drools rule engine updates

Sequential mode is an advanced rule base configuration in the Drools rule engine that enables the Drools rule engine to evaluate rules one time in the order that they are listed in the Drools rule engine agenda without regard to changes in the working memory. As a result, rule execution may be faster in sequential mode, but important updates may not be applied to your rules. Sequential mode applies to stateless KIE sessions only.

To enable sequential mode, set the system property `drools.sequential` to `true`.

For more information about sequential mode or other options for enabling it, see Sequential mode in Phreak.

Use simple operations with event listeners

Limit the number of event listeners and the type of operations they perform. Use event listeners for simple operations, such as debug logging and setting properties. Complicated operations, such as network calls, in listeners can impede rule execution. After you finish working with a KIE session, remove the attached event listeners so that the session can be cleaned, as shown in the following example:

Example event listener removed after use

```
Listener listener = ...;
StatelessKnowledgeSession ksession = createSession();
try {
    ksession.insert(fact);
    ksession.fireAllRules();
    ...
} finally {
    if (session != null) {
        ksession.detachListener(listener);
        ksession.dispose();
    }
}
```

For information about built-in event listeners and debug logging in the Drools rule engine, see Drools rule engine event listeners and debug logging.

Configure `LambdaIntrospector` cache size for an executable model build

You can configure the size of `LambdaIntrospector.methodFingerprintsMap` cache, which is used in an executable model build. The default size of the cache is `32`. When you configure smaller value for the cache size, it reduces memory usage. For example, you can configure system property `drools.lambda.introspector.cache.size` to `0` for minimum memory usage. Note that smaller cache size also slows down the build performance.

Use lambda externalization for executable model

Enable lambda externalization to optimize the memory consumption during runtime. It rewrites lambdas that are generated and used in the executable model. This enables you to reuse the same lambda multiple times with all the patterns and the same constraint. When the rete or phreak is instantiated, the executable model becomes garbage collectible.

To enable lambda externalization for the executable model, include the following property:

```
-Ddrools.externaliseCanonicalModelLambda=true
```

Configure alpha node range index threshold

Alpha node range index is used to evaluate the rule constraint. You can configure the threshold of the alpha node range index using the `drools.alphaNodeRangeIndexThreshold` system property.

The default value of the threshold is 9, indicating that the alpha node range index is enabled when a precedent node contains more than nine alpha nodes with inequality constraints. For example, when you have nine rules similar to `Person(age > 10)`, `Person(age > 20)`, ..., `Person(age > 90)`, then you can have similar nine alpha nodes.

The default value of the threshold is based on the related advantages and overhead. However, if you configure a smaller value for the threshold, then the performance can be improved depending on your rules. For example, you can configure the `drools.alphaNodeRangeIndexThreshold` value to 6, enabling the alpha node range index when you have more than six alpha nodes for a precedent node. You can set a suitable value for the threshold based on the performance test results of your rules.

Enable join node range index

The join node range index feature improves the performance only when there is a large number of facts to be joined, for example, 256*16 combinations. When your application inserts a large number of facts, you can enable the join node range index and evaluate the performance improvement. By default, the join node range index is disabled.

Example `kmodule.xml` file

```
<kbase name="KBase1" betaRangeIndex="enabled">
```

System property for `BetaRangeIndexOption`

```
drools.betaNodeRangeIndexEnabled=true
```

Chapter 3. Rule Language Reference

3.1. Drools Rule Language (DRL)

Drools Rule Language (DRL) is a notation established by the Drools open source business automation project for defining and describing business rules. You define DRL rules in `.drl` text files. A DRL file can contain one or more rules that define at a minimum the rule conditions (`when`) and actions (`then`).

DRL files consist of the following components:

Components in a DRL file

```
package
unit

import

declare    // Optional

query   // Optional

rule "rule name"
    // Attributes
    when
        // Conditions
    then
        // Actions
end

rule "rule2 name"

...
```

The following example DRL rule determines the age limit in a loan application decision service:

Example rule for loan application age limit

```
rule "Underage"
  when
    /applicants[ applicantName : name, age < 21 ]
    $application : /loanApplications[ applicant == applicantName ]
  then
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
end
```

A DRL file can contain single or multiple rules and queries, and can define resource declarations

and attributes that are assigned and used by your rules and queries. The components in a DRL file are grouped in a defined rule unit that serves as a unique namespace for each group of rules. The DRL package followed by the rule unit definition must be listed at the top of a DRL file, and the rules are typically listed last. All other DRL components can follow any order.

Each rule must have a unique name within the rule unit. If you use the same rule name more than once in any DRL file in the unit, the rules fail to compile. Rule names generally must follow standard Java identifier conventions. However, you can enclose rule names with double quotation marks (`rule "rule name"`) to prevent possible compilation errors, especially if you use spaces in rule names.

3.1.1. Packages in DRL

A package is a folder of related assets in Drools, such as data objects, DRL files, decision tables, and other asset types. A package also serves as a unique namespace for each group of rules. A single rule base can contain multiple packages. You typically store all the rules for a package in the same file as the package declaration so that the package is self-contained. However, you can import objects from other packages that you want to use in the rules.

The following example is a package name and namespace for a DRL file in a mortgage application decision service:

Example package definition in a DRL file

```
package org.mortgages;
```

The following railroad diagram shows all the components that may make up a package:
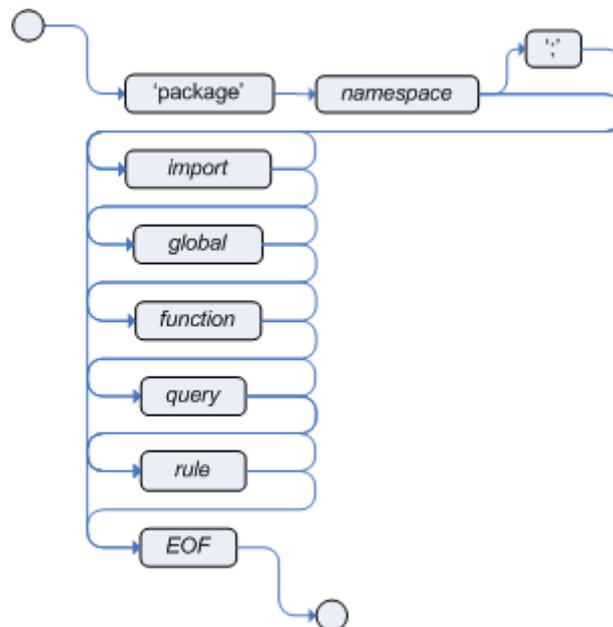
Figure 28. Package

Note that a package *must* have a namespace and be declared using standard Java conventions for package names; i.e., no spaces, unlike rule names which allow spaces. In terms of the order of elements, they can appear in any order in the rule file, with the exception of the `package` and `unit` statements, which must be at the top of the file. In all cases, the semicolons are optional.

Notice that any rule attribute (as described in the section Rule attributes in DRL) may also be written at package level, superseding the attribute's default value. The modified default may still be replaced by an attribute setting within a rule.

3.1.2. Rule units in DRL

A DRL rule unit is a module for rules and a unit of execution. A rule unit collects a set of rules with the declaration of the type of facts that the rules act on. A rule unit also serves as a unique namespace for each group of rules. A single rule base can contain multiple rule units. You typically store all the rules for a unit in the same file as the unit declaration so that the unit is self-contained.

The following example is a rule unit designated in a DRL file in a mortgage application decision service:

Example package definition and rule unit designation in a DRL file

```
package org.mortgages;
unit MortgageRules;
```

To define a rule unit, you declare the relevant fact types and declare the data sources for the types by implementing the `RuleUnitData` interface, and then define the rules in the unit:

Example DRL rule unit file

```
package org.mortgages;
unit MortgageRules;

import org.drools.ruleunits.api.DataSource;
import org.drools.ruleunits.api.DataStream;

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

declare MortgageRules extends RuleUnitData
  persons: DataStream<Person> = DataSource.createStream()
end

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

To separate the fact types from the rule unit for use with other DRL rules, you can declare the types in a separate DRL file and then use the DRL rule file to declare the data sources by using the `RuleUnitData` interface implementation:

Example DRL type declaration as a separate file

```
package org.mortgages;

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

Example DRL rule unit file without explicitly defined types

```
package org.mortgages;
unit MortgageRules;

import org.drools.ruleunits.api.DataSource;
import org.drools.ruleunits.api.DataStream;

declare MortgageRules extends RuleUnitData
  persons: DataStream<Person> = DataSource.createStream()
end

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

In this example, `persons` is a `DataStream` data source for facts of type `Person`. Data sources are typed sources of data that rule units can subscribe to for updates. You interact with the rule unit through the data sources it exposes. A data source can be a `DataStream` source for append-only storage, a `DataStore` source for writable storage to add or remove data, or a `SingletonStore` source for writable storage to set and clear a single element.

As part of your data source declaration, you also import `org.drools.ruleunits.api.DataSource` and the relevant data source support, such as `import org.drools.ruleunits.api.DataStream` in this example.

You can add several rules to the same DRL file, or further break down the rule set and type declarations by creating more files. However you construct your rule sets, ensure that all DRL rule files exist in the same directory and start with the correct `package` and `unit` declarations.

3.1.2.1. Rule unit use case

As an additional rule unit use case, consider the following example decision service that evaluates incoming data from a heat sensor for temperature measurements and produces alerts when the temperature is above a specified threshold.

This example service uses the following `types.drl` file in the `src/main/resources/org/acme` folder of the Drools project to declare the `Temperature` and the `Alert` fact types:

Example DRL type declarations

```
package com.acme;

declare Temperature
  value: double
end

declare Alert
    severity: String
    message: String
end
```

To define DRL rules that pattern-match against `Temperature` values, the example service must expose an entry point for the incoming data to the Drools rule engine and publish alerts on a separate channel. To establish this data source for decision data, the example service uses a rule unit with `DataStream` data sources for `Temperature` objects and for `Alert` objects.

The `DataStream` data source is an append-only store for incoming data, similar to a queue. This type of data source is logical for both sources in this example because the temperature data is coming from an external source (the sensor) and the service publishes the alerts externally as they are produced.

The example service uses the following `MonitoringService.drl` file in the same `src/main/resources/com/acme` folder of the Drools project to declare the data sources for the fact types and defines the rules for the rule unit:

```
package com.acme;
unit MonitoringService;

import org.drools.ruleunits.api.DataSource;
import org.drools.ruleunits.api.DataStream;

declare MonitoringService extends RuleUnitData
  temperatures: DataStream<Temperature> = DataSource.createStream()
  alertData: DataStream<Alert> = DataSource.createStream()
end

rule "tooHot"
when
    $temp : /temperatures[value >= 80]
then
    alertData.append(new Alert("HIGH", "Temperature exceeds threshold: " +
temp.value));
end
```

The rule unit implements the required `RuleUnitData` interface and declares the data sources for the previously defined types. The sample rule raises an alert when the temperature reaches or exceeds 80 degrees.

3.1.2.2. Data sources for DRL rule units

Data sources are typed sources of data that rule units can subscribe to for updates. You interact with the rule unit through the data sources it exposes.

Drools supports the following types of data sources. When you declare data sources in DRL rule files, the sources are internally rendered as shown in these examples.

- `DataStream`: An append-only storage option. Use this storage option when you want to publish or share data values. You can use the notation `DataSource.createStream()` to return a `DataStream<T>` object and use the method `append(T)` to add more data.

  Example DataStream data source definition

  ```
  DataStream<Temperature> temperatures = DataSource.createStream();
  // Append value and notify all subscribers
  temperatures.append(new Temperature(100));
  ```

- `DataStore`: A writable storage option for adding or removing data and then notifying all subscribers that mutable data has been modified. Rules can pattern-match against incoming values and update or remove available values. For users familiar with Drools, this option is equivalent to a typed version of an entry point. In fact, a `DataStore<Object>` is equivalent to an old-style entry point.

Example DataStore data source definition

```
DataStore<Temperature> temperatures = DataSource.createStore();
Temperature temp = new Temperature(100);
// Add value `t` and notify all subscribers
DataHandle t = temperatures.add(temp);
temp.setValue(50);
// Notify all subscribers that the value referenced by `t` has changed
temperatures.update(t, temp);
// Remove value referenced by `t` and notify all subscribers
temperatures.remove(t);
```

- **SingletonStore**: A writable storage option for setting or clearing a single element and then notifying all subscribers that the element has been modified. Rules can pattern-match against the value and update or clear available values. For users familiar with Drools, this option is equivalent to a global. In fact, a `Singleton<Object>` is similar to an old-style global, except that when used in conjuction with rules, you can pattern-match against it.

Example SingletonStore data source definition

```
SingletonStore<Temperature> temperature = DataSource.createSingleton();
Temperature temp = new Temperature(100);
// Add value `temp` and notify all subscribers
temperature.set(temp);
temp.setValue(50);
// Notify all subscribers that the value has changed
temperature.update();

Temperature temp2 = new Temperature(200);
// Overwrite contained value with `temp2` and notify all subscribers
temperature.set(temp2);
temp2.setValue(150);
// Notify all subscribers that the value has changed
temperature.update();

// Clear store and notify all subscribers
temperature.clear();
```

Subscribers to a data source are known as *data processors*. A data processor implements the `DataProcessor<T>` interface. This interface contains callbacks to all the events that a subscribed data source can trigger.

Example DataStream data processor

```java
public interface DataProcessor<T> {
    void insert(DataHandle handle, T object);
    void update(DataHandle handle, T object);
    void delete(DataHandle handle);
}
```

The `DataHandle` method is an internal reference to an object of a data source. Each callback method might or might not be invoked, depending on whether the corresponding data source implements the capability. For example, a `DataStream` source invokes only the `insert` callback, whereas a `SingletonStore` source invokes the `insert` callback on `set` and the `delete` callback on `clear` or before an overwriting `set`.

3.1.2.3. DRL rule unit declaration using Java

As an alternative to declaring fact types and rule units in DRL files, you can also declare types and units using Java classes. In this case, you add the source code to the `src/main/java` folder of your Drools project instead of `src/main/resources`.

For example, the following Java classes define the type and rule unit declarations for the example temperature monitoring service:

Example Temperature class

```java
package com.acme;

public class Temperature {
    private final double value;
    public Temperature(double value) { this.value = value; }
    public double getValue() { return value; }
}
```

Example Alert class

```java
package com.acme;

public class Alert {
    private final String severity
    private final String message;
    public Alert(String severity, String message) {
        this.severity = severity;
        this.message = message;
    }
    public String getSeverity() { return severity; }
    public String getMessage() { return message; }
```

Example rule unit class

```
package com.acme;

import org.drools.ruleunits.api.DataSource;
import org.drools.ruleunits.api.DataStream;

public class MonitoringService implements RuleUnitData {
    private DataStream<Temperature> temperatures = DataSource.createStream();
    private DataStream<Alert> alertData = DataSource.createStream();
    public DataStream<Temperature> getTemperatures() { return temperatures; }
    public DataStream<Alert> getAlertData() { return alertData; }
}
```

In this scenario, the DRL rule files then stand alone in the `src/main/resources` folder and consist of the `unit` and the rules, with no direct declarations, as shown in the following example:

Example DRL rule unit file without declarations

```
package com.acme;
unit MonitoringService;

rule "tooHot"
  when
    $temp : /temperatures[value >= 80]
  then
    alertData.append(new Alert("HIGH", "Temperature exceeds threshold: " +
temp.value));
end
```

3.1.2.4. DRL rule units with BPMN processes

If you use a DRL rule unit as part of a business rule task in a Business Process Model and Notation (BPMN) process in your Drools project, you do not need to create an explicit data type declaration or a rule unit class that implements the `RuleUnitData` interface. Instead, you designate the rule unit in the DRL file as usual and specify the rule unit in the format `unit:PACKAGE_NAME.UNIT_NAME` in the implementation details for the business rule task in the BPMN process. When you build the project, the business process implicitly declares the rule unit as part of the business rule task to execute the DRL file.

For example, the following is a DRL file with a rule unit designation:

Example DRL rule unit file

```
package com.acme;
unit MonitoringService;

rule "tooHot"
  when
    $temp : Temperature( value >= 80 ) from temperature
  then
    alertData.add(new Alert("HIGH", "Temperature exceeds threshold: " + temp.value));
end
```

In the relevant business process in a BPMN 2.0 process modeler, you select the business rule task and for the **Implementation/Execution** property, you set the rule language to `DRL` and the rule flow group to `unit:com.acme.MonitoringService`.

This rule unit syntax specifies that you are using the `com.acme.MonitoringService` rule unit instead of a traditional rule flow group. This is the rule unit that you referenced in the example DRL file. When you build the project, the business process implicitly declares the rule unit as part of the business rule task to execute the DRL file.

3.1.3. Import statements in DRL

Figure 29. Import

Similar to import statements in Java, imports in DRL files identify the fully qualified paths and type names for any objects that you want to use in the rules. You specify the package and data object in the format `packageName.objectName`, with multiple imports on separate lines. The Drools rule engine automatically imports classes from the Java package with the same name as the DRL package and from the package `java.lang`.

The following example is an import statement for a loan application object in a mortgage application decision service:

Example import statement in a DRL file

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

3.1.4. Type declarations and metadata in DRL
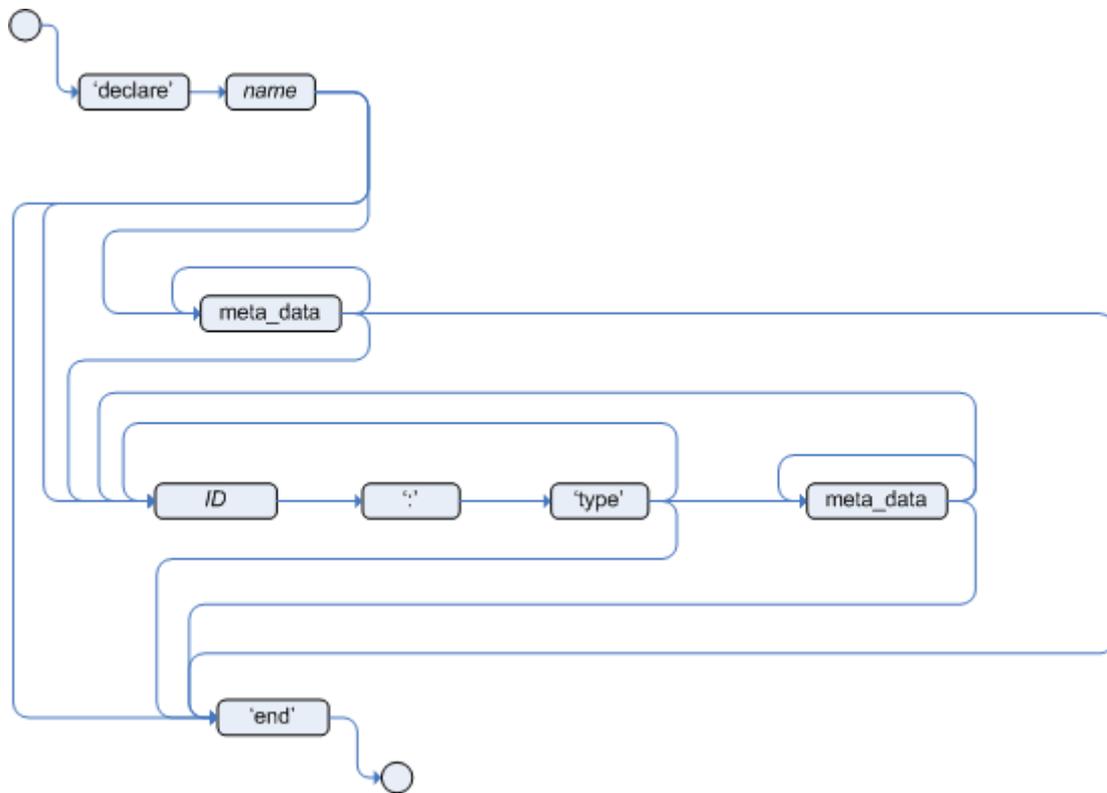
Figure 30. Type declaration
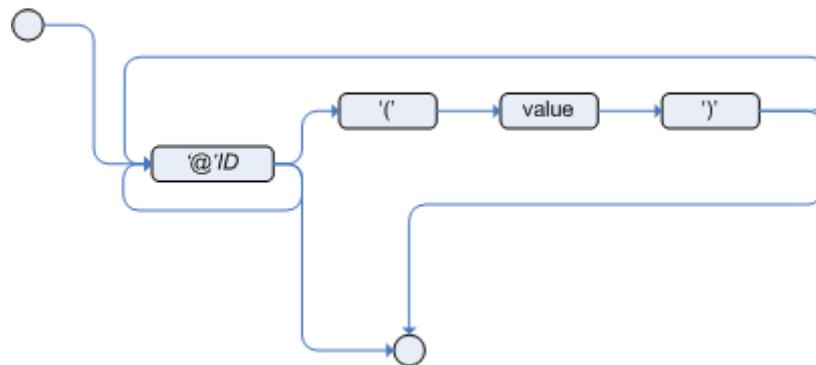
Figure 31. Metadata

Declarations in DRL files define new fact types or metadata for fact types to be used by rules in the DRL file:

- **New fact types:** The default fact type in the `java.lang` package of Drools is `Object`, but you can declare other types in DRL files as needed. Declaring fact types in DRL files enables you to define a new fact model directly in the Drools rule engine, without creating models in a lower-level language like Java. You can also declare a new type when a domain model is already built and you want to complement this model with additional entities that are used mainly during the reasoning process.

- **Metadata for fact types:** You can associate metadata in the format `@KEY( VALUE )` with new or existing facts. Metadata can be any kind of data that is not represented by the fact attributes and is consistent among all instances of that fact type. The metadata can be queried at run time by the Drools rule engine and used in the reasoning process.

3.1.4.1. Type declarations without metadata in DRL

A declaration of a new fact does not require any metadata, but must include a list of attributes or fields. If a type declaration does not include identifying attributes, the Drools rule engine searches for an existing fact class in the classpath and raises an error if the class is missing.

For example, the following DRL file contains a declaration of a new fact type `Person` from a `persons` data source and uses no metadata:

Example declaration of a new fact type with a rule

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

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

In this example, the new fact type `Person` has the three attributes `name`, `dateOfBirth`, and `address`. Each attribute has a type that can be any valid Java type, including another class that you create or a fact type that you previously declared. The `dateOfBirth` attribute has the type `java.util.Date`, from the Java API, and the `address` attribute has the previously defined fact type `Address`.

To avoid writing the fully qualified name of a class every time you declare it, you can define the full class name as part of the `import` clause:

Example type declaration with the fully qualified class name in the import

```
import java.util.Date

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

When you declare a new fact type, the Drools rule engine generates at compile time a Java class representing the fact type. The generated Java class is a one-to-one JavaBeans mapping of the type definition.

For example, the following Java class is generated from the example `Person` type declaration:

Generated Java class for the Person fact type declaration

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

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

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

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

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

You can then use the generated class in your rules like any other fact, as illustrated in the previous rule example with the `Person` type declaration from a `persons` data source:

Example rule that uses the declared Person fact type

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

3.1.4.2. Enumerative type declarations in DRL

DRL supports the declaration of enumerative types in the format `declare enum FACT_TYPE`, followed by a comma-separated list of values ending with a semicolon. You can then use the enumerative list in the rules in the DRL file.

For example, the following enumerative type declaration defines days of the week for an employee scheduling rule:

Example enumerative type declaration with a scheduling rule

```
declare enum DaysOfWeek

SUN("Sunday"),MON("Monday"),TUE("Tuesday"),WED("Wednesday"),THU("Thursday"),FRI("Frida
y"),SAT("Saturday");

    fullName : String
end

rule "Using a declared Enum"
  when
    $emp : /employees[ dayOff == DaysOfWeek.MONDAY ]
  then
    ...
end
```

3.1.4.3. Extended type declarations in DRL

DRL supports type declaration inheritance in the format `declare FACT_TYPE_1 extends FACT_TYPE_2`. To extend a type declared in Java by a subtype declared in DRL, you repeat the parent type in a declaration statement without any fields.

For example, the following type declarations extend a `Student` type from a top-level `Person` type, and a `LongTermStudent` type from the `Student` subtype:

Example extended type declarations

```
import org.people.Person

declare Person end

declare Student extends Person
    school : String
end

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

3.1.4.4. Type declarations with metadata in DRL

You can associate metadata in the format `@KEY( VALUE )` (the value is optional) with fact types or fact attributes. Metadata can be any kind of data that is not represented by the fact attributes and is consistent among all instances of that fact type. The metadata can be queried at run time by the Drools rule engine and used in the reasoning process. Any metadata that you declare before the attributes of a fact type are assigned to the fact type, while metadata that you declare after an attribute are assigned to that particular attribute.

In the following example, the two metadata attributes `@author` and `@dateOfCreation` are declared for the `Person` fact type, and the two metadata items `@key` (literal) and `@maxLength` are declared for the `name` attribute. The `@key` literal metadata attribute has no required value, so the parentheses and the value are omitted.

Example metadata declaration for fact types and attributes

```
import java.util.Date

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

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

For declarations of metadata attributes for existing types, you can identify the fully qualified class name as part of the `import` clause for all declarations or as part of the individual `declare` clause:

Example metadata declaration for an imported type

```
import org.drools.examples.Person

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

Example metadata declaration for a declared type

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

3.1.4.5. Metadata tags for fact type and attribute declarations in DRL

Although you can define custom metadata attributes in DRL declarations, the Drools rule engine also supports the following predefined metadata tags for declarations of fact types or fact type attributes.

> The examples in this section that refer to the `VoiceCall` class assume that the sample application domain model includes the following class details:
>
> *VoiceCall fact class in an example Telecom domain model*
>
> ```
> public class VoiceCall {
>   private String  originNumber;
>   private String  destinationNumber;
>   private Date    callDateTime;
>   private long    callDuration;  // in milliseconds
>
>   // Constructors, getters, and setters
> }
> ```

@role

This tag determines whether a given fact type is handled as a regular fact or an event in the Drools rule engine during complex event processing.

Default parameter: `fact`

Supported parameters: `fact`, `event`

```
@role( fact | event )
```

Example: Declare VoiceCall as event type

```
declare VoiceCall
  @role( event )
end
```

@timestamp

This tag is automatically assigned to every event in the Drools rule engine. By default, the time is provided by the session clock and assigned to the event when it is inserted into the working memory of the Drools rule engine. You can specify a custom time stamp attribute instead of the default time stamp added by the session clock.

Default parameter: The time added by the Drools rule engine session clock

Supported parameters: Session clock time or custom time stamp attribute

```
@timestamp( <em>ATTRIBUTE_NAME</em> )
```

Example: Declare VoiceCall timestamp attribute

```
declare VoiceCall
  @role( event )
  @timestamp( callDateTime )
end
```

@duration

This tag determines the duration time for events in the Drools rule engine. Events can be interval-based events or point-in-time events. Interval-based events have a duration time and persist in the working memory of the Drools rule engine until their duration time has lapsed. Point-in-time events have no duration and are essentially interval-based events with a duration of zero. By default, every event in the Drools rule engine has a duration of zero. You can specify a custom duration attribute instead of the default.

Default parameter: Null (zero)

Supported parameters: Custom duration attribute

```
@duration( <em>ATTRIBUTE_NAME</em> )
```

Example: Declare VoiceCall duration attribute

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

@expires

This tag determines the time duration before an event expires in the working memory of the Drools rule engine. By default, an event expires when the event can no longer match and activate any of the current rules. You can define an amount of time after which an event should expire. This tag definition also overrides the implicit expiration offset calculated from temporal constraints and sliding windows in the KIE base. This tag is available only when the Drools rule engine is running in stream mode.

Default parameter: Null (event expires after event can no longer match and activate rules)

Supported parameters: Custom `timeOffset` attribute in the format `[#d][#h][#m][#s][#ms]`

```
@expires( <em>TIME_OFFSET</em> )
```

Example: Declare expiration offset for VoiceCall events

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

@typesafe

This tab determines whether a given fact type is compiled with or without type safety. By default, all type declarations are compiled with type safety enabled. You can override this behavior to type-unsafe evaluation, where all constraints are generated as MVEL constraints and executed dynamically. This is useful when dealing with collections that do not have any generics or mixed type collections.

Default parameter: `true`

Supported parameters: `true`, `false`

```
@typesafe( <em>BOOLEAN</em> )
```

Example: Declare VoiceCall for type-unsafe evaluation

```
declare VoiceCall
  @role( fact )
  @typesafe( false )
end
```

@serialVersionUID

This tag defines an identifying `serialVersionUID` value for a serializable class in a fact declaration. If a serializable class does not explicitly declare a `serialVersionUID`, the serialization run time calculates a default `serialVersionUID` value for that class based on various aspects of the class, as described in the Java Object Serialization Specification. However, for optimal deserialization results and for greater compatibility with serialized KIE sessions, set the `serialVersionUID` as needed in the relevant class or in your DRL declarations.

Default parameter: Null

Supported parameters: Custom `serialVersionUID` integer

```
@serialVersionUID( <em>INTEGER</em> )
```

Example: Declare serialVersionUID for a VoiceCall class

```
declare VoiceCall
  @serialVersionUID( 42 )
end
```

@key

This tag enables a fact type attribute to be used as a key identifier for the fact type. The generated class can then implement the `equals()` and `hashCode()` methods to determine if two instances of the type are equal to each other. The Drools rule engine can also generate a constructor using all the key attributes as parameters.

Default parameter: None

Supported parameters: None

```
<em>ATTRIBUTE_DEFINITION</em> @key
```

Example: Declare Person type attributes as keys

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

For this example, the Drools rule engine checks the `firstName` and `lastName` attributes to determine if two instances of `Person` are equal to each other, but it does not check the `age` attribute. The Drools rule engine also implicitly generates three constructors: one without parameters, one with the `@key` fields, and one with all fields:

Example constructors from the key declarations

```
Person() // Empty constructor

Person( String firstName, String lastName )

Person( String firstName, String lastName, int age )
```

You can then create instances of the type based on the key constructors, as shown in the following example:

Example instance using the key constructor

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

3.1.5. Queries in DRL
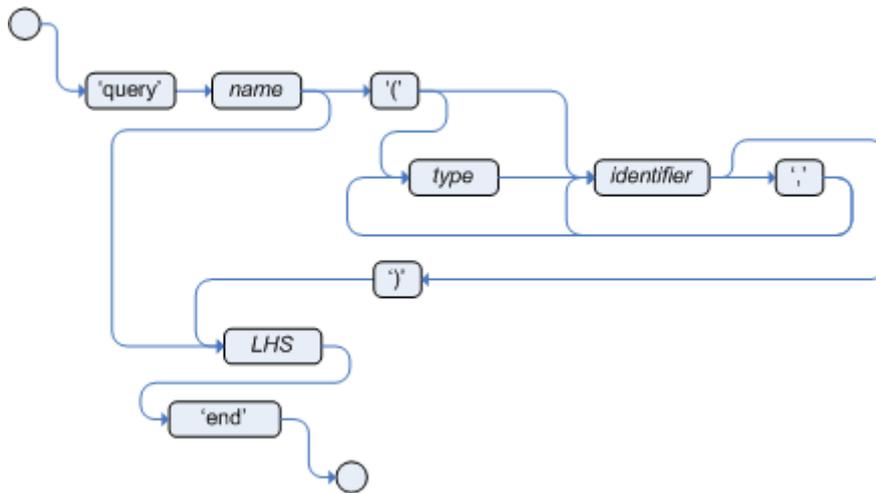
Figure 32. Query

Queries in DRL files search the working memory of the Drools rule engine for facts related to the rules in the DRL file. You add the query definitions in DRL files and then obtain the matching results in your application code. Queries search for a set of defined conditions and do not require `when` or `then` specifications. Query names are scoped to the rule unit, so each query name must be unique within the same rule unit. In Drools, queries are automatically exposed as REST endpoints.

The following example is a query definition for an `Alert` object with a `severity` field set to `HIGH`:

Example query definition in a DRL file

```
package com.acme;
unit MonitoringService;

query highSeverity
    alerts : /alertData[ severity == "HIGH" ]
end
```

Drools automatically exposes this query through an endpoint `/high-severity`.

For this example, assume that the `MonitoringService` rule unit class has the following form:

Example Java rule unit class

```java
package com.acme;

import org.drools.ruleunits.api.DataSource;
import org.drools.ruleunits.api.DataStream;

public class MonitoringService implements RuleUnitData {
    private DataStream<Temperature> temperature = DataSource.createStream();
    private DataStream<Alert> alertData = DataSource.createStream();
    public DataStream<Temperature> getTemperature() { return temperature; }
    public DataStream<Alert> getAlertData() { return alertData; }
}
```

In this case, you can invoke the query using the following command:

Example POST request to the /high-severity endpoint

```
$ curl -X POST \
        -H 'Accept: application/json' \
        -H 'Content-Type: application/json' \
        -d '{ "eventData": [ { "type": "temperature", "value" : 20 }, { "type":
"temperature", "value" : 100 } ] }' \
        http://localhost:8080/high-severity
```

Example response (JSON)

```json
{
    "alerts" : [
        {
            "severity" : "HIGH",
            "message" : "Temperature exceeds threshold: 100"
        }
    ]
}
```

This example submits the data to the `eventData` data source and returns the result of the `highSeverity` query as a response.

3.1.6. Rule attributes in DRL
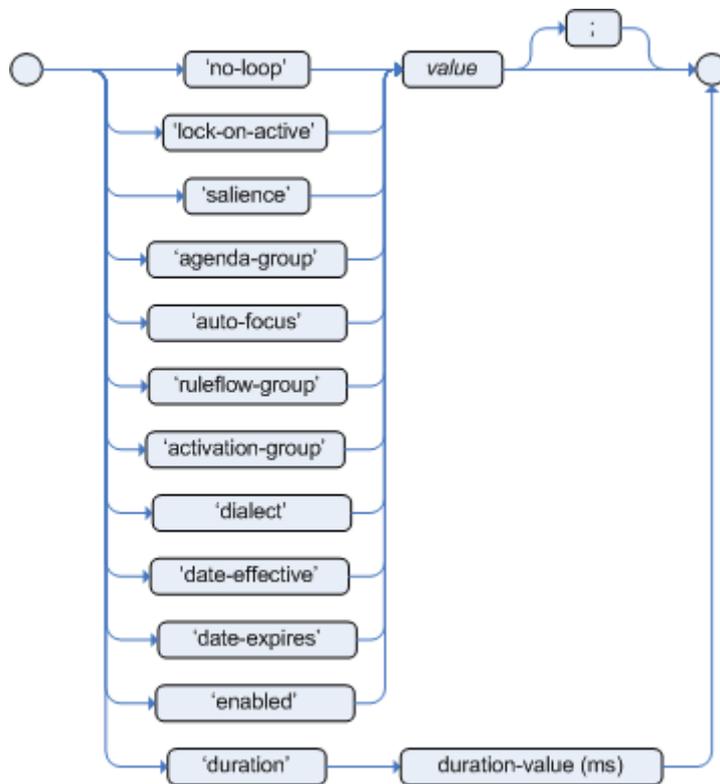
Figure 33. Rule attributes

Rule attributes are additional specifications that you can add to business rules to modify rule behavior. In DRL files, you typically define rule attributes above the rule conditions and actions, with multiple attributes on separate lines, in the following format:

```
rule "rule_name"
    // Attribute
    // Attribute
    when
        // Conditions
    then
        // Actions
end
```

The following table lists the names and supported values of the attributes that you can assign to rules:

Table 4. 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`

Attribute	Value
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"`
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`
activation-group	A string identifying an activation (or XOR) group to which you want to assign the rule. In activation groups, only one rule can be activated. The first rule to fire will cancel all pending activations of all rules in the activation group. Example: `activation-group "GroupName"`
duration	A long integer value defining the duration of time in milliseconds after which the rule can be activated, if the rule conditions are still met. Example: `duration 10000`
timer	A string identifying either `int` (interval) or `cron` timer definitions for scheduling the rule. Example: `timer ( cron:* 0/15 * * * ? )` (every 15 minutes)
calendar	A Quartz calendar definition for scheduling the rule. Example: `calendars "* * 0-7,18-23 ? * *"` (exclude non-business hours)

Attribute	Value
`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`
`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`
`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"`

3.1.7. Rule conditions in DRL
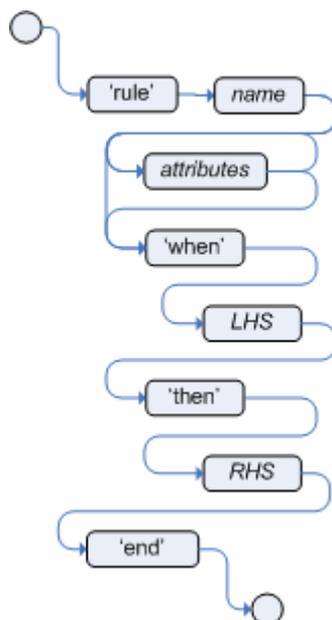
Figure 34. Rule
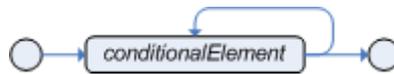
Figure 35. Conditional element in a rule

The `when` part of a DRL rule (also known as the *Left Hand Side (LHS)* of the rule) contains the conditions that must be met to execute an action. Conditions consist of a series of stated OOPath expressions of patterns and constraints, with optional bindings and supported rule condition elements (keywords), based on the available data objects in the package. OOPath is an object-oriented syntax extension to XPath for navigating through related elements while handling collections and filtering constraints.

For example, in a decision service that raises alerts when the temperature reaches or exceeds 80 degrees, a rule `tooHot` contains the `when` condition `/temperature[value >= 80]`.

> **i** DRL uses `when` instead of `if` because `if` is typically part of a procedural execution flow during which a condition is checked at a specific point in time. In contrast, `when` indicates that the condition evaluation is not limited to a specific evaluation sequence or point in time, but instead occurs continually at any time. Whenever the condition is met, the actions are executed.

If the `when` section is empty, then the conditions are considered to be true and the actions in the `then` section are executed the first time the rules are fired. This is useful if you want to use rules to set up the Drools rule engine state.

The following example rule uses empty conditions to insert a fact every time the rule is executed:

Example rule without conditions

```
rule "start-up"
  when
    // Empty
  then   // Actions to be executed once
    alerts.add( new Alert("INFO", "System started") );
end
```

Formally, the core grammar of an OOPath expression is defined in extended Backus-Naur form (EBNF) notation in the following way:

EBNF notation for OOPath expressions

```
OOPExpr = [ID ( ":" | ":=" )] ( "/" | "?/" ) OOPSegment { ( "/" | "?/" | "." )
OOPSegment } ;
OOPSegment = ID ["#" ID] ["[" ( Number | Constraints ) "]"]
```

3.1.7.1. OOPath expressions and constraints

An *OOPath expression* of a pattern in a DRL rule condition is the segment to be matched by the Drools rule engine. An OOPath expression can potentially match each fact that is inserted into the working memory of the Drools rule engine. It can also contain constraints to further define the

facts to be matched.

In the simplest form, with no constraints, an OOPath expression matches a fact in the given data source. In the following example with a `DataSource<Person>` named `persons`, the expression matches against all `Person` objects in the data source of the Drools rule engine:

Example expression for a single fact type

```
/persons
```

Patterns can also refer to superclasses or even interfaces, potentially matching facts from many different classes. For example, the following pattern matches all `Student` subtypes of the `Person` object:

Example pattern for subtypes

```
/persons # Student
```

Square brackets in a pattern enclose the constraints, such as the following constraint on the person's age:

Example pattern with a constraint

```
/persons[ age == 50 ]
```

A *constraint* is an expression that returns `true` or `false`. Constraints in DRL are essentially Java expressions with some enhancements, such as property access, and some differences, such as `equals()` and `!equals()` semantics for `==` and `!=` (instead of the usual `same` and `not same` semantics).

Any JavaBeans property can be accessed directly from pattern constraints. A JavaBeans property is exposed internally using a standard JavaBeans getter that takes no arguments and returns something. For example, the `age` property is written as `age` in DRL instead of the getter `getAge()`:

DRL constraint syntax with JavaBeans properties

```
/persons[ age == 50 ]

// This is equivalent to the following getter format:

/persons[ getAge() == 50 ]
```

Drools uses the standard JDK `Introspector` class to achieve this mapping and follows the standard JavaBeans specification. For optimal Drools rule engine performance, use the property access format, such as `age`, instead of using getters explicitly, such as `getAge()`.

Do not use property accessors to change the state of the object in a way that might affect the rules because the Drools rule engine caches the results of the match between invocations for higher efficiency.

For example, do not use property accessors in the following ways:

```
public int getAge() {
    age++; // Do not do this.
    return age;
}
```

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

Instead of following the second example, insert a fact that wraps the current date in the working memory and update that fact between rule executions as needed.

However, if the getter of a property cannot be found, the compiler uses the property name as a fallback method name, without arguments:

Fallback method if object is not found

```
/persons[ age == 50 ]

// If `Person.getAge()` does not exist, the compiler uses the following syntax:

/persons[ age() == 50 ]
```

You can also nest access properties in patterns, as shown in the following example. Nested properties are indexed by the Drools rule engine.

Example pattern with nested property access

```
/persons[ address.houseNumber == 50 ]

// This is equivalent to the following expression:

/persons[ getAddress().getHouseNumber() == 50 ]
```

You can use any Java expression that returns a `boolean` value as a constraint inside the parentheses of a pattern. Java expressions can be mixed with other expression enhancements, such as property access:

Example pattern with a constraint using property access and Java expression

```
/persons[ age == 50 ]
```

You can change the evaluation priority by using parentheses, as in any logical or mathematical expression:

Example evaluation order of constraints

```
/persons[ age > 100 && ( age % 10 == 0 ) ]
```

You can also reuse Java methods in constraints, as shown in the following example:

Example constraints with reused Java methods

```
/persons[ Math.round( weight / ( height * height ) ) < 25.0 ]
```

> ⚠️ Do not use constraints to change the state of the object in a way that might affect the rules because the Drools rule engine caches the results of the match between invocations for higher efficiency. Any method that is executed on a fact in the rule conditions must be a read-only method. Also, the state of a fact should not change between rule invocations unless those facts are marked as updated in the working memory on every change.
>
> For example, do not use a pattern constraint in the following ways:
>
> ```
> /persons[ incrementAndGetAge() == 10 ] // Do not do this.
> ```
>
> ```
> /persons[ System.currentTimeMillis() % 1000 == 0 ] // Do not do this.
> ```

Standard Java operator precedence applies to constraint operators in DRL, and DRL operators follow standard Java semantics except for the `==` and `!=` operators.

The `==` operator uses null-safe `equals()` semantics instead of the usual `same` semantics. For example, the pattern `/persons[ firstName == "John" ]` is similar to `java.util.Objects.equals(person.getFirstName(), "John")`, and because `"John"` is not null, the pattern is also similar to `"John".equals(person.getFirstName())`.

The `!=` operator uses null-safe `!equals()` semantics instead of the usual `not same` semantics. For example, the pattern `/persons[ firstName != "John" ]` is similar to `!java.util.Objects.equals(person.getFirstName(), "John")`.

If the field and the value of a constraint are of different types, the Drools rule engine uses type coercion to resolve the conflict and reduce compilation errors. For instance, if `"ten"` is provided as a string in a numeric evaluator, a compilation error occurs, whereas `"10"` is coerced to a numeric 10. In coercion, the field type always takes precedence over the value type:

Example constraint with a value that is coerced

```
/persons[ age == "10" ] // "10" is coerced to 10
```

For groups of constraints, you can use a delimiting comma `,` to use implicit `and` connective semantics:

Example patterns with multiple constraints

```
// Person is at least 50 years old and weighs at least 80 kilograms:
/persons[ age > 50, weight > 80 ]

// Person is at least 50 years old, weighs at least 80 kilograms, and is taller than 2
meters:
/persons[ age > 50, weight > 80, height > 2 ]
```

> ℹ️ Although the `&&` and `,` operators have the same semantics, they are resolved with different priorities. The `&&` operator precedes the `||` operator, and both the `&&` and `||` operators together precede the `,` operator. Use the comma operator at the top-level constraint for optimal Drools rule engine performance and human readability.

You cannot embed a comma operator in a composite constraint expression, such as in parentheses:

Example of misused comma in composite constraint expression

```
// Do not use the following format:
/persons[ ( age > 50, weight > 80 ) || height > 2 ]

