Red Hat Decision Manager 7.10

Developing decision services in Red Hat Decision Manager
Abstract

This document describes how to develop decision services with Red Hat Decision Manager using Decision Model and Notation (DMN) models, Drools Rule Language (DRL) files, guided decision tables, and other decision-authoring assets.
# Table of Contents

PREFACE .................................................................................................................................................. 11

MAKING OPEN SOURCE MORE INCLUSIVE .......................................................................................... 12

PART I. DESIGNING A DECISION SERVICE USING DMN MODELS ...................................................... 13

CHAPTER 1. DECISION-AUTHORING ASSETS IN RED HAT DECISION MANAGER .............................. 14

CHAPTER 2. RED HAT DECISION MANAGER BPMN AND DMN MODELERS ....................................... 18

  2.1. INSTALLING THE RED HAT DECISION MANAGER VS CODE EXTENSION BUNDLE ............... 18

  2.2. CONFIGURING THE RED HAT DECISION MANAGER STANDALONE EDITORS .................... 19

CHAPTER 3. CREATING AND EXECUTING DMN AND BPMN MODELS USING MAVEN ..................... 23

CHAPTER 4. DECISION MODEL AND NOTATION (DMN) ........................................................................ 25

  4.1. DMN CONFORMANCE LEVELS .................................................................................................. 25

  4.2. DMN DECISION REQUIREMENTS DIAGRAM (DRD) COMPONENTS ................................... 25

  4.3. RULE EXPRESSIONS IN FEEL ................................................................................................. 29

    4.3.1. Data types in FEEL .............................................................................................................. 30

    4.3.2. Built-in functions in FEEL .................................................................................................. 34

      4.3.2.1. Conversion functions .................................................................................................... 35

      4.3.2.2. Boolean functions ........................................................................................................ 39

      4.3.2.3. String functions ............................................................................................................. 39

      4.3.2.4. List functions ................................................................................................................ 44

      4.3.2.5. Numeric functions ........................................................................................................ 52

    4.3.2.6. Date and time functions .................................................................................................. 56

    4.3.2.7. Range functions .............................................................................................................. 56

    4.3.2.8. Temporal functions ........................................................................................................ 64

    4.3.2.9. Sort functions ................................................................................................................ 66

      4.3.2.10. Context functions ....................................................................................................... 66

    4.3.3. Variable and function names in FEEL ............................................................................... 67

  4.4. DMN DECISION LOGIC IN BOXED EXPRESSIONS ................................................................... 68

    4.4.1. DMN decision tables .......................................................................................................... 68

      4.4.1.1. Hit policies in DMN decision tables ............................................................................. 70

    4.4.2. Boxed literal expressions .................................................................................................. 71

    4.4.3. Boxed context expressions ................................................................................................ 71

    4.4.4. Boxed relation expressions ............................................................................................... 72

    4.4.5. Boxed function expressions ............................................................................................. 73

    4.4.6. Boxed invocation expressions ........................................................................................... 75

    4.4.7. Boxed list expressions .................................................................................................... 76

  4.5. DMN MODEL EXAMPLE ............................................................................................................. 77