// Use the following format instead:
/persons[ ( age > 50 && weight > 80 ) || height > 2 ]
```

3.1.7.2. Bound variables in patterns and constraints

You can bind variables to OOPath expressions of patterns and constraints to refer to matched objects in other portions of a rule. Bound variables can help you define rules more efficiently or more consistently with how you annotate facts in your data model.

For example, the following DRL rule uses the variable `$p` for an OOPath expression with the `Person` fact:

Pattern with a bound variable

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

Similarly, you can also bind variables to nested properties, as shown in the following example:

```
// Two persons of the same age:
/persons[ firstAge : age ]  // Binding
and
/persons[ age == firstAge ] // Constraint expression
```

> Ensure that you separate constraint bindings and constraint expressions for clearer and more efficient rule definitions. Although mixed bindings and expressions are supported, they can complicate patterns and affect evaluation efficiency.
>
> ```
> // Do not use the following format:
> /persons[ age : age * 2 < 100 ]
>
> // Use the following format instead:
> /persons[ age * 2 < 100, $age : age ]
> ```

3.1.7.3. Nested constraints and inline casts

In some cases, you might need to access multiple properties of a nested object, as shown in the following example:

Example pattern to access multiple properties

```
/persons[ name == "mark", address.city == "london", address.country == "uk" ]
```

You can group these property accessors to nested objects for more readable rules, as shown in the following example:

Example pattern with grouped constraints

```
/persons[ name == "mark"]/address[ city == "london", country == "uk" ]
```

When you work with nested objects, you can use the syntax TYPE#SUB_TYPE to cast to a subtype and make the getters from the parent type available to the subtype. You can use either the object name or fully qualified class name, and you can cast to one or multiple subtypes, as shown in the

following examples:

Example patterns with inline casting to a subtype

```
// Inline casting with subtype name:
/persons[ name == "mark"]/address#LongAddress[ country == "uk" ]

// Inline casting with fully qualified class name:
/persons[ name == "mark"]/address#org.domain.LongAddress[ country == "uk" ]

// Multiple inline casts:
/persons[ name == "mark" ]/address#LongAddress/country#DetailedCountry[ population >
10000000 ]
```

These example patterns cast `Address` to `LongAddress`, and additionally to `DetailedCountry` in the last example, making the parent getters available to the subtypes in each case.

3.1.7.4. Date literal in constraints

By default, the Drools rule engine supports the date format `dd-mmm-yyyy`. You can customize the date format, including a time format mask if needed, by providing an alternative format mask with the system property `drools.dateformat="dd-mmm-yyyy hh:mm"`. You can also customize the date format by changing the language locale with the `drools.defaultlanguage` and `drools.defaultcountry` system properties. For example, the locale of Thailand is set as `drools.defaultlanguage=th` and `drools.defaultcountry=TH`.

Example pattern with a date literal restriction

```
/persons[ bornBefore < "27-Oct-2009" ]
```

3.1.7.5. Auto-boxing and primitive types

Drools attempts to preserve numbers in their primitive or object wrapper form, so a variable bound to an int primitive when used in a code block or expression will no longer need manual unboxing; unlike early Drools versions where all primitives were autoboxed, requiring manual unboxing. A variable bound to an object wrapper will remain as an object; the existing JDK 1.5 and JDK 5 rules to handle auto-boxing and unboxing apply in this case. When evaluating field constraints, the system attempts to coerce one of the values into a comparable format; so a primitive is comparable to an object wrapper.

3.1.7.6. Supported operators in DRL constraints

DRL supports standard Java semantics for operators in constraints, with some exceptions and with some additional operators that are unique in DRL. The following list summarizes the operators that are handled differently in DRL constraints than in standard Java semantics or that are unique in DRL constraints.

`/`, `#`

 Use the `/` operator to group property accessors to nested objects, and use the `#` operator to cast to

a subtype in nested objects. Casting to a subtype makes the getters from the parent type available to the subtype. You can use either the object name or fully qualified class name, and you can cast to one or multiple subtypes.

Example constraints with nested objects

```
// Ungrouped property accessors:
/persons[ name == "mark", address.city == "london", address.country == "uk" ]

// Grouped property accessors:
/persons[ name == "mark"]/address[ city == "london", country == "uk" ]
```

Example constraints with inline casting to a subtype

```
// Inline casting with subtype name:
/persons[ name == "mark", address#LongAddress.country == "uk" ]

// Inline casting with fully qualified class name:
/persons[ name == "mark", address#org.domain.LongAddress.country == "uk" ]

// Multiple inline casts:
/persons[ name == "mark", address#LongAddress.country#DetailedCountry.population >
10000000 ]
```

!.

Use this operator to dereference a property in a null-safe way. The value to the left of the `!.` operator must be not null (interpreted as `!= null`) in order to give a positive result for pattern matching.

Example constraint with null-safe dereferencing

```
/persons[ $streetName : address!.street ]

// This is internally rewritten in the following way:

/persons[ address != null, $streetName : address.street ]
```

[]

Use this operator to access a `List` value by index or a `Map` value by key.

Example constraints with `List` and `Map` access

```
// The following format is the same as `childList(0).getAge() == 18`:
/persons[childList[0].age == 18]

// The following format is the same as `credentialMap.get("jdoe").isValid()`:
/persons[credentialMap["jdoe"].valid]
```

<, <=, >, >=

Use these operators on properties with natural ordering. For example, for `Date` fields, the `<` operator means *before*, and for `String` fields, the operator means *alphabetically before*. These properties apply only to comparable properties.

Example constraints with `before` operator

```
/persons[ birthDate < $otherBirthDate ]

/persons[ firstName < $otherFirstName ]
```

==, !=

Use these operators as `equals()` and `!equals()` methods in constraints, instead of the usual `same` and `not same` semantics.

Example constraint with null-safe equality

```
/persons[ firstName == "John" ]

// This is similar to the following formats:

java.util.Objects.equals(person.getFirstName(), "John")
"John".equals(person.getFirstName())
```

Example constraint with null-safe not equality

```
/persons[ firstName != "John" ]

// This is similar to the following format:

!java.util.Objects.equals(person.getFirstName(), "John")
```

&&, ||

Use these operators to create an abbreviated combined relation condition that adds more than one restriction on a field. You can group constraints with parentheses `()` to create a recursive syntax pattern.

Example constraints with abbreviated combined relation

```
// Simple abbreviated combined relation condition using a single `&&`:
/persons[age > 30 && < 40]

// Complex abbreviated combined relation using groupings:
/persons[age ((> 30 && < 40) || (> 20 && < 25))]

// Mixing abbreviated combined relation with constraint connectives:
/persons[age > 30 && < 40 || location == "london"]
```
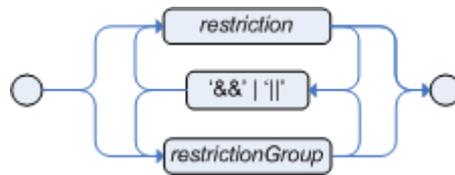
Figure 36. Abbreviated combined relation condition

Figure 37. Abbreviated combined relation condition withparentheses

matches, not matches

Use these operators to indicate that a field matches or does not match a specified Java regular expression. Typically, the regular expression is a `String` literal, but variables that resolve to a valid regular expression are also supported. These operators apply only to `String` properties. If you use `matches` against a `null` value, the resulting evaluation is always `false`. If you use `not matches` against a `null` value, the resulting evaluation is always `true`. As in Java, regular expressions that you write as `String` literals must use a double backslash `\\` to escape.

Example constraint to match or not match a regular expression

```
/persons[ country matches "(USA)?\\S*UK" ]

/persons[ country not matches "(USA)?\\S*UK" ]
```

contains, not contains

Use these operators to verify whether a field that is an `Array` or a `Collection` contains or does not contain a specified value. These operators apply to `Array` or `Collection` properties, but you can also use these operators in place of `String.contains()` and `!String.contains()` constraints checks.

Example constraints with `contains` and `not contains` for a Collection

```
// Collection with a specified field:
/familyTree[ countries contains "UK" ]

/familyTree[ countries not contains "UK" ]


// Collection with a variable:
/familyTree[ countries contains $var ]

/familyTree[ countries not contains $var ]
```

Example constraints with `contains` *and* `not contains` *for a String literal*

```
// Sting literal with a specified field:
/persons[ fullName contains "Jr" ]

/persons[ fullName not contains "Jr" ]


// String literal with a variable:
/persons[ fullName contains $var ]

/persons[ fullName not contains $var ]
```

> ℹ️ For backward compatibility, the `excludes` operator is a supported synonym for `not contains`.

`memberOf`, `not memberOf`

Use these operators to verify whether a field is a member of or is not a member of an `Array` or a `Collection` that is defined as a variable. The `Array` or `Collection` must be a variable.

Example constraints with `memberOf` *and* `not memberOf` *with a Collection*

```
/familyTree[ person memberOf $europeanDescendants ]

/familyTree[ person not memberOf $europeanDescendants ]
```

`soundslike`

Use this operator to verify whether a word has almost the same sound, using English pronunciation, as the given value (similar to the `matches` operator). This operator uses the Soundex algorithm.

Example constraint with `soundslike`

```
// Match firstName "Jon" or "John":
/persons[ firstName soundslike "John" ]
```

`str`

Use this operator to verify whether a field that is a `String` starts with or ends with a specified value. You can also use this operator to verify the length of the `String`.

Example constraints with `str`

```
// Verify what the String starts with:
/messages[ routingValue str[startsWith] "R1" ]

// Verify what the String ends with:
/messages[ routingValue str[endsWith] "R2" ]

// Verify the length of the String:
/messages[ routingValue str[length] 17 ]
```

`in`, `notin`

Use these operators to specify more than one possible value to match in a constraint (compound value restriction). This functionality of compound value restriction is supported only in the `in` and `not in` operators. The second operand of these operators must be a comma-separated list of values enclosed in parentheses. You can provide values as variables, literals, return values, or qualified identifiers. These operators are internally rewritten as a list of multiple restrictions using the operators `==` or `!=`.
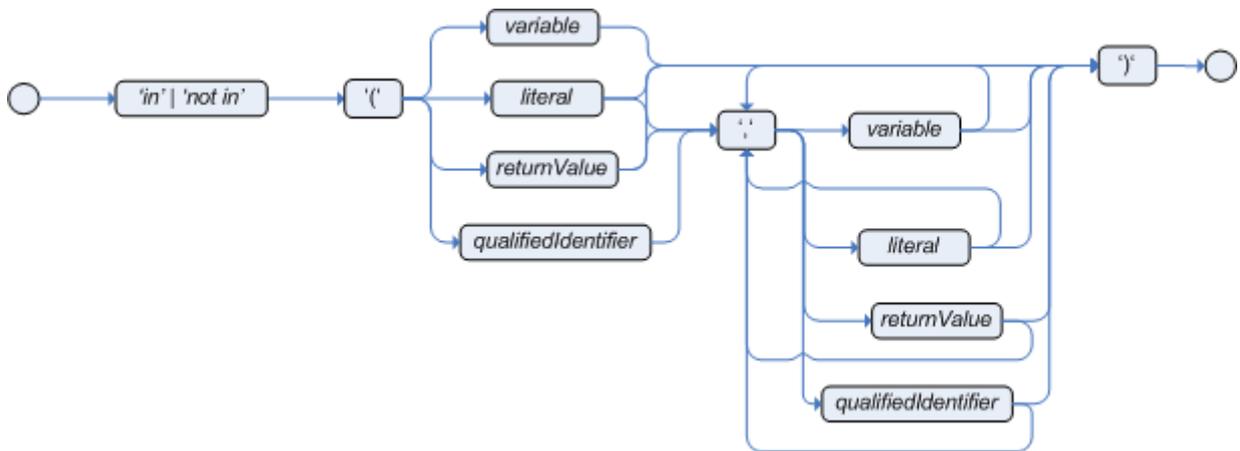
Figure 38. compoundValueRestriction

Example constraints with `in` *and* `notin`

```
/persons[ $color : favoriteColor ]
/colors[ type in ( "red", "blue", $color ) ]

/persons[ $color : favoriteColor ]
/colors[ type notin ( "red", "blue", $color ) ]
```

3.1.7.7. Operator precedence in DRL pattern constraints

DRL supports standard Java operator precedence for applicable constraint operators, with some exceptions and with some additional operators that are unique in DRL. The following table lists DRL operator precedence where applicable, from highest to lowest precedence:

Table 5. Operator precedence in DRL pattern constraints

Operator type	Operators	Notes
Nested or null-safe property access	/, !.	Not standard Java semantics
List or Map access	[]	Not standard Java semantics
Constraint binding	:	Not standard Java semantics
Multiplicative	*, /%	
Additive	+, -	
Shift	>>, >>>, <<	
Relational	<, <=, >, >=, instanceof	
Equality	== !=	Uses equals() and !equals() semantics, not standard Java same and not same semantics
Non-short-circuiting AND	&	
Non-short-circuiting exclusive OR	^	
Non-short-circuiting inclusive OR	|	
Logical AND	&&	
Logical OR	||	
Ternary	? :	
Comma-separated AND	,	Not standard Java semantics

3.1.7.8. Supported rule condition elements in DRL (keywords)

DRL supports the following rule condition elements (keywords) that you can use with the patterns that you define in DRL rule conditions:

and

Use this to group conditional components into a logical conjunction. Infix and prefix and are supported. You can group patterns explicitly with parentheses (). By default, all listed patterns are combined with and when no conjunction is specified.
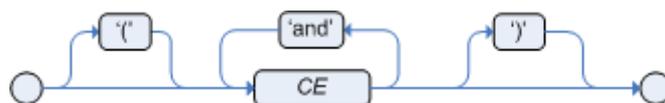
Figure 39. infixAnd

Figure 40. prefixAnd

Example patterns with and

```
//Infix `and`:
colorType: /colors/type and /persons[ favoriteColor == colorType ]

//Infix `and` with grouping:
(colorType: /colors/type and (/persons[ favoriteColor == colorType ] or /persons[
favoriteColor == colorType ])

// Prefix `and`:
(and colorType: /colors/type /persons[ favoriteColor == colorType ])

// Default implicit `and`:
colorType: /colors/type
/persons[ favoriteColor == colorType ]
```

> Do not use a leading declaration binding with the and keyword (as you can with
> or, for example). A declaration can only reference a single fact at a time, and if
> you use a declaration binding with and, then when and is satisfied, it matches
> both facts and results in an error.
>
> *Example misuse of and*
>
> ```
> // Causes compile error:
> $person : (/persons[ name == "Romeo" ] and /persons[ name ==
> "Juliet"])
> ```

or

Use this to group conditional components into a logical disjunction. Infix and prefix or are
supported. You can group patterns explicitly with parentheses (). You can also use pattern
binding with or, but each pattern must be bound separately.

Figure 41. infixOr
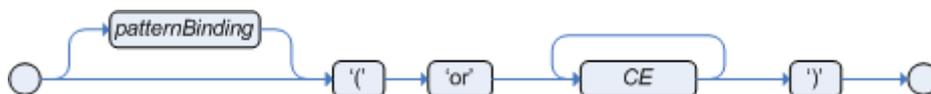
Figure 42. prefixOr

Example patterns with or

```
//Infix `or`:
colorType: /colors/type or /persons[ favoriteColor == colorType]

//Infix `or` with grouping:
colorType: /colors/type or (/persons[ favoriteColor == colorType] and /persons[
favoriteColor == colorType])

// Prefix `or`:
(or colorType: /colors/type /persons[ favoriteColor == colorType])
```

Example patterns with or and pattern binding

```
pensioner : ( /persons[ sex == "f", age > 60 ] or /persons[ sex == "m", age > 65 ]
)

(or pensioner : /persons[ sex == "f", age > 60 ]
    pensioner : /persons[ sex == "m", age > 65 ])
```

The behavior of the or condition element is different from the connective || operator for constraints and restrictions in field constraints. The Drools rule engine does not directly interpret the or element but uses logical transformations to rewrite a rule with or as a number of sub-rules. This process ultimately results in a rule that has a single or as the root node and one sub-rule for each of its condition elements. Each sub-rule is activated and executed like any normal rule, with no special behavior or interaction between the sub-rules.

Therefore, consider the or condition element a shortcut for generating two or more similar rules that, in turn, can create multiple activations when two or more terms of the disjunction are true.

exists

Use this to specify facts and constraints that must exist. This option is triggered on only the first match, not subsequent matches. If you use this element with multiple patterns, enclose the patterns with parentheses ().
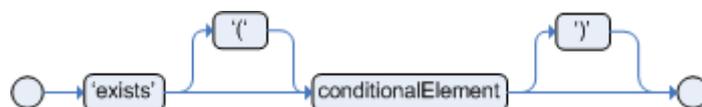
Figure 43. Exists

Example patterns with exists

```
exists /persons[ firstName == "John"]

exists (/persons[ firstName == "John", age == 42 ])

exists (/persons[ firstName == "John" ] and
        /persons[ lastName == "Doe" ])
```

not

Use this to specify facts and constraints that must not exist. If you use this element with multiple patterns, enclose the patterns with parentheses ().

Figure 44. Not

Example patterns with not

```
not /persons[ firstName == "John"]

not (/persons[ firstName == "John", age == 42 )]

not (/persons[ firstName == "John" ] and
     /persons[ lastName == "Doe" ])
```

forall

Use this to verify whether all facts that match the first pattern match all the remaining patterns. When a forall construct is satisfied, the rule evaluates to true. This element is a scope delimiter, so it can use any previously bound variable, but no variable bound inside of it is available for use outside of it.
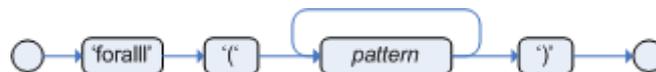
Figure 45. Forall

Example rule with forall

```
rule "All full-time employees have red ID badges"
  when
    forall( $emp : /employees[ type == "fulltime" ]
                   /employees[ this == $emp, badgeColor = "red" ] )
  then
    // True, all full-time employees have red ID badges.
end
```

In this example, the rule selects all employee objects whose type is "fulltime". For each fact that matches this pattern, the rule evaluates the patterns that follow (badge color) and if they match, the rule evaluates to true.

To state that all facts of a given type in the working memory of the Drools rule engine must match a set of constraints, you can use forall with a single pattern for simplicity.

Example rule with forall *and a single pattern*

```
rule "All full-time employees have red ID badges"
  when
    forall( /employees[ badgeColor = "red" ] )
  then
    // True, all full-time employees have red ID badges.
end
```

You can use forall constructs with multiple patterns or nest them with other condition elements, such as inside a not element construct.

Example rule with forall *and multiple patterns*

```
rule "All employees have health and dental care programs"
  when
    forall( $emp : /employees
            /healthCare[ employee == $emp ]
            /dentalCare[ employee == $emp ]
          )
  then
    // True, all employees have health and dental care.
end
```

Example rule with forall *and* not

```
rule "Not all employees have health and dental care"
  when
    not ( forall( $emp : /employees
            /healthCare[ employee == $emp ]
            /dentalCare[ employee == $emp ] )
        )
  then
    // True, not all employees have health and dental care.
end
```

> ℹ The format forall(p1 p2 p3 ···) is equivalent to not(p1 and not(and p2 p3 ···)).

accumulate

Use this to iterate over a collection of objects, execute custom actions for each of the elements, and return one or more result objects (if the constraints evaluate to true). You can use predefined functions in your accumulate conditions or implement custom functions as needed. You can also use the abbreviation acc for accumulate in rule conditions.

Use the following format to define accumulate conditions in rules:

Preferred format for `accumulate`

```
accumulate( <em>SOURCE_PATTERN</em>; <em>FUNCTIONS</em> [;<em>CONSTRAINTS</em>] )
```
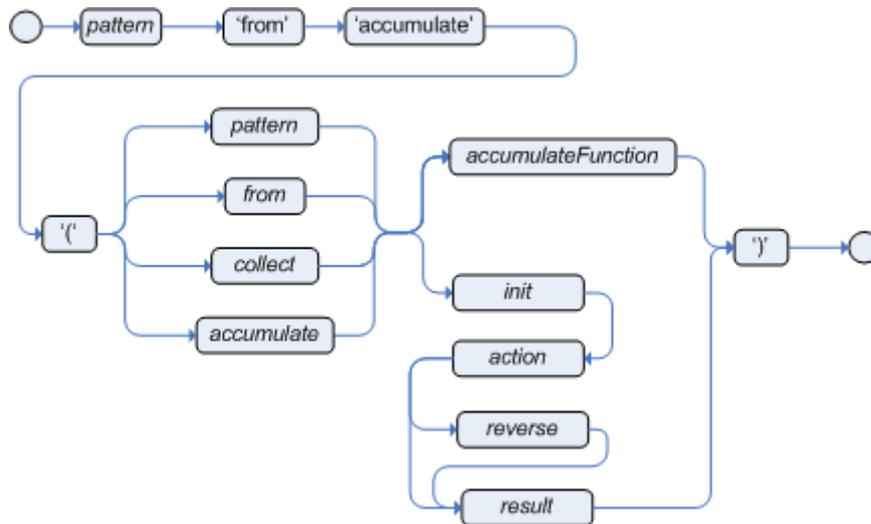

Figure 46. Accumulate

> ℹ️ Although the Drools rule engine supports alternate formats for the `accumulate` element for backward compatibility, this format is preferred for optimal performance in rules and applications.

The Drools rule engine supports the following predefined `accumulate` functions. These functions accept any expression as input.

- `average`

- `min`

- `max`

- `count`

- `sum`

- `collectList`

- `collectSet`

In the following example rule, `min`, `max`, and `average` are `accumulate` functions that calculate the minimum, maximum, and average temperature values over all the readings for each sensor:

Example rule with `accumulate` *to calculate temperature values*

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

The following example rule uses the `average` function with `accumulate` to calculate the average profit for all items in an order:

Example rule with `accumulate` *to calculate average profit*

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

To use custom, domain-specific functions in `accumulate` conditions, create a Java class that implements the `org.kie.api.runtime.rule.AccumulateFunction` interface. For example, the following Java class defines a custom implementation of an `AverageData` function:

Example Java class with custom implementation of `average` *function*

```
// An implementation of an accumulator capable of calculating average values

public class AverageAccumulateFunction implements org.kie.api.runtime.rule
.AccumulateFunction<AverageAccumulateFunction.AverageData> {

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

    }

    public void writeExternal(ObjectOutput out) throws IOException {

    }

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

```java
        public double total = 0;

        public AverageData() {}

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

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

    }

    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#createContext()
     */
    public AverageData createContext() {
        return new AverageData();
    }

    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#init(java.io.Serializable)
     */
    public void init(AverageData context) {
        context.count = 0;
        context.total = 0;
    }

    /* (non-Javadoc)
     * @see
org.kie.api.runtime.rule.AccumulateFunction#accumulate(java.io.Serializable,
java.lang.Object)
     */
    public void accumulate(AverageData context,
                           Object value) {
        context.count++;
        context.total += ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see
org.kie.api.runtime.rule.AccumulateFunction#reverse(java.io.Serializable,
java.lang.Object)
     */
    public void reverse(AverageData context, Object value) {
        context.count--;
        context.total -= ((Number) value).doubleValue();
```

```
        }

        /* (non-Javadoc)
         * @see
org.kie.api.runtime.rule.AccumulateFunction#getResult(java.io.Serializable)
         */
        public Object getResult(AverageData context) {
            return new Double( context.count == 0 ? 0 : context.total / context.count
);
        }

        /* (non-Javadoc)
         * @see org.kie.api.runtime.rule.AccumulateFunction#supportsReverse()
         */
        public boolean supportsReverse() {
            return true;
        }

        /* (non-Javadoc)
         * @see org.kie.api.runtime.rule.AccumulateFunction#getResultType()
         */
        public Class< ? > getResultType() {
            return Number.class;
        }
```

```
}
```

To use the custom function in a DRL rule, import the function using the `import accumulate` statement:

Format to import a custom function

```
import accumulate <em>CLASS_NAME</em> <em>FUNCTION_NAME</em>
```

Example rule with the imported average *function*

```
import accumulate AverageAccumulateFunction.AverageData average

rule "Average profit"
  when
    $order : /orders
    accumulate( /orderItems[ order == $order, $cost : cost, $price : price ];
                $avgProfit : average( 1 - $cost / $price ) )
  then
    // Average profit for `$order` is `$avgProfit`.
end
```

For backward compatibility, the Drools rule engine also supports the configuration of `accumulate` functions through configuration files and system properties, but this is a deprecated method. To configure the `average` function from the previous example using the configuration file or system property, set a property as shown in the following example:

```
drools.accumulate.function.average =
AverageAccumulateFunction.AverageData
```

Note that `drools.accumulate.function` is a required prefix, `average` is how the function is used in the DRL files, and `AverageAccumulateFunction.AverageData` is the fully qualified name of the class that implements the function behavior.

3.1.7.9. Railroad diagrams for rule condition elements in DRL
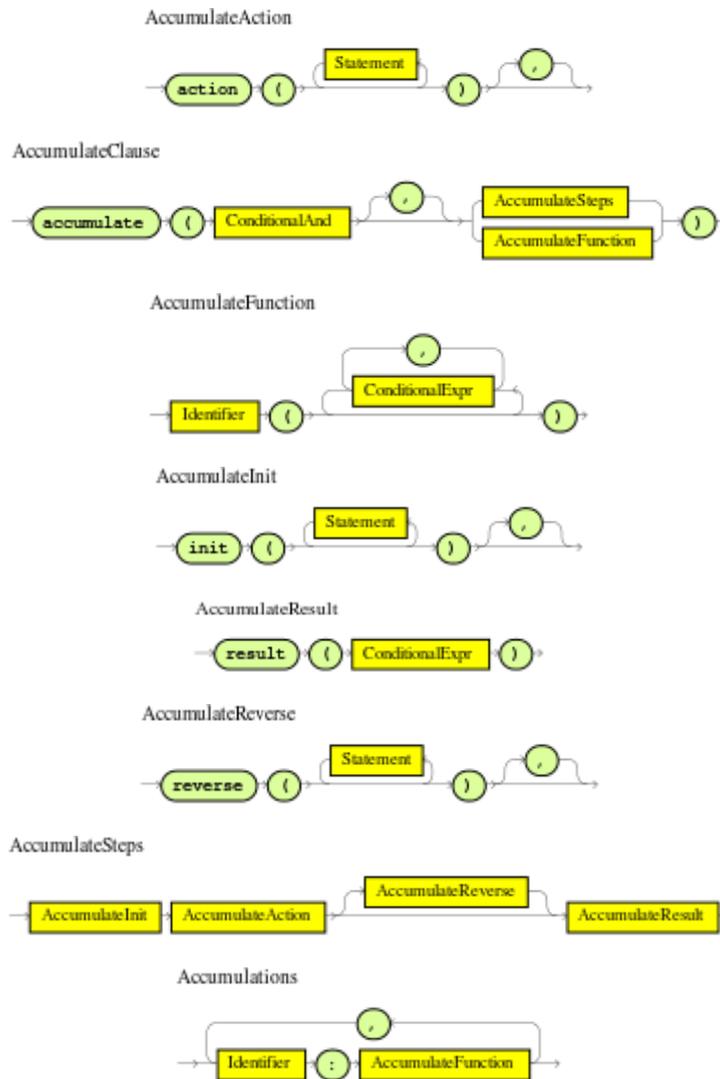
AdditiveExpr
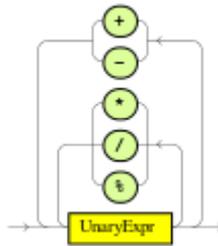
Annotation
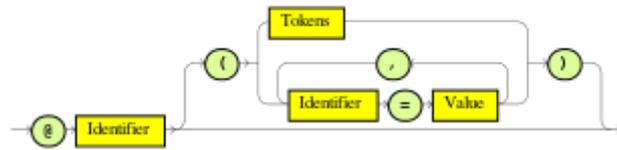
Arguments

ArrayCreatorRest

ArrayInitializer
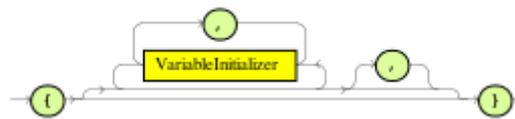
AssignmentOperator
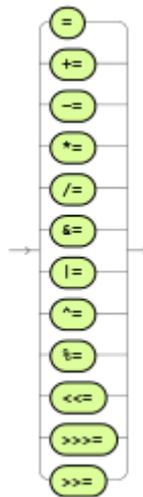
BindingPattern

Block

BooleanLitenal
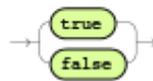
CompilationUnit
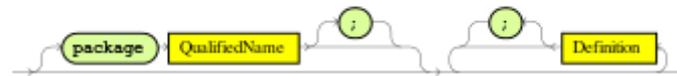
ConditionalAnd
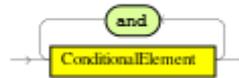
ConditionalElementAccumulate
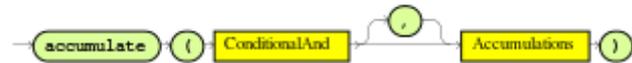
ConditionalElementEval

ConditionalElementExists
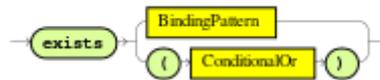
ConditionalElementForall
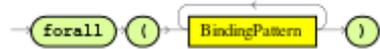
ConditionalElementNot
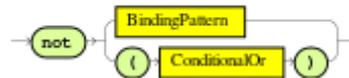
ConditionalElement
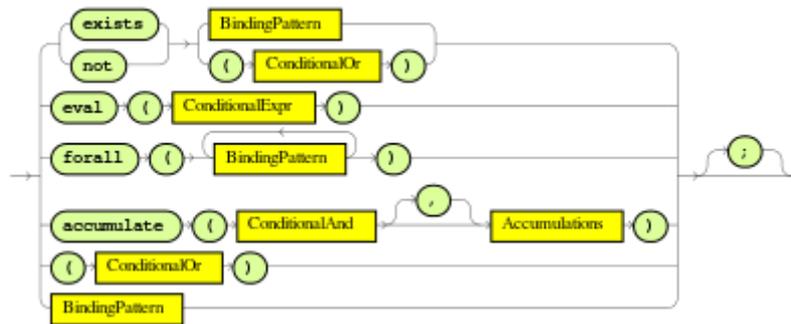
ConditionalExpr
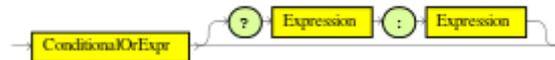
ConditionalOrExpr
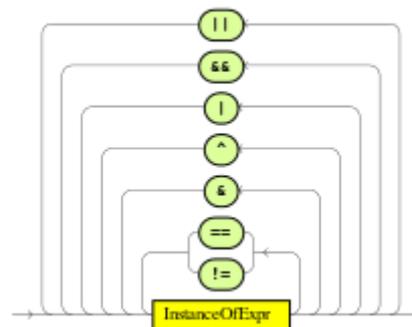
ConditionalOr

ConditionalAnd

Constraints

ConditionalOrExpr ; ConditionalOrExpr

CreatedName

TypeArguments

Identifier

PrimitiveType

Creator

NonWildcardTypeArguments CreatedName ArrayCreatorRest

Arguments

Definition

RuleAttribute

ImportDefinition

GlobalDefinition

FunctionDefinition

TypeDefinition

RuleDefinition

QueryDefinition

Digit

'0'..'9'
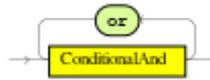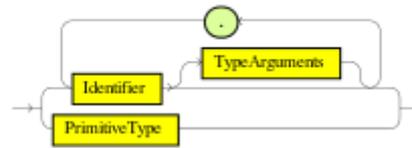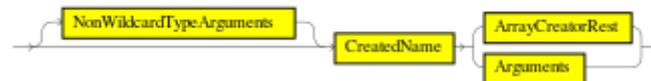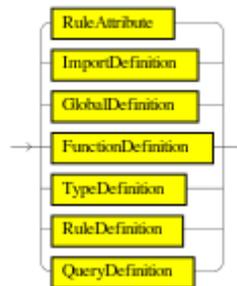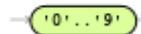
ExplicitGenericInvocationSuffix

super SuperSuffix

Identifier Arguments

ExplicitGenericInvocation

NonWildcardTypeArguments Arguments

Exponent

+

−

e

E

Digit
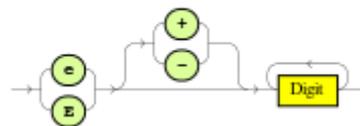
ExpressionList

Expression

Expression

AssignmentOperator Expression

ConditionalExpr

Field

Identifier : QualifiedName = ConditionalExpr Annotation ;

Fraction

Fraction

Digit . Digit

. Digit
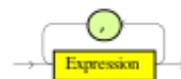
FromAccumulateClause
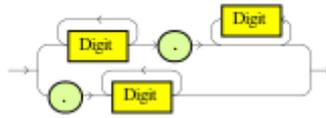
from AccumulateClause

FromClause

from ConditionalOrExpr
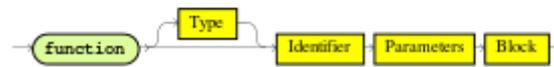
FromCollectClause

from collect (SourcePattern)

FunctionDefinition

function Type Identifier Parameters Block

GlobalDefinition

global Type Identifier

IdentifierSuffix

[ ] . class

[Expression]

Arguments

ImportDefinition

import function static QualifiedName . *

InExpr

RelationalExpr not in (, Expression)

InlineListExpr

[ExpressionList]

InlineMapExpr

[, Expression : Expression]

InnerCreator

Identifier Arguments

InstanceOfExpr

InExpr instanceof Type

IntLiteral

Literal

ModifyStatement

NonWildcardTypeArguments

OrRestriction

OverClause

Parameters
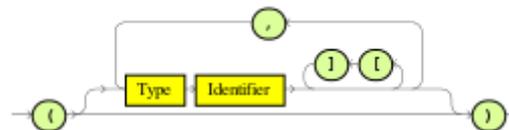
Placeholders
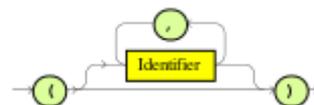
Primary
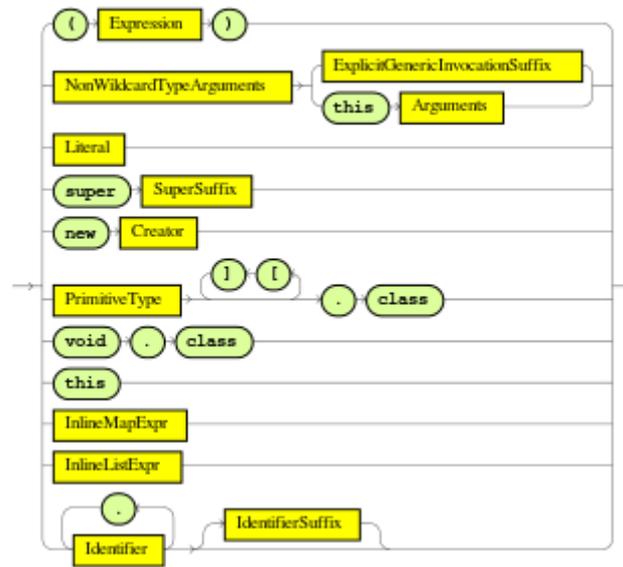
PrimitiveType
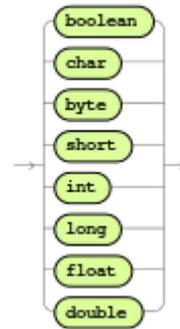
QualifiedName
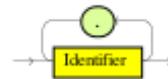
QueryDefinition
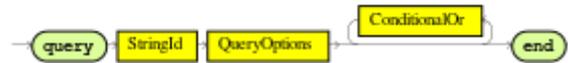
QueryOptions
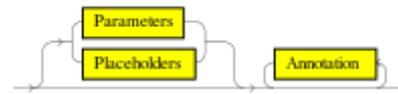
RealLiteral
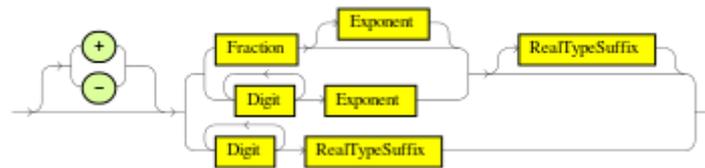
RealTypeSuffix
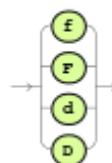
RelationalExpr
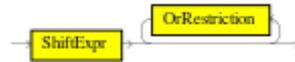
RelationalOperator

RhsStatement
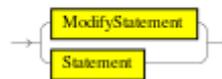
RuleAttributes
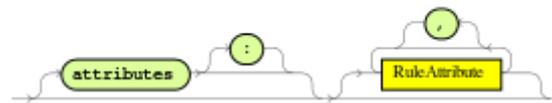
RuleAttribute
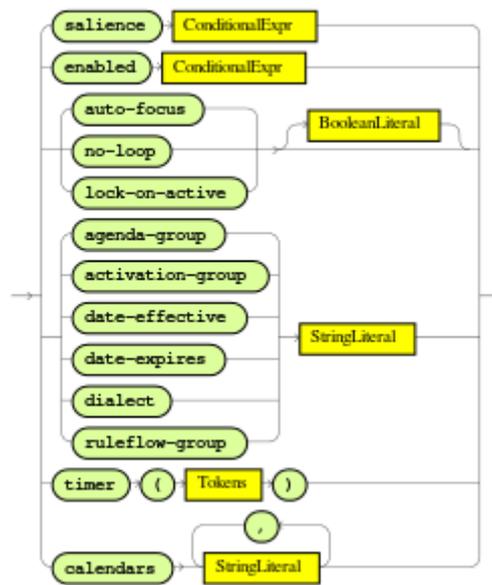
RuleDefinition

RuleOptions

Selector
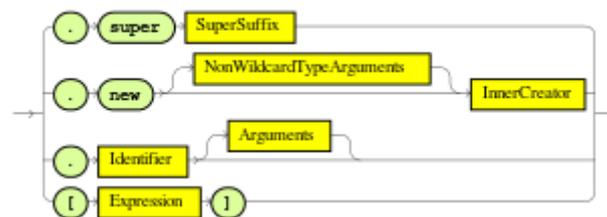
ShiftExpr

SingleRestriction
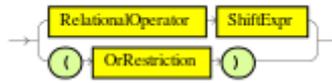
SourcePattern
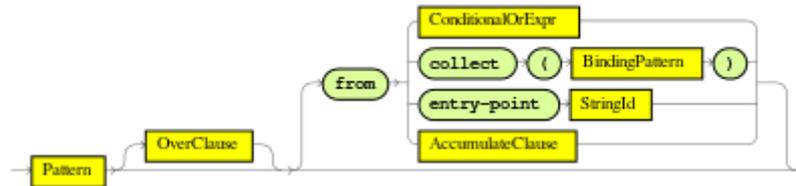
StringId
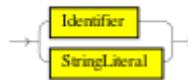
SuperSuffix
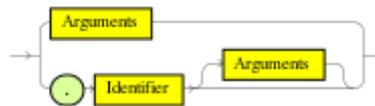
ThenPart

TypeArguments
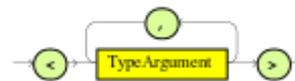
TypeArgument
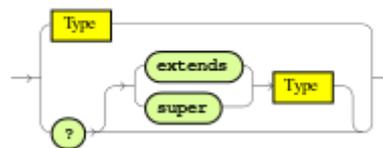
TypeDefinition

TypeOptions

Type

UnaryExprNotPlusMinus

UnaryExpr

Value

VariableInitializer
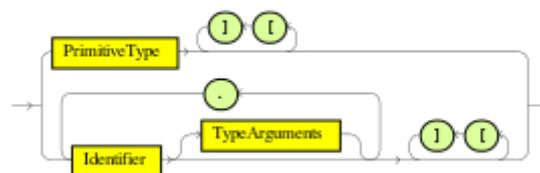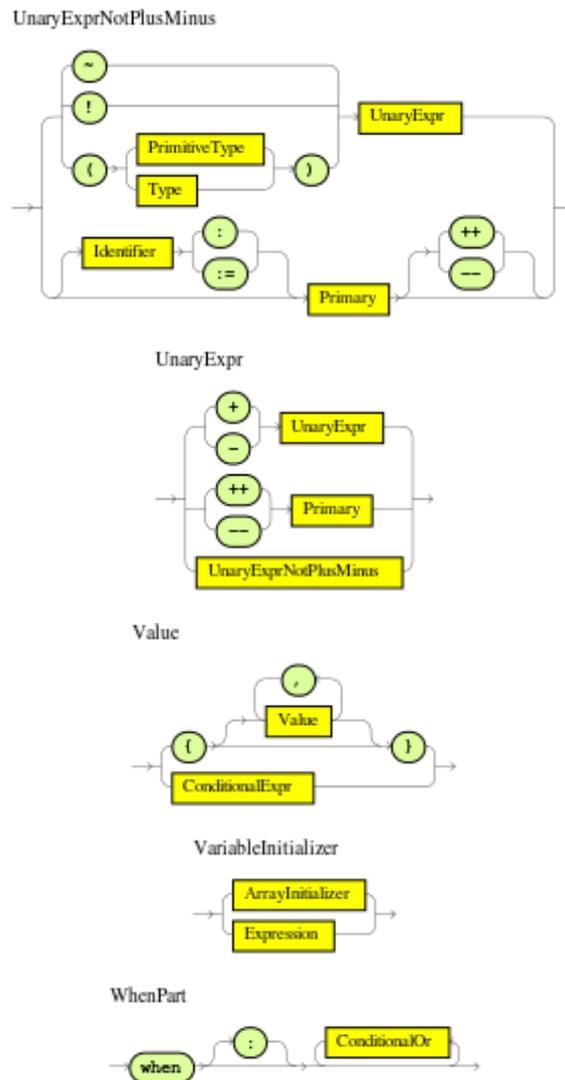
WhenPart

3.1.8. Rule actions in DRL

The `then` part of the rule (also known as the *Right Hand Side (RHS)* of the rule) contains the actions to be performed when the conditional part of the rule has been met. Rule actions are typically determined by one or more *data sources* that you define as part of your DRL rule unit. For example, if a bank requires loan applicants to have over 21 years of age (with a rule condition `/applicants[ applicantName : name, age < 21 ]`) and a loan applicant is under 21 years old, the `then` action of an `"Underage"` rule would be `setApproved( false )` based on a defined data source, declining the loan because the applicant is under age.

The main purpose of rule actions is to to insert, delete, or modify data in the working memory of the Drools rule engine. Effective rule actions are small, declarative, and readable. If you need to use imperative or conditional code in rule actions, then divide the rule into multiple smaller and more declarative rules.

Example rule for loan application age limit

```
rule "Underage"
  when
    /applicants[ applicantName : name, age < 21 ]
    $application : /loanApplications[ applicant == applicantName ]
  then
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
end
```

For more information about using data sources for rule actions, see Rule units in DRL.

3.1.9. Comments in DRL files

DRL supports single-line comments prefixed with a double forward slash `//` and multi-line comments enclosed with a forward slash and asterisk `/* ⋯ */`. You can use DRL comments to annotate rules or any related components in DRL files. DRL comments are ignored by the Drools rule engine when the DRL file is processed.

Example rule with comments

```
rule "Underage"
  // This is a single-line comment.
  when
    /applicants[ applicantName : name, age < 21 ]  // This is an in-line comment
    $application : /loanApplications[ applicant == applicantName ]
  then
    /* This is a multi-line comment
    in the rule actions. */
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
end
```


Figure 47. Multi-line comment

> ❗ The hash symbol `#` is not supported for DRL comments.

3.1.10. Error messages for DRL troubleshooting

Drools provides standardized messages for DRL errors to help you troubleshoot and resolve problems in your DRL files. The error messages use the following format:
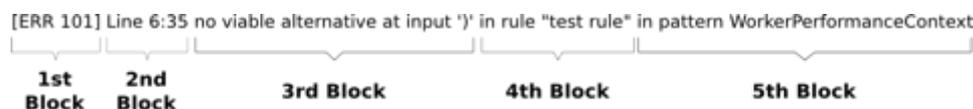
Figure 48. Error message format for DRL file problems

- **1st Block:** Error code
- **2nd Block:** Line and column in the DRL source where the error occurred
- **3rd Block:** Description of the problem
- **4th Block:** Component in the DRL source (rule, function, query) where the error occurred
- **5th Block:** Pattern in the DRL source where the error occurred (if applicable)

Drools supports the following standardized error messages:

101: no viable alternative

Indicates that the parser reached a decision point but could not identify an alternative.

Example rule with incorrect spelling

```
1: rule "simple rule"
2:   when
3:     exists /persons
4:     exits /students  // Must be `exists`
5:   then
6: end
```

Error message

```
[ERR 101] Line 4:4 no viable alternative at input 'exits' in rule "simple rule"
```

Example rule without a rule name

```
1: package org.drools.examples;
2: rule    // Must be `rule "rule name"` (or `rule rule_name` if no spacing)
3:   when
4:     Object()
5:   then
6:     System.out.println("A RHS");
7: end
```

Error message

```
[ERR 101] Line 3:2 no viable alternative at input 'when'
```

In this example, the parser encountered the keyword `when` but expected the rule name, so it flags `when` as the incorrect expected token.

Example rule with incorrect syntax

```
1: rule "simple rule"
2:   when
3:     /students[ name == "Andy ]  // Must be `"Andy"`
4:   then
5: end
```

Error message

```
[ERR 101] Line 0:-1 no viable alternative at input '<eof>' in rule "simple rule" in
pattern student
```

> A line and column value of `0:-1` means the parser reached the end of the source file (`<eof>`) but encountered incomplete constructs, usually due to missing quotation marks "…", apostrophes '…', or parentheses (…).

102: mismatched input

Indicates that the parser expected a particular symbol that is missing at the current input position.

Example rule with an incomplete rule statement

```
1: rule "simple rule"
2:   when
3:     $p : /persons[
         // Must be a complete rule statement
```

Error message

```
[ERR 102] Line 0:-1 mismatched input '<eof>' expecting ']' in rule "simple rule" in
pattern person
```

> A line and column value of `0:-1` means the parser reached the end of the source file (`<eof>`) but encountered incomplete constructs, usually due to missing quotation marks "…", apostrophes '…', or parentheses (…).

Example rule with incorrect syntax

```
1: package org.drools.examples;
2:
3: rule "Wrong syntax"
4:    when
5:      not /cars[ ( type == "tesla", price == 10000 ) || ( type == "kia", price ==
1000 ) ]
          // Must use `&&` operators instead of commas `,`
6:    then
7:      System.out.println("OK");
8: end
```

Error messages

```
[ERR 102] Line 5:36 mismatched input ',' expecting ')' in rule "Wrong syntax" in
pattern car
[ERR 101] Line 5:57 no viable alternative at input 'type' in rule "Wrong syntax"
[ERR 102] Line 5:106 mismatched input ']' expecting 'then' in rule "Wrong syntax"
```

In this example, the syntactic problem results in multiple error messages related to each other. The single solution of replacing the commas , with && operators resolves all errors. If you encounter multiple errors, resolve one at a time in case errors are consequences of previous errors.

103: failed predicate

Indicates that a validating semantic predicate evaluated to false. These semantic predicates are typically used to identify component keywords in DRL files, such as declare, rule, exists, not, and others.

Example rule with an invalid keyword

```
 1: package nesting;
 2:
 3: import org.drools.compiler.Person
 4: import org.drools.compiler.Address
 5:
 6: Some text  // Must be a valid DRL keyword
 7:
 8: rule "test something"
 9:    when
10:      $p: /persons[ name=="Michael" ]
11:    then
12:      $p.name = "other";
13:      System.out.println(p.name);
14: end
```

```
[ERR 103] Line 6:0 rule 'rule_key' failed predicate:
{(validateIdentifierKey(DroolsSoftKeywords.RULE))}? in rule
```

The `Some text` line is invalid because it does not begin with or is not a part of a DRL keyword construct, so the parser fails to validate the rest of the DRL file.

> **i** This error is similar to `102: mismatched input`, but usually involves DRL keywords.

105: did not match anything

Indicates that the parser reached a sub-rule in the grammar that must match an alternative at least once, but the sub-rule did not match anything. The parser has entered a branch with no way out.

Example rule with invalid text in an empty condition

```
1: rule "empty condition"
2:   when
3:     None  // Must remove `None` if condition is empty
4:   then
5:      insert( new Person() );
6: end
```

Error message

```
[ERR 105] Line 2:2 required (...)+ loop did not match anything at input 'WHEN' in
rule "empty condition"
```

In this example, the condition is intended to be empty but the word None is used. This error is resolved by removing None, which is not a valid DRL keyword, data type, or pattern construct.

3.1.11. Legacy DRL conventions

The following Drools Rule Language (DRL) conventions are no longer applicable or optimal in Drools but might be available for backward compatibility.

3.1.11.1. Legacy functions in DRL
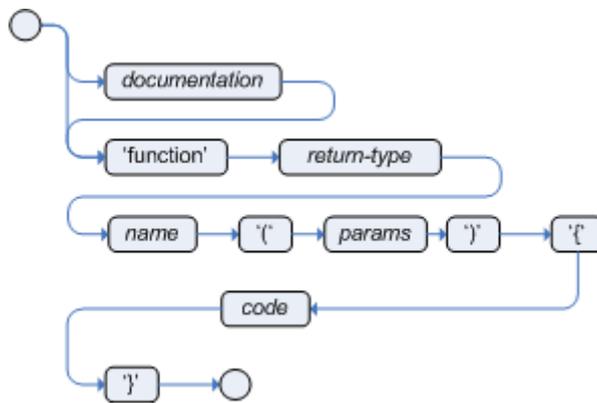
Figure 49. Function

Functions in DRL files put semantic code in your rule source file instead of in Java classes. Functions are especially useful if an action (then) part of a rule is used repeatedly and only the parameters differ for each rule. Above the rules in the DRL file, you can declare the function or import a static method from a helper class as a function, and then use the function by name in an action (then) part of the rule.

The following examples illustrate a function that is either declared or imported in a DRL file:

Example function declaration with a rule (option 1)

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

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

Example function import with a rule (option 2)

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

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

3.1.11.2. Legacy rule attributes

The following attributes were used in earlier versions of the Drools rule engine to provide grouping of rules across a rule base. These attributes are superseded by DRL rule units and are only available for backward compatibility reasons. If you need to group your rules, use DRL rule units as a clearer

and simpler grouping method.

Table 6. Legacy rule attributes

Attribute	Value
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"
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"

3.1.11.3. Legacy DRL rule condition syntax

In Drools, the preferred syntax for DRL rule conditions is through OOPath expressions. For legacy use cases, you can write rules using traditional pattern matching. In this case, you must explicitly indicate the data source using the from clause, as shown in the following comparative examples:

Example PersonRules *DRL file using OOPath notation*

```
package org.acme
unit PersonRules;

import org.acme.Person;

rule isAdult
    when
        $person: /persons[ age > 18 ]
    then
    modify($person) {
        setAdult(true)
    };
end
```

Example `PersonRules` DRL file using traditional notation

```
package org.acme
unit PersonRules;

import org.acme.Person;

rule isAdult
    when
        $person: Person(age > 18) from person
    then
    modify($person) {
        setAdult(true)
    };
end
```

3.1.11.4. Legacy DRL rule condition elements

The following rule condition elements (keywords) are obsolete in Drools:

`from`

(Obsolete with OOPath notation)

Use this to specify a data source for a pattern. This enables the Drools rule engine to reason over data that is not in the working memory. The data source can be a sub-field on a bound variable or the result of a method call. The expression used to define the object source is any expression that follows regular MVEL syntax. Therefore, the `from` element enables you to easily use object property navigation, execute method calls, and access maps and collection elements.

Figure 50. from

Example rule with `from` and pattern binding

```
rule "Validate zipcode"
  when
    Person( $personAddress : address )
    Address( zipcode == "23920W" ) from $personAddress
  then
    // Zip code is okay.
  end
```

Example rule with from *and a graph notation*

```
rule "Validate zipcode"
  when
    $p : Person()
    $a : Address( zipcode == "23920W" ) from $p.address
  then
    // Zip code is okay.
end
```

Example rule with from *to iterate over all objects*

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

For large collections of objects, instead of adding an object with a large graph that the Drools rule engine must iterate over frequently, add the collection directly to the KIE session and then join the collection in the condition, as shown in the following example:

```
when
  $order : Order()
  OrderItem( value > 100, order == $order )
```

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

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

Using `from` with `lock-on-active` rule attribute can result in rules not being executed. You can address this issue in one of the following ways:

- Avoid using the `from` element when you can insert all facts into the working memory of the Drools rule engine or use nested object references in your constraint expressions.

- Place the variable used in the `modify()` block as the last sentence in your rule condition.

- Avoid using the `lock-on-active` rule attribute when you can explicitly manage how rules within the same ruleflow group place activations on one another.

The pattern that contains a `from` clause cannot be followed by another pattern starting with a parenthesis. The reason for this restriction is that the DRL parser reads the `from` expression as `"from $l (String() or Number())"` and it cannot differentiate this expression from a function call. The simplest workaround to this is to wrap the `from` clause in parentheses, as shown in the following example:

Example rules with `from` used incorrectly and correctly

```
// Do not use `from` in this way:
rule R
  when
    $l : List()
    String() from $l
    (String() or Number())
  then
    // Actions
end

// Use `from` in this way instead:
rule R
  when
    $l : List()
    (String() from $l)
    (String() or Number())
  then
    // Actions
end
```

entry-point

(Superseded by rule unit data sources)

Use this to define an entry point, or *event stream*, corresponding to a data source for the pattern. This element is typically used with the `from` condition element. You can declare an entry point for events so that the Drools rule engine uses data from only that entry point to evaluate the rules. You can declare an entry point either implicitly by referencing it in DRL rules or explicitly in your Java application.

Example rule with `from entry-point`

```
rule "Authorize withdrawal"
  when
    WithdrawRequest( $ai : accountId, $am : amount ) from entry-point "ATM Stream"
    CheckingAccount( accountId == $ai, balance > $am )
  then
    // Authorize withdrawal.
end
```

collect

(Obsolete with OOPath notation)

Use this to define a collection of objects that the rule can use as part of the condition. The rule obtains the collection either from a specified source or from the working memory of the Drools rule engine. The result pattern of the `collect` element can be any concrete class that implements the `java.util.Collection` interface and provides a default no-arg public constructor. You can use Java collections like `List`, `LinkedList`, and `HashSet`, or your own class. If variables are bound

before the `collect` element in a condition, you can use the variables to constrain both your source and result patterns. However, any binding made inside the `collect` element is not available for use outside of it.
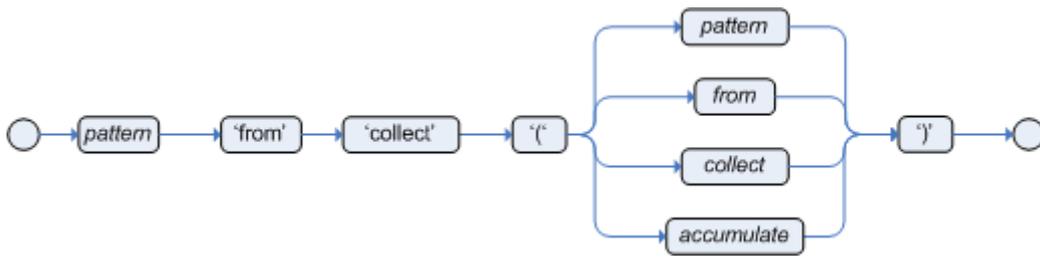
Figure 51. Collect

Example rule with `collect`

```
import java.util.List

rule "Raise priority when system has more than three pending alarms"
  when
    $system : System()
    $alarms : List( size >= 3 )
            from collect( Alarm( system == $system, status == 'pending' ) )
  then
    // Raise priority because `$system` has three or more `$alarms` pending.
end
```

In this example, the rule assesses all pending alarms in the working memory of the Drools rule engine for each given system and groups them in a `List`. If three or more alarms are found for a given system, the rule is executed.

You can also use the `collect` element with nested `from` elements, as shown in the following example:

Example rule with `collect` *and nested* `from`

```
import java.util.LinkedList;

rule "Send a message to all parents"
  when
    $town : Town( name == 'Paris' )
    $mothers : LinkedList()
            from collect( Person( children > 0 )
                        from $town.getPeople()
                        )
  then
    // Send a message to all parents.
end
```

`accumulate` alternate syntax for a single function with return type

The accumulate syntax evolved over time with the goal of becoming more compact and

expressive. Nevertheless, Drools still supports previous syntaxes for backward compatibility purposes.

In case the rule is using a single accumulate function on a given accumulate, the author may add a pattern for the result object and use the "from" keyword to link it to the accumulate result.

Example: a rule to apply a 10% discount on orders over $100 could be written in the following way:

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

In the above example, the accumulate element is using only one function (sum), and so, the rules author opted to explicitly write a pattern for the result type of the accumulate function (Number) and write the constraints inside it. There are no problems in using this syntax over the compact syntax presented before, except that is is a bit more verbose. Also note that it is not allowed to use both the return type and the functions binding in the same accumulate statement.

Compile-time checks are performed in order to ensure the pattern used with the "from" keyword is assignable from the result of the accumulate function used.

> **ⓘ** With this syntax, the "from" binds to the single result returned by the accumulate function, and it does not iterate.

In the above example, "$total" is bound to the result returned by the accumulate sum() function.

As another example however, if the result of the accumulate function is a collection, "from" still binds to the single result and it does not iterate:

```
rule "Person names"
when
  $x : Object() from accumulate(MyPerson( $val : name );
                                collectList( $val ) )
then
  // $x is a List
end
```

The bound "$x : Object()" is the List itself, returned by the collectList accumulate function used.

This is an important distinction to highlight, as the "from" keyword can also be used separately of accumulate, to iterate over the elements of a collection:

```
rule "Iterate the numbers"
when
    $xs : List()
    $x : Integer() from $xs
then
  // $x matches and binds to each Integer in the collection
end
```

While this syntax is still supported for backward compatibility purposes, for this and other reasons we encourage rule authors to make use instead of the preferred `accumulate` syntax (described previously), to avoid any potential pitfalls.

`accumulate` with inline custom code

Another possible syntax for the `accumulate` is to define inline custom code, instead of using accumulate functions.

> The use of accumulate with inline custom code is not a good practice for several reasons, including difficulties on maintaining and testing rules that use them, as well as the inability of reusing that code. Implementing your own accumulate functions is very simple and straightforward, they are easy to unit test and to use. This form of accumulate is supported for backward compatibility only.
>
> Only limited support for inline accumulate is provided while using the executable model. For example, you cannot use an external binding in the code while using the MVEL dialect:
>
> ```
> rule R
> dialect "mvel"
> when
>     String( $l : length )
>     $sum : Integer() from accumulate (
>                        Person( age > 18, $age : age ),
>                        init( int sum = 0 * $l; ),
>                        action( sum += $age; ),
>                        reverse( sum -= $age; ),
>                        result( sum )
>                 )
> ```

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

```
<em>RESULT_PATTERN</em> from accumulate( <em>SOURCE_PATTERN</em>,
                            init( <em>INIT_CODE</em> ),
                            action( <em>ACTION_CODE</em> ),
                            reverse( <em>REVERSE_CODE</em> ),
                            result( <em>RESULT_EXPRESSION</em> ) ) )
```

The meaning of each of the elements is the following:

- *SOURCE_PATTERN*: the source pattern is a regular pattern that the Drools rule engine will try to match against each of the source objects.

- *INIT_CODE*: this is a semantic block of code in the selected dialect that will be executed once for each tuple, before iterating over the source objects.

- *ACTION_CODE*: this is a semantic block of code in the selected dialect that will be executed for each of the source objects.

- *REVERSE_CODE*: this is an optional semantic block of code in the selected dialect that if present will be executed for each source object that no longer matches the source pattern. The objective of this code block is to undo any calculation done in the *ACTION_CODE* block, so that the Drools rule engine can do decremental calculation when a source object is modified or deleted, hugely improving performance of these operations.

- *RESULT_EXPRESSION*: this is a semantic expression in the selected dialect that is executed after all source objects are iterated.

- *RESULT_PATTERN*: this is a regular pattern that the Drools rule engine tries to match against the object returned from the *RESULT_EXPRESSION*. If it matches, the `accumulate` conditional element evaluates to *true* and the Drools rule engine proceeds with the evaluation of the next CE in the rule. If it does not matches, the `accumulate` CE evaluates to *false* and the Drools rule engine stops evaluating CEs for that rule.

It is easier to understand if we look at an example:

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

In the above example, for each `Order` in the Working Memory, the Drools rule engine will execute the *INIT_CODE* initializing the total variable to zero. Then it will iterate over all `OrderItem` objects for that order, executing the *action* for each one (in the example, it will sum the value of all items into the total variable). After iterating over all `OrderItem` objects, it will return the value corresponding to the *result expression* (in the above example, the value of variable `total`). Finally, the Drools rule engine will try to match the result with the `Number` pattern, and if the double value is greater than 100, the rule will fire.

The example used Java as the semantic dialect, and as such, note that the usage of the semicolon as statement delimiter is mandatory in the init, action and reverse code blocks. The result is an

expression and, as such, it does not admit ';'. If the user uses any other dialect, he must comply to that dialect's specific syntax.

As mentioned before, the *REVERSE_CODE* is optional, but it is strongly recommended that the user writes it in order to benefit from the *improved performance on update and delete*.

The `accumulate` CE can be used to execute any action on source objects. The following example instantiates and populates a custom object:

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

3.1.12. Creating DRL rules for your Drools project

You can create and manage DRL rules for your Drools project in your integrated development environment (IDE). For Drools service, VSCode is the preferred IDE. In each DRL rule file, you define rule conditions, actions, and other components related to the rule, based on the data objects you create or import in the package.

In Drools, you typically define DRL rules in rule units. A DRL rule unit is a module for rules and a unit of execution. A rule unit collects a set of rules with the declaration of the type of facts that the rules act on. A rule unit also serves as a unique namespace for each group of rules. A single rule base can contain multiple rule units. You typically store all the rules for a unit in the same file as the unit declaration so that the unit is self-contained.

For this procedure, create the following example DRL type declarations and DRL rule unit to define DRL rules in a decision service for a loan application:

Example DRL type declarations for a loan application

```
package org.mortgages;

declare Bankruptcy
    name: String
    yearOfOccurrence: int
end

declare Applicant
    name: String
    age: int
end

declare LoanApplication
    applicant: String
    approved: boolean
    explanation: String
end
```

Example DRL rule unit file for a loan application

```
package org.mortgages;
unit MortgageRules;

import org.drools.ruleunits.api.DataSource;
import org.drools.ruleunits.api.DataStream;

declare MortgageRules extends RuleUnitData
  bankruptcies: DataStore<Bankruptcy> = DataSource.createStore()
  applicants: DataStore<Applicant> = DataSource.createStore()
  loanApplications: DataStore<LoanApplication> = DataSource.createStore()
end

rule "Bankruptcy history"
    salience 10
    when
    $a : /loanApplications[ applicantName: applicant ]
    exists (/bankruptcies[ name == applicantName, yearOfOccurrence > 1990 ||
amountOwed > 100000 ])
    then
        $a.setApproved( false );
        $a.setExplanation( "has been bankrupt" );
        loanApplications.remove( $a );
end

rule "Underage"
    salience 15
    when
    /applicants[ applicantName : name, age < 21 ]
    $application : /loanApplications[ applicant == applicantName ]
    then
        $application.setApproved( false );
        $application.setExplanation( "Underage" );
        loanApplications.remove( $a );
end
```

Prerequisites

- You have created a Drools project and have included any Java data objects required for your Drools service.

Procedure

1. In your VSCode IDE, open your Drools project and create a `src/main/resources/org/mortgages` folder. This folder serves as the package for your DRL files in this example.

2. In your new `src/main/resources/org/mortgages` folder, add the following `ApplicationTypes.drl` file to define the fact types for the loan application service:

Example DRL type declarations for a loan application

```
package org.mortgages;

declare Bankruptcy
    name: String
    yearOfOccurrence: int
end

declare Applicant
    name: String
    age: int
end

declare LoanApplication
    applicant: String
    approved: boolean
    explanation: String
end
```

This DRL file defines the fact types that you can declare in any rule units in the same package for the decision service. Declarations in DRL files define new fact types or metadata for fact types to be used by rules in a DRL files. If you declare these types directly in the DRL rule unit file, you cannot declare them in any other rule units.

This example defines the following fact types:

- `Bankruptcy`: Provides data for bankruptcy status, if applicable

- `Applicant`: Provides data about the loan applicant

- `LoanApplication`: Provides data about loan approval status for a specified applicant, with an explanation if needed

3. In the same `src/main/resources/org/mortgages` folder of your Drools project, create the following `LoanApplication.drl` file to declare the DRL rule unit and data sources:

Example DRL file with rule unit and data sources

```
package org.mortgages;
unit MortgageRules;

import org.drools.ruleunits.api.DataSource;
import org.drools.ruleunits.api.DataStore;

declare MortgageRules extends RuleUnitData
  bankruptcies: DataStore<Bankruptcy> = DataSource.createStore()
  applicants: DataStore<Applicant> = DataSource.createStore()
  loanApplications: DataStore<LoanApplication> = DataSource.createStore()
end
...
```

In this example, the rule unit is named `MortgageRules` and the previously defined fact types are declared as `DataStore` data sources.

Data sources are typed sources of data that rule units can subscribe to for updates. You interact with the rule unit through the data sources it exposes. A data source can be a `DataStream` source for append-only storage, a `DataStore` source for writable storage to add or remove data, or a `SingletonStore` source for writable storage to set and clear a single element.

This example uses the `DataStore` data source to enable application data to be added or removed as part of the decision service.

4. To complete the DRL rule unit file, add the following rules for `"Bankruptcy history"` and `"Underage"` logic:

Example DRL rule unit file for a loan application

```
package org.mortgages;
unit MortgageRules;

import org.drools.ruleunits.api.DataSource;
import org.drools.ruleunits.api.DataStream;

declare MortgageRules extends RuleUnitData
   bankruptcies: DataStore<Bankruptcy> = DataSource.createStore()
   applicants: DataStore<Applicant> = DataSource.createStore()
   loanApplications: DataStore<LoanApplication> = DataSource.createStore()
end

rule "Bankruptcy history"
    salience 10
    when
    $a : /loanApplications[ applicantName: applicant ]
    exists (/bankruptcies[ name == applicantName, yearOfOccurrence > 1990 ||
amountOwed > 100000 ])
    then
        $a.setApproved( false );
        $a.setExplanation( "has been bankrupt" );
        loanApplications.remove( $a );
end

rule "Underage"
    salience 15
    when
    /applicants[ applicantName : name, age < 21 ]
    $application : /loanApplications[ applicant == applicantName ]
    then
        $application.setApproved( false );
        $application.setExplanation( "Underage" );
        loanApplications.remove( $a );
end
```

The example rules consist of the following rule components:

- `rule`: Use this segment to define each rule in the DRL file. Rules consist of a rule name in the format `rule "rule name"`, followed by optional attributes that define rule behavior, such as `salience` or `no-loop`, followed by `when` and `then` definitions. Each rule must have a unique name within the rule package.

  In this example, the `"Bankruptcy history"` rule has a defined salience of `10` and the `"Underage"` rule has a defined salience of `15`. These values ensure that the `"Bankruptcy history"` rule is executed first.

- `when` and `then`: Use the `when` portion to define the condition patterns and constraints in OOPath syntax and use the `then` portion to define the actions to be executed when the conditions are met.

  In this example, the `"Bankruptcy history"` rule states that if an applicant has owed more than 100,000 USD of unresolved debt since 1990 (beginning 1991), then the applicant is considered to have been bankrupt and is not approved for a loan. The application is removed from memory.

  If the applicant passes the bankruptcy check, then the `"Underage"` rule states that if the applicant is younger than 21 years old, then the applicant is not approved for the loan. The application is removed from memory.

  If the applicant passes both checks, then the loan is approved.

5. After you define all components of the data sources and rules, save all DRL files.

3.1.13. Performance tuning considerations with DRL

The following key concepts or suggested practices can help you optimize DRL rules and Drools rule engine performance. These concepts are summarized in this section as a convenience and are explained in more detail in the cross-referenced documentation, where applicable. This section will expand or change as needed with new releases of Drools.

Define the property and value of pattern constraints from left to right

In DRL pattern constraints, ensure that the fact property name is on the left side of the operator and that the value (constant or a variable) is on the right side. The property name must always be the key in the index and not the value. For example, write `Person( firstName == "John" )` instead of `Person( "John" == firstName )`. Defining the constraint property and value from right to left can hinder Drools rule engine performance.

For more information about DRL patterns and constraints, see Rule conditions in DRL.

Use equality operators more than other operator types in pattern constraints when possible

Although the Drools rule engine supports many DRL operator types that you can use to define your business rule logic, the equality operator `==` is evaluated most efficiently by the Drools rule engine. Whenever practical, use this operator instead of other operator types. For example, the pattern `Person( firstName == "John" )` is evaluated more efficiently than `Person( firstName !=`

`"OtherName"`). In some cases, using only equality operators might be impractical, so consider all of your business logic needs and options as you use DRL operators.

List the most restrictive rule conditions first

For rules with multiple conditions, list the conditions from most to least restrictive so that the Drools rule engine can avoid assessing the entire set of conditions if the more restrictive conditions are not met.

For example, the following conditions are part of a travel-booking rule that applies a discount to travelers who book both a flight and a hotel together. In this scenario, customers rarely book hotels with flights to receive this discount, so the hotel condition is rarely met and the rule is rarely executed. Therefore, the first condition ordering is more efficient because it prevents the Drools rule engine from evaluating the flight condition frequently and unnecessarily when the hotel condition is not met.

Preferred condition order: hotel and flight

```
when
   $h:hotel() // Rarely booked
   $f:flight()
```

Inefficient condition order: flight and hotel

```
when
   $f:flight()
   $h:hotel() // Rarely booked
```

For more information about DRL patterns and constraints, see Rule conditions in DRL.

Avoid iterating over large collections of objects with excessive `from` clauses

Avoid using the `from` condition element in DRL rules to iterate over large collections of objects, as shown in the following example:

Example conditions with `from` clause

```
when
   $c: Company()
   $e : Employee ( salary > 100000.00) from $c.employees
```

In such cases, the Drools rule engine iterates over the large graph every time the rule condition is evaluated and impedes rule evaluation.

Alternatively, instead of adding an object with a large graph that the Drools rule engine must iterate over frequently, add the collection directly to the KIE session and then join the collection in the condition, as shown in the following example:

Example conditions without `from` clause

```
when
  $c: Company();
  Employee (salary > 100000.00, company == $c)
```

In this example, the Drools rule engine iterates over the list only one time and can evaluate rules more efficiently.

For more information about the `from` element or other DRL condition elements, see Rule conditions in DRL.

Use Drools rule engine event listeners instead of `System.out.println` statements in rules for debug logging

You can use `System.out.println` statements in your rule actions for debug logging and console output, but doing this for many rules can impede rule evaluation. As a more efficient alternative, use the built-in Drools rule engine event listeners when possible. If these listeners do not meet your requirements, use a system logging utility supported by the Drools rule engine, such as Logback, Apache Commons Logging, or Apache Log4j.

For more information about supported Drools rule engine event listeners and logging utilities, see Drools rule engine event listeners and debug logging.

Use the `drools-metric` module to identify the obstruction in your rules

You can use the `drools-metric` module to identify slow rules especially when you process many rules. The `drools-metric` module can also assist in analyzing the Drools rule engine performance. Note that the `drools-metric` module is not for production environment use. However, you can perform the analysis in your test environment.

To analyze the Drools rule engine performance using `drools-metric`, first add `drools-metric` to your project dependencies:

Example project dependency for `drools-metric`

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-metric</artifactId>
</dependency>
```

If you want to use `drools-metric` to enable trace logging, configure a logger for `org.drools.metric.util.MetricLogUtils` as shown in the following example:

Example logback.xml configuration file

```
<configuration>
  <logger name="org.drools.metric.util.MetricLogUtils" level="trace"/>
  ...
<configuration>
```

Alternatively, you can use `drools-metric` to expose the data using Micrometer. To expose the data, enable the Micrometer registry of your choice as shown in the following example:

Example project dependency for Micrometer

```xml
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-jmx</artifactId> <!-- Discover more registries at micrometer.io. -->
</dependency>
```

Example Java code for Micrometer

```java
Metrics.addRegitry(new JmxMeterRegistry(s -> null, Clock.SYSTEM));
```

Regardless of whether you want to use logging or Micrometer, you need to enable `MetricLogUtils` by setting the system property `drools.metric.logger.enabled` to `true`. Optionally, you can change the microseconds threshold of metric reporting by setting the `drools.metric.logger.threshold` system property.

> ℹ️ Only node executions exceeding the threshold are reported. The default value is `500`.

After configuring the `drools-metric` to use logging, rule execution produces logs as shown in the following example:

Example rule execution output

```
TRACE [JoinNode(6) - [ClassObjectType class=com.sample.Order]], evalCount:1000,
elapsedMicro:5962
TRACE [JoinNode(7) - [ClassObjectType class=com.sample.Order]], evalCount:100000,
elapsedMicro:95553
TRACE [ AccumulateNode(8) ], evalCount:4999500, elapsedMicro:2172836
TRACE [EvalConditionNode(9)]:
cond=com.sample.Rule_Collect_expensive_orders_combination930932360Eval1Invoker@ee2a
6922], evalCount:49500, elapsedMicro:18787
```

This example includes the following key parameters:

- `evalCount` is the number of constraint evaluations against inserted facts during the node execution. When `evalCount` is used with Micrometer, a counter with the data is called `org.drools.metric.evaluation.count`.

- `elapsedMicro` is the elapsed time of the node execution in microseconds. When `elapsedMicro` is used with Micrometer, look for a timer called `org.drools.metric.elapsed.time`.

If you find an outstanding `evalCount` or `elapsedMicro` log, correlate the node name with `ReteDumper.dumpAssociatedRulesRete()` output to identify the rule associated with the node.

Example ReteDumper usage

```
ReteDumper.dumpAssociatedRulesRete(kbase);
```

Example ReteDumper output

```
[ AccumulateNode(8) ] : [Collect expensive orders combination]
...
```

3.2. Domain Specific Languages

Domain Specific Languages (or DSLs) are a way of creating a rule language that is dedicated to your problem domain. A set of DSL definitions consists of transformations from DSL "sentences" to DRL constructs, which lets you use of all the underlying rule language and engine features. Given a DSL, you write rules in DSL rule (or DSLR) files, which will be translated into DRL files.

DSL and DSLR files are plain text files, and you can use any text editor to create and modify them. But there are also DSL and DSLR editors, both in the IDE as well as in the web based BRMS, and you can use those as well, although they may not provide you with the full DSL functionality.

3.2.1. When to Use a DSL

DSLs can serve as a layer of separation between rule authoring (and rule authors) and the technical intricacies resulting from the modelling of domain object and the Drools rule engine's native language and methods. If your rules need to be read and validated by domain experts (such as business analysts, for instance) who are not programmers, you should consider using a DSL; it hides implementation details and focuses on the rule logic proper. DSL sentences can also act as "templates" for conditional elements and consequence actions that are used repeatedly in your rules, possibly with minor variations. You may define DSL sentences as being mapped to these repeated phrases, with parameters providing a means for accommodating those variations.

DSLs have no impact on the Drools rule engine at runtime, they are just a compile time feature, requiring a special parser and transformer.

3.2.2. DSL Basics

The Drools DSL mechanism allows you to customise conditional expressions and consequence actions. A global substitution mechanism ("keyword") is also available.

Example 1. Example DSL mapping

```
[when]Something is {colour}=Something(colour=="{colour}")
```

In the preceding example, `[when]` indicates the scope of the expression, i.e., whether it is valid for the LHS or the RHS of a rule. The part after the bracketed keyword is the expression that you use in the rule; typically a natural language expression, but it doesn't have to be. The part to the right of

the equal sign ("=") is the mapping of the expression into the rule language. The form of this string depends on its destination, RHS or LHS. If it is for the LHS, then it ought to be a term according to the regular LHS syntax; if it is for the RHS then it might be a Java statement.

Whenever the DSL parser matches a line from the rule file written in the DSL with an expression in the DSL definition, it performs three steps of string manipulation. First, it extracts the string values appearing where the expression contains variable names in braces (here: `{colour}`). Then, the values obtained from these captures are then interpolated wherever that name, again enclosed in braces, occurs on the right hand side of the mapping. Finally, the interpolated string replaces whatever was matched by the entire expression in the line of the DSL rule file.

Note that the expressions (i.e., the strings on the left hand side of the equal sign) are used as regular expressions in a pattern matching operation against a line of the DSL rule file, matching all or part of a line. This means you can use (for instance) a '?' to indicate that the preceding character is optional. One good reason to use this is to overcome variations in natural language phrases of your DSL. But, given that these expressions are regular expression patterns, this also means that all "magic" characters of Java's pattern syntax have to be escaped with a preceding backslash ('\').

It is important to note that the compiler transforms DSL rule files line by line. In the previous example, all the text after "Something is " to the end of the line is captured as the replacement value for "{colour}", and this is used for interpolating the target string. This may not be exactly what you want. For instance, when you intend to merge different DSL expressions to generate a composite DRL pattern, you need to transform a DSLR line in several independent operations. The best way to achieve this is to ensure that the captures are surrounded by characteristic text - words or even single characters. As a result, the matching operation done by the parser plucks out a substring from somewhere within the line. In the example below, quotes are used as distinctive characters. Note that the characters that surround the capture are not included during interpolation, just the contents between them.

As a rule of thumb, use quotes for textual data that a rule editor may want to enter. You can also enclose the capture with words to ensure that the text is correctly matched. Both is illustrated by the following example. Note that a single line such as `Something is "green" and another solid thing` is now correctly expanded.

Example 2. Example with quotes

```
[when]something is "{colour}"=Something(colour=="{colour}")
[when]another {state} thing=OtherThing(state=="{state})"
```

It is a good idea to avoid punctuation (other than quotes or apostrophes) in your DSL expressions as much as possible. The main reason is that punctuation is easy to forget for rule authors using your DSL. Another reason is that parentheses, the period and the question mark are magic characters, requiring escaping in the DSL definition.

In a DSL mapping, the braces "{" and "}" should only be used to enclose a variable definition or reference, resulting in a capture. If they should occur literally, either in the expression or within the replacement text on the right hand side, they must be escaped with a preceding backslash ("\"):

```
[then]do something= if (foo) \{ doSomething(); \}
```

ℹ️ If braces "{" and "}" should appear in the replacement string of a DSL definition, escape them with a backslash ('\').

Example 3. Examples of DSL mapping entries

```
# This is a comment to be ignored.
[when]There is a person with name of "{name}"=Person(name=="{name}")
[when]Person is at least {age} years old and lives in "{location}"=
      Person(age >= {age}, location=="{location}")
[then]Log "{message}"=System.out.println("{message}");
[when]And = and
```

Given the above DSL examples, the following examples show the expansion of various DSLR snippets:

Example 4. Examples of DSL expansions

```
There is a person with name of "Kitty"
    ==> Person(name="Kitty")
Person is at least 42 years old and lives in "Atlanta"
    ==> Person(age >= 42, location="Atlanta")
Log "boo"
    ==> System.out.println("boo");
There is a person with name of "Bob" And Person is at least 30 years old and lives
in "Utah"
    ==> Person(name="Bob") and Person(age >= 30, location="Utah")
```

ℹ️ Don't forget that if you are capturing plain text from a DSL rule line and want to use it as a string literal in the expansion, you must provide the quotes on the right hand side of the mapping.

You can chain DSL expressions together on one line, as long as it is clear to the parser where one ends and the next one begins and where the text representing a parameter ends. (Otherwise you risk getting all the text until the end of the line as a parameter value.) The DSL expressions are tried, one after the other, according to their order in the DSL definition file. After any match, all remaining DSL expressions are investigated, too.

The resulting DRL text may consist of more than one line. Line ends are in the replacement text are written as \n.

3.2.3. Adding Constraints to Facts

A common requirement when writing rule conditions is to be able to add an arbitrary combination of constraints to a pattern. Given that a fact type may have many fields, having to provide an individual DSL statement for each combination would be plain folly.

The DSL facility allows you to add constraints to a pattern by a simple convention: if your DSL expression starts with a hyphen (minus character, "-") it is assumed to be a field constraint and, consequently, is is added to the last pattern line preceding it.

For an example, lets take look at class Cheese, with the following fields: type, price, age and country. We can express some LHS condition in normal DRL like the following

```
Cheese(age < 5, price == 20, type=="stilton", country=="ch")
```

The DSL definitions given below result in three DSL phrases which may be used to create any combination of constraint involving these fields.

```
[when]There is a Cheese with=Cheese()
[when]- age is less than {age}=age<{age}
[when]- type is '{type}'=type=='{type}'
[when]- country equal to '{country}'=country=='{country}'
```

You can then write rules with conditions like the following:

```
There is a Cheese with
        - age is less than 42
        - type is 'stilton'
```

```
 The parser will pick up a line beginning with "-" and add it as a constraint to  the
preceding pattern, inserting a comma when it is required.
For the preceding example, the resulting DRL is:
```

```
Cheese(age<42, type=='stilton')
```

Combining all numeric fields with all relational operators (according to the DSL expression "age is less than..." in the preceding example) produces an unwieldy amount of DSL entries. But you can define DSL phrases for the various operators and even a generic expression that handles any field constraint, as shown below. (Notice that the expression definition contains a regular expression in addition to the variable name.)

```
[when][]is less than or equal to=<=
[when][]is less than=<
[when][]is greater than or equal to=>=
[when][]is greater than=>
[when][]is equal to===
[when][]equals===
[when][]There is a Cheese with=Cheese()
[when][]- {field:\w*} {operator} {value:\d*}={field} {operator} {value}
```

Given these DSL definitions, you can write rules with conditions such as:

```
There is a Cheese with
    - age is less than 42
    - rating is greater than 50
    - type equals 'stilton'
```

In this specific case, a phrase such as "is less than" is replaced by <, and then the line matches the last DSL entry. This removes the hyphen, but the final result is still added as a constraint to the preceding pattern. After processing all of the lines, the resulting DRL text is:

```
Cheese(age<42, rating > 50, type=='stilton')
```

> The order of the entries in the DSL is important if separate DSL expressions are intended to match the same line, one after the other.

3.2.4. Developing a DSL

A good way to get started is to write representative samples of the rules your application requires, and to test them as you develop. This will provide you with a stable framework of conditional elements and their constraints. Rules, both in DRL and in DSLR, refer to entities according to the data model representing the application data that should be subject to the reasoning process defined in rules. Notice that writing rules is generally easier if most of the data model's types are facts.

Given an initial set of rules, it should be possible to identify recurring or similar code snippets and to mark variable parts as parameters. This provides reliable leads as to what might be a handy DSL entry. Also, make sure you have a full grasp of the jargon the domain experts are using, and base your DSL phrases on this vocabulary.

You may postpone implementation decisions concerning conditions and actions during this first design phase by leaving certain conditional elements and actions in their DRL form by prefixing a line with a greater sign (">"). (This is also handy for inserting debugging statements.)

During the next development phase, you should find that the DSL configuration stabilizes pretty quickly. New rules can be written by reusing the existing DSL definitions, or by adding a parameter to an existing condition or consequence entry.

Try to keep the number of DSL entries small. Using parameters lets you apply the same DSL sentence for similar rule patterns or constraints. But do not exaggerate: authors using the DSL should still be able to identify DSL phrases by some fixed text.

3.2.5. DSL and DSLR Reference

A DSL file is a text file in a line-oriented format. Its entries are used for transforming a DSLR file into a file according to DRL syntax.

- A line starting with "#" or "//" (with or without preceding white space) is treated as a comment. A comment line starting with "#/" is scanned for words requesting a debug option, see below.

- Any line starting with an opening bracket ("[") is assumed to be the first line of a DSL entry definition.

- Any other line is appended to the preceding DSL entry definition, with the line end replaced by a space.

A DSL entry consists of the following four parts:

- A scope definition, written as one of the keywords "when" or "condition", "then" or "consequence", "*" and "keyword", enclosed in brackets ("[" and "]"). This indicates whether the DSL entry is valid for the condition or the consequence of a rule, or both. A scope indication of "keyword" means that the entry has global significance, i.e., it is recognized anywhere in a DSLR file.

- A type definition, written as a Java class name, enclosed in brackets. This part is optional unless the next part begins with an opening bracket. An empty pair of brackets is valid, too.

- A DSL expression consists of a (Java) regular expression, with any number of embedded *variable definitions,* terminated by an equal sign ("="). A variable definition is enclosed in braces ("{" and "}"). It consists of a variable name and two optional attachments, separated by colons (":"). If there is one attachment, it is a regular expression for matching text that is to be assigned to the variable; if there are two attachments, the first one is a hint for the GUI editor and the second one the regular expression.

  Note that all characters that are "magic" in regular expressions must be escaped with a preceding backslash ("\") if they should occur literally within the expression.

- The remaining part of the line after the delimiting equal sign is the replacement text for any DSLR text matching the regular expression. It may contain variable references, i.e., a variable name enclosed in braces. Optionally, the variable name may be followed by an exclamation mark ("!") and a transformation function, see below.

  Note that braces ("{" and "}") must be escaped with a preceding backslash ("\") if they should occur literally within the replacement string.

Debugging of DSL expansion can be turned on, selectively, by using a comment line starting with "#/" which may contain one or more words from the table presented below. The resulting output is written to standard output.

Table 7. Debug options for DSL expansion

Word	Description
result	Prints the resulting DRL text, with line numbers.
steps	Prints each expansion step of condition and consequence lines.
keyword	Dumps the internal representation of all DSL entries with scope "keyword".
when	Dumps the internal representation of all DSL entries with scope "when" or "*".
then	Dumps the internal representation of all DSL entries with scope "then" or "*".
usage	Displays a usage statistic of all DSL entries.

Below are some sample DSL definitions, with comments describing the language features they illustrate.

```
# Comment: DSL examples

#/ debug: display result and usage

# keyword definition: replaces "regula" by "rule"
[keyword][]regula=rule

# conditional element: "T" or "t", "a" or "an", convert matched word
[when][][Tt]here is an? {entity:\w+}=
        ${entity!lc}: {entity!ucfirst} ()

# consequence statement: convert matched word, literal braces
[then][]update {entity:\w+}=modify( ${entity!lc} )\{ \}
```

The transformation of a DSLR file proceeds as follows:

1. The text is read into memory.

2. Each of the "keyword" entries is applied to the entire text. First, the regular expression from the keyword definition is modified by replacing white space sequences with a pattern matching any number of white space characters, and by replacing variable definitions with a capture made from the regular expression provided with the definition, or with the default (".*?"). Then, the DSLR text is searched exhaustively for occurrences of strings matching the modified regular expression. Substrings of a matching string corresponding to variable captures are extracted and replace variable references in the corresponding replacement text, and this text replaces the matching string in the DSLR text.

3. Sections of the DSLR text between "when" and "then", and "then" and "end", respectively, are located and processed in a uniform manner, line by line, as described below.

   For a line, each DSL entry pertaining to the line's section is taken in turn, in the order it appears in the DSL file. Its regular expression part is modified: white space is replaced by a pattern

matching any number of white space characters; variable definitions with a regular expression are replaced by a capture with this regular expression, its default being ".*?". If the resulting regular expression matches all or part of the line, the matched part is replaced by the suitably modified replacement text.

Modification of the replacement text is done by replacing variable references with the text corresponding to the regular expression capture. This text may be modified according to the string transformation function given in the variable reference; see below for details.

If there is a variable reference naming a variable that is not defined in the same entry, the expander substitutes a value bound to a variable of that name, provided it was defined in one of the preceding lines of the current rule.

4. If a DSLR line in a condition is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a pattern CE, i.e., a type name followed by a pair of parentheses. if this pair is empty, the expanded line (which should contain a valid constraint) is simply inserted, otherwise a comma (",") is inserted beforehand.

If a DSLR line in a consequence is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a "modify" statement, ending in a pair of braces ("{" and "}"). If this pair is empty, the expanded line (which should contain a valid method call) is simply inserted, otherwise a comma (",") is inserted beforehand.

> It is currently *not* possible to use a line with a leading hyphen to insert text into other conditional element forms (e.g., "accumulate") or it may only work for the first insertion (e.g., "eval").

All string transformation functions are described in the following table.

Table 8. String transformation functions

Name	Description
uc	Converts all letters to upper case.
lc	Converts all letters to lower case.
ucfirst	Converts the first letter to upper case, and all other letters to lower case.
num	Extracts all digits and "-" from the string. If the last two digits in the original string are preceded by "." or ",", a decimal period is inserted in the corresponding position.
a?*b*/*c*	Compares the string with string *a*, and if they are equal, replaces it with *b*, otherwise with *c*. But *c* can be another triplet *a*, *b*, *c*, so that the entire structure is, in fact, a translation table.

The following DSL examples show how to use string transformation functions.