CHAPTER 5. DMN SUPPORT IN RED HAT DECISION MANAGER ............................................................ 86

  5.1. CONFIGURABLE DMN PROPERTIES IN RED HAT DECISION MANAGER ............................... 86

  5.2. CONFIGURABLE DMN VALIDATION IN RED HAT DECISION MANAGER ............................... 88

CHAPTER 6. CREATING AND EDITING DMN MODELS IN BUSINESS CENTRAL .............................. 90

  6.1. DEFINING DMN DECISION LOGIC IN BOXED EXPRESSIONS IN BUSINESS CENTRAL .......... 98

  6.2. CREATING CUSTOM DATA TYPES FOR DMN BOXED EXPRESSIONS IN BUSINESS CENTRAL .... 107

  6.3. INCLUDED MODELS IN DMN FILES IN BUSINESS CENTRAL ................................................ 117

      6.3.1. Including other DMN models within a DMN file in Business Central ............................. 117

      6.3.2. Including PMML models within a DMN file in Business Central .................................. 120

  6.4. CREATING DMN MODELS WITH MULTIPLE DIAGRAMS IN BUSINESS CENTRAL ............. 126
30.1. "ADD A CONDITION" 378
   30.1.1. Inserting an any other value in condition column cells 380
30.2. "ADD A CONDITION BRL FRAGMENT" 380
30.3. "ADD A METADATA COLUMN" 383
30.4. "ADD AN ACTION BRL FRAGMENT" 383
30.5. "ADD AN ATTRIBUTE COLUMN" 386
30.6. "DELETE AN EXISTING FACT" 387
30.7. "EXECUTE A WORK ITEM" 387
30.8. "SET THE VALUE OF A FIELD" 388
30.9. "SET THE VALUE OF A FIELD WITH A WORK ITEM RESULT" 388

CHAPTER 31. VIEWING RULE NAME COLUMN IN GUIDED DECISION TABLES ............... 390

CHAPTER 32. EDITING OR DELETING COLUMNS IN GUIDED DECISION TABLES ............ 391

CHAPTER 33. ADDING ROWS AND DEFINING RULES IN GUIDED DECISION TABLES ....... 392

CHAPTER 34. DEFINING ENUMERATIONS FOR DROP-DOWN LISTS IN RULE ASSETS ....... 394
   34.1. ADVANCED ENUMERATION OPTIONS FOR RULE ASSETS 395

CHAPTER 35. REAL-TIME VERIFICATION AND VALIDATION OF GUIDED DECISION TABLES .... 398
   35.1. TYPES OF PROBLEMS IN GUIDED DECISION TABLES 398
   35.2. TYPES OF NOTIFICATIONS 399
   35.3. DISABLING VERIFICATION AND VALIDATION OF GUIDED DECISION TABLES 399

CHAPTER 36. CONVERTING A GUIDED DECISION TABLE TO A SPREADSHEET DECISION TABLE .... 401
84.6. TEMPORAL OPERATORS FOR EVENTS ........................................ 603
84.7. SESSION CLOCK IMPLEMENTATIONS IN THE DECISION ENGINE 611
84.8. EVENT STREAMS AND ENTRY POINTS ........................................ 613
  84.8.1. Declaring entry points for rule data ................................. 613
84.9. SLIDING WINDOWS OF TIME OR LENGTH .............................. 615
  84.9.1. Declaring sliding windows for rule data ......................... 615
84.10. MEMORY MANAGEMENT FOR EVENTS ................................... 616

CHAPTER 85. DECISION ENGINE QUERIES AND LIVE QUERIES ............................. 618

CHAPTER 86. DECISION ENGINE EVENT LISTENERS AND DEBUG LOGGING .................. 620
  86.1. CONFIGURING A LOGGING UTILITY IN THE DECISION ENGINE ............. 621

CHAPTER 87. EXAMPLE DECISIONS IN RED HAT DECISION MANAGER FOR AN IDE .............. 623
  87.1. IMPORTING AND EXECUTING RED HAT DECISION MANAGER EXAMPLE DECISIONS IN AN IDE 623
  87.2. HELLO WORLD EXAMPLE DECISIONS (BASIC RULES AND DEBUGGING) ............... 626
  87.3. STATE EXAMPLE DECISIONS (FORWARD CHAINING AND CONFLICT RESOLUTION) .... 629
    State example using salience .................................. 632
    State example using agenda groups .............................. 635
    Dynamic facts in the State example ............................. 636
  87.4. FIBONACCI EXAMPLE DECISIONS (RECURSION AND CONFLICT RESOLUTION) ......... 637
  87.5. PRICING EXAMPLE DECISIONS (DECISION TABLES) ............................ 643
    Spreadsheet decision table setup ................................ 644
    Base pricing rules ........................................... 647
    Promotional discount rules ................................. 648
  87.6. PET STORE EXAMPLE DECISIONS (AGENDA GROUPS, GLOBAL VARIABLES, CALLBACKS, AND GUI INTEGRATION) ........................................ 648
    Rule execution behavior in the Pet Store example ............ 649
    Pet Store rule file imports, global variables, and Java functions 651
    Pet Store rules with agenda groups ................................ 652
    Pet Store example execution .................................... 656
  87.7. HONEST POLITICIAN EXAMPLE DECISIONS (TRUTH MAINTENANCE AND SALIENCE) .... 660
    Politician and Hope classes ..................................... 661
    Rule definitions for politician honesty .......................... 662
    Example execution and audit trail .............................. 663
  87.8. SUDOKU EXAMPLE DECISIONS (COMPLEX PATTERN MATCHING, CALLBACKS, AND GUI INTEGRATION) ........................................ 666
    Sudoku example execution and interaction ......................... 666
    Sudoku example classes ....................................... 672
    Sudoku validation rules (validate.drl) ............................ 672
    Sudoku solving rules (sudoku.drl) ................................ 673
  87.9. CONWAY’S GAME OF LIFE EXAMPLE DECISIONS (RULEFLOW GROUPS AND GUI INTEGRATION) ................. 680
    Conway example execution and interaction ......................... 681
    Conway example rules with ruleflow groups ........................ 682
  87.10. HOUSE OF DOOM EXAMPLE DECISIONS (BACKWARD CHAINING AND RECURSION) .......... 686
    Recursive query and related rules .................................. 690
    Transitive closure rule ........................................... 691
    Reactive query rule .............................................. 692
    Queries with unbound arguments in rules .......................... 693