```
# definitions for conditions
[when][]There is an? {entity}=${entity!lc}: {entity!ucfirst}()
[when][]- with an? {attr} greater than {amount}={attr} <= {amount!num}
[when][]- with a {what} {attr}={attr} {what!positive?>0/negative?%lt;0/zero?==0/ERROR}
```

A file containing a DSL definition has to be put under the resources folder or any of its subfolders like any other drools artifact. It must have the extension `.dsl`, or alternatively be marked with type `ResourceType.DSL`. when programmatically added to a `KieFileSystem`. For a file using DSL definition, the extension `.dslr` should be used, while it can be added to a `KieFileSystem` with type `ResourceType.DSLR`.

For parsing and expanding a DSLR file the DSL configuration is read and supplied to the parser. Thus, the parser can "recognize" the DSL expressions and transform them into native rule language expressions.

Chapter 4. Decision Model and Notation (DMN)

Decision Model and Notation (DMN) is a standard established by the Object Management Group (OMG) for describing and modeling operational decisions. DMN defines an XML schema that enables DMN models to be shared between DMN-compliant platforms and across organizations so that business analysts and business rules developers can collaborate in designing and implementing DMN decision services. The DMN standard is similar to and can be used together with the Business Process Model and Notation (BPMN) standard for designing and modeling business processes.

For general information about the background and applications of DMN, see the Drools DMN landing page.

4.1. DMN conformance levels

The DMN specification defines three incremental levels of conformance in a software implementation. A product that claims compliance at one level must also be compliant with any preceding levels. For example, a conformance level 3 implementation must also include the supported components in conformance levels 1 and 2. For the formal definitions of each conformance level, see the OMG Decision Model and Notation specification.

The following list summarizes the three DMN conformance levels:

Conformance level 1

A DMN conformance level 1 implementation supports decision requirement diagrams (DRDs), decision logic, and decision tables, but decision models are not executable. Any language can be used to define the expressions, including natural, unstructured languages.

Conformance level 2

A DMN conformance level 2 implementation includes the requirements in conformance level 1, and supports Simplified Friendly Enough Expression Language (S-FEEL) expressions and fully executable decision models.

Conformance level 3

A DMN conformance level 3 implementation includes the requirements in conformance levels 1 and 2, and supports Friendly Enough Expression Language (FEEL) expressions, the full set of boxed expressions, and fully executable decision models.

Drools DMN engine provides runtime support for DMN 1.1, 1.2, 1.3, and 1.4 models at conformance level 3.

KIE DMN Editor provides design support for DMN 1.2 models at conformance level 3.

You can design your DMN models directly with KIE DMN Editor online, directly with KIE DMN Editor in VSCode, or import existing DMN models into your Drools projects for deployment and execution.

Any DMN 1.1 and 1.3 models (do not contain DMN 1.3 features) that you import or open into KIE DMN Editor and save are converted to DMN 1.2 models.

4.2. DMN decision requirements diagram (DRD) components

A decision requirements diagram (DRD) is a visual representation of your DMN model. A DRD can represent part or all of the overall decision requirements graph (DRG) for the DMN model. DRDs trace business decisions using decision nodes, business knowledge models, sources of business knowledge, input data, and decision services.

The following table summarizes the components in a DRD:

Table 9. DRD components

Component		Description	Notation
Elements	Decision	Node where one or more input elements determine an output based on defined decision logic.	Decision
	Business knowledge model	Reusable function with one or more decision elements. Decisions that have the same logic but depend on different sub-input data or sub-decisions use business knowledge models to determine which procedure to follow.	Business knowledge model
	Knowledge source	External authorities, documents, committees, or policies that regulate a decision or business knowledge model. Knowledge sources are references to real-world factors rather than executable business rules.	Knowledge source
	Input data	Information used in a decision node or a business knowledge model. Input data usually includes business-level concepts or objects relevant to the business, such as loan applicant data used in a lending strategy.	Input data
	Decision service	Top-level decision containing a set of reusable decisions published as a service for invocation. A decision service can be invoked from an external application or a BPMN business process.	Decision service
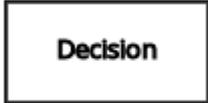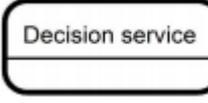
Component		Description	Notation
Requirement connectors	Information requirement	Connection from an input data node or decision node to another decision node that requires the information.	
	Knowledge requirement	Connection from a business knowledge model to a decision node or to another business knowledge model that invokes the decision logic.	
	Authority requirement	Connection from an input data node or a decision node to a dependent knowledge source or from a knowledge source to a decision node, business knowledge model, or another knowledge source.	
Artifacts	Text annotation	Explanatory note associated with an input data node, decision node, business knowledge model, or knowledge source.	
	Association	Connection from an input data node, decision node, business knowledge model, or knowledge source to a text annotation.	
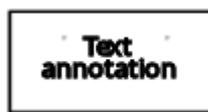
The following table summarizes the permitted connectors between DRD elements:

Table 10. DRD connector rules

Starts from	Connects to	Connection type	Example
Decision	Decision	Information requirement	
Business knowledge model	Decision	Knowledge requirement	
	Business knowledge model		
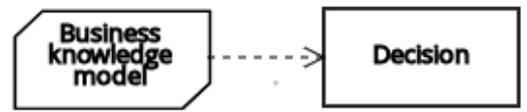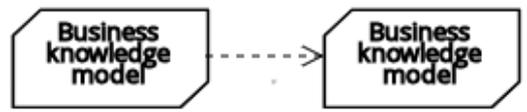
Starts from	Connects to	Connection type	Example
Decision service	Decision	Knowledge requirement	
	Business knowledge model		
Input data	Decision	Information requirement	
	Knowledge source	Authority requirement	
Knowledge source	Decision	Authority requirement	
	Business knowledge model		
	Knowledge source		
Decision	Text annotation	Association	
Business knowledge model			
Knowledge source			
Input data			
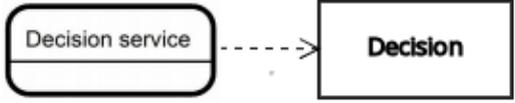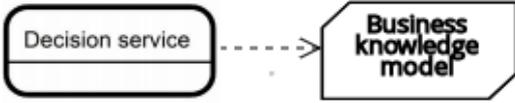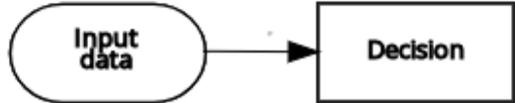
The following example DRD illustrates some of these DMN components in practice:
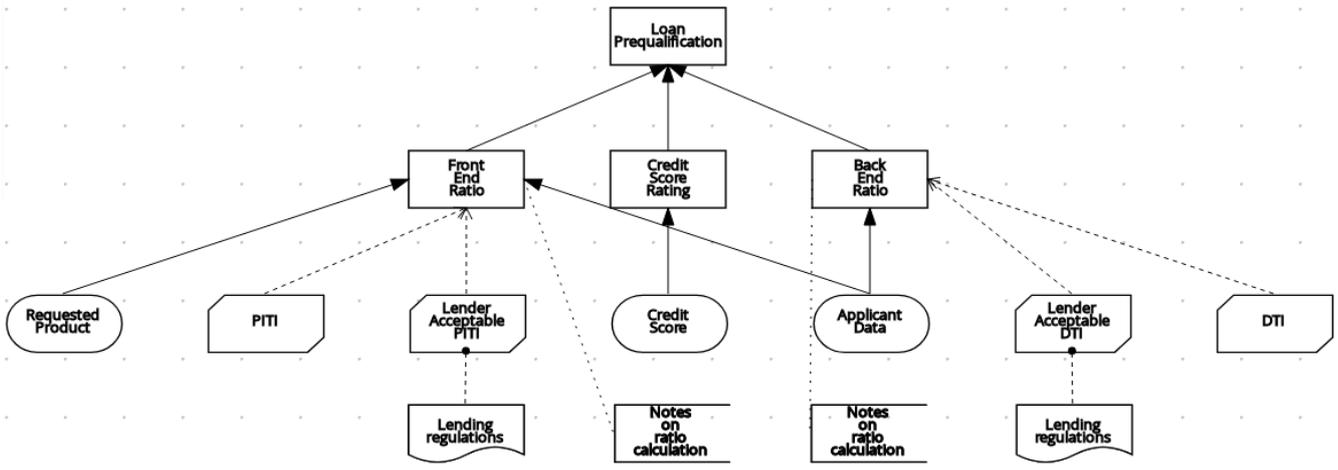
Figure 52. Example DRD: Loan prequalification

The following example DRD illustrates DMN components that are part of a reusable decision service:
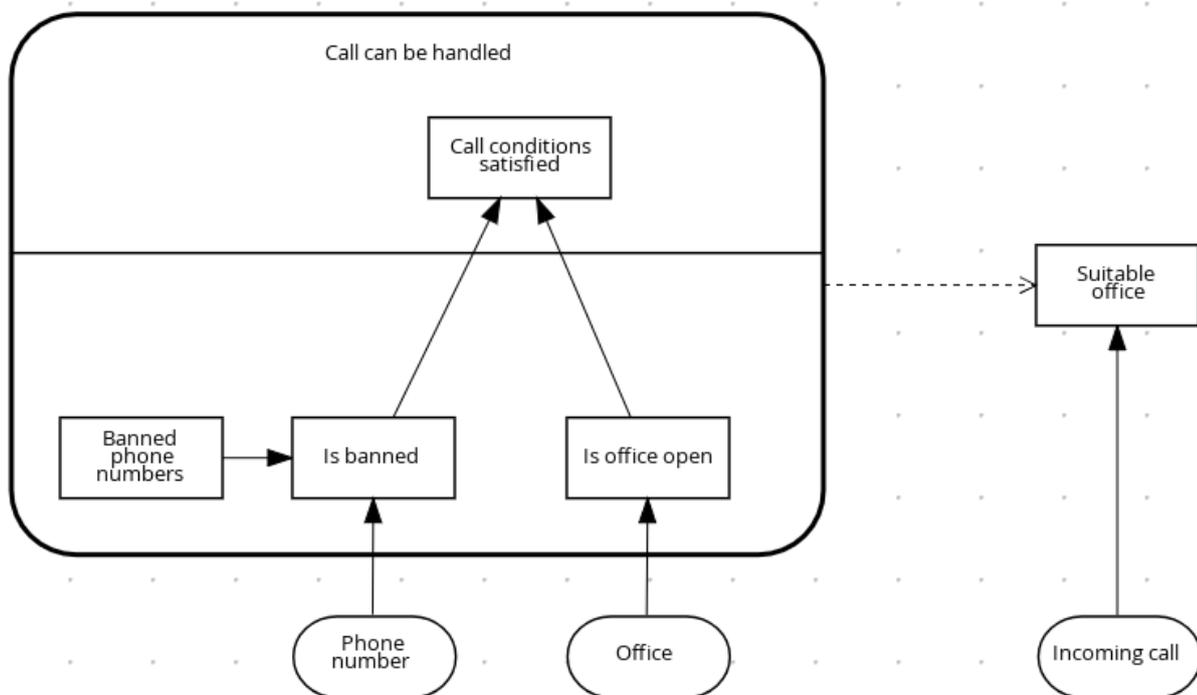
Figure 53. Example DRD: Phone call handling as a decision service

In a DMN decision service node, the decision nodes in the bottom segment incorporate input data from outside of the decision service to arrive at a final decision in the top segment of the decision service node. The resulting top-level decisions from the decision service are then implemented in any subsequent decisions or business knowledge requirements of the DMN model. You can reuse DMN decision services in other DMN models to apply the same decision logic with different input data and different outgoing connections.

4.3. Rule expressions in FEEL

Friendly Enough Expression Language (FEEL) is an expression language defined by the Object Management Group (OMG) DMN specification. FEEL expressions define the logic of a decision in a DMN model. FEEL is designed to facilitate both decision modeling and execution by assigning

semantics to the decision model constructs. FEEL expressions in decision requirements diagrams (DRDs) occupy table cells in boxed expressions for decision nodes and business knowledge models.

The following sections provide just some highlights about some of the FEEL features. For complete information about FEEL, and built-in FEEL functions in DMN, you can reference the Drools DMN engine, DMN FEEL handbook.

4.3.1. Data types in FEEL

Friendly Enough Expression Language (FEEL) supports the following data types:

- Numbers
- Strings
- Boolean values
- Dates
- Time
- Date and time
- Days and time duration
- Years and months duration
- Functions
- Contexts
- Ranges (or intervals)
- Lists

> ℹ The DMN specification currently does not provide an explicit way of declaring a variable as a `function`, `context`, `range`, or `list`, but Drools extends the DMN built-in types to support variables of these types.

The following list describes each data type:

Numbers

Numbers in FEEL are based on the IEEE 754-2008 Decimal 128 format, with 34 digits of precision. Internally, numbers are represented in Java as `BigDecimals` with `MathContext` `DECIMAL128`. FEEL supports only one number data type, so the same type is used to represent both integers and floating point numbers.

FEEL numbers use a dot (`.`) as a decimal separator. FEEL does not support `-INF`, `+INF`, or `NaN`. FEEL uses `null` to represent invalid numbers.

Drools extends the DMN specification and supports additional number notations:

- **Scientific:** You can use scientific notation with the suffix `e<exp>` or `E<exp>`. For example, `1.2e3` is the same as writing the expression `1.2*10**3`, but is a literal instead of an expression.
- **Hexadecimal:** You can use hexadecimal numbers with the prefix `0x`. For example, `0xff` is the

same as the decimal number `255`. Both uppercase and lowercase letters are supported. For example, `0XFF` is the same as `0xff`.

- **Type suffixes:** You can use the type suffixes `f`, `F`, `d`, `D`, `l`, and `L`. These suffixes are ignored.

Strings

Strings in FEEL are any sequence of characters delimited by double quotation marks.

Example

```
"John Doe"
```

Boolean values

FEEL uses three-valued boolean logic, so a boolean logic expression may have values `true`, `false`, or `null`.

Dates

Date literals are not supported in FEEL, but you can use the built-in `date()` function to construct date values. Date strings in FEEL follow the format defined in the [XML Schema Part 2: Datatypes](#) document. The format is `"YYYY-MM-DD"` where `YYYY` is the year with four digits, `MM` is the number of the month with two digits, and `DD` is the number of the day.

Example:

```
date( "2017-06-23" )
```

Date objects have time equal to `"00:00:00"`, which is midnight. The dates are considered to be local, without a timezone.

Time

Time literals are not supported in FEEL, but you can use the built-in `time()` function to construct time values. Time strings in FEEL follow the format defined in the [XML Schema Part 2: Datatypes](#) document. The format is `"hh:mm:ss[.uuu][(+-)hh:mm]"` where `hh` is the hour of the day (from `00` to `23`), `mm` is the minutes in the hour, and `ss` is the number of seconds in the minute. Optionally, the string may define the number of milliseconds (`uuu`) within the second and contain a positive (`+`) or negative (`-`) offset from UTC time to define its timezone. Instead of using an offset, you can use the letter `z` to represent the UTC time, which is the same as an offset of `-00:00`. If no offset is defined, the time is considered to be local.

Examples:

```
time( "04:25:12" )
time( "14:10:00+02:00" )
time( "22:35:40.345-05:00" )
time( "15:00:30z" )
```

Time values that define an offset or a timezone cannot be compared to local times that do not

define an offset or a timezone.

Date and time

Date and time literals are not supported in FEEL, but you can use the built-in `date and time()` function to construct date and time values. Date and time strings in FEEL follow the format defined in the XML Schema Part 2: Datatypes document. The format is `"<date>T<time>"`, where `<date>` and `<time>` follow the prescribed XML schema formatting, conjoined by `T`.

Examples:

```
date and time( "2017-10-22T23:59:00" )
date and time( "2017-06-13T14:10:00+02:00" )
date and time( "2017-02-05T22:35:40.345-05:00" )
date and time( "2017-06-13T15:00:30z" )
```

Date and time values that define an offset or a timezone cannot be compared to local date and time values that do not define an offset or a timezone.

> **!** If your implementation of the DMN specification does not support spaces in the XML schema, use the keyword `dateTime` as a synonym of `date and time`.

Days and time duration

Days and time duration literals are not supported in FEEL, but you can use the built-in `duration()` function to construct days and time duration values. Days and time duration strings in FEEL follow the format defined in the XML Schema Part 2: Datatypes document, but are restricted to only days, hours, minutes and seconds. Months and years are not supported.

Examples:

```
duration( "P1DT23H12M30S" )
duration( "P23D" )
duration( "PT12H" )
duration( "PT35M" )
```

> **!** If your implementation of the DMN specification does not support spaces in the XML schema, use the keyword `dayTimeDuration` as a synonym of `days and time duration`.

Years and months duration

Years and months duration literals are not supported in FEEL, but you can use the built-in `duration()` function to construct days and time duration values. Years and months duration strings in FEEL follow the format defined in the XML Schema Part 2: Datatypes document, but are restricted to only years and months. Days, hours, minutes, or seconds are not supported.

Examples:

```
duration( "P3Y5M" )
duration( "P2Y" )
duration( "P10M" )
duration( "P25M" )
```

> ⚠️ If your implementation of the DMN specification does not support spaces in the XML schema, use the keyword `yearMonthDuration` as a synonym of `years and months duration`.

Functions

FEEL has `function` literals (or anonymous functions) that you can use to create functions. The DMN specification currently does not provide an explicit way of declaring a variable as a `function`, but Drools extends the DMN built-in types to support variables of functions.

Example:

```
function(a, b) a + b
```

In this example, the FEEL expression creates a function that adds the parameters `a` and `b` and returns the result.

Contexts

FEEL has `context` literals that you can use to create contexts. A `context` in FEEL is a list of key and value pairs, similar to maps in languages like Java. The DMN specification currently does not provide an explicit way of declaring a variable as a `context`, but Drools extends the DMN built-in types to support variables of contexts.

Example:

```
{ x : 5, y : 3 }
```

In this example, the expression creates a context with two entries, `x` and `y`, representing a coordinate in a chart.

In DMN 1.2, another way to create contexts is to create an item definition that contains the list of keys as attributes, and then declare the variable as having that item definition type.

The Drools DMN API supports DMN `ItemDefinition` structural types in a `DMNContext` represented in two ways:

- User-defined Java type: Must be a valid JavaBeans object defining properties and getters for each of the components in the DMN `ItemDefinition`. If necessary, you can also use the `@FEELProperty` annotation for those getters representing a component name which would result in an invalid Java identifier.

- `java.util.Map` interface: The map needs to define the appropriate entries, with the keys

corresponding to the component name in the DMN `ItemDefinition`.

Ranges (or intervals)

FEEL has `range` literals that you can use to create ranges or intervals. A `range` in FEEL is a value that defines a lower and an upper bound, where either can be open or closed. The DMN specification currently does not provide an explicit way of declaring a variable as a `range`, but Drools extends the DMN built-in types to support variables of ranges.

The syntax of a range is defined in the following formats:

```
range          := interval_start endpoint '..' endpoint interval_end
interval_start := open_start | closed_start
open_start     := '(' | ']'
closed_start   := '['
interval_end   := open_end | closed_end
open_end       := ')' | '['
closed_end     := ']'
endpoint       := expression
```

The expression for the endpoint must return a comparable value, and the lower bound endpoint must be lower than the upper bound endpoint.

For example, the following literal expression defines an interval between 1 and 10, including the boundaries (a closed interval on both endpoints):

```
[ 1 .. 10 ]
```

The following literal expression defines an interval between 1 hour and 12 hours, including the lower boundary (a closed interval), but excluding the upper boundary (an open interval):

```
[ duration("PT1H") .. duration("PT12H") )
```

You can use ranges in decision tables to test for ranges of values, or use ranges in simple literal expressions. For example, the following literal expression returns `true` if the value of a variable `x` is between 0 and 100:

```
x in [ 1 .. 100 ]
```

Lists

FEEL has `list` literals that you can use to create lists of items. A `list` in FEEL is represented by a comma-separated list of values enclosed in square brackets. The DMN specification currently does not provide an explicit way of declaring a variable as a `list`, but Drools extends the DMN built-in types to support variables of lists.

Example:

```
[ 2, 3, 4, 5 ]
```

All lists in FEEL contain elements of the same type and are immutable. Elements in a list can be accessed by index, where the first element is 1. Negative indexes can access elements starting from the end of the list so that -1 is the last element.

For example, the following expression returns the second element of a list x:

```
x[2]
```

The following expression returns the second-to-last element of a list x:

```
x[-2]
```

Elements in a list can also be counted by the function count, which uses the list of elements as the parameter.

For example, the following expression returns 4:

```
count([ 2, 3, 4, 5 ])
```

4.3.2. Built-in functions in FEEL

To promote interoperability with other platforms and systems, Friendly Enough Expression Language (FEEL) includes a library of built-in functions. The built-in FEEL functions are implemented in the Drools Decision Model and Notation (DMN) engine so that you can use the functions in your DMN decision services.

For detailed information about FEEL and built-in FEEL functions in DMN, you can reference the Drools DMN engine, DMN FEEL handbook.

4.3.3. Variable and function names in FEEL

Unlike many traditional expression languages, Friendly Enough Expression Language (FEEL) supports spaces and a few special characters as part of variable and function names. A FEEL name must start with a letter, ?, or _ element. The unicode letter characters are also allowed. Variable names cannot start with a language keyword, such as and, true, or every. The remaining characters in a variable name can be any of the starting characters, as well as digits, white spaces, and special characters such as +, -, /, *, ', and ..

For example, the following names are all valid FEEL names:

- Age
- Birth Date

- Flight 234 pre-check procedure

Several limitations apply to variable and function names in FEEL:

Ambiguity

The use of spaces, keywords, and other special characters as part of names can make FEEL ambiguous. The ambiguities are resolved in the context of the expression, matching names from left to right. The parser resolves the variable name as the longest name matched in scope. You can use `( )` to disambiguate names if necessary.

Spaces in names

The DMN specification limits the use of spaces in FEEL names. According to the DMN specification, names can contain multiple spaces but not two consecutive spaces.

In order to make the language easier to use and avoid common errors due to spaces, Drools removes the limitation on the use of consecutive spaces. Drools supports variable names with any number of consecutive spaces, but normalizes them into a single space. For example, the variable references `First Name` with one space and `First Name` with two spaces are both acceptable in Drools.

Drools also normalizes the use of other white spaces, like the non-breakable white space that is common in web pages, tabs, and line breaks. From a Drools FEEL engine perspective, all of these characters are normalized into a single white space before processing.

The keyword `in`

The keyword `in` is the only keyword in the language that cannot be used as part of a variable name. Although the specifications allow the use of keywords in the middle of variable names, the use of `in` in variable names conflicts with the grammar definition of `for`, `every` and `some` expression constructs.

4.4. DMN decision logic in boxed expressions

Boxed expressions in DMN are tables that you use to define the underlying logic of decision nodes and business knowledge models in a decision requirements diagram (DRD). Some boxed expressions can contain other boxed expressions, but the top-level boxed expression corresponds to the decision logic of a single DRD artifact. While DRDs represent the flow of a DMN decision model, boxed expressions define the actual decision logic of individual nodes. DRDs and boxed expressions together form a complete and functional DMN decision model.

The following are the types of DMN boxed expressions:

- Decision tables
- Literal expressions
- Contexts
- Relations
- Functions
- Invocations

- Lists

> KIE DMN Editor does not provide boxed list expressions, but supports a FEEL `list` data type that you can use in boxed literal expressions. For more information about the `list` data type and other FEEL data types in Drools, see Data types in FEEL.

All Friendly Enough Expression Language (FEEL) expressions that you use in your boxed expressions must conform to the FEEL syntax requirements in the OMG Decision Model and Notation specification.

4.4.1. DMN decision tables

A decision table in DMN is a visual representation of one or more business rules in a tabular format. You use decision tables to define rules for a decision node that applies those rules at a given point in the decision model. Each rule consists of a single row in the table, and includes columns that define the conditions (input) and outcome (output) for that particular row. The definition of each row is precise enough to derive the outcome using the values of the conditions. Input and output values can be FEEL expressions or defined data type values.

For example, the following decision table determines credit score ratings based on a defined range of a loan applicant's credit score:
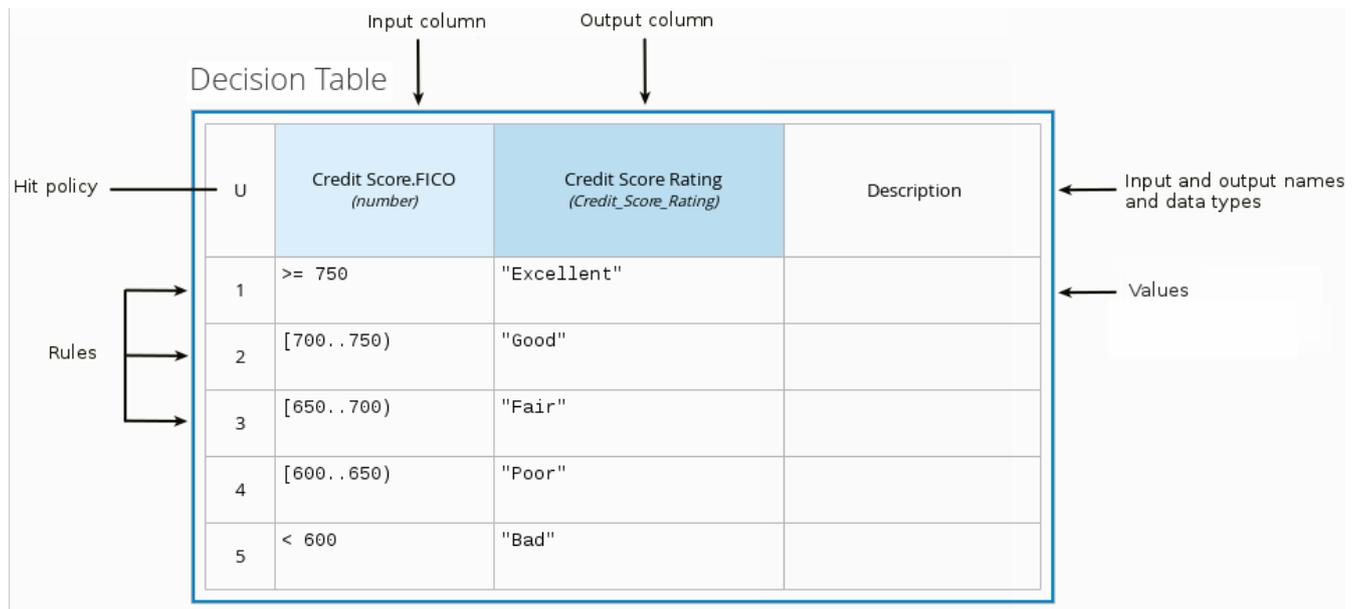
Figure 54. Decision table for credit score rating

The following decision table determines the next step in a lending strategy for applicants depending on applicant loan eligibility and the bureau call type:

Strategy (Decision Table)

U	Eligibility (string)	BureauCallType (string)	Strategy (tStrategy)	Description
1	"INELIGIBLE"	-	"DECLINE"	Disregard BureauCallType when ineligible.
2	"ELIGIBLE"	"FULL","MINI"	"BUREAU"	
3	"ELIGIBLE"	"NONE"	"THROUGH"	

Figure 55. Decision table for lending strategy

The following decision table determines applicant qualification for a loan as the concluding decision node in a loan prequalification decision model:

Loan Pre-Qualification (Decision Table)

F	Credit Score Rating (Credit_Score_Rating)	Back End Ratio (Back_End_Ratio)	Front End Ratio (Front_End_Ratio)	Loan Pre-Qualification (Loan_Qualification)		Description
				Qualification (string)	Reason (string)	
1	"Poor", "Bad"	-	-	"Not Qualified"	"Credit Score too low."	
2	-	"Insufficient"	"Sufficient"	"Not Qualified"	"Debt to income ratio is too high."	
3	-	"Sufficient"	"Insufficient"	"Not Qualified"	"Mortgage payment to income ratio is too high."	
4	-	"Insufficient"	"Insufficient"	"Not Qualified"	"Debt to income ratio is too high AND mortgage payment to income ratio is too high."	
5	"Fair", "Good", "Excellent"	"Sufficient"	"Sufficient"	"Qualified"	"The borrower has been successfully prequalified for the requested loan."	

Figure 56. Decision table for loan prequalification

Decision tables are a popular way of modeling rules and decision logic, and are used in many methodologies (such as DMN) and implementation frameworks (such as Drools).

> ❗ Drools supports both DMN decision tables and Drools-native decision tables, but they are different types of assets with different syntax requirements and are not interchangeable. For more information about Drools-native decision tables in Drools, see [decision-tables-con_decision-tables].

4.4.1.1. Hit policies in DMN decision tables

Hit policies determine how to reach an outcome when multiple rules in a decision table match the provided input values. For example, if one rule in a decision table applies a sales discount to military personnel and another rule applies a discount to students, then when a customer is both a student and in the military, the decision table hit policy must indicate whether to apply one discount or the other (**Unique**, **First**) or both discounts (**Collect Sum**). You specify the single character of the hit policy (**U**, **F**, **C+**) in the upper-left corner of the decision table.

The following decision table hit policies are supported in DMN:

- **Unique (U):** Permits only one rule to match. Any overlap raises an error.

- **Any (A):** Permits multiple rules to match, but they must all have the same output. If multiple matching rules do not have the same output, an error is raised.

- **Priority (P):** Permits multiple rules to match, with different outputs. The output that comes first in the output values list is selected.

- **First (F):** Uses the first match in rule order.

- **Collect (C+, C>, C<, C#):** Aggregates output from multiple rules based on an aggregation function.

  - **Collect (C):** Aggregates values in an arbitrary list.

  - **Collect Sum (C+):** Outputs the sum of all collected values. Values must be numeric.

  - **Collect Min (C<):** Outputs the minimum value among the matches. The resulting values must be comparable, such as numbers, dates, or text (lexicographic order).

  - **Collect Max (C>):** Outputs the maximum value among the matches. The resulting values must be comparable, such as numbers, dates or text (lexicographic order).

  - **Collect Count (C#):** Outputs the number of matching rules.

4.4.2. Boxed literal expressions

A boxed literal expression in DMN is a literal FEEL expression as text in a table cell, typically with a labeled column and an assigned data type. You use boxed literal expressions to define simple or complex node logic or decision data directly in FEEL for a particular node in a decision. Literal FEEL expressions must conform to FEEL syntax requirements in the OMG Decision Model and Notation specification.

For example, the following boxed literal expression defines the minimum acceptable PITI calculation (principal, interest, taxes, and insurance) in a lending decision, where `acceptable rate` is a variable defined in the DMN model:
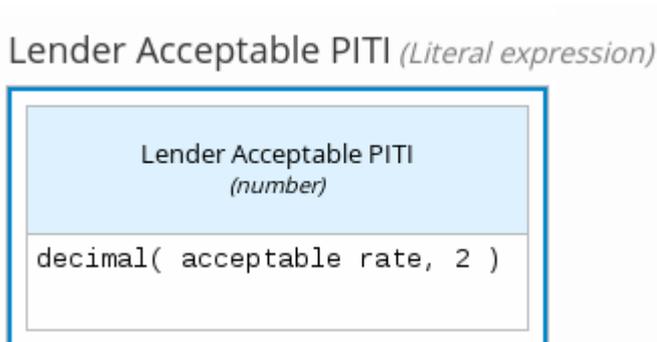
Figure 57. Boxed literal expression for minimum PITI value

The following boxed literal expression sorts a list of possible dating candidates (soul mates) in an online dating application based on their score on criteria such as age, location, and interests:

Figure 58. Boxed literal expression for matching online dating candidates

4.4.3. Boxed context expressions

A boxed context expression in DMN is a set of variable names and values with a result value. Each name-value pair is a context entry. You use context expressions to represent data definitions in decision logic and set a value for a desired decision element within the DMN decision model. A value in a boxed context expression can be a data type value or FEEL expression, or can contain a nested sub-expression of any type, such as a decision table, a literal expression, or another context expression.

For example, the following boxed context expression defines the factors for sorting delayed passengers in a flight-rebooking decision model, based on defined data types (`tPassengerTable`, `tFlightNumberList`):

Figure 59. Boxed context expression for flight passenger waiting list

The following boxed context expression defines the factors that determine whether a loan applicant can meet minimum mortgage payments based on principal, interest, taxes, and insurance (PITI), represented as a front-end ratio calculation with a sub-context expression:

Front End Ratio (Context)

#		Front End Ratio (Front_End_Ratio)		
1	Client PITI (number)	#	PITI	
		1	pmt (number)	(Requested Product.Amount*((Requested Product.Rate/100)/12))/ (1-(1/(1+(Requested Product.Rate/100)/12)**-Requested Product.Term))
		2	tax (number)	Applicant Data.Monthly.Tax
		3	insurance (number)	Applicant Data.Monthly.Insurance
		4	income (number)	Applicant Data.Monthly.Income
	<result>	if Client PITI <= Lender Acceptable PITI() then "Sufficient" else "Insufficient"		

Figure 60. Boxed context expression for front-end client PITI ratio

4.4.4. Boxed relation expressions

A boxed relation expression in DMN is a traditional data table with information about given entities, listed as rows. You use boxed relation tables to define decision data for relevant entities in a decision at a particular node. Boxed relation expressions are similar to context expressions in that they set variable names and values, but relation expressions contain no result value and list all variable values based on a single defined variable in each column.

For example, the following boxed relation expression provides information about employees in an employee rostering decision:

Employee Information (Relation)

#	Name (string)	Dept (string)	Salary (number)
1	"John"	"Sales"	100000
2	"Mary"	"Finances"	120000

Figure 61. Boxed relation expression with employee information

4.4.5. Boxed function expressions

A boxed function expression in DMN is a parameterized boxed expression containing a literal FEEL expression, a nested context expression of an external JAVA or PMML function, or a nested boxed expression of any type. By default, all business knowledge models are defined as boxed function expressions. You use boxed function expressions to call functions on your decision logic and to define all business knowledge models.

For example, the following boxed function expression determines airline flight capacity in a flight-rebooking decision model:

Flight Capacity *(Function)*

F	Flight Capacity *(boolean)*
	(flight, rebooked list)
	flight.Capacity > count(rebooked list[Flight Number = flight.Flight Number])

Figure 62. Boxed function expression for flight capacity

The following boxed function expression contains a basic Java function as a context expression for determining absolute value in a decision model calculation:

Absolute *(Function)*

J	Absolute *(number)*	
	(value)	
	1	class *(string)*	"java.lang.Math"
	2	method signature *(string)*	"abs(double)"

Figure 63. Boxed function expression for absolute value

The following boxed function expression determines a monthly mortgage installment as a business knowledge model in a lending decision, with the function value defined as a nested context expression:

Figure 64. Boxed function expression for installment calculation in business knowledge model

The following boxed function expression uses a PMML model included in the DMN file to define the minimum acceptable PITI calculation (principal, interest, taxes, and insurance) in a lending decision:

Figure 65. Boxed function expression with an included PMML model in business knowledge model

4.4.6. Boxed invocation expressions

A boxed invocation expression in DMN is a boxed expression that invokes a business knowledge model. A boxed invocation expression contains the name of the business knowledge model to be invoked and a list of parameter bindings. Each binding is represented by two boxed expressions on a row: The box on the left contains the name of a parameter and the box on the right contains the binding expression whose value is assigned to the parameter to evaluate the invoked business knowledge model. You use boxed invocations to invoke at a particular decision node a business knowledge model defined in the decision model.

For example, the following boxed invocation expression invokes a `Reassign Next Passenger` business knowledge model as the concluding decision node in a flight-rebooking decision model:

Figure 66. Boxed invocation expression to reassign flight passengers

The following boxed invocation expression invokes an `InstallmentCalculation` business knowledge model to calculate a monthly installment amount for a loan before proceeding to affordability decisions:

Figure 67. Boxed invocation expression for required monthly installment

4.4.7. Boxed list expressions

A boxed list expression in DMN represents a FEEL list of items. You use boxed lists to define lists of relevant items for a particular node in a decision. You can also use literal FEEL expressions for list items in cells to create more complex lists.

For example, the following boxed list expression identifies approved credit score agencies in a loan application decision service:

Figure 68. Boxed list expression for approved credit score agencies

The following boxed list expression also identifies approved credit score agencies but uses FEEL logic to define the agency status (Inc., LLC, SA, GA) based on a DMN input node:

Figure 69. Boxed list expression using FEEL logic for approved credit score agency status
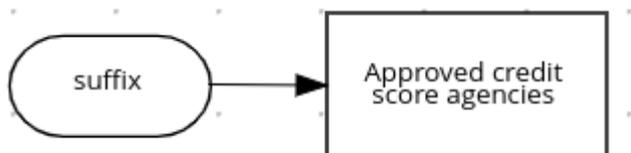
4.5. DMN model example

The following is a real-world DMN model example that demonstrates how you can use decision modeling to reach a decision based on input data, circumstances, and company guidelines. In this scenario, a flight from San Diego to New York is canceled, requiring the affected airline to find alternate arrangements for its inconvenienced passengers.

First, the airline collects the information necessary to determine how best to get the travelers to their destinations:

Input data

- List of flights
- List of passengers

Decisions

- Prioritize the passengers who will get seats on a new flight

- Determine which flights those passengers will be offered

Business knowledge models

- The company process for determining passenger priority

- Any flights that have space available

- Company rules for determining how best to reassign inconvenienced passengers

The airline then uses the DMN standard to model its decision process in the following decision requirements diagram (DRD) for determining the best rebooking solution:
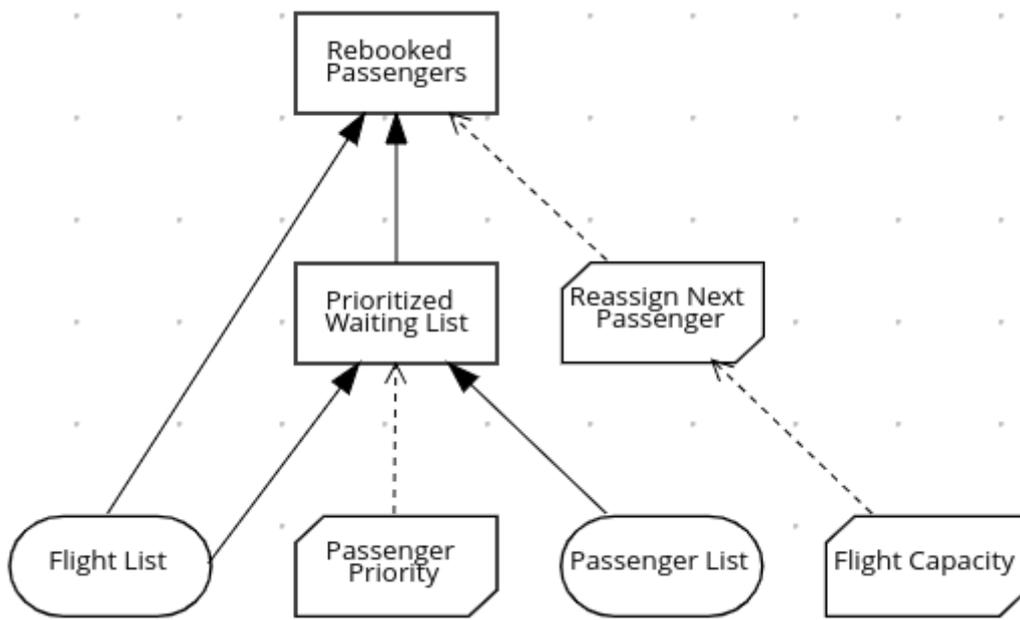
Figure 70. DRD for flight rebooking

Similar to flowcharts, DRDs use shapes to represent the different elements in a process. Ovals contain the two necessary input data, rectangles contain the decision points in the model, and rectangles with clipped corners (business knowledge models) contain reusable logic that can be repeatedly invoked.

The DRD draws logic for each element from boxed expressions that provide variable definitions using FEEL expressions or data type values.

Some boxed expressions are basic, such as the following decision for establishing a prioritized waiting list:

Prioritized Waiting List *(Context)*

#	Prioritized Waiting List (tPassengerTable)	
1	Cancelled Flights *(tFlightNumberList)*	`Flight List[ Status = "cancelled" ].Flight Number`
2	Waiting List *(tPassengerTable)*	`Passenger List[ list contains( Cancelled Flights, Flight Number ) ]`
	<result>	`sort( Waiting List, Passenger Priority )`

Figure 71. Boxed context expression example for prioritized wait list

Some boxed expressions are more complex with greater detail and calculation, such as the following business knowledge model for reassigning the next delayed passenger:

Reassign Next Passenger *(Function)*

Figure 72. Boxed function expression for passenger reassignment

The following is the DMN source file for this decision model:

```
<dmn:definitions xmlns="https://www.drools.org/kie-dmn/Flight-rebooking" xmlns:dmn=
"http://www.omg.org/spec/DMN/20151101/dmn.xsd" xmlns:feel=
"http://www.omg.org/spec/FEEL/20140401" id="_0019_flight_rebooking" name="0019-flight-
```

```xml
rebooking" namespace="https://www.drools.org/kie-dmn/Flight-rebooking">
  <dmn:itemDefinition id="_tFlight" name="tFlight">
    <dmn:itemComponent id="_tFlight_Flight" name="Flight Number">
      <dmn:typeRef>feel:string</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_tFlight_From" name="From">
      <dmn:typeRef>feel:string</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_tFlight_To" name="To">
      <dmn:typeRef>feel:string</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_tFlight_Dep" name="Departure">
      <dmn:typeRef>feel:dateTime</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_tFlight_Arr" name="Arrival">
      <dmn:typeRef>feel:dateTime</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_tFlight_Capacity" name="Capacity">
      <dmn:typeRef>feel:number</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_tFlight_Status" name="Status">
      <dmn:typeRef>feel:string</dmn:typeRef>
    </dmn:itemComponent>
  </dmn:itemDefinition>
  <dmn:itemDefinition id="_tFlightTable" isCollection="true" name="tFlightTable">
    <dmn:typeRef>tFlight</dmn:typeRef>
  </dmn:itemDefinition>
  <dmn:itemDefinition id="_tPassenger" name="tPassenger">
    <dmn:itemComponent id="_tPassenger_Name" name="Name">
      <dmn:typeRef>feel:string</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_tPassenger_Status" name="Status">
      <dmn:typeRef>feel:string</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_tPassenger_Miles" name="Miles">
      <dmn:typeRef>feel:number</dmn:typeRef>
    </dmn:itemComponent>
    <dmn:itemComponent id="_tPassenger_Flight" name="Flight Number">
      <dmn:typeRef>feel:string</dmn:typeRef>
    </dmn:itemComponent>
  </dmn:itemDefinition>
  <dmn:itemDefinition id="_tPassengerTable" isCollection="true" name="tPassengerTable
">
    <dmn:typeRef>tPassenger</dmn:typeRef>
  </dmn:itemDefinition>
  <dmn:itemDefinition id="_tFlightNumberList" isCollection="true" name=
"tFlightNumberList">
    <dmn:typeRef>feel:string</dmn:typeRef>
  </dmn:itemDefinition>
  <dmn:inputData id="i_Flight_List" name="Flight List">
    <dmn:variable name="Flight List" typeRef="tFlightTable"/>
```

```
      </dmn:inputData>
      <dmn:inputData id="i_Passenger_List" name="Passenger List">
        <dmn:variable name="Passenger List" typeRef="tPassengerTable"/>
      </dmn:inputData>
      <dmn:decision name="Prioritized Waiting List" id="d_PrioritizedWaitingList">
        <dmn:variable name="Prioritized Waiting List" typeRef="tPassengerTable"/>
        <dmn:informationRequirement>
          <dmn:requiredInput href="#i_Passenger_List"/>
        </dmn:informationRequirement>
        <dmn:informationRequirement>
          <dmn:requiredInput href="#i_Flight_List"/>
        </dmn:informationRequirement>
        <dmn:knowledgeRequirement>
          <dmn:requiredKnowledge href="#b_PassengerPriority"/>
        </dmn:knowledgeRequirement>
        <dmn:context>
          <dmn:contextEntry>
            <dmn:variable name="Cancelled Flights" typeRef="tFlightNumberList"/>
            <dmn:literalExpression>
              <dmn:text>Flight List[ Status = "cancelled" ].Flight Number</dmn:text>
            </dmn:literalExpression>
          </dmn:contextEntry>
          <dmn:contextEntry>
            <dmn:variable name="Waiting List" typeRef="tPassengerTable"/>
            <dmn:literalExpression>
              <dmn:text>Passenger List[ list contains( Cancelled Flights, Flight Number )
]</dmn:text>
            </dmn:literalExpression>
          </dmn:contextEntry>
          <dmn:contextEntry>
            <dmn:literalExpression>
              <dmn:text>sort( Waiting List, passenger priority )</dmn:text>
            </dmn:literalExpression>
          </dmn:contextEntry>
        </dmn:context>
      </dmn:decision>
      <dmn:decision name="Rebooked Passengers" id="d_RebookedPassengers">
        <dmn:variable name="Rebooked Passengers" typeRef="tPassengerTable"/>
        <dmn:informationRequirement>
          <dmn:requiredDecision href="#d_PrioritizedWaitingList"/>
        </dmn:informationRequirement>
        <dmn:informationRequirement>
          <dmn:requiredInput href="#i_Flight_List"/>
        </dmn:informationRequirement>
        <dmn:knowledgeRequirement>
          <dmn:requiredKnowledge href="#b_ReassignNextPassenger"/>
        </dmn:knowledgeRequirement>
        <dmn:invocation>
          <dmn:literalExpression>
            <dmn:text>reassign next passenger</dmn:text>
          </dmn:literalExpression>
```

```xml
      <dmn:binding>
        <dmn:parameter name="Waiting List"/>
        <dmn:literalExpression>
          <dmn:text>Prioritized Waiting List</dmn:text>
        </dmn:literalExpression>
      </dmn:binding>
      <dmn:binding>
        <dmn:parameter name="Reassigned Passengers List"/>
        <dmn:literalExpression>
          <dmn:text>[]</dmn:text>
        </dmn:literalExpression>
      </dmn:binding>
      <dmn:binding>
        <dmn:parameter name="Flights"/>
        <dmn:literalExpression>
          <dmn:text>Flight List</dmn:text>
        </dmn:literalExpression>
      </dmn:binding>
    </dmn:invocation>
</dmn:decision>
<dmn:businessKnowledgeModel id="b_PassengerPriority" name="passenger priority">
  <dmn:encapsulatedLogic>
    <dmn:formalParameter name="Passenger1" typeRef="tPassenger"/>
    <dmn:formalParameter name="Passenger2" typeRef="tPassenger"/>
    <dmn:decisionTable hitPolicy="UNIQUE">
      <dmn:input id="b_Passenger_Priority_dt_i_P1_Status" label="Passenger1.Status">
        <dmn:inputExpression typeRef="feel:string">
          <dmn:text>Passenger1.Status</dmn:text>
        </dmn:inputExpression>
        <dmn:inputValues>
          <dmn:text>"gold", "silver", "bronze"</dmn:text>
        </dmn:inputValues>
      </dmn:input>
      <dmn:input id="b_Passenger_Priority_dt_i_P2_Status" label="Passenger2.Status">
        <dmn:inputExpression typeRef="feel:string">
          <dmn:text>Passenger2.Status</dmn:text>
        </dmn:inputExpression>
        <dmn:inputValues>
          <dmn:text>"gold", "silver", "bronze"</dmn:text>
        </dmn:inputValues>
      </dmn:input>
      <dmn:input id="b_Passenger_Priority_dt_i_P1_Miles" label="Passenger1.Miles">
        <dmn:inputExpression typeRef="feel:string">
          <dmn:text>Passenger1.Miles</dmn:text>
        </dmn:inputExpression>
      </dmn:input>
      <dmn:output id="b_Status_Priority_dt_o" label="Passenger1 has priority">
        <dmn:outputValues>
          <dmn:text>true, false</dmn:text>
        </dmn:outputValues>
        <dmn:defaultOutputEntry>
```

```
          <dmn:text>false</dmn:text>
        </dmn:defaultOutputEntry>
      </dmn:output>
      <dmn:rule id="b_Passenger_Priority_dt_r1">
        <dmn:inputEntry id="b_Passenger_Priority_dt_r1_i1">
          <dmn:text>"gold"</dmn:text>
        </dmn:inputEntry>
        <dmn:inputEntry id="b_Passenger_Priority_dt_r1_i2">
          <dmn:text>"gold"</dmn:text>
        </dmn:inputEntry>
        <dmn:inputEntry id="b_Passenger_Priority_dt_r1_i3">
          <dmn:text>>= Passenger2.Miles</dmn:text>
        </dmn:inputEntry>
        <dmn:outputEntry id="b_Passenger_Priority_dt_r1_o1">
          <dmn:text>true</dmn:text>
        </dmn:outputEntry>
      </dmn:rule>
      <dmn:rule id="b_Passenger_Priority_dt_r2">
        <dmn:inputEntry id="b_Passenger_Priority_dt_r2_i1">
          <dmn:text>"gold"</dmn:text>
        </dmn:inputEntry>
        <dmn:inputEntry id="b_Passenger_Priority_dt_r2_i2">
          <dmn:text>"silver","bronze"</dmn:text>
        </dmn:inputEntry>
        <dmn:inputEntry id="b_Passenger_Priority_dt_r2_i3">
          <dmn:text>-</dmn:text>
        </dmn:inputEntry>
        <dmn:outputEntry id="b_Passenger_Priority_dt_r2_o1">
          <dmn:text>true</dmn:text>
        </dmn:outputEntry>
      </dmn:rule>
      <dmn:rule id="b_Passenger_Priority_dt_r3">
        <dmn:inputEntry id="b_Passenger_Priority_dt_r3_i1">
          <dmn:text>"silver"</dmn:text>
        </dmn:inputEntry>
        <dmn:inputEntry id="b_Passenger_Priority_dt_r3_i2">
          <dmn:text>"silver"</dmn:text>
        </dmn:inputEntry>
        <dmn:inputEntry id="b_Passenger_Priority_dt_r3_i3">
          <dmn:text>>= Passenger2.Miles</dmn:text>
        </dmn:inputEntry>
        <dmn:outputEntry id="b_Passenger_Priority_dt_r3_o1">
          <dmn:text>true</dmn:text>
        </dmn:outputEntry>
      </dmn:rule>
      <dmn:rule id="b_Passenger_Priority_dt_r4">
        <dmn:inputEntry id="b_Passenger_Priority_dt_r4_i1">
          <dmn:text>"silver"</dmn:text>
        </dmn:inputEntry>
        <dmn:inputEntry id="b_Passenger_Priority_dt_r4_i2">
          <dmn:text>"bronze"</dmn:text>
```

```
          </dmn:inputEntry>
          <dmn:inputEntry id="b_Passenger_Priority_dt_r4_i3">
            <dmn:text>-</dmn:text>
          </dmn:inputEntry>
          <dmn:outputEntry id="b_Passenger_Priority_dt_r4_o1">
            <dmn:text>true</dmn:text>
          </dmn:outputEntry>
        </dmn:rule>
        <dmn:rule id="b_Passenger_Priority_dt_r5">
          <dmn:inputEntry id="b_Passenger_Priority_dt_r5_i1">
            <dmn:text>"bronze"</dmn:text>
          </dmn:inputEntry>
          <dmn:inputEntry id="b_Passenger_Priority_dt_r5_i2">
            <dmn:text>"bronze"</dmn:text>
          </dmn:inputEntry>
          <dmn:inputEntry id="b_Passenger_Priority_dt_r5_i3">
            <dmn:text>>= Passenger2.Miles</dmn:text>
          </dmn:inputEntry>
          <dmn:outputEntry id="b_Passenger_Priority_dt_r5_o1">
            <dmn:text>true</dmn:text>
          </dmn:outputEntry>
        </dmn:rule>
      </dmn:decisionTable>
    </dmn:encapsulatedLogic>
    <dmn:variable name="passenger priority" typeRef="feel:boolean"/>
  </dmn:businessKnowledgeModel>
  <dmn:businessKnowledgeModel id="b_ReassignNextPassenger" name="reassign next
passenger">
    <dmn:encapsulatedLogic>
      <dmn:formalParameter name="Waiting List" typeRef="tPassengerTable"/>
      <dmn:formalParameter name="Reassigned Passengers List" typeRef="tPassengerTable
"/>
      <dmn:formalParameter name="Flights" typeRef="tFlightTable"/>
      <dmn:context>
        <dmn:contextEntry>
          <dmn:variable name="Next Passenger" typeRef="tPassenger"/>
          <dmn:literalExpression>
            <dmn:text>Waiting List[1]</dmn:text>
          </dmn:literalExpression>
        </dmn:contextEntry>
        <dmn:contextEntry>
          <dmn:variable name="Original Flight" typeRef="tFlight"/>
          <dmn:literalExpression>
            <dmn:text>Flights[ Flight Number = Next Passenger.Flight Number
][1]</dmn:text>
          </dmn:literalExpression>
        </dmn:contextEntry>
        <dmn:contextEntry>
          <dmn:variable name="Best Alternate Flight" typeRef="tFlight"/>
          <dmn:literalExpression>
            <dmn:text>Flights[ From = Original Flight.From and To = Original Flight.To
```

```
and Departure > Original Flight.Departure and Status = "scheduled" and has capacity(
item, Reassigned Passengers List ) ][1]</dmn:text>
          </dmn:literalExpression>
        </dmn:contextEntry>
        <dmn:contextEntry>
          <dmn:variable name="Reassigned Passenger" typeRef="tPassenger"/>
          <dmn:context>
            <dmn:contextEntry>
              <dmn:variable name="Name" typeRef="feel:string"/>
              <dmn:literalExpression>
                <dmn:text>Next Passenger.Name</dmn:text>
              </dmn:literalExpression>
            </dmn:contextEntry>
            <dmn:contextEntry>
              <dmn:variable name="Status" typeRef="feel:string"/>
              <dmn:literalExpression>
                <dmn:text>Next Passenger.Status</dmn:text>
              </dmn:literalExpression>
            </dmn:contextEntry>
            <dmn:contextEntry>
              <dmn:variable name="Miles" typeRef="feel:number"/>
              <dmn:literalExpression>
                <dmn:text>Next Passenger.Miles</dmn:text>
              </dmn:literalExpression>
            </dmn:contextEntry>
            <dmn:contextEntry>
              <dmn:variable name="Flight Number" typeRef="feel:string"/>
              <dmn:literalExpression>
                <dmn:text>Best Alternate Flight.Flight Number</dmn:text>
              </dmn:literalExpression>
            </dmn:contextEntry>
          </dmn:context>
        </dmn:contextEntry>
        <dmn:contextEntry>
          <dmn:variable name="Remaining Waiting List" typeRef="tPassengerTable"/>
          <dmn:literalExpression>
            <dmn:text>remove( Waiting List, 1 )</dmn:text>
          </dmn:literalExpression>
        </dmn:contextEntry>
        <dmn:contextEntry>
          <dmn:variable name="Updated Reassigned Passengers List" typeRef=
"tPassengerTable"/>
          <dmn:literalExpression>
            <dmn:text>append( Reassigned Passengers List, Reassigned Passenger
)</dmn:text>
          </dmn:literalExpression>
        </dmn:contextEntry>
        <dmn:contextEntry>
          <dmn:literalExpression>
            <dmn:text>if count( Remaining Waiting List ) > 0 then reassign next
passenger( Remaining Waiting List, Updated Reassigned Passengers List, Flights ) else
```

```
Updated Reassigned Passengers List</dmn:text>
          </dmn:literalExpression>
        </dmn:contextEntry>
      </dmn:context>
    </dmn:encapsulatedLogic>
    <dmn:variable name="reassign next passenger" typeRef="tPassengerTable"/>
    <dmn:knowledgeRequirement>
      <dmn:requiredKnowledge href="#b_HasCapacity"/>
    </dmn:knowledgeRequirement>
  </dmn:businessKnowledgeModel>
  <dmn:businessKnowledgeModel id="b_HasCapacity" name="has capacity">
    <dmn:encapsulatedLogic>
      <dmn:formalParameter name="flight" typeRef="tFlight"/>
      <dmn:formalParameter name="rebooked list" typeRef="tPassengerTable"/>
      <dmn:literalExpression>
        <dmn:text>flight.Capacity > count( rebooked list[ Flight Number =
flight.Flight Number ] )</dmn:text>
      </dmn:literalExpression>
    </dmn:encapsulatedLogic>
    <dmn:variable name="has capacity" typeRef="feel:boolean"/>
  </dmn:businessKnowledgeModel>
</dmn:definitions>
```

Chapter 5. DMN support in Drools

Drools DMN engine provides runtime support for DMN 1.1, 1.2, 1.3, and 1.4 models at conformance level 3.

KIE DMN Editor provides design support for DMN 1.2 models at conformance level 3.

You can integrate DMN models with your Drools decision services in several ways:

- Design your DMN models using the KIE DMN Editor online.

- Design your DMN models using the KIE DMN Editor in VSCode.

- Import DMN files into your project by opening them in KIE DMN Editor. Any DMN 1.1 and 1.3 models (do not contain DMN 1.3 features) that you import or open into KIE DMN Editor and save are converted to DMN 1.2 models.

- Package DMN files as part of your project knowledge JAR (KJAR) file without KIE DMN Editor.

The following table summarizes the design and runtime support for each DMN version in Drools:

Table 11. DMN support in Drools

DMN version	DMN engine support	DMN modeler support	
	Execution	Open	Save
DMN 1.1	yes	yes	no
DMN 1.2	yes	yes	yes
DMN 1.3	yes	yes	no
DMN 1.4	yes	no	no

In addition to all DMN conformance level 3 requirements, Drools DMN engine also includes enhancements and fixes to FEEL and DMN model components to optimize the experience of implementing DMN decision services with Drools. From a platform perspective, DMN models are like any other business asset in Drools, such as DRL files or spreadsheet decision tables, that you can include in your Drools project and deploy in order to start your DMN decision services.

For more information about including external DMN files with your Drools project packaging and deployment method, see [_builddeployutilizeandrunsection].

You can design a new DMN decision service using a Kogito microservice as an alternative for the cloud-native capabilities of DMN decision services. You could also migrate your existing DMN service to a Kogito microservice. For more information about Kogito or migrating to Kogito microservices, see the Kogito website for documentation.

5.1. FEEL enhancements in Drools DMN engine

Drools DMN engine includes the following enhancements and other changes to FEEL in the current DMN implementation:

- *Space Sensitivity*: This DMN implementation of the FEEL language is space insensitive. The goal is to avoid non-deterministic behavior based on the context and differences in behavior based on invisible characters, such as white spaces. This means that for this implementation, a variable named `first name` with one space is exactly the same as `first name` with two spaces in it.

- *List functions* `or()` *and* `and()` : The specification defines two list functions named `or()` and `and()`. However, according to the FEEL grammar, these are not valid function names, as `and` and `or` are reserved keywords. This implementation renames these functions to `any()` and `all()` respectively, in anticipation for DMN 1.2.

- *Keyword* `in` *cannot be used in variable names*: The specification defines that any keyword can be reused as part of a variable name, but the ambiguities caused with the `for ⋯ in ⋯ return` loop prevent the reuse of the `in` keyword. All other keywords are supported as part of variable names.

- *Keywords are not supported in attributes of anonymous types*: FEEL is not a strongly typed language and the parser must resolve ambiguity in name parts of an attribute of an anonymous type. The parser supports reusable keywords as part of a variable name defined in the scope, but the parser does not support keywords in attributes of an anonymous type. For example, `for item in Order.items return Federal Tax for Item( item )` is a valid and supported FEEL expression, where a function named `Federal Tax for Item(⋯)` can be defined and invoked correctly in the scope. However, the expression `for i in [ {x and y : true, n : 1}, {x and y : false, n: 2} ] return i.x and y` is not supported because anonymous types are defined in the iteration context of the `for` expression and the parser cannot resolve the ambiguity.

- *Support for date and time literals on ranges*: According to the grammar rules #8, #18, #19, #34 and #62, `date and time` literals are supported in ranges (pages 110-111). Chapter 10.3.2.7 on page 114, on the other hand, contradicts the grammar and says they are not supported. This implementation chose to follow the grammar and support `date and time` literals on ranges, as well as extend the specification to support any arbitrary expression (see extensions below).

- *Invalid time syntax*: Chapter 10.3.2.3.4 on page 112 and bullet point about `time` on page 131 both state that the `time` string lexical representation follows the XML Schema Datatypes specification as well as ISO 8601. According to the XML Schema specification (https://www.w3.org/TR/xmlschema-2/#time), the lexical representation of a time follows the pattern `hh:mm:ss.sss` without any leading character. The DMN specification uses a leading "T" in several examples, that we understand is a typo and not in accordance with the standard.

- *Support for scientific and hexadecimal notations*: This implementation supports scientific and hexadecimal notation for numbers. For example, `1.2e5` (scientific notation), `0xD5` (hexadecimal notation).

- *Support for expressions as end points in ranges*: This implementation supports expressions as endpoints for ranges. For example, `[date("2016-11-24")..date("2016-11-27")]`

- *Support for additional types*: The specification only defines the following as basic types of the language:
  - number
  - string
  - boolean

- days and time duration

- years and month duration

- time

- date and time

  For completeness and orthogonality, this implementation also supports the following types:

- context

- list

- range

- function

- unary test

- *Support for unary tests*: For completeness and orthogonality, unary tests are supported as first class citizens in the language. They are functions with an implicit single parameter and can be invoked in the same way as functions. For example,

UnaryTestAsFunction.feel

```
{
    is minor : < 18,
    Bob is minor : is minor( bob.age )
}
```

- *Support for additional built-in functions*: The following additional functions are supported:

  - `now()` : Returns the current local date and time.

  - `today()` : Returns the current local date.

  - `decision table()` : Returns a decision table function, although the specification mentions a decision table. The function on page 114 is not implementable as defined.

  - `string( mask, p… )` : Returns a string formatted as per the mask. See Java String.format() for details on the mask syntax. For example, `string( "%4.2f", 7.1298 )` returns the string `"7.12"`.

- *Support for additional date and time arithmetics*: Subtracting two dates returns a day and time duration with the number of days between the two dates, ignoring daylight savings. For example,

DateArithmetic.feel

```
date( "2017-05-12" ) - date( "2017-04-25" ) = duration( "P17D" )
```

5.2. DMN model enhancements in Drools DMN engine

Drools DMN engine includes the following enhancements to DMN model support in the current

DMN implementation:

- *Support for types with spaces on names*: The DMN XML schema defines type refs such as QNames. The QNames do not allow spaces. Therefore, it is not possible to use types like FEEL `date and time`, `days and time duration` or `years and months duration`. This implementation does parse such typerefs as strings and allows type names with spaces. However, in order to comply with the XML schema, it also adds the following aliases to such types that can be used instead:

  - `date and time` = `dateTime`

  - `days and time duration` = `duration` or `dayTimeDuration`

  - `years and months duration` = `duration` or `yearMonthDuration`

  Note that, for the "duration" types, the user can simply use `duration` and the Drools DMN engine will infer the proper duration, either `days and time duration` or `years and months duration`.

- *Lists support heterogeneous element types*: Currently this implementation supports lists with heterogeneous element types. This is an experimental extension and does limit the functionality of some functions and filters. This decision will be re-evaluated in the future.

- *TypeRef link between Decision Tables and Item Definitions*: On decision tables/input clause, if no values list is defined, the Drools DMN engine automatically checks the type reference and applies the allowed values check if it is defined.

5.3. Configurable DMN properties in Drools DMN engine

Drools DMN engine provides the following DMN properties that you can configure when you execute your DMN models on your client application. You can configure some of these properties using the `kmodule.xml` file in your Drools project (preferred method also for the previous traditional KIE Server deployment).

org.kie.dmn.strictConformance

When enabled, this property disables by default any extensions or profiles provided beyond the DMN standard, such as some helper functions or enhanced features of DMN 1.2 backported into DMN 1.1. You can use this property to configure the Drools DMN engine to support only pure DMN features, such as when running the DMN Technology Compatibility Kit (TCK).

Default value: `false`

```
-Dorg.kie.dmn.strictConformance=true
```

org.kie.dmn.runtime.typecheck

When enabled, this property enables verification of actual values conforming to their declared types in the DMN model, as input or output of DRD elements. You can use this property to verify whether data supplied to the DMN model or produced by the DMN model is compliant with what is specified in the model.

Default value: `false`

```
-Dorg.kie.dmn.runtime.typecheck=true
```

org.kie.dmn.decisionservice.coercesingleton

By default, this property makes the result of a decision service defining a single output decision be the single value of the output decision value. When disabled, this property makes the result of a decision service defining a single output decision be a `context` with the single entry for that decision. You can use this property to adjust your decision service outputs according to your project requirements.

Default value: `true`

```
-Dorg.kie.dmn.decisionservice.coercesingleton=false
```

org.kie.dmn.profiles.$PROFILE_NAME

When valorized with a Java fully qualified name, this property loads a DMN profile onto the Drools DMN engine at start time. You can use this property to implement a predefined DMN profile with supported features different from or beyond the DMN standard. For example, if you are creating DMN models using the Signavio DMN modeller, use this property to implement features from the Signavio DMN profile into your DMN decision service.

```
-Dorg.kie.dmn.profiles.signavio=org.kie.dmn.signavio.KieDMNSignavioProfile
```

org.kie.dmn.runtime.listeners.$LISTENER_NAME

When valorized with a Java fully qualified name, this property loads and registers a DMN Runtime Listener onto the Drools DMN engine at start time. You can use this property to register a DMN listener in order to be notified of several events during DMN model evaluations.

To configure this property, modify this property in the `kmodule.xml` file of your project (preferred method also for the previous traditional KIE Server deployment).

```xml
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
  <configuration>
    <property key="org.kie.dmn.runtime.listeners.mylistener" value=
"org.acme.MyDMNListener"/>
  </configuration>
</kmodule>
```

Alternatively, you can configure this property globally for your Drools environment; modify this property using a command terminal or any other global application configuration mechanism.

```
-Dorg.kie.dmn.runtime.listeners.mylistener=org.acme.MyDMNListener
```

org.kie.dmn.compiler.execmodel

When enabled, this property enables DMN decision table logic to be compiled into executable rule models during run time. You can use this property to evaluate DMN decision table logic more efficiently. This property is helpful when the executable model compilation was not originally performed during project compile time. Enabling this property may result in added compile time during the first evaluation by the Drools DMN engine, but subsequent compilations are more efficient.

Default value: `false`

```
-Dorg.kie.dmn.compiler.execmodel=true
```

5.4. Configurable DMN validation in Drools

By default, the `kie-maven-plugin` component in the `pom.xml` file of your Drools project uses the following `<validateDMN>` configurations to perform pre-compilation validation of DMN model assets and to perform DMN decision table static analysis:

- `VALIDATE_SCHEMA`: DMN model files are verified against the DMN specification XSD schema to ensure that the files are valid XML and compliant with the specification.

- `VALIDATE_MODEL`: The pre-compilation analysis is performed for the DMN model to ensure that the basic semantic is aligned with the DMN specification.

- `ANALYZE_DECISION_TABLE`: DMN decision tables are statically analyzed for gaps or overlaps and to ensure that the semantic of the decision table follows best practices.

You can modify the default DMN validation and DMN decision table analysis behavior to perform only a specified validation during the project build, or you can disable this default behavior completely, as shown in the following examples:

Default configuration for DMN validation and decision table analysis

```
<plugin>
  <groupId>org.kie</groupId>
  <artifactId>kie-maven-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <validateDMN>VALIDATE_SCHEMA,VALIDATE_MODEL,ANALYZE_DECISION_TABLE</validateDMN>
  </configuration>
</plugin>
```

Configuration to perform only the DMN decision table static analysis

```xml
<plugin>
  <groupId>org.kie</groupId>
  <artifactId>kie-maven-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <validateDMN>ANALYZE_DECISION_TABLE</validateDMN>
  </configuration>
</plugin>
```

Configuration to perform only the XSD schema validation

```xml
<plugin>
  <groupId>org.kie</groupId>
  <artifactId>kie-maven-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <validateDMN>VALIDATE_SCHEMA</validateDMN>
  </configuration>
</plugin>
```

Configuration to perform only the DMN model validation

```xml
<plugin>
  <groupId>org.kie</groupId>
  <artifactId>kie-maven-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <validateDMN>VALIDATE_MODEL</validateDMN>
  </configuration>
</plugin>
```

Configuration to disable all DMN validation