CHAPTER 88. PERFORMANCE TUNING CONSIDERATIONS WITH THE DECISION ENGINE .......... 695

CHAPTER 89. ADDITIONAL RESOURCES .................................................. 697
<table>
<thead>
<tr>
<th>PART X. INTEGRATING MACHINE LEARNING WITH RED HAT DECISION MANAGER</th>
<th>698</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 90. PRAGMATIC AI</td>
<td>699</td>
</tr>
<tr>
<td>CHAPTER 91. CREDIT CARD FRAUD DISPUTE USE CASE</td>
<td>702</td>
</tr>
<tr>
<td>91.1. USING A PMML MODEL WITH A DMN MODEL TO RESOLVE CREDIT CARD TRANSACTION DISPUTES</td>
<td>702</td>
</tr>
<tr>
<td>91.2. CREDIT CARD TRANSACTION DISPUTE EXERCISE PMML FILE</td>
<td>712</td>
</tr>
<tr>
<td>CHAPTER 92. ADDITIONAL RESOURCES</td>
<td>721</td>
</tr>
<tr>
<td>APPENDIX A. VERSIONING INFORMATION</td>
<td>722</td>
</tr>
<tr>
<td>APPENDIX B. CONTACT INFORMATION</td>
<td>723</td>
</tr>
</tbody>
</table>
As a developer of business decisions, you can use Red Hat Decision Manager to develop decision services using Decision Model and Notation (DMN) models, Drools Rule Language (DRL) rules, guided decision tables, and other rule-authoring assets.
MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright’s message.
PART I. DESIGNING A DECISION SERVICE USING DMN MODELS

As a business analyst or business rules developer, you can use Decision Model and Notation (DMN) to model a decision service graphically. The decision requirements of a DMN decision model are determined by a decision requirements graph (DRG) that is depicted in one or more decision requirements diagrams (DRDs). A DRD can represent part or all of the overall DRG for the DMN model. DRDs trace business decisions from start to finish, with each decision node using logic defined in DMN boxed expressions such as decision tables.

Red Hat Decision Manager provides design and runtime support for DMN 1.2 models at conformance level 3, and runtime-only support for DMN 1.1 and 1.3 models at conformance level 3. You can design your DMN models directly in Business Central or with the Red Hat Decision Manager DMN modeler in VSCode, or import existing DMN models into your Red Hat Decision Manager projects for deployment and execution. Any DMN 1.1 and 1.3 models (do not contain DMN 1.3 features) that you import into Business Central, open in the DMN designer, and save are converted to DMN 1.2 models.

For more information about DMN, see the Object Management Group (OMG) Decision Model and Notation specification.

For a step-by-step tutorial with an example DMN decision service, see Getting started with decision services.
CHAPTER 1. DECISION-AUTHORING ASSETS IN RED HAT DECISION MANAGER

Red Hat Decision Manager supports several assets that you can use to define business decisions for your decision service. Each decision-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights the main decision-authoring assets supported in Red Hat Decision Manager projects to help you decide or confirm the best method for defining decisions in your decision service.

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

<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Decision Model and Notation (DMN) models | ● Are decision models based on a notation standard defined by the Object Management Group (OMG)  
● Use graphical decision requirements diagrams (DRDs) that represent part or all of the overall decision requirements graph (DRG) to trace business decision flows  
● Use an XML schema that allows the DMN models to be shared between DMN-compliant platforms  
● Support Friendly Enough Expression Language (FEEL) to define decision logic in DMN decision tables and other DMN boxed expressions  
● Are optimal for creating comprehensive, illustrative, and stable decision flows | Business Central or other DMN-compliant editor | Designing a decision service using DMN models |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided decision tables   | • Are tables of rules that you create in a UI-based table designer in Business Central  
• Are a wizard-led alternative to spreadsheet decision tables  
• Provide fields and options for acceptable input  
• Support template keys and values for creating rule templates  
• Support hit policies, real-time validation, and other additional features not supported in other assets  
• Are optimal for creating rules in a controlled tabular format to minimize compilation errors | Business Central      | Designing a decision service using guided decision tables                                           |
| Spreadsheet decision tables | • Are XLS or XLSX spreadsheet decision tables that you can upload into Business Central  
• Support template keys and values for creating rule templates  
• Are optimal for creating rules in decision tables already managed outside of Business Central  
• Have strict syntax requirements for rules to be compiled properly when uploaded | Spreadsheet editor     | Designing a decision service using spreadsheet decision tables                                     |
| Guided rules             | • Are individual rules that you create in a UI-based rule designer in Business Central  
• Provide fields and options for acceptable input  
• Are optimal for creating single rules in a controlled format to minimize compilation errors | Business Central      | Designing a decision service using guided rules                                                     |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided rule templates    | • Are reusable rule structures that you create in a UI-based template designer in Business Central  
                               • Provide fields and options for acceptable input  
                               • Support template keys and values for creating rule templates (fundamental to the purpose of this asset)  
                               • Are optimal for creating many rules with the same rule structure but with different defined field values | Business Central | Designing a decision service using guided rule templates |
| DRL rules                | • Are individual rules that you define directly in `.drl` text files  
                               • Provide the most flexibility for defining rules and other technicalities of rule behavior  
                               • Can be created in certain standalone environments and integrated with Red Hat Decision Manager  
                               • Are optimal for creating rules that require advanced DRL options  
                               • Have strict syntax requirements for rules to be compiled properly | Business Central or integrated development environment (IDE) | Designing a decision service using DRL rules |
### Asset Highlights

<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Predictive Model Markup Language (PMML) models | - Are predictive data-analytic models based on a notation standard defined by the Data Mining Group (DMG)  
- Use an XML schema that allows the PMML models to be shared between PMML-compliant platforms  
- Support Regression, Scorecard, Tree, Mining, and other model types  
- Can be included with a standalone Red Hat Decision Manager project or imported into a project in Business Central  
- Are optimal for incorporating predictive data into decision services in Red Hat Decision Manager | PMML or XML editor | Designing a decision service using PMML models |
CHAPTER 2. RED HAT DECISION MANAGER BPMN AND DMN MODELERS

Red Hat Decision Manager provides the following extensions or applications that you can use to design Business Process Model and Notation (BPMN) process models and Decision Model and Notation (DMN) decision models using graphical modelers.

- **Business Central** Enables you to view and design BPMN models, DMN models, and test scenario files in a related embedded designer.
  
  To use Business Central, you can set up a development environment containing a Business Central to design business rules and processes, and a KIE Server to execute and test the created business rules and processes.

- **Red Hat Decision Manager VSCode extension** Enables you to view and design BPMN models, DMN models, and test scenario files in Visual Studio Code (VSCode). The VSCode extension requires VSCode 1.46.0 or later.
  
  To install the Red Hat Decision Manager VSCode extension, select the **Extensions** menu option in VSCode and search for and install the **Red Hat Business Automation Bundle** extension.