```xml
<plugin>
  <groupId>org.kie</groupId>
  <artifactId>kie-maven-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <validateDMN>disable</validateDMN>
  </configuration>
</plugin>
```

> **ℹ** If you enter an unrecognized `<validateDMN>` configuration flag, all pre-compilation validation is disabled and the Maven plugin emits related log messages.

Chapter 6. Creating and editing DMN models in KIE DMN Editor

You can use the KIE DMN Editor to design DMN decision requirements diagrams (DRDs) and define decision logic for a complete and functional DMN decision model. Drools provides design support for DMN 1.2 models at conformance level 3, and includes enhancements and fixes to FEEL and DMN model components to optimize the experience of implementing DMN decision services with Drools. Drools DMN engine also provides runtime support for DMN 1.1, 1.2, 1.3, and 1.4 models at conformance level 3, but any DMN 1.1 and 1.3 models (do not contain DMN 1.3 features) that you import or open into KIE DMN Editor and save are converted to DMN 1.2 models

Procedure

1. Create or import a DMN file in your KIE DMN Editor.

    To create a DMN file, follow the instruction on the KIE DMN Editor online, or create a new `.dmn` file in VSCode.

    To import an existing DMN file, follow the instruction on the KIE DMN Editor online, or double click to open a `.dmn` file in VSCode.

    > If you imported a DMN file that does not contain layout information, the imported decision requirements diagram (DRD) is formatted automatically in the DMN designer. Click **Save** in the DMN designer to save the DRD layout.
    >
    > If an imported DRD is not automatically formatted, you can select the **Perform automatic layout** icon in the upper-right toolbar in the DMN designer to format the DRD.

2. Begin adding components to your new or imported DMN decision requirements diagram (DRD) by clicking and dragging one of the DMN nodes from the left toolbar:
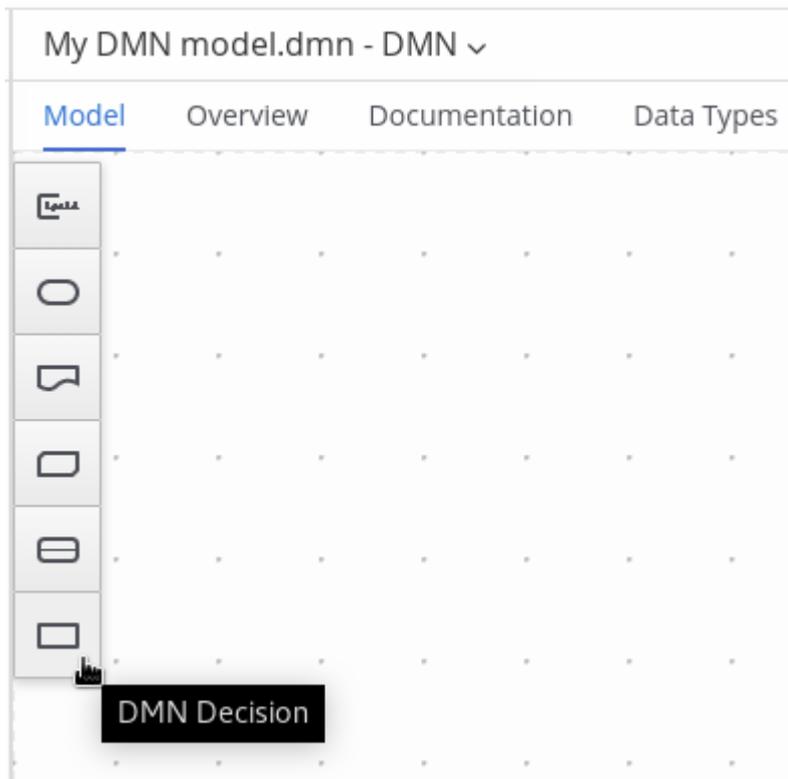
Figure 73. Adding DRD components

The following DRD components are available:

- **Decision**: Use this node for a DMN decision, where one or more input elements determine an output based on defined decision logic.

- **Business knowledge model**: Use this node for reusable functions with one or more decision elements. Decisions that have the same logic but depend on different sub-input data or sub-decisions use business knowledge models to determine which procedure to follow.

- **Knowledge source**: Use this node for external authorities, documents, committees, or policies that regulate a decision or business knowledge model. Knowledge sources are references to real-world factors rather than executable business rules.

- **Input data**: Use this node for information used in a decision node or a business knowledge model. Input data usually includes business-level concepts or objects relevant to the business, such as loan applicant data used in a lending strategy.

- **Text annotation**: Use this node for explanatory notes associated with an input data node, decision node, business knowledge model, or knowledge source.

- **Decision service**: Use this node to enclose a set of reusable decisions implemented as a decision service for invocation. A decision service can be used in other DMN models and can be invoked from an external application or a BPMN business process.

3. In the DMN designer canvas, double-click the new DRD node to enter an informative node name.

4. If the node is a decision or business knowledge model, select the node to display the node options and click the **Edit** icon to open the DMN boxed expression designer to define the decision logic for the node:
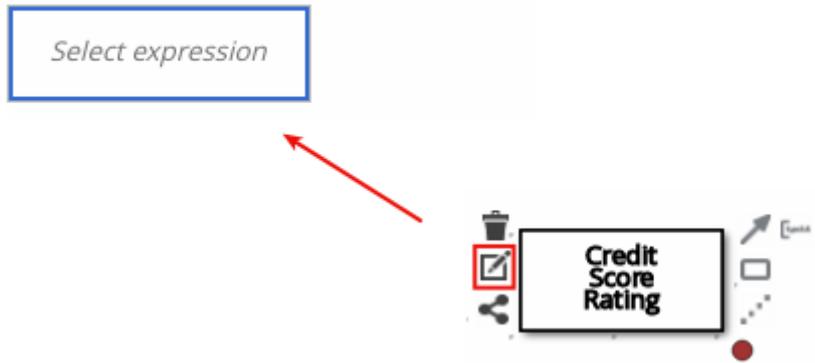
*Figure 74. Opening a new decision node boxed expression*
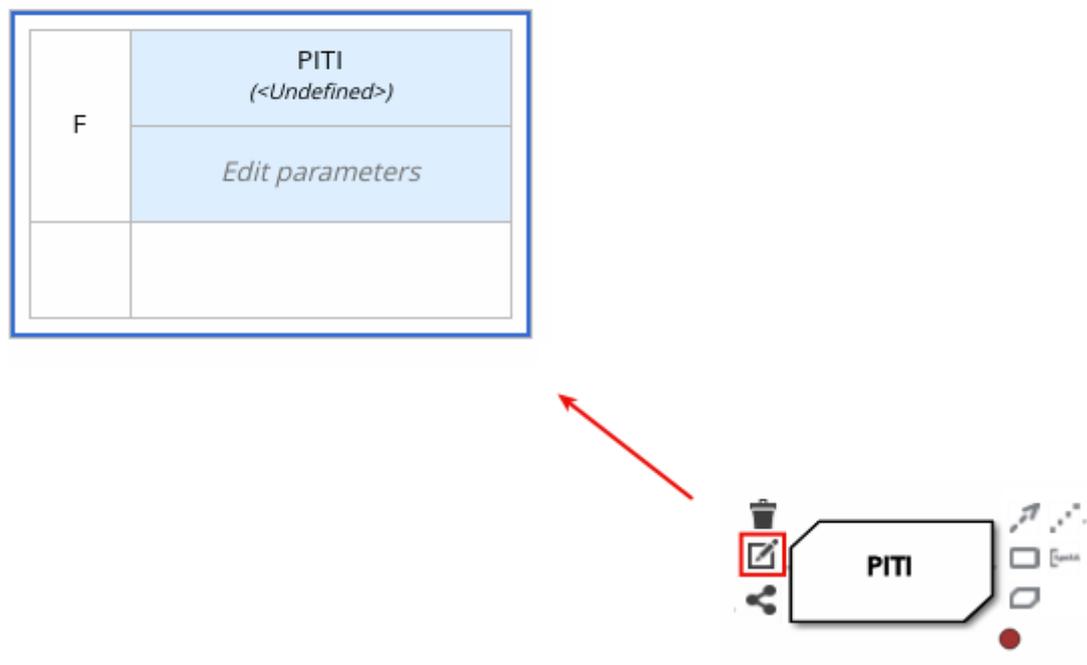
Figure 75. Opening a new business knowledge model boxed expression

By default, all business knowledge models are defined as boxed function expressions containing a literal FEEL expression, a nested context expression of an external JAVA or PMML function, or a nested boxed expression of any type.

For decision nodes, you click the undefined table to select the type of boxed expression you want to use, such as a boxed literal expression, boxed context expression, decision table, or other DMN boxed expression.
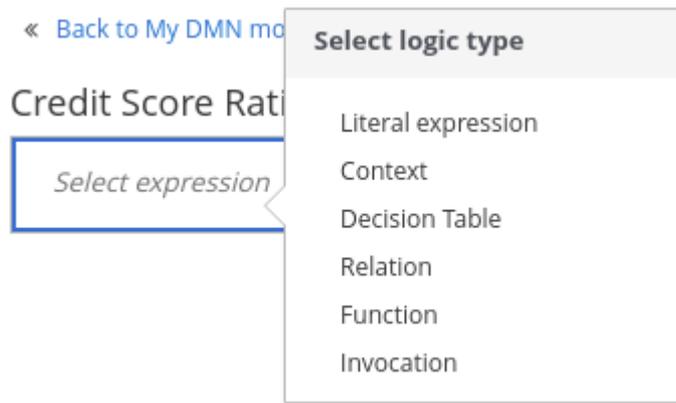
Figure 76. Selecting the logic type for a decision node

For business knowledge models, you click the top-left function cell to select the function type, or right-click the function value cell, select **Clear**, and select a boxed expression of another type.
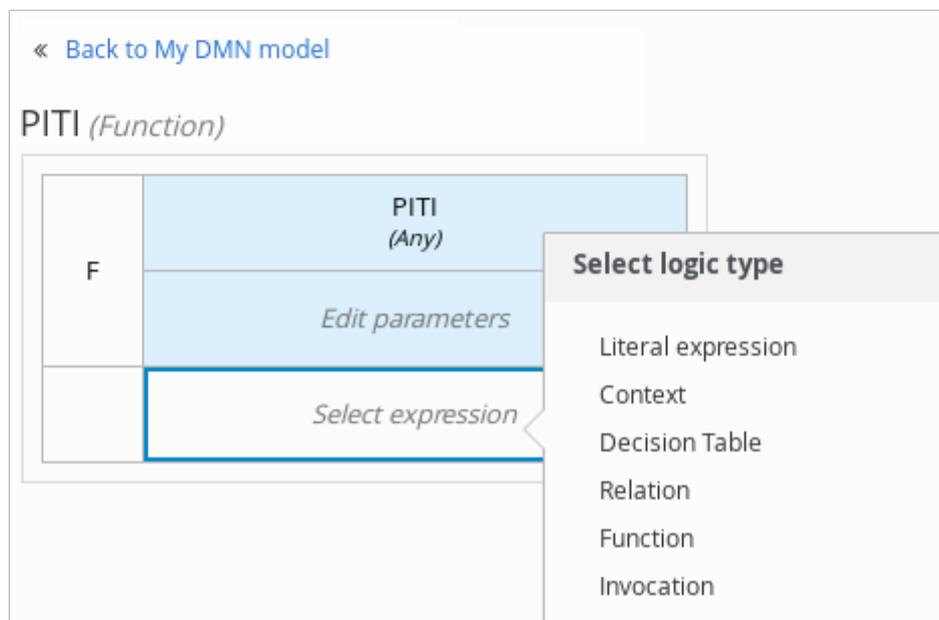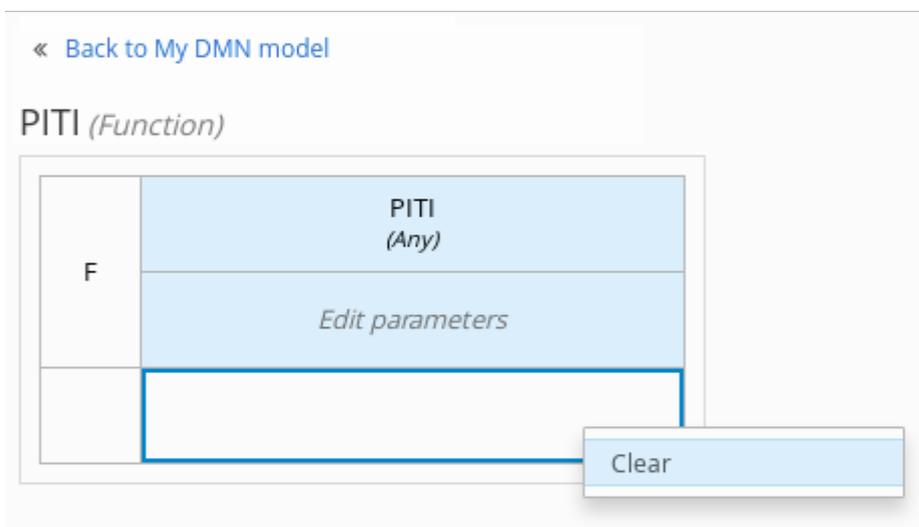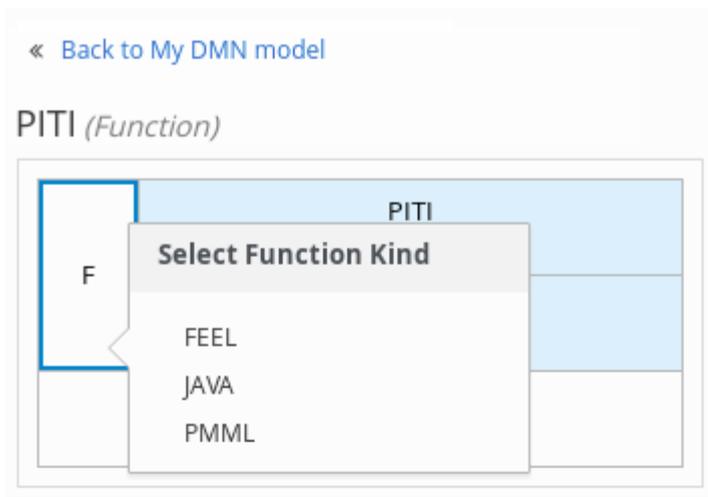
Figure 77. Selecting the function or other logic type for a business knowledge model

5. In the selected boxed expression designer for either a decision node (any expression type) or business knowledge model (function expression), click the applicable table cells to define the table name, variable data types, variable names and values, function parameters and bindings,

or FEEL expressions to include in the decision logic.

You can right-click cells for additional actions where applicable, such as inserting or removing table rows and columns or clearing table contents.

The following is an example decision table for a decision node that determines credit score ratings based on a defined range of a loan applicant's credit score:

Figure 78. Decision node decision table for credit score rating

The following is an example boxed function expression for a business knowledge model that calculates mortgage payments based on principal, interest, taxes, and insurance (PITI) as a literal expression:

Figure 79. Business knowledge model function for PITI calculation

6. After you define the decision logic for the selected node, click **Back to "<MODEL_NAME>"** to return to the DRD view.

7. For the selected DRD node, use the available connection options to create and connect to the next node in the DRD, or click and drag a new node onto the DRD canvas from the left toolbar.

The node type determines which connection options are supported. For example, an **Input data** node can connect to a decision node, knowledge source, or text annotation using the applicable connection type, whereas a **Knowledge source** node can connect to any DRD element. A **Decision** node can connect only to another decision or a text annotation.

The following connection types are available, depending on the node type:

◦ **Information requirement**: Use this connection from an input data node or decision node to another decision node that requires the information.

◦ **Knowledge requirement**: Use this connection from a business knowledge model to a decision node or to another business knowledge model that invokes the decision logic.

◦ **Authority requirement**: Use this connection from an input data node or a decision node to a dependent knowledge source or from a knowledge source to a decision node, business knowledge model, or another knowledge source.

◦ **Association**: Use this connection from an input data node, decision node, business knowledge model, or knowledge source to a text annotation.
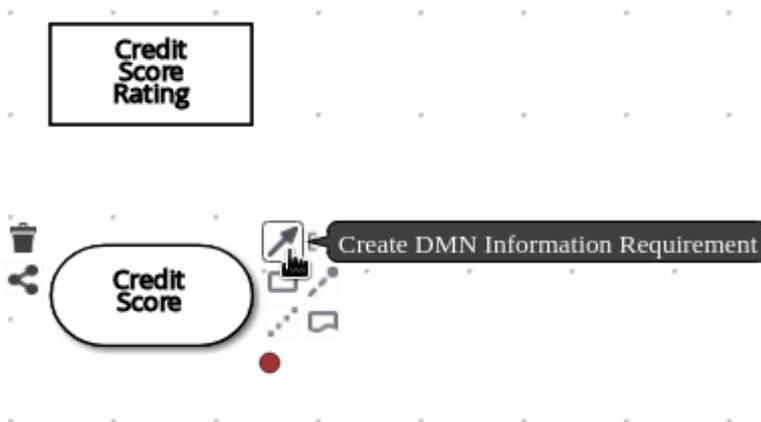
Figure 80. Connecting credit score input to the credit score rating decision
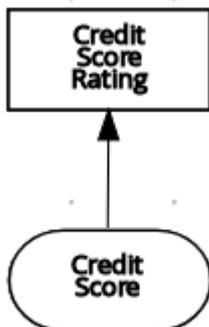
8. Continue adding and defining the remaining DRD components of your decision model. Periodically click **Save** in the DMN designer to save your work.

As you periodically save a DRD, the DMN designer performs a static validation of the DMN model and might produce error messages until the model is defined completely. After you finish defining the DMN model completely, if any errors remain, troubleshoot the specified problems accordingly.

9. After you add and define all components of the DRD, click **Save** to save and validate the completed DRD.

   To adjust the DRD layout, you can select the **Perform automatic layout** icon in the upper-right toolbar of the DMN designer.

   The following is an example DRD for a loan prequalification decision model:
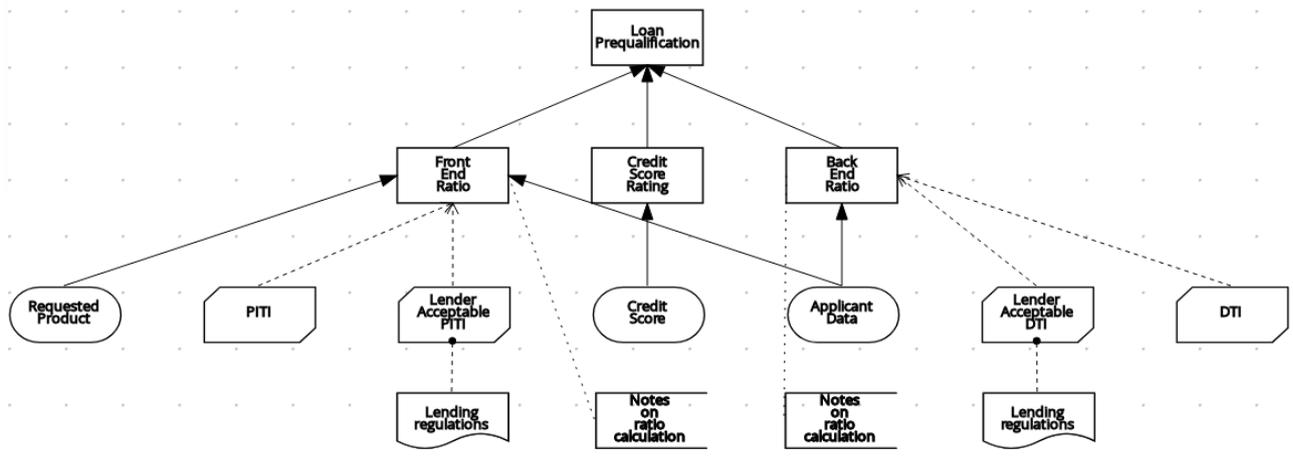
Figure 81. Completed DRD for loan prequalification

   The following is an example DRD for a phone call handling decision model using a reusable decision service:
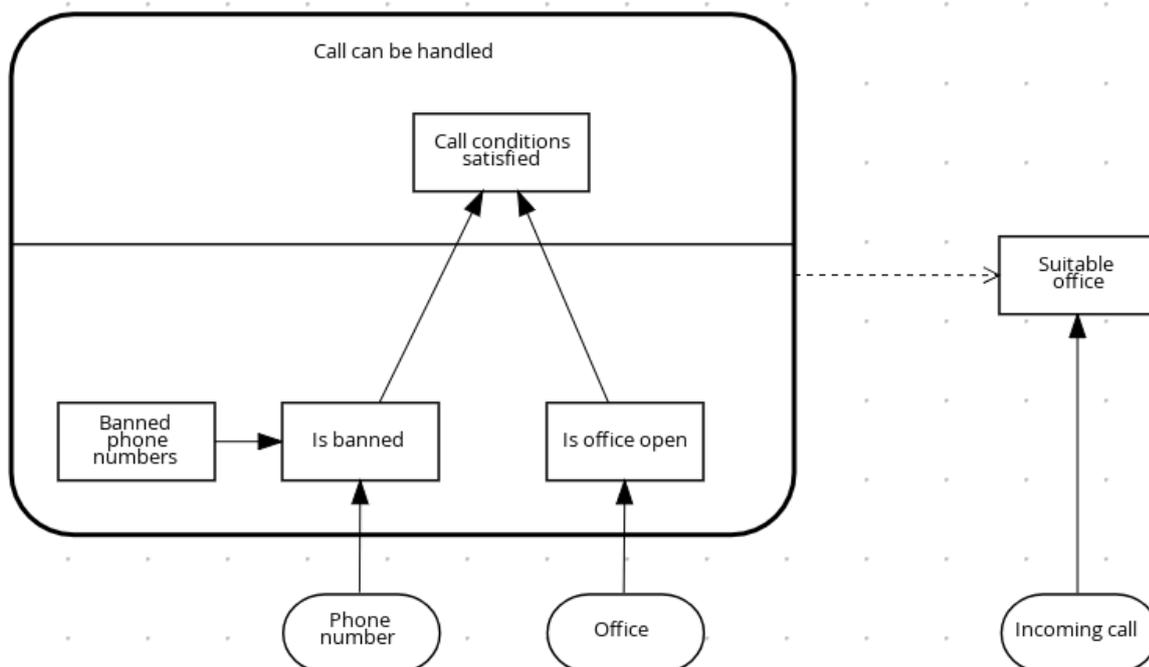
Figure 82. Completed DRD for phone call handling with a decision service

   In a DMN decision service node, the decision nodes in the bottom segment incorporate input

data from outside of the decision service to arrive at a final decision in the top segment of the decision service node. The resulting top-level decisions from the decision service are then implemented in any subsequent decisions or business knowledge requirements of the DMN model. You can reuse DMN decision services in other DMN models to apply the same decision logic with different input data and different outgoing connections.

6.1. Defining DMN decision logic in boxed expressions in KIE DMN Editor

Boxed expressions in DMN are tables that you use to define the underlying logic of decision nodes and business knowledge models in a decision requirements diagram (DRD). Some boxed expressions can contain other boxed expressions, but the top-level boxed expression corresponds to the decision logic of a single DRD artifact. While DRDs represent the flow of a DMN decision model, boxed expressions define the actual decision logic of individual nodes. DRDs and boxed expressions together form a complete and functional DMN decision model.

You can use the KIE DMN Editor to define decision logic for your DRD components using built-in boxed expressions.

Prerequisites

- A DMN file is created or imported in KIE DMN Editor.

Procedure

1. In the DMN designer canvas, select a decision node or business knowledge model node that you want to define and click the **Edit** icon to open the DMN boxed expression designer:
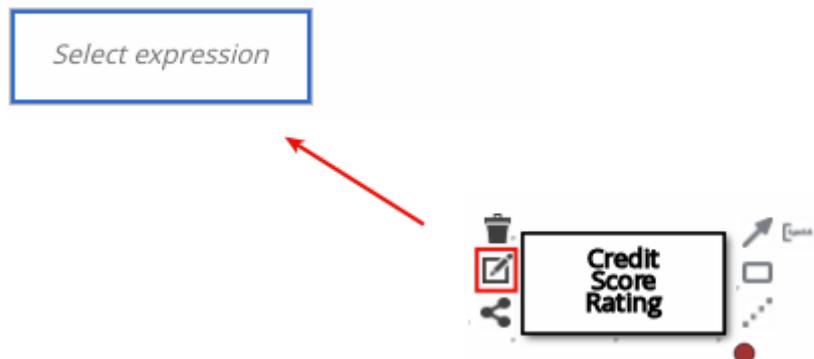
Figure 83. Opening a new decision node boxed expression
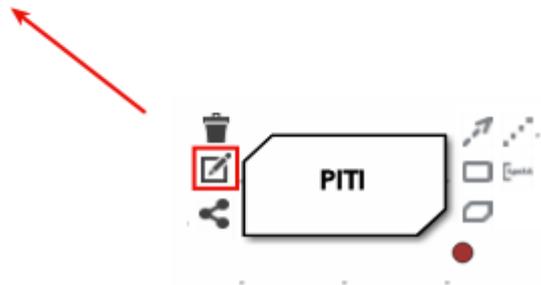
Figure 84. Opening a new business knowledge model boxed expression

By default, all business knowledge models are defined as boxed function expressions containing a literal FEEL expression, a nested context expression of an external JAVA or PMML function, or a nested boxed expression of any type.

For decision nodes, you click the undefined table to select the type of boxed expression you want to use, such as a boxed literal expression, boxed context expression, decision table, or other DMN boxed expression.
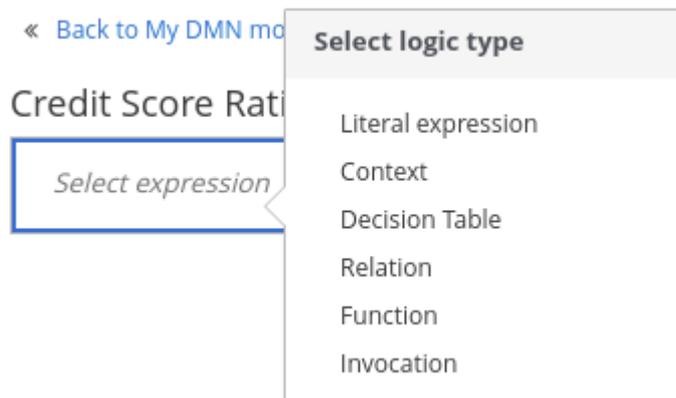
Figure 85. Selecting the logic type for a decision node

For business knowledge model nodes, you click the top-left function cell to select the function type, or right-click the function value cell, select **Clear**, and select a boxed expression of another type.
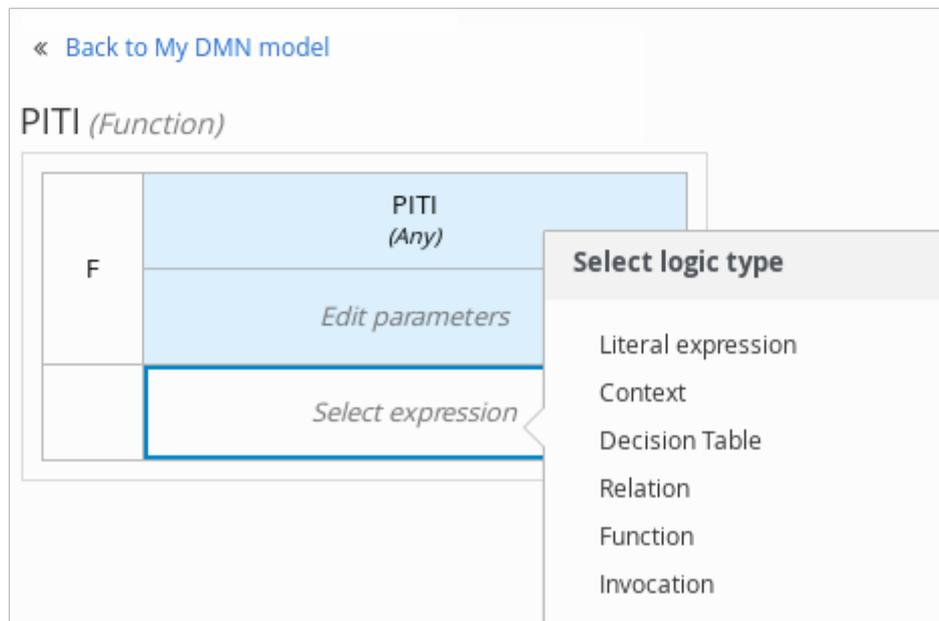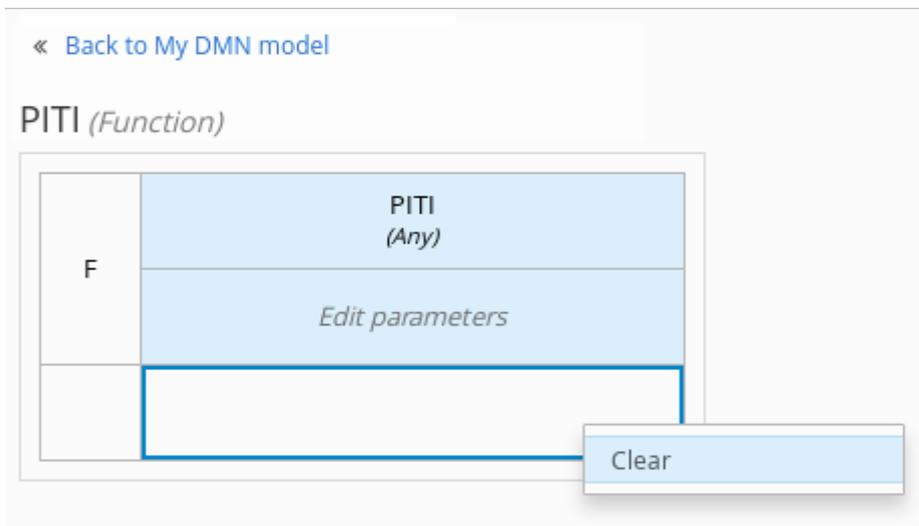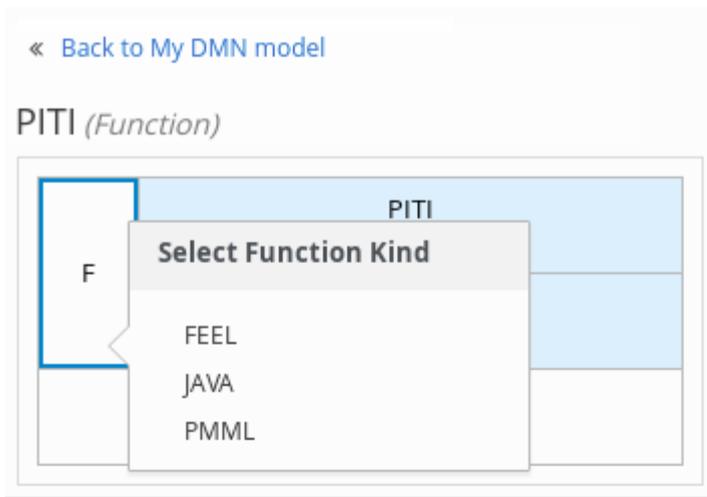
Figure 86. Selecting the function or other logic type for a business knowledge model

2. For this example, use a decision node and select **Decision Table** as the boxed expression type.

   A decision table in DMN is a visual representation of one or more rules in a tabular format.

Each rule consists of a single row in the table, and includes columns that define the conditions (input) and outcome (output) for that particular row.

3. Click the input column header to define the name and data type for the input condition. For example, name the input column **Credit Score.FICO** with a `number` data type. This column specifies numeric credit score values or ranges of loan applicants.

4. Click the output column header to define the name and data type for the output values. For example, name the output column **Credit Score Rating** and next to the **Data Type** option, click **Manage** to go to the **Data Types** page where you can create a custom data type with score ratings as constraints.
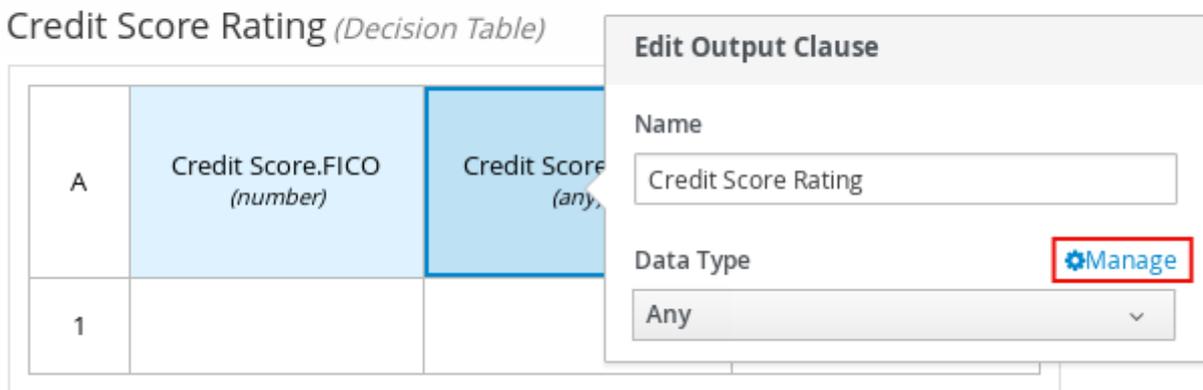
Figure 87. Managing data types for a column header value

5. On the **Data Types** page, click **New Data Type** to add a new data type or click **Import Data Object** to import an existing data object from your project that you want to use as a DMN data type.

If you import a data object from your project as a DMN data type and then that object is updated, you must re-import the data object as a DMN data type to apply the changes in your DMN model.

For this example, click **New Data Type** and create a **Credit_Score_Rating** data type as a `string`:
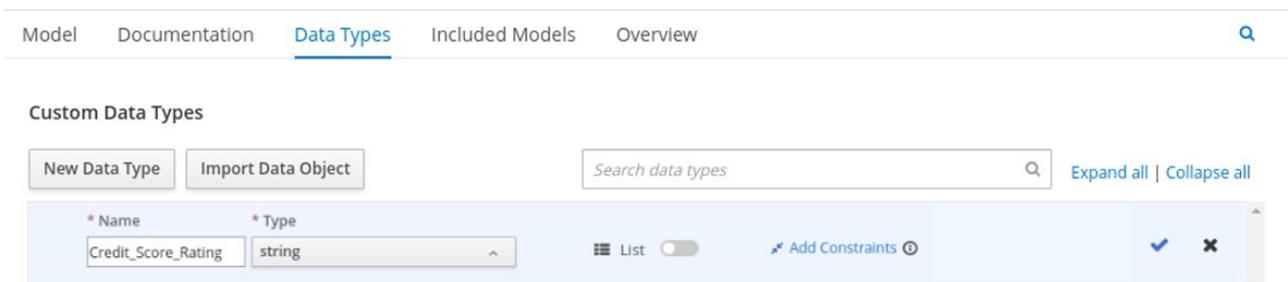
Figure 88. Adding a new data type

6. Click **Add Constraints**, select **Enumeration** from the drop-down options, and add the following constraints:

   ◦ `"Excellent"`

- ◦ `"Good"`

- ◦ `"Fair"`

- ◦ `"Poor"`

- ◦ `"Bad"`
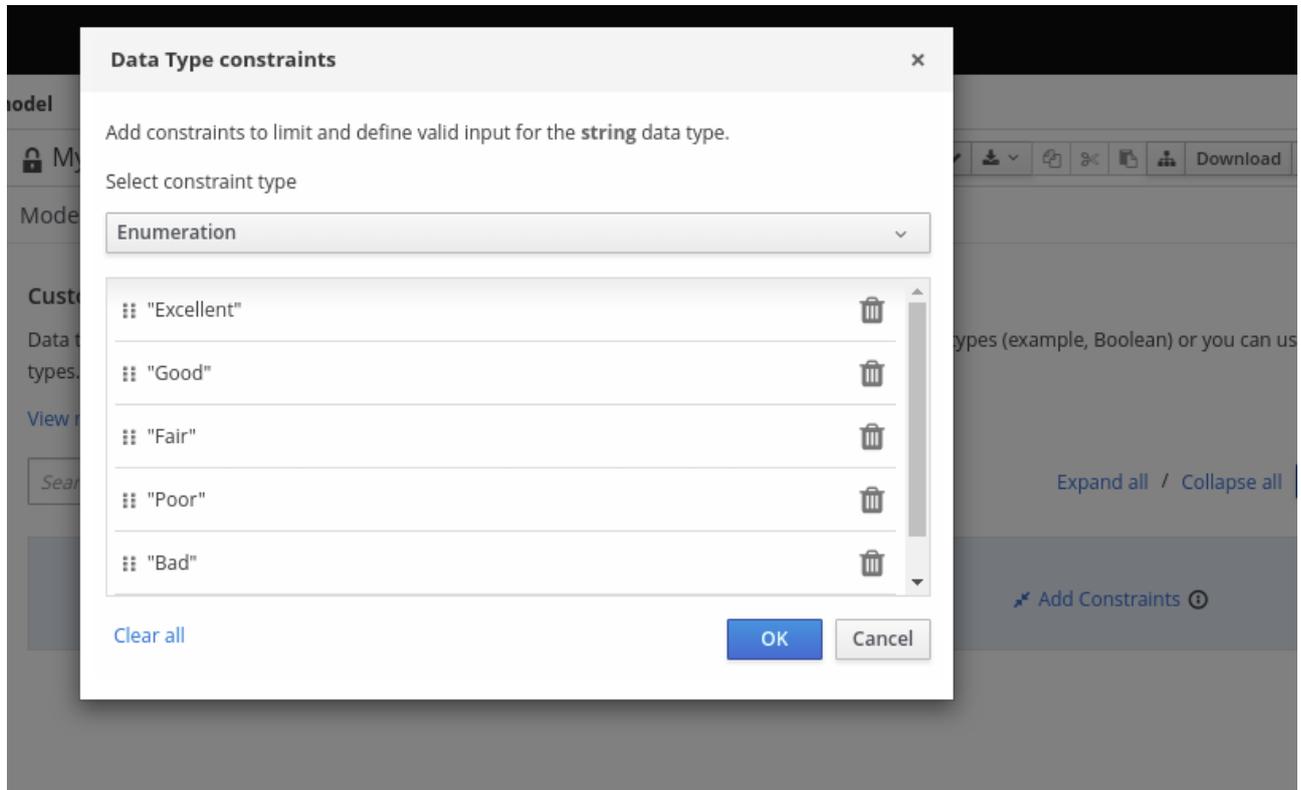
Figure 89. Adding constraints to the new data type

To change the order of data type constraints, you can click the left end of the constraint row and drag the row as needed:
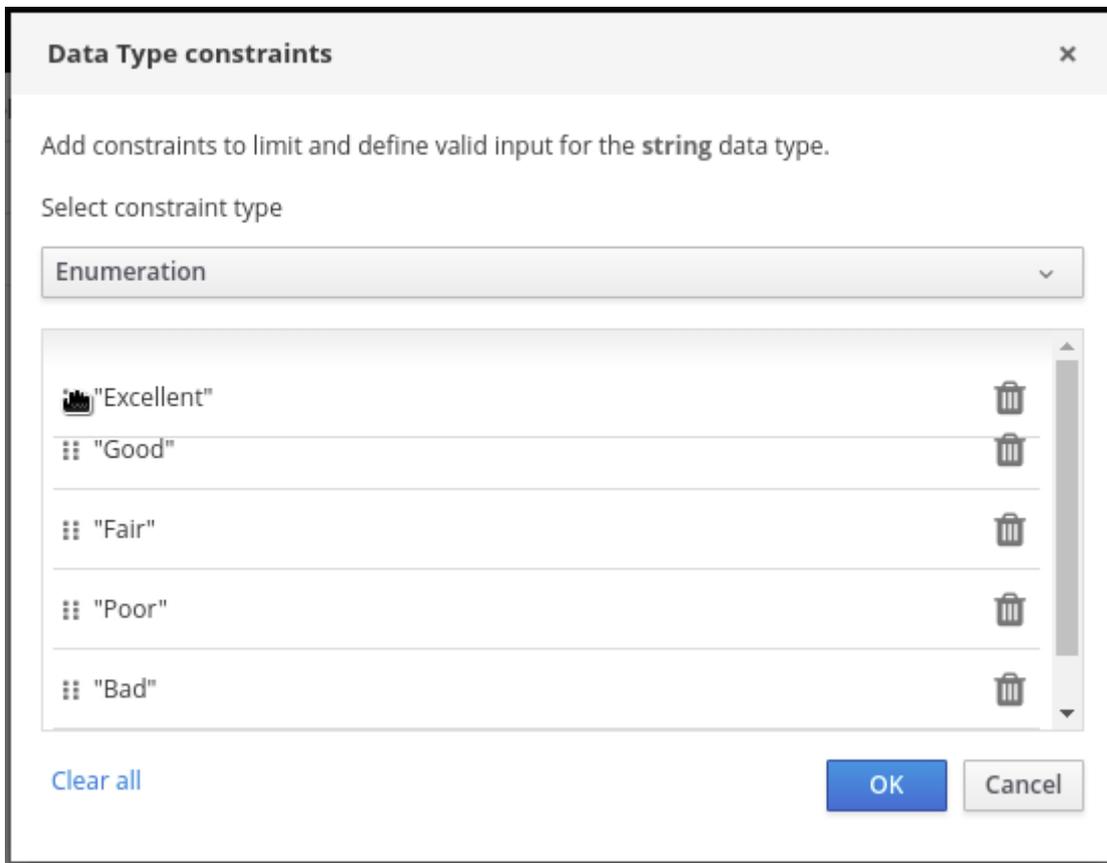
Figure 90. Dragging constraints to change constraint order

For information about constraint types and syntax requirements for the specified data type, see the Decision Model and Notation specification.

7. Click **OK** to save the constraints and click the check mark to the right of the data type to save the data type.

8. Return to the **Credit Score Rating** decision table, click the **Credit Score Rating** column header, and set the data type to this new custom data type.

9. Use the **Credit Score.FICO** input column to define credit score values or ranges of values, and use the **Credit Score Rating** column to specify one of the corresponding ratings you defined in the **Credit_Score_Rating** data type.

   Right-click any value cell to insert or delete rows (rules) or columns (clauses).

Credit Score Rating *(Decision Table)*

U	Credit Score.FICO (number)	Credit Score Rating (Credit_Score_Rating)	Description
1	>= 750	"Excellent"	
2	[700..750)	"Good"	
3	[650..700)	"Fair"	
4	[600..650)	"Poor"	
5	< 600	"Bad"	

Figure 91. Decision node decision table for credit score rating

10. After you define all rules, click the top-left corner of the decision table to define the rule **Hit Policy** and **Builtin Aggregator** (for **COLLECT** hit policy only).

   The hit policy determines how to reach an outcome when multiple rules in a decision table match the provided input values. The built-in aggregator determines how to aggregate rule values when you use the **COLLECT** hit policy.

Credit Score Rating *(Decision Table)*

Figure 92. Defining the decision table hit policy