- **Standalone BPMN and DMN editors** Enable you to view and design BPMN and DMN models embedded in your web applications. To download the necessary files, you can either use the NPM artifacts from the [NPM registry](https://<YOUR_PAGE>/dmn/index.js) or download the JavaScript files directly for the DMN standalone editor library at [https://<YOUR_PAGE>/dmn/index.js](https://<YOUR_PAGE>/dmn/index.js) and for the BPMN standalone editor library at [https://<YOUR_PAGE>/bpmn/index.js](https://<YOUR_PAGE>/bpmn/index.js).

### 2.1. INSTALLING THE RED HAT DECISION MANAGER VS CODE EXTENSION BUNDLE

Red Hat Decision Manager provides a **Red Hat Business Automation Bundle** VSCode extension that enables you to design Decision Model and Notation (DMN) decision models, Business Process Model and Notation (BPMN) 2.0 business processes, and test scenarios directly in VSCode. VSCode is the preferred integrated development environment (IDE) for developing new business applications. Red Hat Decision Manager also provides individual [DMN Editor](https://<YOUR_PAGE>/dmn/index.js) and [BPMN Editor](https://<YOUR_PAGE>/bpmn/index.js) VSCode extensions for DMN or BPMN support only, if needed.

**IMPORTANT**

The editors in the VSCode are partially compatible with the editors in the Business Central, and several Business Central features are not supported in the VSCode.

**Prerequisites**

- The latest stable version of [VSCode](https://<YOUR_PAGE>/bpmn/index.js) is installed.

**Procedure**

1. In your VSCode IDE, select the **Extensions** menu option and search for **Red Hat Business Automation Bundle** for DMN, BPMN, and test scenario file support.

   For DMN or BPMN file support only, you can also search for the individual [DMN Editor](https://<YOUR_PAGE>/dmn/index.js) or [BPMN Editor](https://<YOUR_PAGE>/bpmn/index.js) extensions.

2. When the **Red Hat Business Automation Bundle** extension appears in VSCode, select it and click **Install**.
3. For optimal VSCode editor behavior, after the extension installation is complete, reload or close and re-launch your instance of VSCode.

After you install the VSCode extension bundle, any .dmn, .bpmn, or .bpmn2 files that you open or create in VSCode are automatically displayed as graphical models. Additionally, any .scsesim files that you open or create are automatically displayed as tabular test scenario models for testing the functionality of your business decisions.

If the DMN, BPMN, or test scenario modelers open only the XML source of a DMN, BPMN, or test scenario file and displays an error message, review the reported errors and the model file to ensure that all elements are correctly defined.

NOTE

For new DMN or BPMN models, you can also enter dmn.new or bpmn.new in a web browser to design your DMN or BPMN model in the online modeler. When you finish creating your model, you can click Download in the online modeler page to import your DMN or BPMN file into your Red Hat Decision Manager project in VSCode.

2.2. CONFIGURING THE RED HAT DECISION MANAGER STANDALONE EDITORS

Red Hat Decision Manager provides standalone editors that are distributed in a self-contained library providing an all-in-one JavaScript file for each editor. The JavaScript file uses a comprehensive API to set and control the editor.

You can install the standalone editors in three ways:

- Download each JavaScript file manually
- Use the NPM package

Procedure

1. Install the standalone editors using one of the following methods:
   Download each JavaScript file manually For this method, follow these steps:
   a. Download the JavaScript files.
   b. Add the downloaded Javascript files to your hosted application.
   c. Add the following <script> tag to your HTML page:

   **Script tag for your HTML page for the DMN editor**
   ```html
   <script src="https://<YOUR_PAGE>/dmn/index.js"></script>
   ```

   **Script tag for your HTML page for the BPMN editor**
   ```html
   <script src="https://<YOUR_PAGE>/bpmn/index.js"></script>
   ```

   **Use the NPM package** For this method, follow these steps:
   a. Add the NPM package to your package.json file:
Adding the NPM package

```
npm install @redhat/kogito-tooling-kie-editors-standalone
```

2. Import each editor library to your TypeScript file:

```
import * as DmnEditor from "@redhat/kogito-tooling-kie-editors-standalone/dist/dmn"
import * as BpmnEditor from "@redhat/kogito-tooling-kie-editors-standalone/dist/bpmn"
```

After you install the standalone editors, open the required editor by using the provided editor API, as shown in the following example for opening a DMN editor. The API is the same for each editor.

Opening the DMN standalone editor

```javascript
const editor = DmnEditor.open(
    container: document.getElementById("dmn-editor-container"),
    initialContent: Promise.resolve(""),
    readOnly: false,
    origin: "",
    resources: new Map(
        ["MyIncludedModel.dmn",
            {contentType: "text",
                content: Promise.resolve("")
            }
        ]
    )
);
```

Use the following parameters with the editor API:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>container</td>
<td>HTML element in which the editor is appended.</td>
</tr>
<tr>
<td>initialContent</td>
<td>Promise to a DMN model content. This parameter can be empty, as shown in the following examples:</td>
</tr>
<tr>
<td></td>
<td>- Promise.resolve(&quot;&quot;&quot;)</td>
</tr>
<tr>
<td></td>
<td>- Promise.resolve(&quot;&lt;DIAGRAM_CONTENT_DIRECTLY HERE&gt;&quot;)</td>
</tr>
<tr>
<td></td>
<td>- fetch(&quot;MyDmnModel.dmn&quot;).then(content =&gt; content.text())</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><code>readOnly</code> (Optional)</td>
<td>Enables you to allow changes in the editor. Set to <strong>false</strong> (default) to allow content editing and <strong>true</strong> for read-only mode in editor.</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>Only the DMN editor supports read-only mode for now.</td>
</tr>
<tr>
<td><code>origin</code> (Optional)</td>
<td>Origin of the repository. The default value is <code>window.location.origin</code>.</td>
</tr>
<tr>
<td><code>resources</code> (Optional)</td>
<td>Map of resources for the editor. For example, this parameter is used to provide included models for the DMN editor or work item definitions for the BPMN editor. Each entry in the map contains a resource name and an object that consists of <code>content-type (text or binary)</code> and <code>content</code> (similar to the <code>initialContent</code> parameter).</td>
</tr>
</tbody>
</table>

The returned object contains the methods that are required to manipulate the editor.

**Table 2.2. Returned object methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getContent(): Promise&lt;string&gt;</code></td>
<td>Returns a promise containing the editor content.</td>
</tr>
<tr>
<td><code>setContent(content: string): void</code></td>
<td>Sets the content of the editor.</td>
</tr>
<tr>
<td><code>getPreview(): Promise&lt;string&gt;</code></td>
<td>Returns a promise containing an SVG string of the current diagram.</td>
</tr>
<tr>
<td><code>subscribeToContentChanges(callback: (isDirty: boolean) =&gt; void): (isDirty: boolean) =&gt; void</code></td>
<td>Sets a callback to be called when the content changes in the editor and returns the same callback to be used for unsubscription.</td>
</tr>
<tr>
<td><code>unsubscribeToContentChanges(callback: (isDirty: boolean) =&gt; void): void</code></td>
<td>Unsubscribes the passed callback when the content changes in the editor.</td>
</tr>
<tr>
<td><code>markAsSaved(): void</code></td>
<td>Resets the editor state that indicates that the content in the editor is saved. Also, it activates the subscribed callbacks related to content change.</td>
</tr>
<tr>
<td><code>undo(): void</code></td>
<td>Undoes the last change in the editor. Also, it activates the subscribed callbacks related to content change.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>redo(): void</td>
<td>Redoes the last undone change in the editor. Also, it activates the subscribed callbacks related to content change.</td>
</tr>
<tr>
<td>close(): void</td>
<td>Closes the editor.</td>
</tr>
<tr>
<td>getElementPosition(selector: string): Promise&lt;Rect&gt;</td>
<td>Provides an alternative to extend the standard query selector when an element lives inside a canvas or a video component. The <code>selector</code> parameter must follow the <code>&lt;PROVIDER&gt;::&lt;SELECT&gt;</code> format, such as <code>Canvas:::MySquare</code> or <code>Video:::PresenterHand</code>. This method returns a <code>Rect</code> representing the element position.</td>
</tr>
<tr>
<td>envelopeApi: MessageBusClientApi&lt;KogitoEditorEnvelopeApi&gt;</td>
<td>This is an advanced editor API. For more information about advanced editor API, see <code>MessageBusClientApi</code> and <code>KogitoEditorEnvelopeApi</code>.</td>
</tr>
</tbody>
</table>
CHAPTER 3. CREATING AND EXECUTING DMN AND BPMN MODELS USING MAVEN

You can use Maven archetypes to develop DMN and BPMN models in VSCode using the Red Hat Decision Manager VSCode extension instead of Business Central. You can then integrate your archetypes with your Red Hat Decision Manager decision and process services in Business Central as needed. This method of developing DMN and BPMN models is helpful for building new business applications using the Red Hat Decision Manager VSCode extension.