The following example is a more complex decision table that determines applicant qualification for a loan as the concluding decision node in the same loan prequalification decision model:

Figure 93. Decision table for loan prequalification

For boxed expression types other than decision tables, you follow these guidelines similarly to navigate the boxed expression tables and define variables and parameters for decision logic, but according to the requirements of the boxed expression type. Some boxed expressions, such as boxed literal expressions, can be single-column tables, while other boxed expressions, such as function, context, and invocation expressions, can be multi-column tables with nested boxed expressions of other types.

For example, the following boxed context expression defines the parameters that determine

whether a loan applicant can meet minimum mortgage payments based on principal, interest, taxes, and insurance (PITI), represented as a front-end ratio calculation with a sub-context expression:

Front End Ratio *(Context)*

#				
		Front End Ratio *(Front_End_Ratio)*		
1	Client PITI *(number)*	#		PITI
		1	pmt *(number)*	(Requested Product.Amount*((Requested Product.Rate/100)/12))/ (1-(1/(1+(Requested Product.Rate/100)/12)**-Requested Product.Term))
		2	tax *(number)*	Applicant Data.Monthly.Tax
		3	insurance *(number)*	Applicant Data.Monthly.Insurance
		4	income *(number)*	Applicant Data.Monthly.Income
	<result>			if Client PITI <= Lender Acceptable PITI() then "Sufficent" else "Insufficient"

Figure 94. Boxed context expression for front-end client PITI ratio

The following boxed function expression determines a monthly mortgage installment as a business knowledge model in a lending decision, with the function value defined as a nested context expression:

InstallmentCalculation *(Function)*

F			
		InstallmentCalculation *(number)*	
		(ProductType, Rate, Term, Amount)	
	1	MonthlyFee *(number)*	if ProductType ="STANDARD LOAN" then 20.00 else if ProductType ="SPECIAL LOAN" then 25.00 else null
	2	MonthlyRepayment *(number)*	(Amount *Rate/12) / (1 - (1 + Rate/12)**-Term)
		<result>	MonthlyRepayment+MonthlyFee

Figure 95. Boxed function expression for installment calculation in business knowledge model

For more information and examples of each boxed expression type, see DMN decision logic in boxed expressions.

6.2. Creating custom data types for DMN boxed expressions in KIE DMN Editor

In DMN boxed expressions in KIE DMN Editor, data types determine the structure of the data that

you use within an associated table, column, or field in the boxed expression. You can use default DMN data types (such as String, Number, Boolean) or you can create custom data types to specify additional fields and constraints that you want to implement for the boxed expression values.

Custom data types that you create for a boxed expression can be simple or structured:

- **Simple** data types have only a name and a type assignment. Example: `Age (number)`.
- **Structured** data types contain multiple fields associated with a parent data type. Example: A single type `Person` containing the fields `Name (string)`, `Age (number)`, `Email (string)`.

Prerequisites

- A DMN file is created or imported in KIE DMN Editor.

Procedure

1. In the DMN designer canvas, select a decision node or business knowledge model for which you want to define the data types and click the **Edit** icon to open the DMN boxed expression designer.

2. If the boxed expression is for a decision node that is not yet defined, click the undefined table to select the type of boxed expression you want to use, such as a boxed literal expression, boxed context expression, decision table, or other DMN boxed expression.
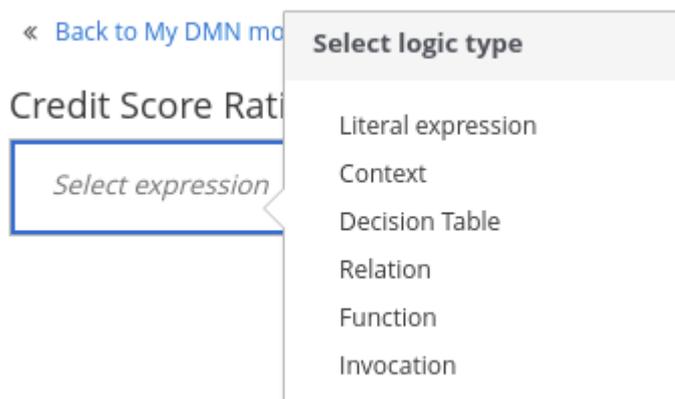
Figure 96. Selecting the logic type for a decision node

3. Click the cell for the table header, column header, or parameter field (depending on the boxed expression type) for which you want to define the data type and click **Manage** to go to the **Data Types** page where you can create a custom data type.

Figure 97. Managing data types for a column header value

You can also set and manage custom data types for a specified decision node or business knowledge model node by selecting the **Properties** icon in the upper-right corner of the DMN designer:

Figure 98. Managing data types in decision requirements diagram (DRD) properties

The data type that you define for a specified cell in a boxed expression determines the structure of the data that you use within that associated table, column, or field in the boxed expression.

In this example, an output column **Credit Score Rating** for a DMN decision table defines a set of custom credit score ratings based on an applicant's credit score.

4. On the **Data Types** page, click **New Data Type** to add a new data type or click **Import Data Object** to import an existing data object from your project that you want to use as a DMN data type.

If you import a data object from your project as a DMN data type and then that object is updated, you must re-import the data object as a DMN data type to apply the changes in your DMN model.

For this example, click **New Data Type** and create a **Credit_Score_Rating** data type as a `string`:
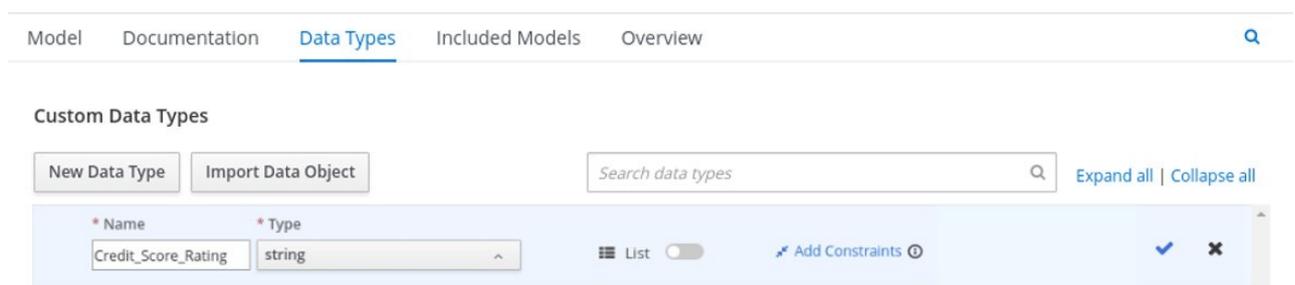
Figure 99. Adding a new data type

If the data type requires a list of items, enable the **List** setting.

5. Click **Add Constraints**, select **Enumeration** from the drop-down options, and add the following constraints:

   ◦ `"Excellent"`

   ◦ `"Good"`

   ◦ `"Fair"`

   ◦ `"Poor"`

   ◦ `"Bad"`
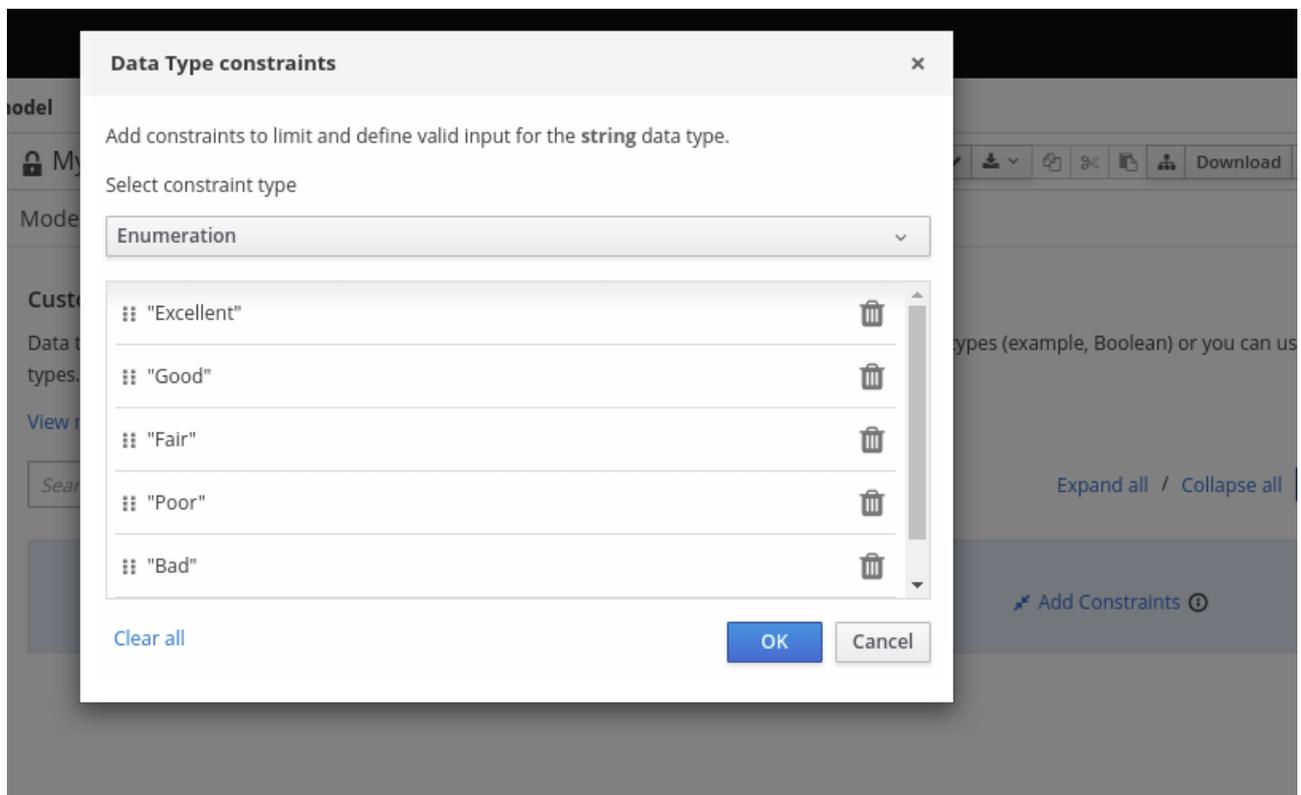
Figure 100. Adding constraints to the new data type

To change the order of data type constraints, you can click the left end of the constraint row and drag the row as needed:
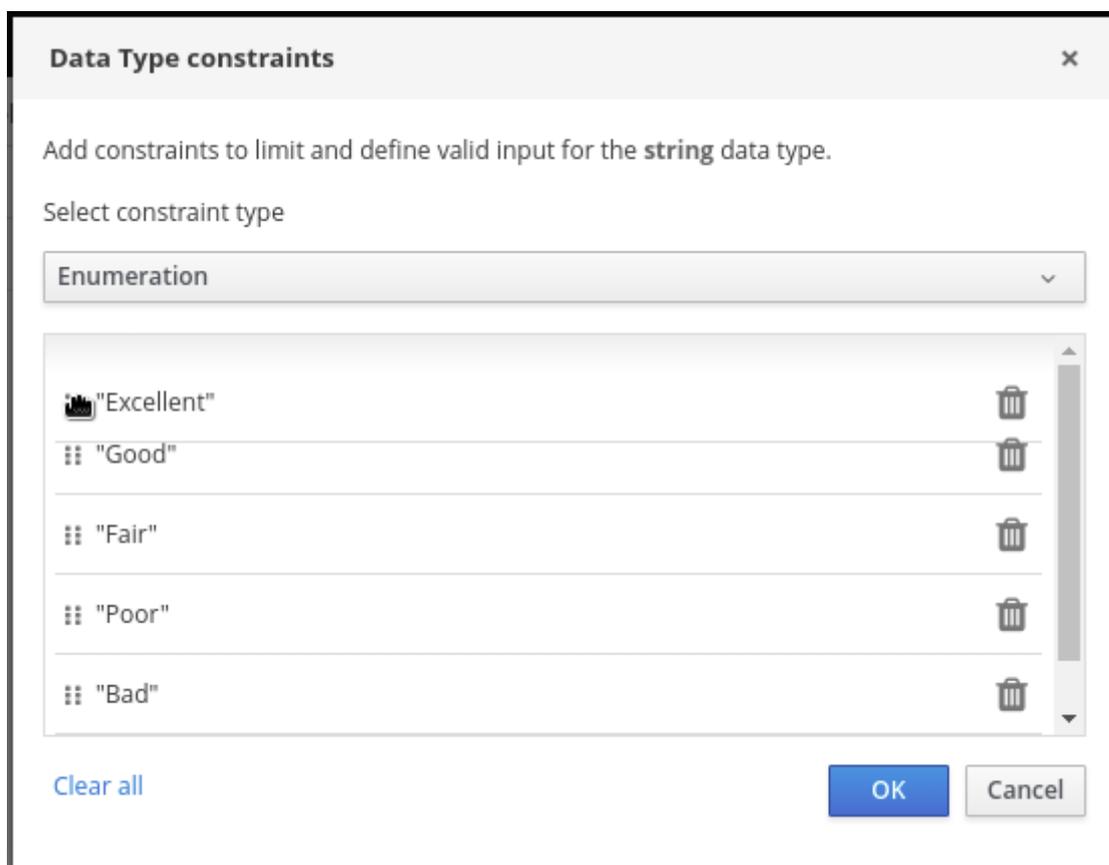
Figure 101. Dragging constraints to change constraint order

For information about constraint types and syntax requirements for the specified data type, see the Decision Model and Notation specification.

6. Click **OK** to save the constraints and click the check mark to the right of the data type to save the data type.

7. Return to the **Credit Score Rating** decision table, click the **Credit Score Rating** column header, set the data type to this new custom data type, and define the rule values for that column with the rating constraints that you specified.

« Back to Loan Pre-Qualification

Credit Score Rating *(Decision Table)*

U	Credit Score.FICO *(number)*	Credit Score Rating *(Credit_Score_Rating)*	Description
1	>= 750	"Excellent"	
2	[700..750)	"Good"	
3	[650..700)	"Fair"	
4	[600..650)	"Poor"	
5	< 600	"Bad"	

Figure 102. Decision table for credit score rating

In the DMN decision model for this scenario, the **Credit Score Rating** decision flows into the following **Loan Prequalification** decision that also requires custom data types:

Loan Pre-Qualification *(Decision Table)*

F	Credit Score Rating *(<Undefined>)*	Back End Ratio *(<Undefined>)*	Front End Ratio *(<Undefined>)*	Loan Pre-Qualification *(<Undefined>)*		Description
				Qualification *(string)*	Reason *(string)*	
1						

Figure 103. Decision table for loan prequalification

8. Continuing with this example, return to the **Data Types** window, click **New Data Type**, and create a **Loan_Qualification** data type as a `Structure` with no constraints.

When you save the new structured data type, the first sub-field appears so that you can begin defining nested data fields in this parent data type. You can use these sub-fields in association

with the parent structured data type in boxed expressions, such as nested column headers in decision tables or nested table parameters in context or function expressions.

For additional sub-fields, select the addition icon next to the **Loan_Qualification** data type:
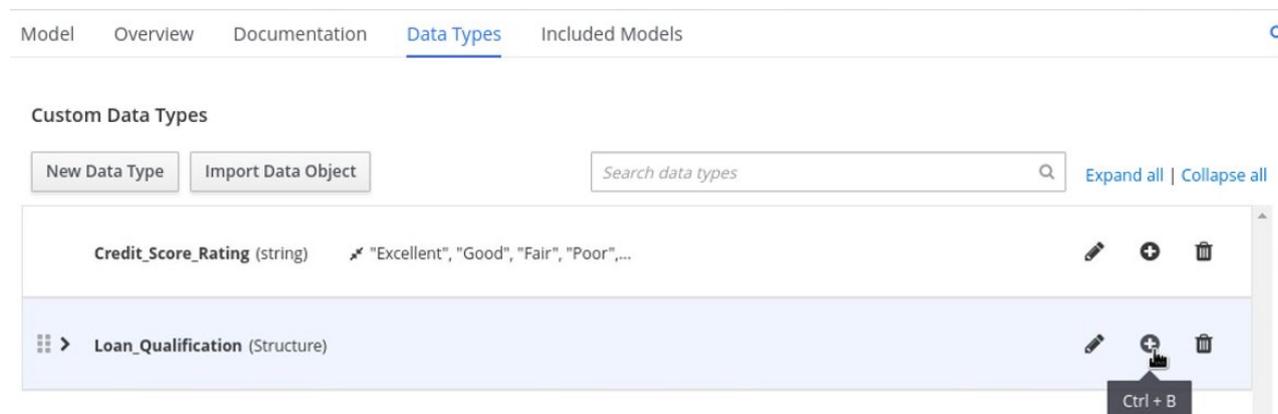
Figure 104. Adding a new structured data type with nested fields

9. For this example, under the structured **Loan_Qualification** data type, add a **Qualification** field with `"Qualified"` and `"Not Qualified"` enumeration constraints, and a **Reason** field with no constraints. Add also a simple **Back_End_Ratio** and a **Front_End_Ratio** data type, both with `"Sufficient"` and `"Insufficient"` enumeration constraints.

Click the check mark to the right of each data type that you create to save your changes.
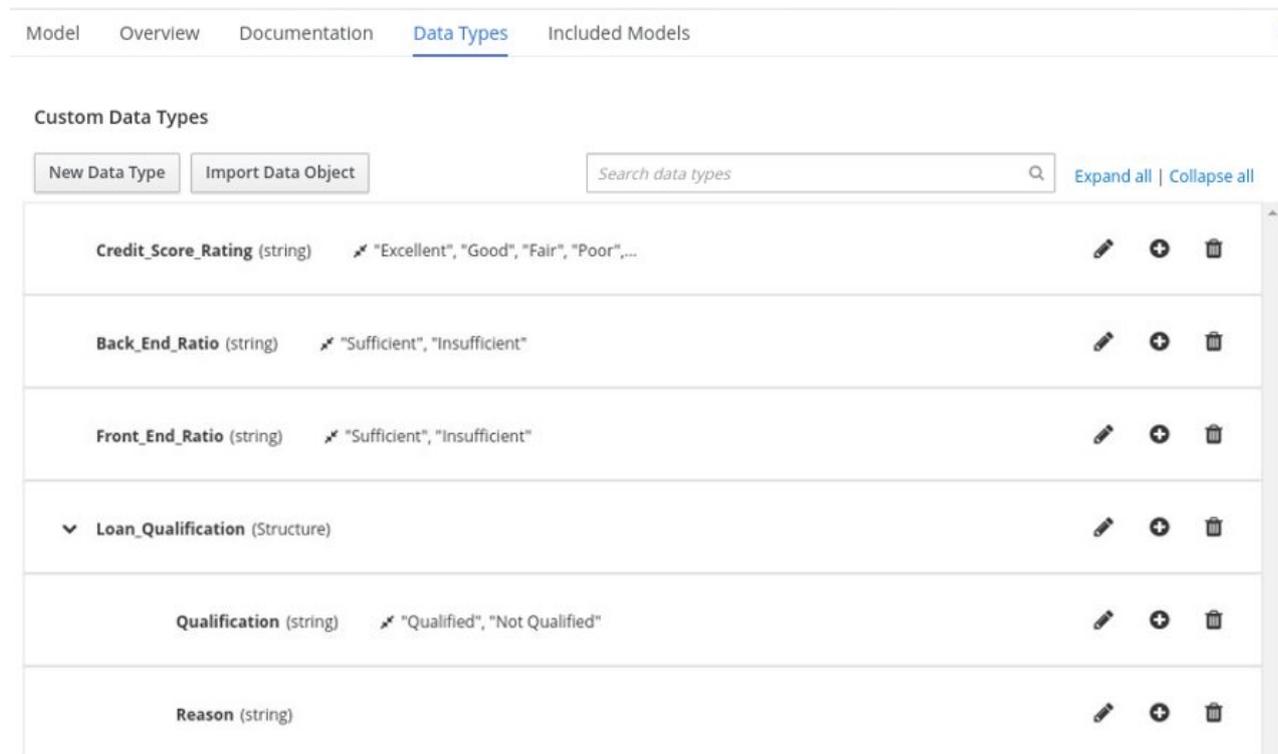
Figure 105. Adding nested data types with constraints

To change the order or nesting of data types, you can click the left end of the data type row and drag the row as needed:

Figure 106. Dragging data types to change data type order or nesting

10. Return to the decision table and, for each column, click the column header cell, set the data type to the new corresponding custom data type, and define the rule values as needed for the column with the constraints that you specified, if applicable.

Loan Pre-Qualification *(Decision Table)*

F	Credit Score Rating (Credit_Score_Rating)	Back End Ratio (Back_End_Ratio)	Front End Ratio (Front_End_Ratio)	Loan Pre-Qualification (Loan_Qualification)		Description
				Qualification (string)	Reason (string)	
1	"Poor", "Bad"	-	-	"Not Qualified"	"Credit Score too low."	
2	-	"Insufficient"	"Sufficient"	"Not Qualified"	"Debt to income ratio is too high."	
3	-	"Sufficient"	"Insufficient"	"Not Qualified"	"Mortgage payment to income ratio is too high."	
4	-	"Insufficient"	"Insufficient"	"Not Qualified"	"Debt to income ratio is too high AND mortgage payment to income ratio is too high."	
5	"Fair", "Good", "Excellent"	"Sufficient"	"Sufficient"	"Qualified"	"The borrower has been successfully prequalified for the requested loan."	

Figure 107. Decision table for loan prequalification

For boxed expression types other than decision tables, you follow these guidelines similarly to navigate the boxed expression tables and define custom data types as needed.

For example, the following boxed function expression uses custom `tCandidate` and `tProfile` structured data types to associate data for online dating compatibility:

Evaluate Match *(Function)*

F	Evaluate Match *(tCandidate)*		
	(Lonely Soul, Candidate)		
	1	Profile1 *(tProfile)*	`Lonely Soul`
	2	Profile2 *(tProfile)*	`Candidate`
	3	Is Match *(boolean)*	`Is Soul a Match(Lonely Soul, Candidate) and Is Soul a Match(Candidate, Lonely Soul)`
	4	Score *(number)*	`Number of Matching Interests(Lonely Soul, Candidate) - absolute( Lonely Soul.Age - Candidate.Age )`
		<result>	*Select expression*

Figure 108. Boxed function expression for online dating compatibility

Figure 109. Custom data type definitions for online dating compatibility
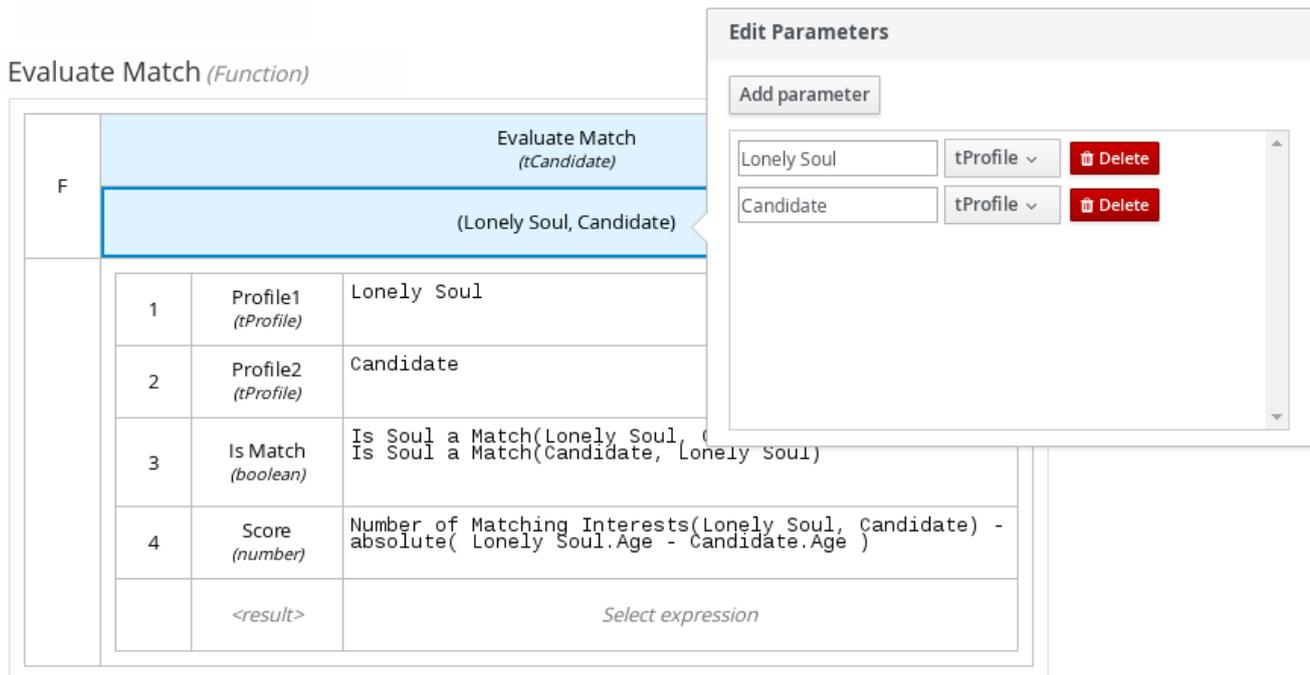
Figure 110. Parameter definitions with custom data types for online dating compatibility

6.3. Included models in DMN files in KIE DMN Editor

In the KIE DMN Editor, you can use the **Included Models** tab to include other DMN models and Predictive Model Markup Language (PMML) models from your project in a specified DMN file. When you include a DMN model within another DMN file, you can use all of the nodes and logic from both models in the same decision requirements diagram (DRD). When you include a PMML model within a DMN file, you can invoke that PMML model as a boxed function expression for a DMN decision node or business knowledge model node.

You cannot include DMN or PMML models from other projects in KIE DMN Editor.

6.3.1. Including other DMN models within a DMN file in KIE DMN Editor

In KIE DMN Editor, you can include other DMN models from your project in a specified DMN file. When you include a DMN model within another DMN file, you can use all of the nodes and logic from both models in the same decision requirements diagram (DRD), but you cannot edit the nodes from the included model. To edit nodes from included models, you must update the source file for the included model directly. If you update the source file for an included DMN model, open the DMN file where the DMN model is included (or close an re-open) to verify the changes.

You cannot include DMN models from other projects in KIE DMN Editor.

Prerequisites

- The DMN models are created or imported (as `.dmn` files) in the same project in KIE DMN Editor as the DMN file in which you want to include the models.

Procedure

1. In the KIE DMN Editor, click the **Included Models** tab.

2. Click **Include Model**, select a DMN model from your project in the **Models** list, enter a unique

name for the included model, and click **Include**:
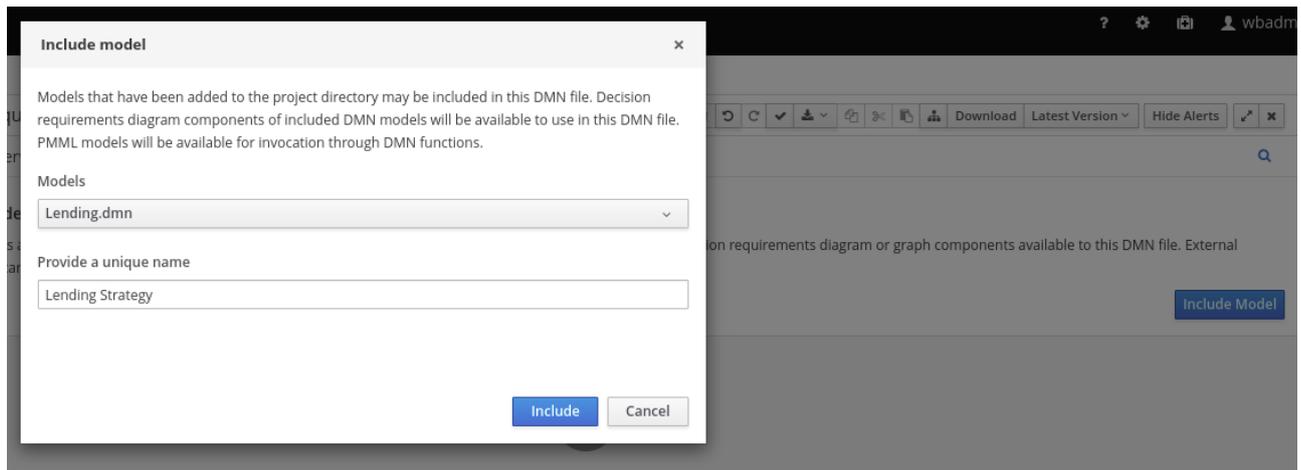
Figure 111. Including a DMN model

The DMN model is added to this DMN file, and all DRD nodes from the included model are listed under **Decision Components** in the **Decision Navigator** view:
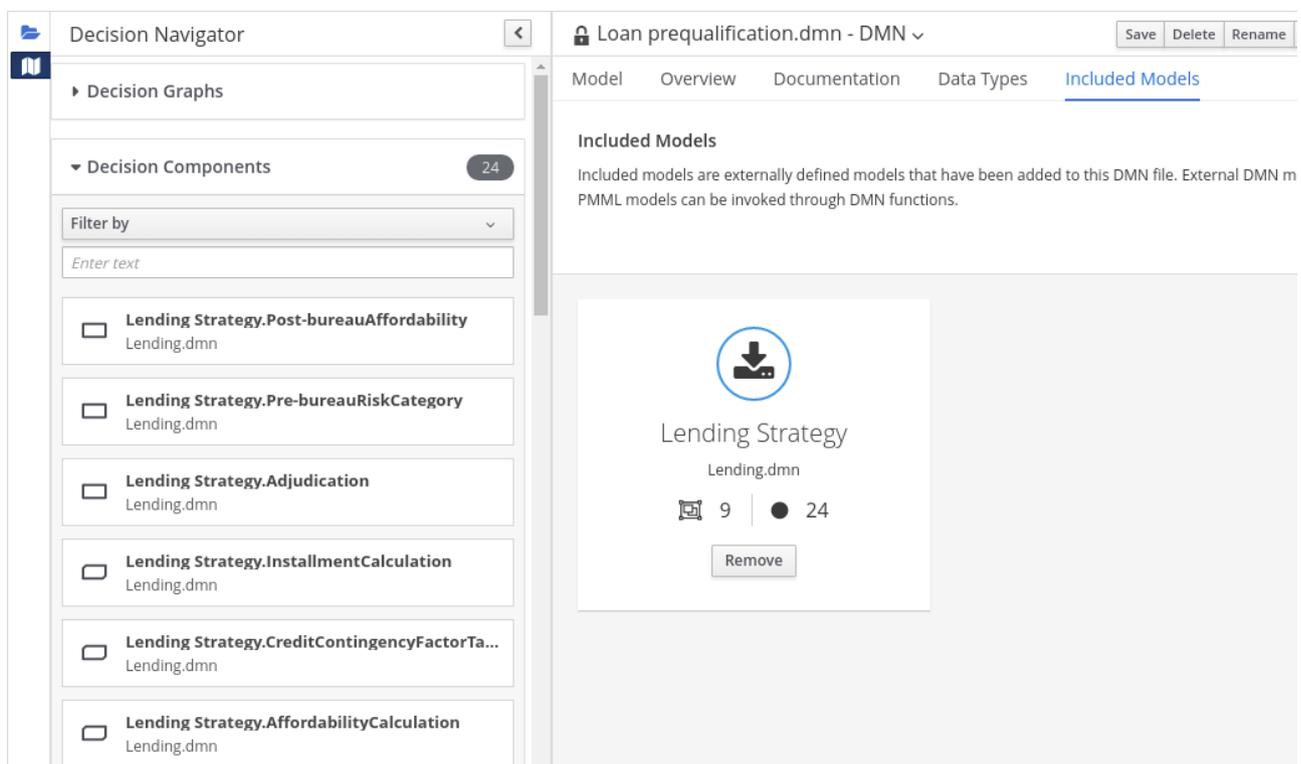
Figure 112. DMN file with decision components from the included DMN model

All data types from the included model are also listed in read-only mode in the **Data Types** tab for the DMN file:
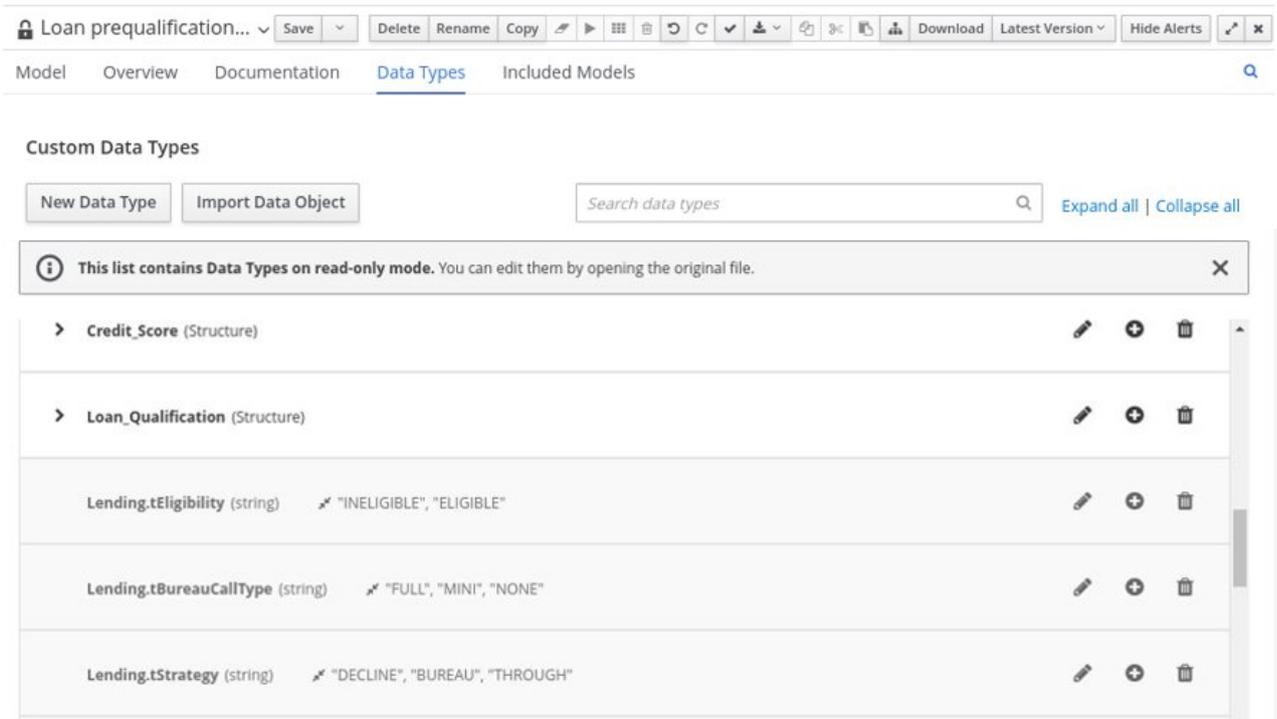
Figure 113. DMN file with data types from the included DMN model

3. In the **Model** tab of the DMN designer, click and drag the included DRD components onto the canvas to begin implementing them in your DRD:
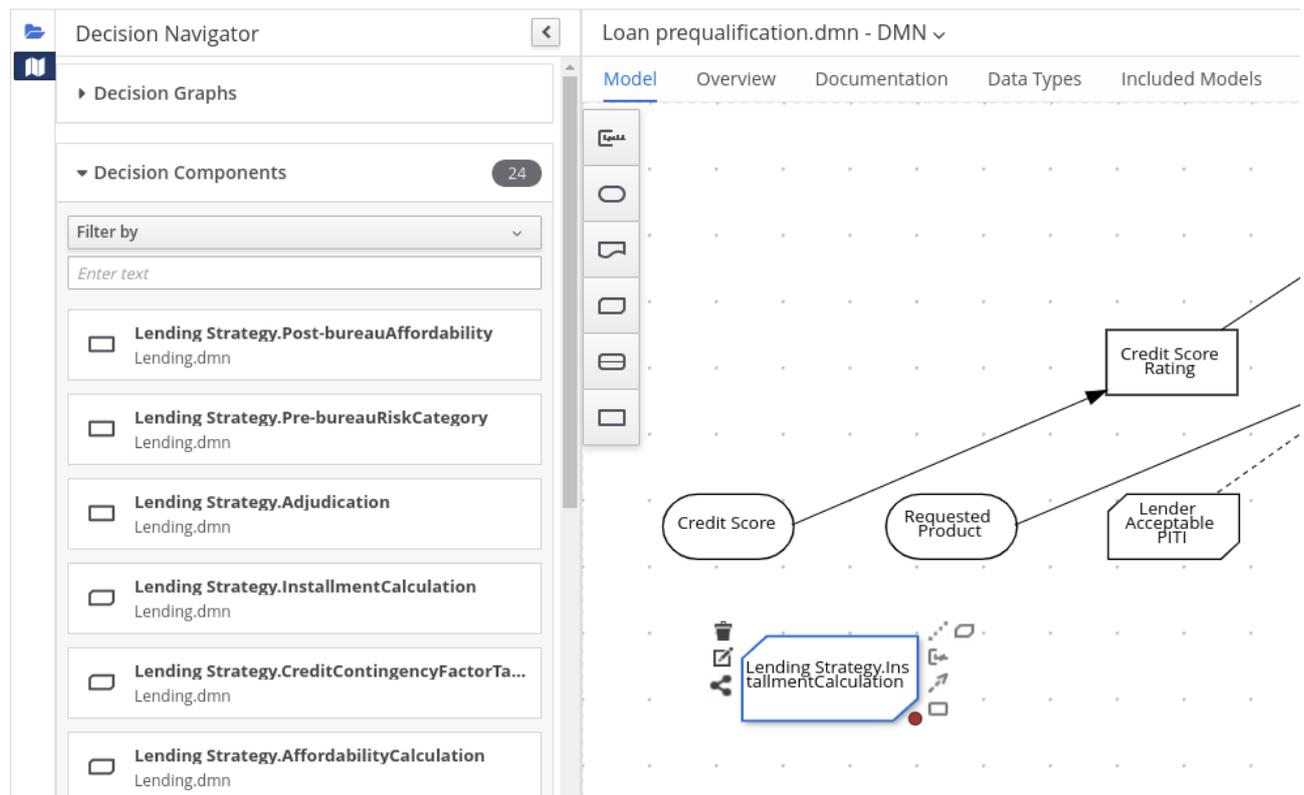
Figure 114. Adding DRD components from the included DMN model

To edit DRD nodes or data types from included models, you must update the source file for the included model directly. If you update the source file for an included DMN model, open the DMN file where the DMN model is included (or close an re-open) to verify the changes.

To edit the included model name or to remove the included model from the DMN file, use the

Included Models tab in the DMN designer.

> ❗ When you remove an included model, any nodes from that included model that are currently used in the DRD are also removed.

6.3.2. Including PMML models within a DMN file in KIE DMN Editor

In KIE DMN Editor, you can include Predictive Model Markup Language (PMML) models from your project in a specified DMN file. When you include a PMML model within a DMN file, you can invoke that PMML model as a boxed function expression for a DMN decision node or business knowledge model node. If you update the source file for an included PMML model, you must remove and re-include the PMML model in the DMN file to apply the source changes.

You cannot include PMML models from other projects in KIE DMN Editor.

Prerequisites

- The PMML models are imported (as `.pmml` files) in the same project in KIE DMN Editor as the DMN file in which you want to include the models.

Procedure

1. In your DMN project, add the following dependencies to the project `pom.xml` file to enable PMML evaluation:

```xml
<!-- Required for the PMML compiler -->
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>kie-pmml-trusty</artifactId>
  <version>${drools.version}</version>
  <scope>provided</scope>
</dependency>

<!-- Alternative dependencies for JPMML Evaluator, override `kie-pmml-trusty`
dependency -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-dmn-jpmml</artifactId>
  <version>${drools.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.jpmml</groupId>
  <artifactId>pmml-evaluator</artifactId>
  <version>1.5.1</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.jpmml</groupId>
  <artifactId>pmml-evaluator-extension</artifactId>
  <version>1.5.1</version>
  <scope>provided</scope>
</dependency>
```

If you want to use the full PMML specification implementation with the Java Evaluator API for PMML (JPMML), use the alternative set of JPMML dependencies in your DMN project. If the JPMML dependencies and the standard `kie-pmml-trusty` dependency are both present, the `kie-pmml-trusty` dependency is disabled. For information about JPMML licensing terms, see Openscoring.io.

> The legacy `kie-pmml` dependency is deprecated with Drools 7.48.0 and was be replaced by `kie-pmml-trusty` dependency in the following Drools release.

> Instead of specifying a Drools `<version>` for individual dependencies, consider adding the Drools bill of materials (BOM) dependency to `dependencyManagement` section of your project `pom.xml` file. 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>org.kie</groupId>
>   <artifactId>kie-dmn-bom</artifactId>
>   <version>${drools.version}</version>
>   <scope>import</scope>
>   <type>pom</type>
> </dependency>
> ```

2. If you added the JPMML dependencies in your DMN project to use the JPMML Evaluator, download the following JAR files and add them to the classpath (in traditional KIE Server deployment these were needed in the `~/kie-server.war/WEB-INF/lib` and `~/business-central.war/WEB-INF/lib` directories of the distribution):

   - KIE JPMML Integration 7.59.0.Final JAR file from the online Maven repository

   - JPMML Evaluator 1.5.1 JAR file from the online Maven repository

   - JPMML Evaluator Extensions 1.5.1 JAR file from the online Maven repository

   These artifacts are required to enable JPMML evaluation.

> Red Hat supports integration with the Java Evaluator API for PMML (JPMML) for PMML execution in Drools. However, Red Hat does not support the JPMML libraries directly. If **you** include JPMML libraries in your Drools distribution, see the Openscoring.io licensing terms for JPMML.

3. In the KIE DMN Editor, click the **Included Models** tab.

4. Click **Include Model**, select a PMML model from your project in the **Models** list, enter a unique name for the included model, and click **Include**:
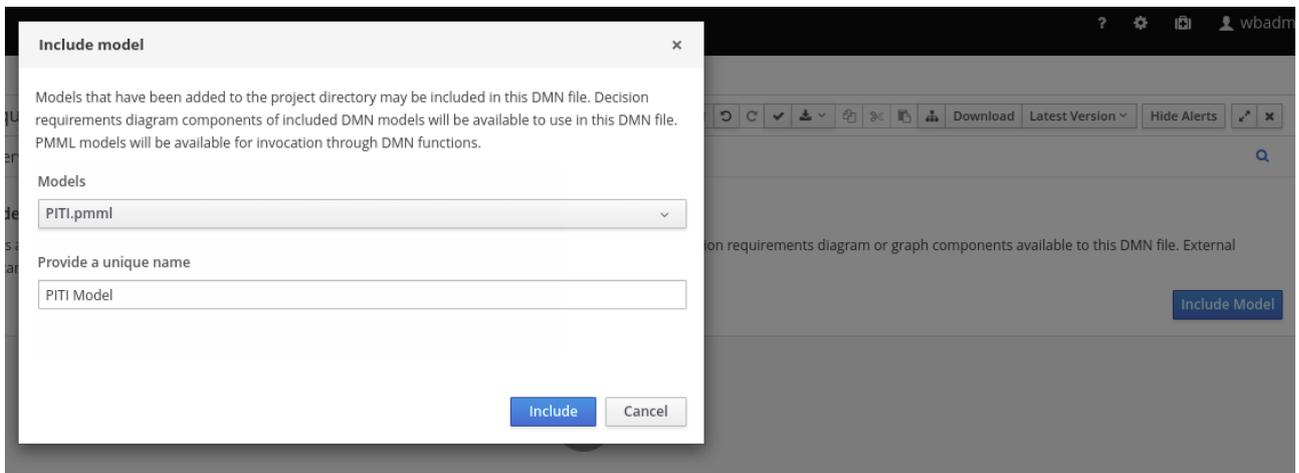
Figure 115. Including a PMML model

The PMML model is added to this DMN file:
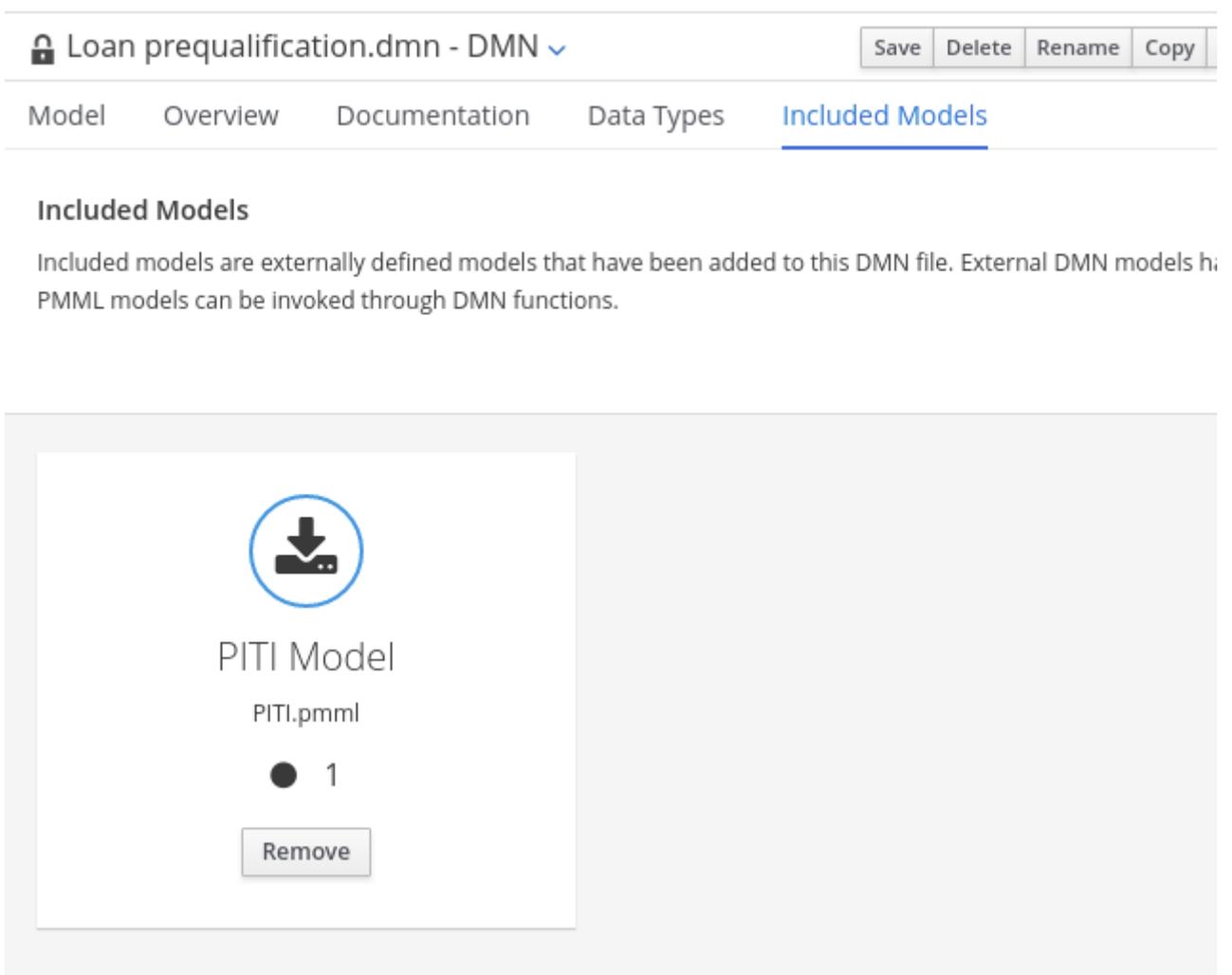
Figure 116. DMN file with included PMML model

5. In the **Model** tab of the DMN designer, select or create the decision node or business knowledge model node in which you want to invoke the PMML model and click the **Edit** icon to open the DMN boxed expression designer:
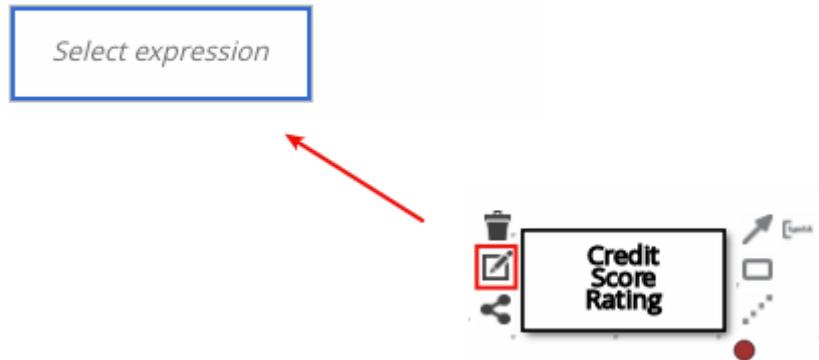
Figure 117. Opening a new decision node boxed expression
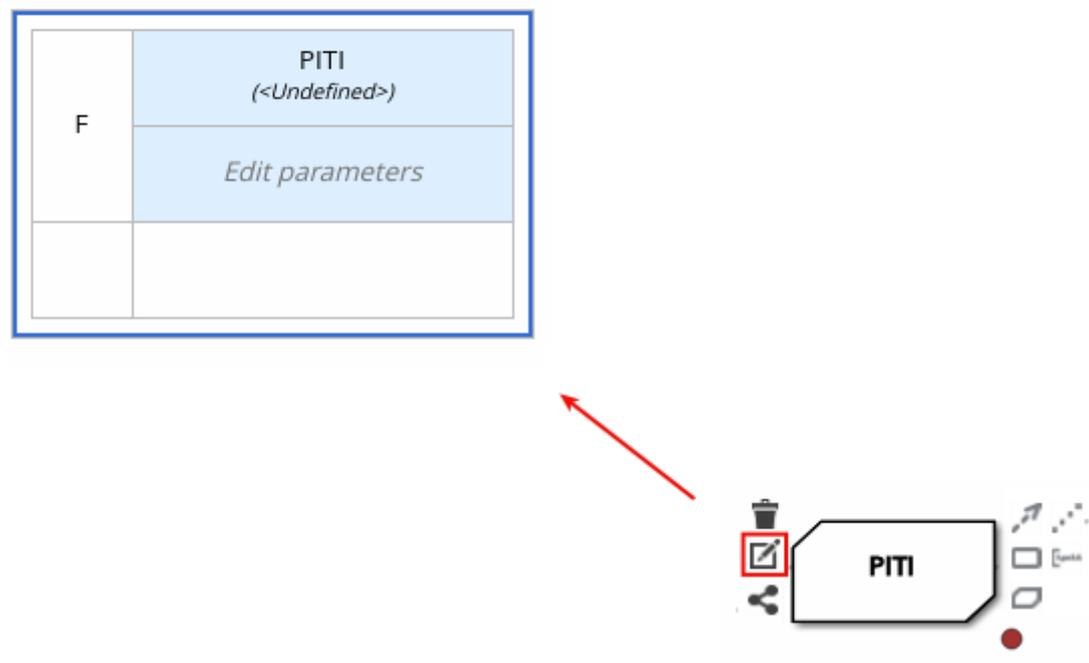
Figure 118. Opening a new business knowledge model boxed expression

6. Set the expression type to **Function** (default for business knowledge model nodes), click the top-left function cell, and select **PMML**.

7. In the **document** and **model** rows in the table, double-click the undefined cells to specify the included PMML document and the relevant PMML model within that document:

PITI *(Function)*

	PITI *(<Undefined>)*	
P	*Edit parameters*	
	1	document *(string)*	PITI Model ▼
	2	model *(string)*	*Second select PMML model*

Figure 119. Adding a PMML model in a DMN business knowledge model

PITI *(Function)*

	PITI *(number)*	
P	(fld1, fld2, fld3)	
	1	document *(string)*	"PITI Model"
	2	model *(string)*	"LinReg"

Figure 120. Example PMML definition in a DMN business knowledge model

If you update the source file for an included PMML model, you must remove and re-include the PMML model in the DMN file to apply the source changes.

To edit the included model name or to remove the included model from the DMN file, use the **Included Models** tab in the DMN designer.

6.4. Creating DMN models with multiple diagrams in KIE DMN Editor

For complex DMN models, you can use the DMN designer in KIE DMN Editor to design multiple DMN decision requirements diagrams (DRDs) that represent parts of the overall decision requirements graph (DRG) for the DMN decision model. In simple cases, you can use a single DRD to represent all of the overall DRG for the decision model, but in complex cases, a single DRD can

become large and difficult to follow. Therefore, to better organize DMN decision models with many decision requirements, you can divide the model into smaller nested DRDs that constitute the larger central DRD representation of the overall DRG.

Prerequisites

- You understand how to design DRDs in KIE DMN Editor. For information about creating DRDs, see Creating and editing DMN models in KIE DMN Editor.

Procedure

1. In KIE DMN Editor, navigate to your DMN project and create or import a DMN file in the project.

2. Open the new or imported DMN file to view the DRD in the DMN designer, and begin designing or modifying the DRD using the DMN nodes in the left toolbar.

3. For any DMN nodes that you want to define in a separate nested DRD, select the node, click the **DRD Actions** icon, and select from the available options.
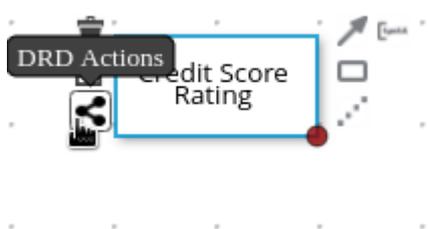
Figure 121. DRD actions icon for subdividing a DRD

The following options are available:

- **Create**: Use this option to create a nested DRD where you can separately define the DMN components and diagram for the selected node.

- **Add to**: If you already created a nested DRD, use this option to add the selected node to an existing DRD.

- **Remove**: If the node that you selected is already within a nested DRD, use this option to remove the node from that nested DRD.

After you create a nested DRD within your DMN decision model, the new DRD opens in a separate DRD canvas and the available DRD and components are listed in the **Decision Navigator** left menu. You can use the **Decision Navigator** menu to rename or remove a nested DRD.
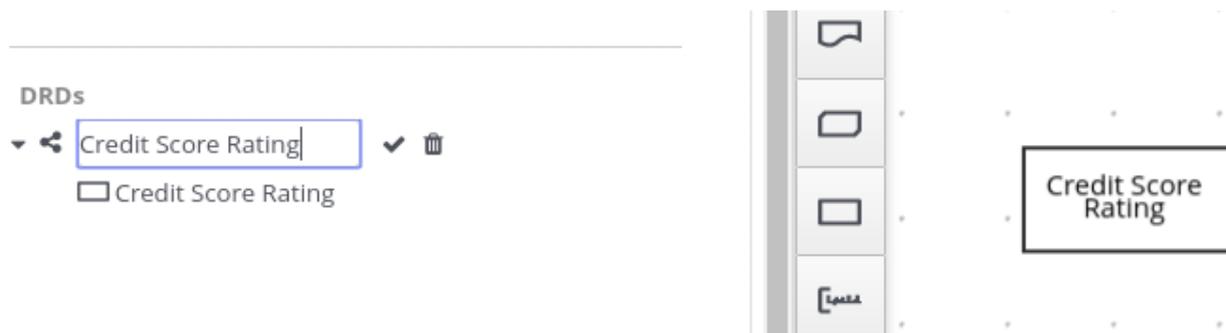
Figure 122. Rename new nested DRD in the Decision Navigator menu

4. In the separate canvas for the new nested DRD, design the flow and logic for all required components in this portion of the DMN model, as usual.

5. Continue adding and defining any other nested DRDs for your decision model and save the completed DMN file.

For example, the following DRD for a loan prequalification decision model contains all DMN components for the model without any nested DRDs. This example relies on the single DRD for all components and logic, resulting in a large and complex diagram.
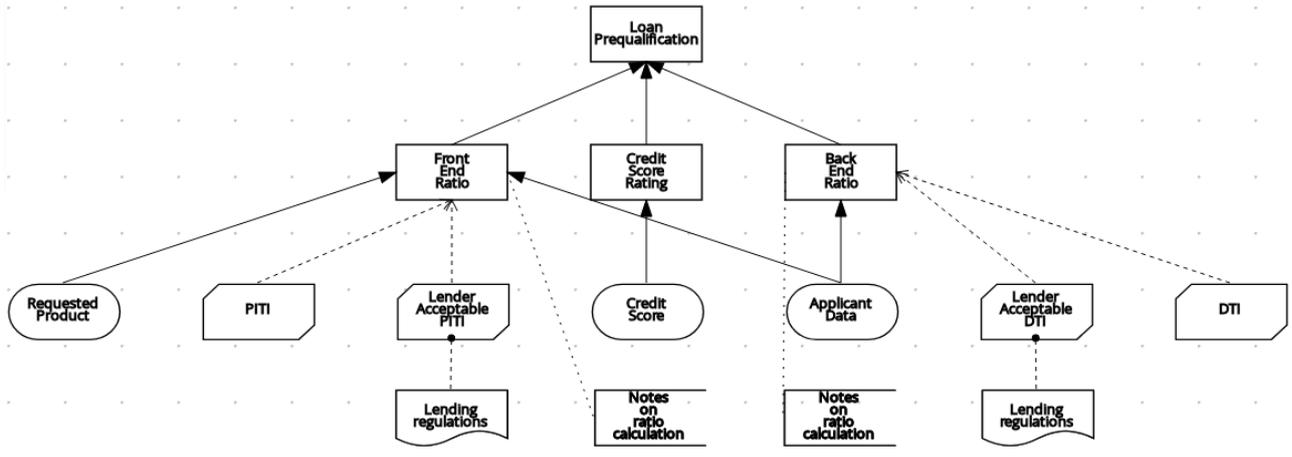
Figure 123. Single DRD for loan prequalification

Alternatively, by following the steps in this procedure, you can divide this example DRD into multiple nested DRDs to better organize the decision requirements, as shown in the following example:
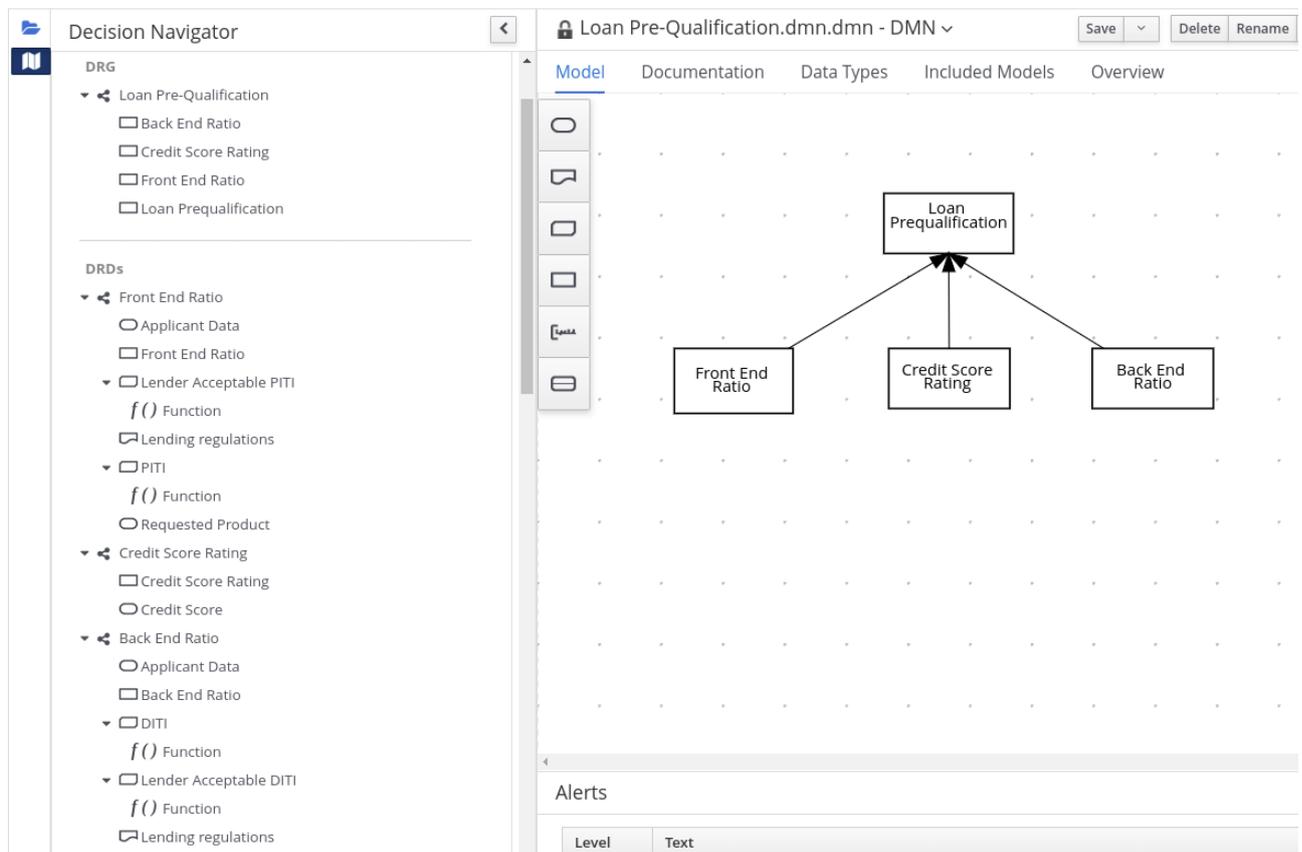
Figure 124. Multiple nested DRDs for loan prequalification
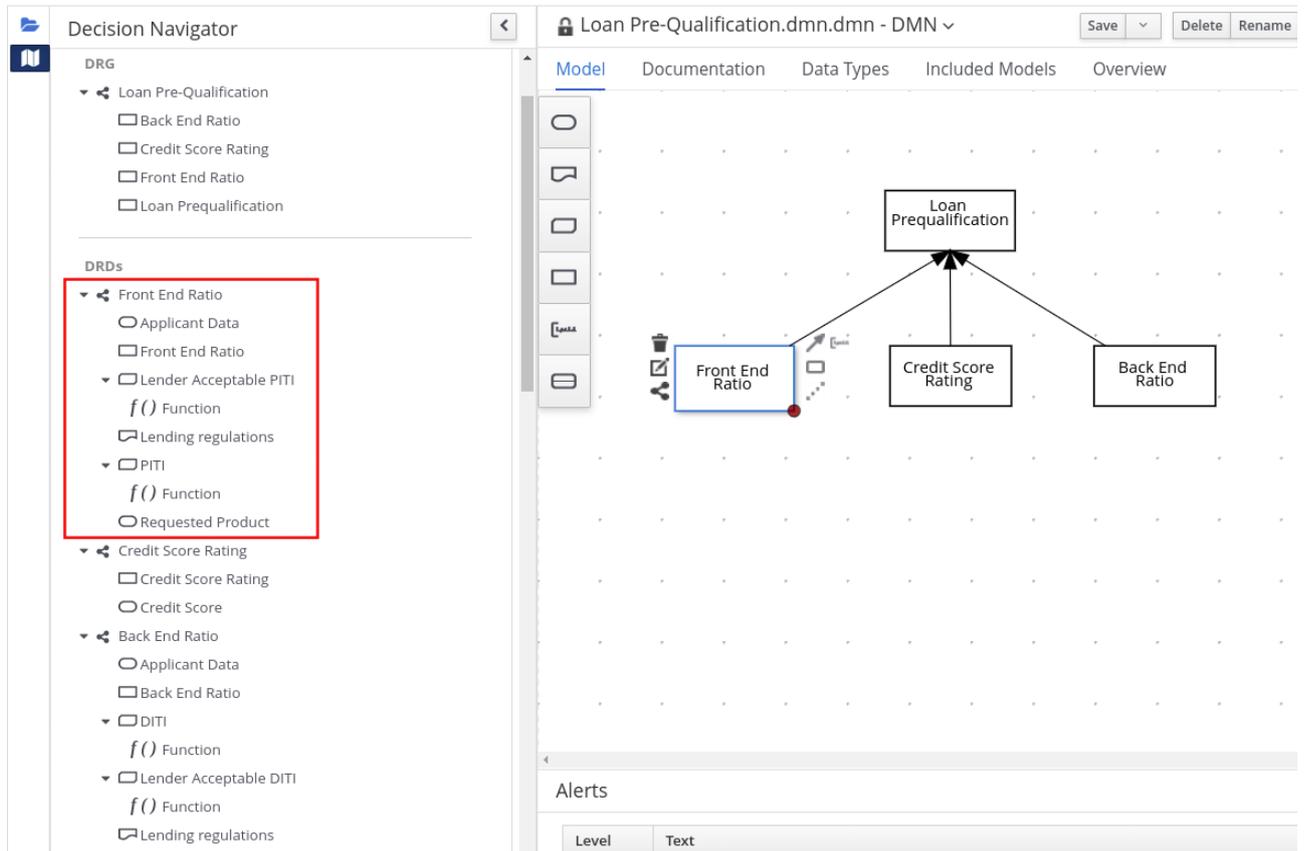
Figure 125. Overview of front end ratio DRD
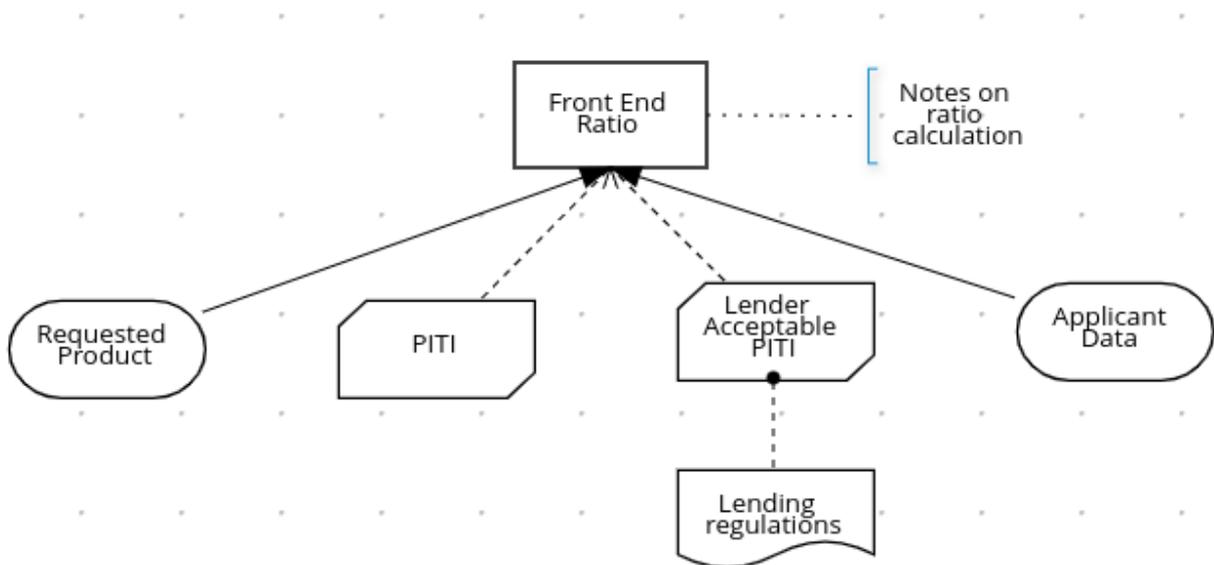
Figure 126. DRD for front end ratio
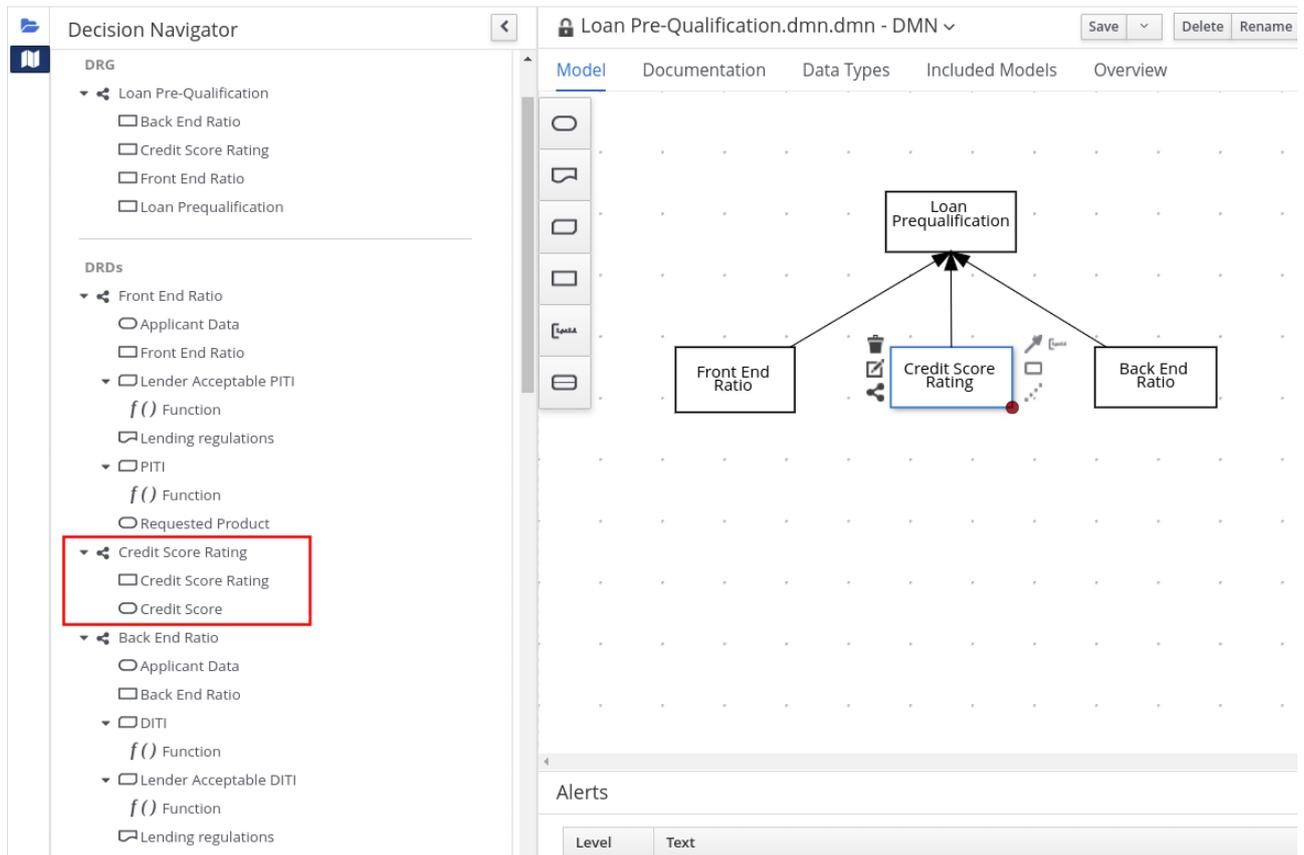
Figure 127. Overview of credit score rating DRD
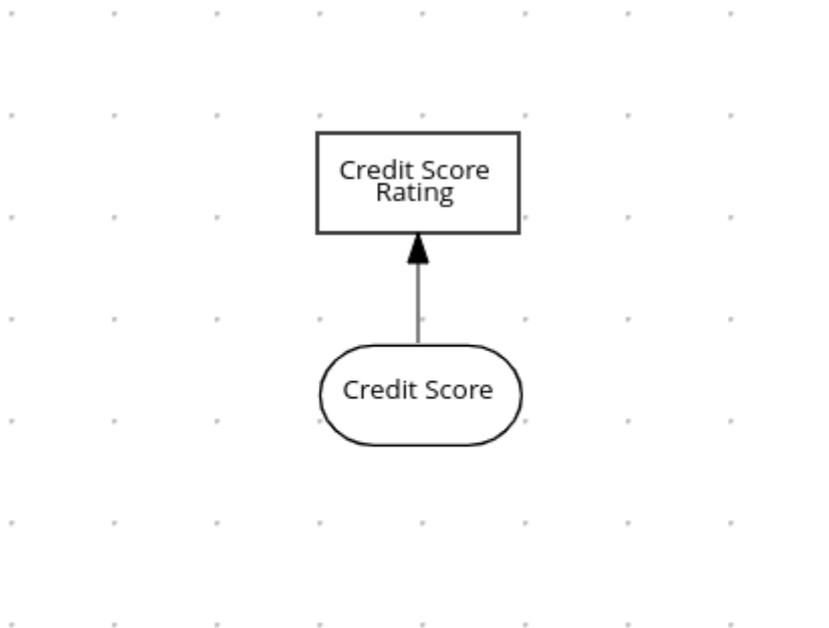
Figure 128. DRD for credit score rating
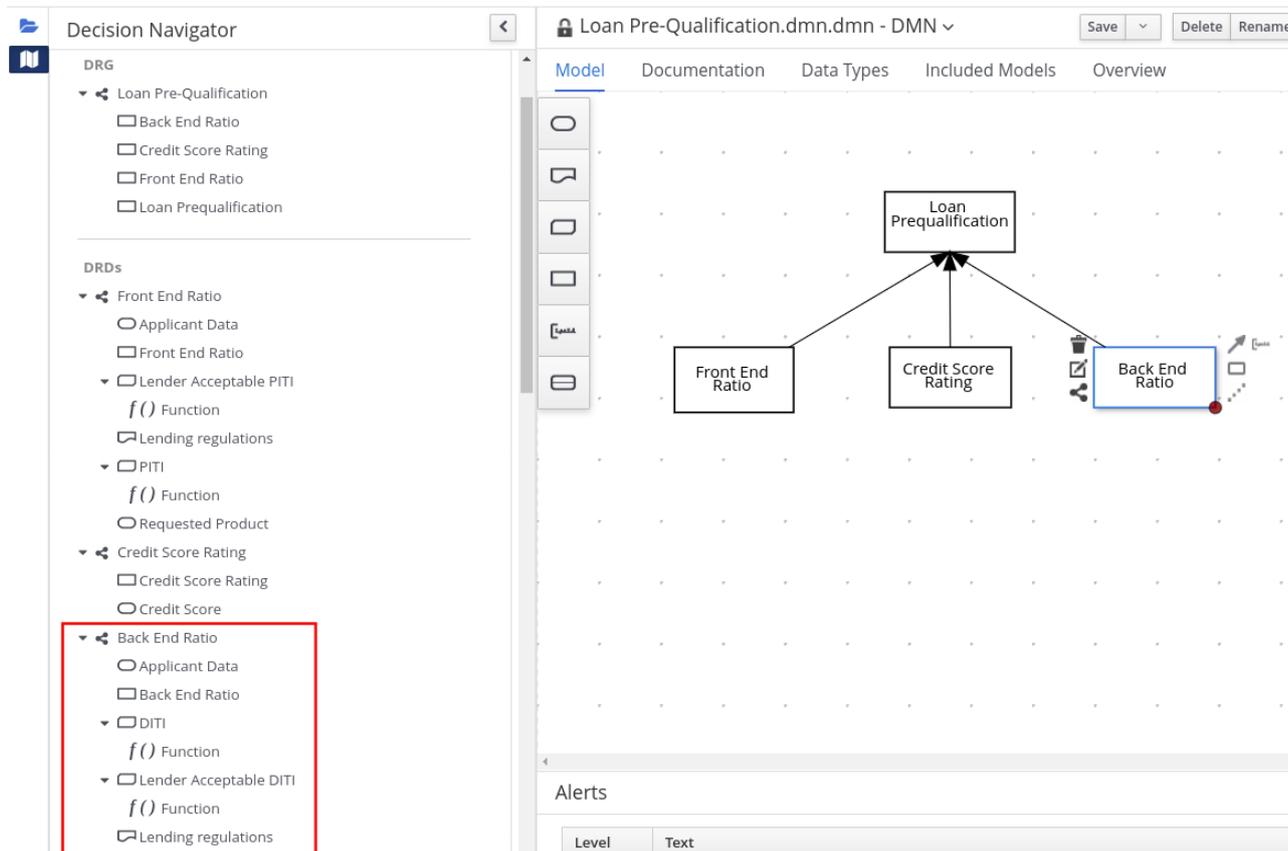
Figure 129. Overview of back end ratio DRD
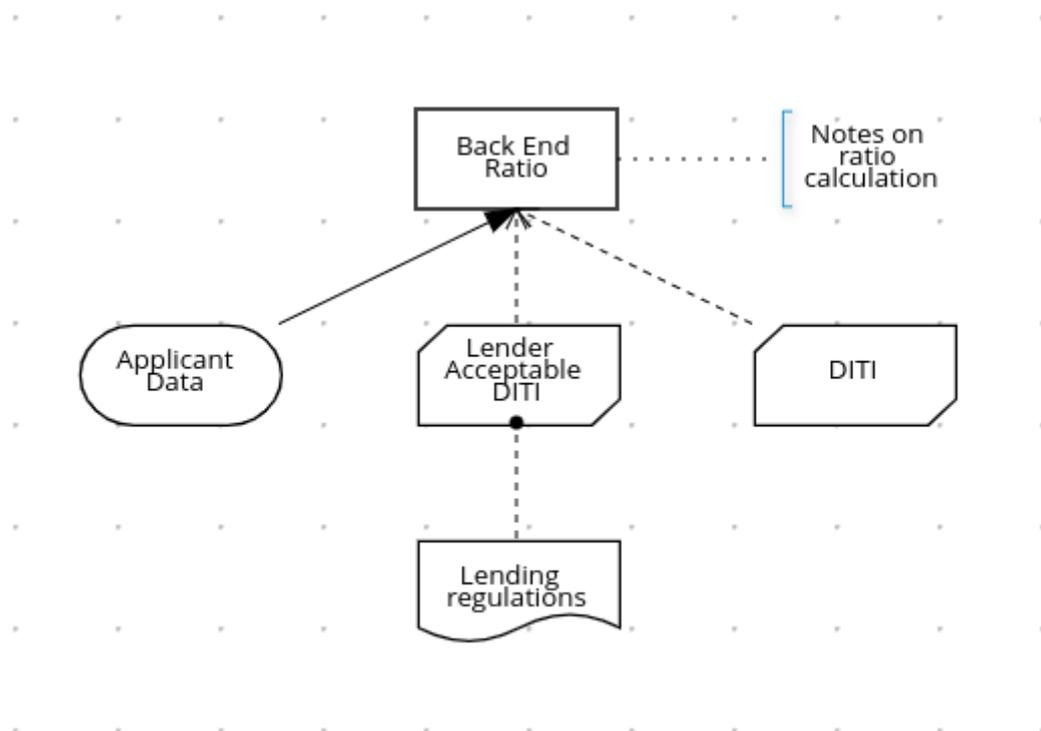
Figure 130. DRD for back end ratio

6.5. DMN model documentation in KIE DMN Editor

In the KIE DMN Editor, you can use the **Documentation** tab to generate a report of your DMN model that you can print or download as an HTML file for offline use. The DMN model report

contains all decision requirements diagrams (DRDs), data types, and boxed expressions in your DMN model. You can use this report to share your DMN model details or as part of your internal reporting workflow.
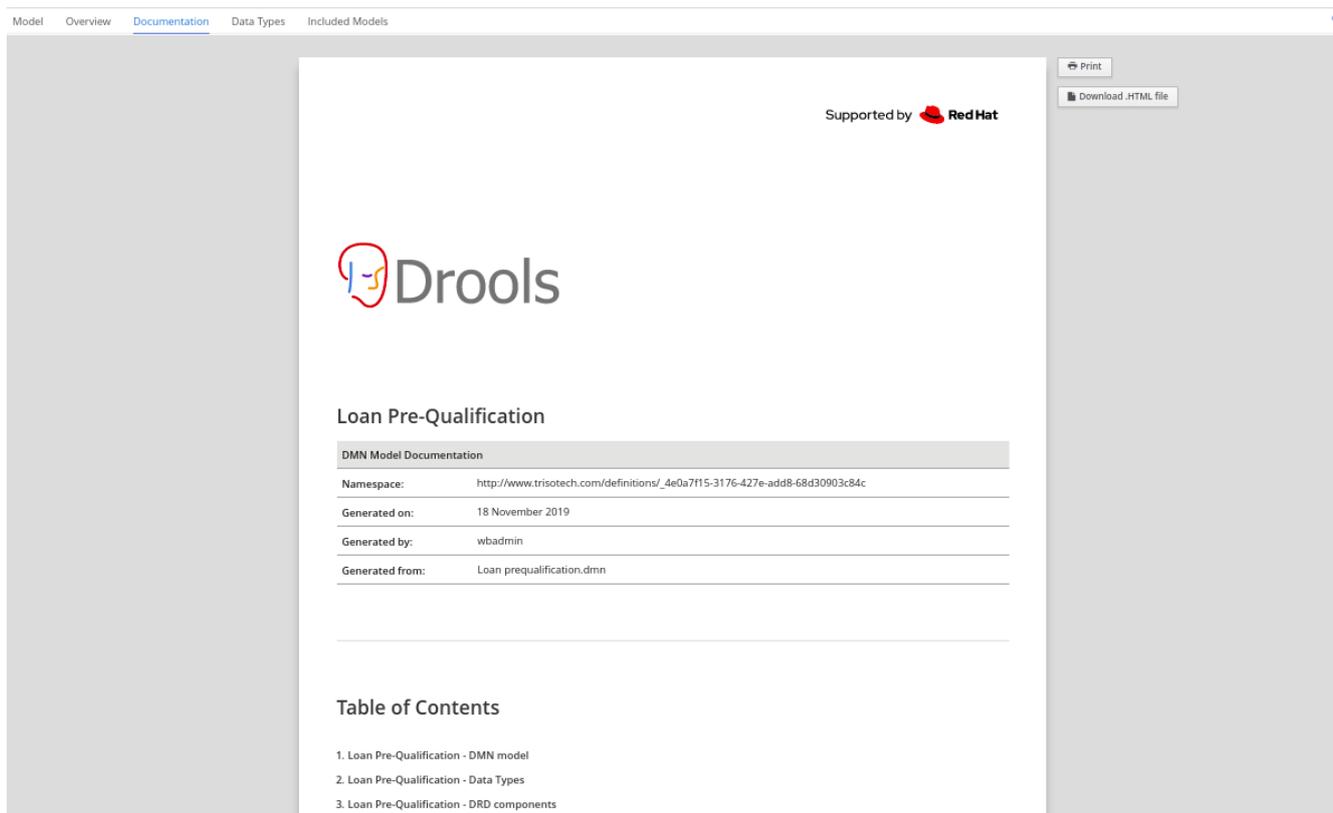
Figure 131. Example DMN model report

6.6. DMN designer navigation and properties in KIE DMN Editor

The DMN designer in KIE DMN Editor provides the following additional features to help you navigate through the components and properties of decision requirements diagrams (DRDs).

DMN file and diagram views

In the upper-left corner of the DMN designer, select the **Project Explorer** view to navigate between all DMN and other files or select the **Decision Navigator** view to navigate between the decision components, graphs, and boxed expressions of a selected DRD:
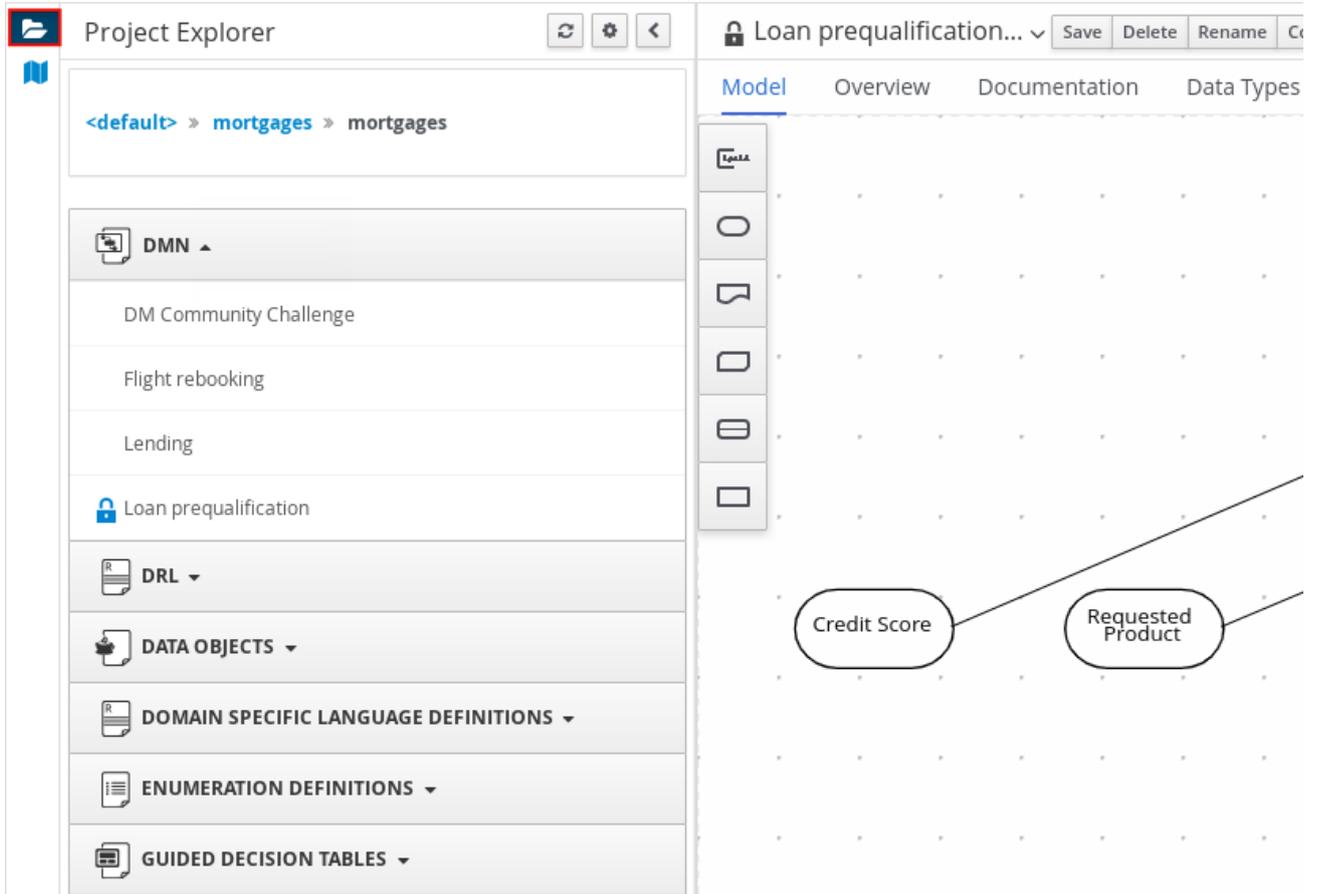
Figure 132. Project Explorer view
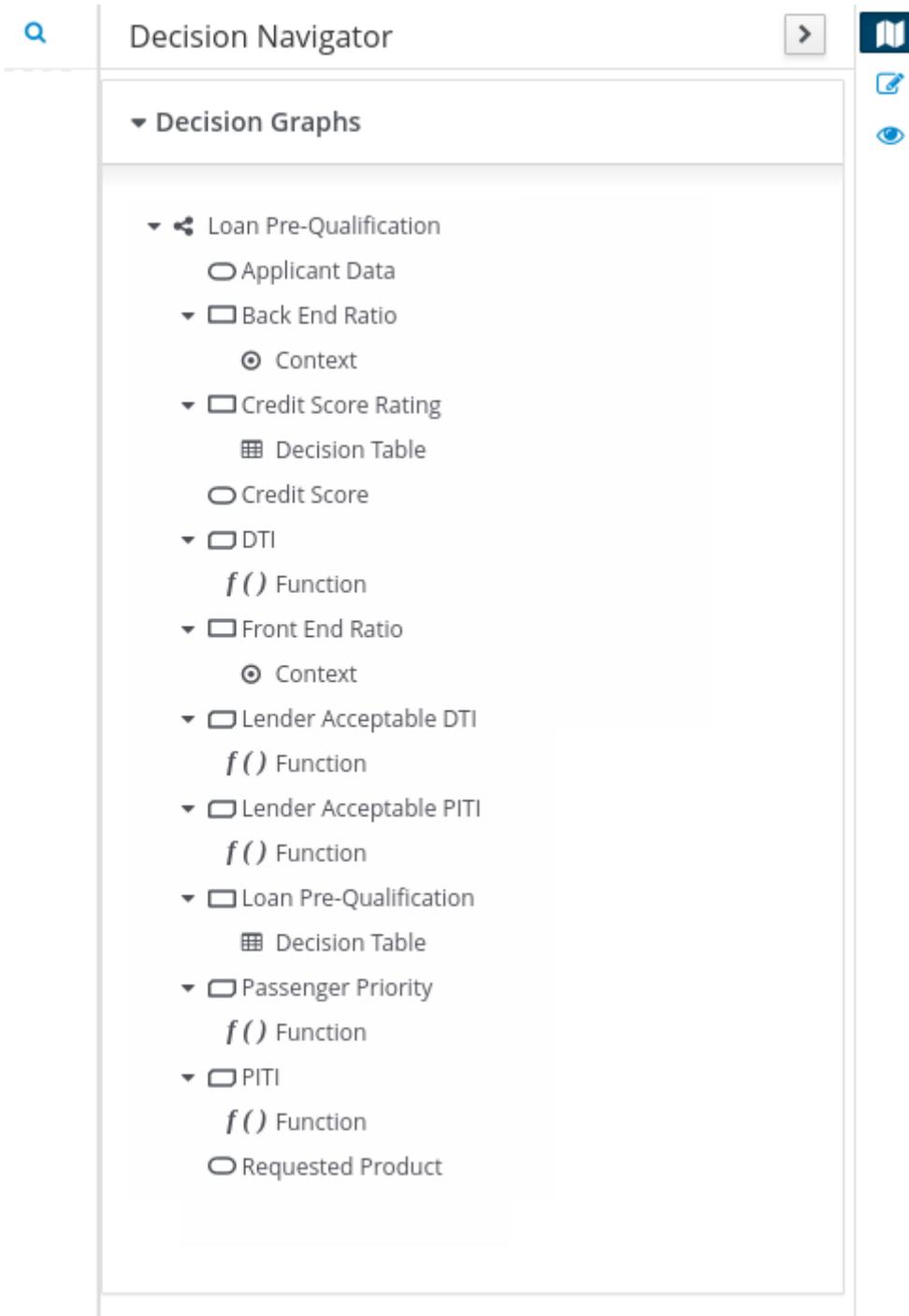
Figure 133. Decision Navigator view
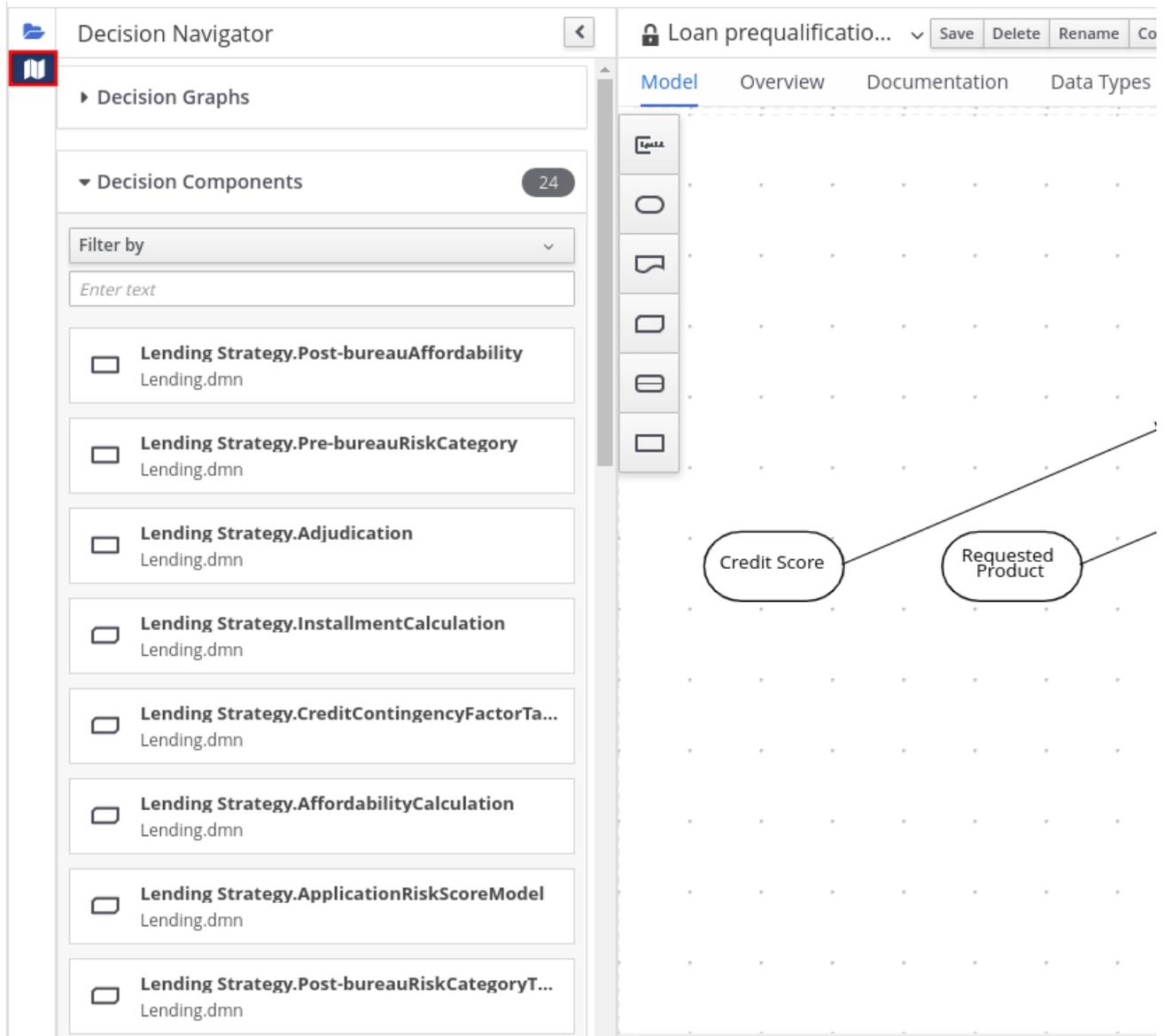
> ℹ️ The DRD components from any DMN models included in the DMN file (in the **Included Models** tab) are also listed in the **Decision Components** panel for the DMN file.

In the upper-right corner of the DMN designer, select the **Explore diagram** icon to view an elevated preview of the selected DRD and to navigate between the nodes of the selected DRD:
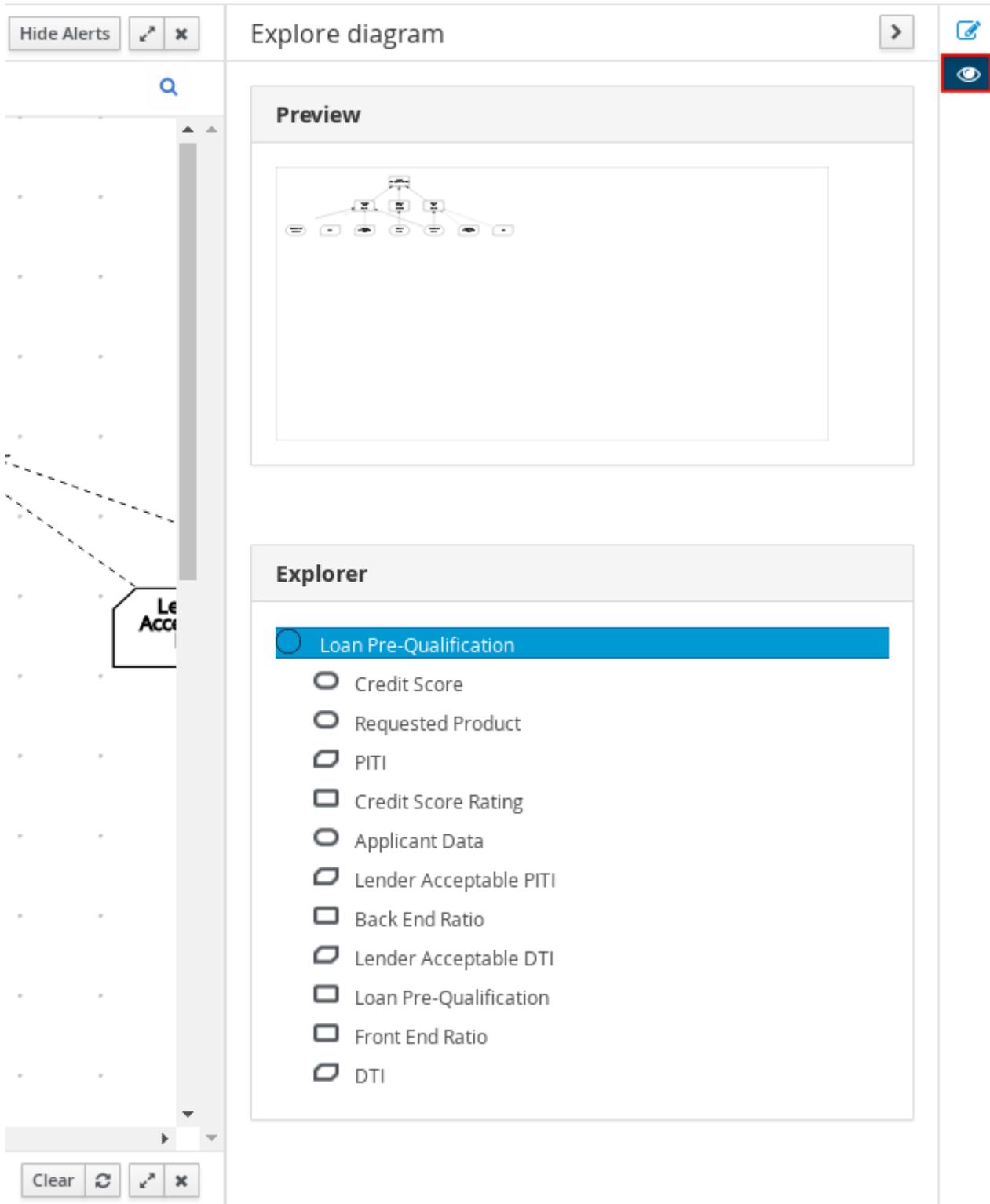
Figure 134. Explore diagram view

DRD properties and design

In the upper-right corner of the DMN designer, select the **Properties** icon to modify the identifying information, data types, and appearance of a selected DRD, DRD node, or boxed expression cell:

Figure 135. DRD node properties

To view the properties of the entire DRD, click the DRD canvas background instead of a specific node.

DRD search

In the upper-right corner of the DMN designer, use the search bar to search for text that appears in your DRD. The search feature is especially helpful in complex DRDs with many nodes:
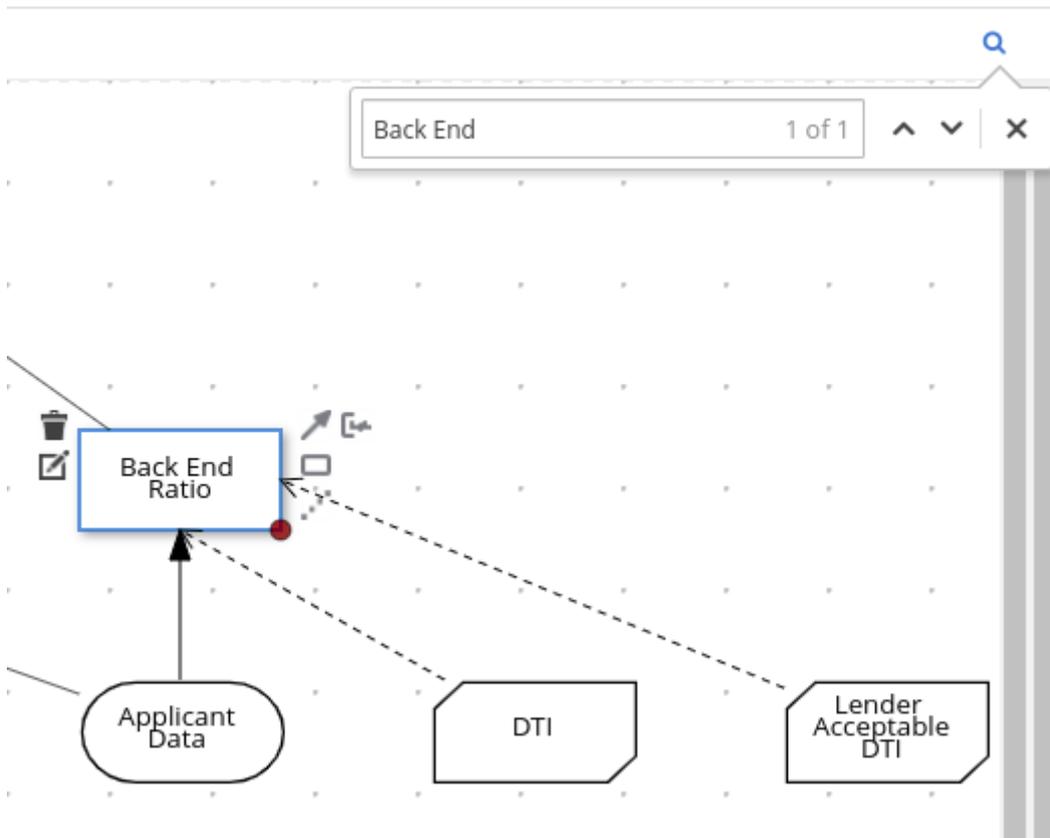
Figure 136. DRD search

DMN decision service details

Select a decision service node in the DMN designer to view additional properties, including **Input Data**, **Encapsulated Decisions**, and **Output Decisions** in the **Properties** panel.
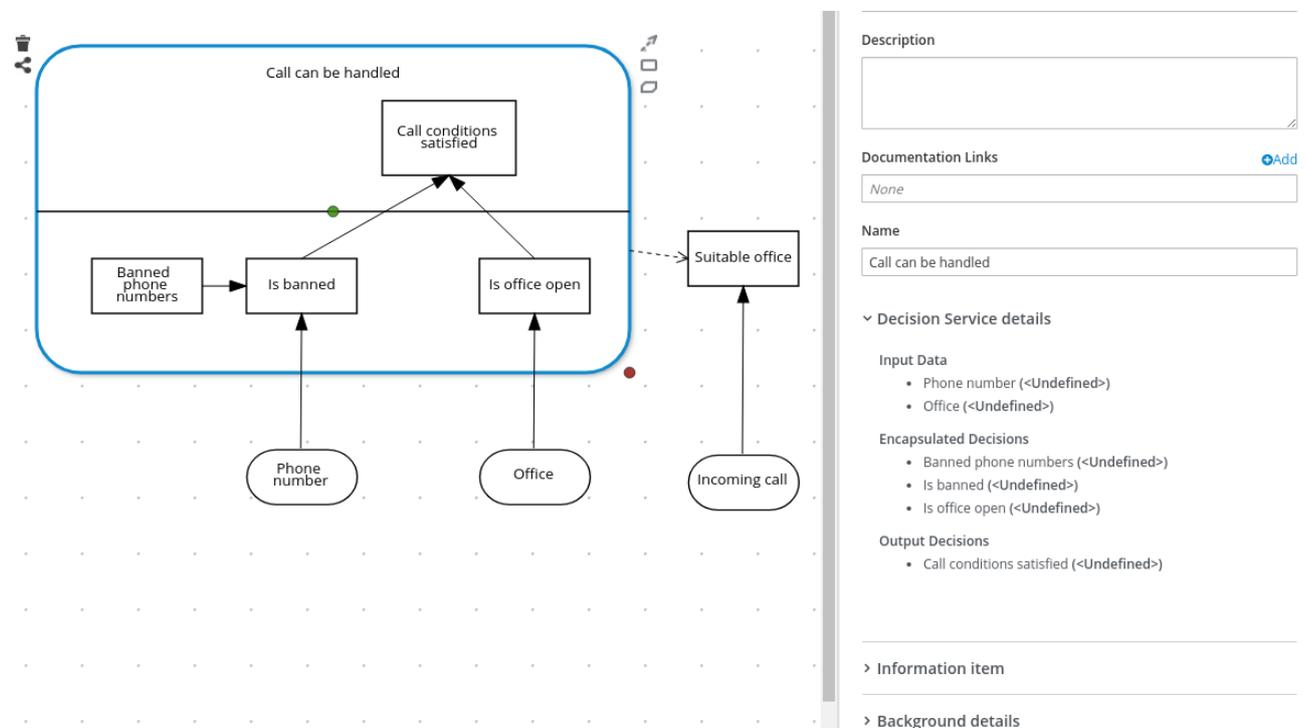

Figure 137. Decision Service details

Chapter 7. DMN model execution

You can create or import DMN files in your Drools project using KIE DMN Editor or package the DMN files as part of your project knowledge JAR (KJAR) file without KIE DMN Editor.

After you implement your DMN files in your Drools project, you can execute the DMN decision service by instantiating a KIE container that contains it directly as a dependency of the calling application. Other options for creating and deploying DMN knowledge packages are also available, and most are similar for all types of knowledge assets, such as DRL files or process definitions.

Alternatively, you could package your DMN files as part of a Kogito cloud-native microservice.

For information about including external DMN assets with your project packaging and deployment method, see [_builddeployutilizeandrunsection].

7.1. Embedding a DMN call directly in a Java application

A KIE container is local when the knowledge assets are either embedded directly into the calling program or are physically pulled in using Maven dependencies for the KJAR. You typically embed knowledge assets directly into a project if there is a tight relationship between the version of the code and the version of the DMN definition. Any changes to the decision take effect after you have intentionally updated and redeployed the application. A benefit of this approach is that proper operation does not rely on any external dependencies to the run time, which can be a limitation of locked-down environments.

Using Maven dependencies enables further flexibility because the specific version of the decision can dynamically change, (for example, by using a `kmodule.xml` or a system property), and it can be periodically scanned for updates and automatically updated. This introduces an external dependency on the deploy time of the service, but executes the decision locally, reducing reliance on an external service being available during run time.

Prerequisites

- You have built the DMN project as a KJAR artifact and deployed it to a Maven repository, or you have included your DMN assets as part of your project classpath:

```
mvn clean install
```

  For more information about project packaging and deployment and executable models, see [_builddeployutilizeandrunsection].

Procedure

1. In your client application, add the following dependencies to the relevant classpath of your Java project:

```
<!-- Required for the DMN runtime API -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-dmn-core</artifactId>
  <version>${drools.version}</version>
</dependency>

<!-- Required if not using classpath KIE container -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>${drools.version}</version>
</dependency>
```

The `<version>` is the Maven artifact version for Drools currently used in your project (for example, 7.59.0.Final).

> Instead of specifying a Drools `<version>` for individual dependencies, consider adding the Drools bill of materials (BOM) dependency to `dependencyManagement` section of your project `pom.xml` file. 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>org.kie</groupId>
>   <artifactId>kie-dmn-bom</artifactId>
>   <version>${drools.version}</version>
>   <scope>import</scope>
>   <type>pom</type>
> </dependency>
> ```

2. Create a KIE container from `classpath` or `ReleaseId`:

```
KieServices kieServices = KieServices.Factory.get();

ReleaseId releaseId = kieServices.newReleaseId( "org.acme", "my-kjar", "1.0.0" );
KieContainer kieContainer = kieServices.newKieContainer( releaseId );
```

Alternative option:

```
KieServices kieServices = KieServices.Factory.get();

KieContainer kieContainer = kieServices.getKieClasspathContainer();
```

3. Obtain `DMNRuntime` from the KIE container and a reference to the DMN model to be evaluated, by using the model `namespace` and `modelName`:

```
DMNRuntime dmnRuntime = KieRuntimeFactory.of(kieContainer.getKieBase()).get
(DMNRuntime.class);

String namespace = "http://www.redhat.com/_c7328033-c355-43cd-b616-0aceef80e52a";
String modelName = "dmn-movieticket-ageclassification";

DMNModel dmnModel = dmnRuntime.getModel(namespace, modelName);
```

4. Execute the decision services for the desired model:

```
DMNContext dmnContext = dmnRuntime.newContext();   ①

for (Integer age : Arrays.asList(1,12,13,64,65,66)) {
    dmnContext.set("Age", age);   ②
    DMNResult dmnResult =
        dmnRuntime.evaluateAll(dmnModel, dmnContext);   ③

    for (DMNDecisionResult dr : dmnResult.getDecisionResults()) {   ④
        log.info("Age: " + age + ", " +
                "Decision: '" + dr.getDecisionName() + "', " +
                "Result: " + dr.getResult());
    }
}
```

① Instantiate a new DMN Context to be the input for the model evaluation. Note that this example is looping through the Age Classification decision multiple times.

② Assign input variables for the input DMN context.

③ Evaluate all DMN decisions defined in the DMN model.

④ Each evaluation may result in one or more results, creating the loop.

This example prints the following output:

```
Age 1 Decision 'AgeClassification' : Child
Age 12 Decision 'AgeClassification' : Child
Age 13 Decision 'AgeClassification' : Adult
Age 64 Decision 'AgeClassification' : Adult
Age 65 Decision 'AgeClassification' : Senior
Age 66 Decision 'AgeClassification' : Senior
```

7.2. Executing a DMN service using Kogito

Interacting with the REST endpoints of Kogito cloud-native microservice including DMN models

provides the most separation between the calling code and the decision logic definition. The calling code is completely free of direct dependencies, and you can implement it in an entirely different development platform such as `Node.js` or `.NET`.

For a quick getting started guide for Kogito on Quarkus, see the Quarkus guide for using Kogito DMN support to add Decision Automation capabilities to a Quarkus application.

For more information about Kogito or migrating to Kogito microservices, see the Kogito website for documentation.