Procedure

1. In a command terminal, navigate to a local folder where you want to store the new Red Hat Decision Manager project.

2. Enter the following command to generate a project within a defined folder using the following Maven archetype:

```
mvn archetype:generate
-DarchetypeGroupId=org.kie
-DarchetypeArtifactId=kie-kjar-archetype
-DarchetypeVersion=7.48.0.Final-redhat-00004
```

This command generates a Maven project with required dependencies and generates required directories and files to build your business application. You can set up and use Git version-control system (recommended) when developing a project.

If you want to generate multiple projects in the same directory, you can specify the `artifactId` and `groupId` of the generated business application by adding `-DgroupId=<groupid> -DartifactId=<artifactId>` to the previous command.

3. In your VSCode IDE, click File, select Open Folder, and navigate to the folder that is generated using the previous command.

4. Before creating the first asset, set a package for your business application, for example, `org.kie.businessapp`, and create respective directories in the following paths:

   - `PROJECT_HOME/src/main/java`
   - `PROJECT_HOME/src/main/resources`
   - `PROJECT_HOME/src/test/resources`

For example, you can create `PROJECT_HOME/src/main/java/org/kie/businessapp` for `org.kie.businessapp` package.

5. Use VSCode to create assets for your business application. You can create the assets supported by Red Hat Decision Manager VSCode extension using the following ways:

   - To create a business process, create a new file with `.bpmn` or `.bpmn2` in `PROJECT_HOME/src/main/java/org/kie/businessapp` directory, such as `Process.bpmn`. 
To create a DMN model, create a new file with .dmn in `PROJECT_HOME/src/main/java/org/kie/businessapp` directory, such as `AgeDecision.dmn`.

To create a test scenario simulation model, create a new file with .scesim in `PROJECT_HOME/src/main/java/org/kie/businessapp` directory, such as `TestAgeScenario.scesim`.

6. After you create the assets in your Maven archetype, navigate to the root directory (contains `pom.xml`) of the project in the command line and run the following command to build the knowledge JAR (KJAR) of your project:

```shell
mvn clean install
```

If the build fails, address any problems described in the command line error messages and try again to validate the project until the build is successful. However, if the build is successful, you can find the artifact of your business application in `PROJECT_HOME/target` directory.

**NOTE**

Use `mvn clean install` command often to validate your project after each major change during development.

You can deploy the generated knowledge JAR (KJAR) of your business application on a running KIE Server using the REST API. For more information about using REST API, see *Interacting with Red Hat Decision Manager using KIE APIs*.
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 more information about the background and applications of DMN, see the OMG Decision Model and Notation specification.

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.

Red Hat Decision Manager provides design and runtime support for DMN 1.2 models at conformance level 3, and runtime-only support for DMN 1.1 and 1.3 models at conformance level 3. You can design your DMN models directly in Business Central or with the Red Hat Decision Manager DMN modeler in VSCode, or import existing DMN models into your Red Hat Decision Manager projects for deployment and execution. Any DMN 1.1 and 1.3 models (do not contain DMN 1.3 features) that you import into Business Central, open in the DMN designer, 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 4.1. DRD components
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decision</td>
<td>Node where one or more input elements determine an output based on defined decision logic.</td>
<td><img src="image" alt="Decision" /></td>
</tr>
<tr>
<td>Business knowledge model</td>
<td>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.</td>
<td><img src="image" alt="Business knowledge model" /></td>
</tr>
<tr>
<td>Knowledge source</td>
<td>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.</td>
<td><img src="image" alt="Knowledge source" /></td>
</tr>
<tr>
<td>Input data</td>
<td>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.</td>
<td><img src="image" alt="Input data" /></td>
</tr>
<tr>
<td>Decision service</td>
<td>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.</td>
<td><img src="image" alt="Decision service" /></td>
</tr>
<tr>
<td>Requirement connectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Information requirement</td>
<td>Connection from an input data node or decision node to another decision node that requires the information.</td>
<td><img src="image" alt="Information requirement" /></td>
</tr>
<tr>
<td>Knowledge requirement</td>
<td>Connection from a business knowledge model to a decision node or to another business knowledge model that invokes the decision logic.</td>
<td><img src="image" alt="Knowledge requirement" /></td>
</tr>
<tr>
<td>Authority requirement</td>
<td>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.</td>
<td><img src="image" alt="Authority requirement" /></td>
</tr>
</tbody>
</table>
The following table summarizes the permitted connectors between DRD elements:

### Table 4.2. DRD connector rules

<table>
<thead>
<tr>
<th>Starts from</th>
<th>Connects to</th>
<th>Connection type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision</td>
<td>Decision</td>
<td>Information requirement</td>
<td><img src="image" alt="Decision to Decision Example" /></td>
</tr>
<tr>
<td>Business knowledge model</td>
<td>Decision</td>
<td>Knowledge requirement</td>
<td><img src="image" alt="Business Knowledge Model to Decision Example" /></td>
</tr>
<tr>
<td></td>
<td>Business knowledge model</td>
<td></td>
<td><img src="image" alt="Business Knowledge Model to Business Knowledge Model Example" /></td>
</tr>
<tr>
<td>Decision service</td>
<td>Decision</td>
<td>Knowledge requirement</td>
<td><img src="image" alt="Decision Service to Decision Example" /></td>
</tr>
<tr>
<td></td>
<td>Business knowledge model</td>
<td></td>
<td><img src="image" alt="Decision Service to Business Knowledge Model Example" /></td>
</tr>
<tr>
<td>Input data</td>
<td>Decision</td>
<td>Information requirement</td>
<td><img src="image" alt="Input Data to Decision Example" /></td>
</tr>
<tr>
<td>Starts from</td>
<td>Connects to</td>
<td>Connection type</td>
<td>Example</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------</td>
<td>----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Knowledge source</td>
<td>Authority requirement</td>
<td></td>
<td><img src="image1" alt="Knowledge source to Authority requirement" /></td>
</tr>
<tr>
<td>Knowledge source</td>
<td>Decision</td>
<td>Authority requirement</td>
<td><img src="image2" alt="Knowledge source to Decision" /></td>
</tr>
<tr>
<td>Business knowledge model</td>
<td></td>
<td></td>
<td><img src="image3" alt="Business knowledge model" /></td>
</tr>
<tr>
<td>Knowledge source</td>
<td></td>
<td></td>
<td><img src="image4" alt="Knowledge source" /></td>
</tr>
<tr>
<td>Decision</td>
<td>Text annotation</td>
<td>Association</td>
<td><img src="image5" alt="Decision to Text annotation" /></td>
</tr>
<tr>
<td>Business knowledge model</td>
<td></td>
<td></td>
<td><img src="image6" alt="Business knowledge model" /></td>
</tr>
<tr>
<td>Knowledge source</td>
<td></td>
<td></td>
<td><img src="image7" alt="Knowledge source" /></td>
</tr>
<tr>
<td>Input data</td>
<td></td>
<td></td>
<td><img src="image8" alt="Input data" /></td>
</tr>
</tbody>
</table>

The following example DRD illustrates some of these DMN components in practice:
Figure 4.1. Example DRD: Loan prequalification

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.
For more information about FEEL in DMN, see the OMG Decision Model and Notation specification.

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

**NOTE**

The DMN specification currently does not provide an explicit way of declaring a variable as a **function**, **context**, **range**, or **list**, but Red Hat Decision Manager 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.

Red Hat Decision Manager 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:**

```java
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:**

```java
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:**

```java
date and time( "2017-10-22T23:59:00" )
```
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.

IMPORTANT

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" )`

IMPORTANT

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" )`

IMPORTANT

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 Red Hat Decision Manager 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 Red Hat Decision Manager 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 Red Hat Decision Manager 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 Red Hat Decision Manager 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):

\[
[ \text{duration}("PT1H") .. \text{duration}("PT12H") ]
\]

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 \text{true} if the value of a variable \( x \) is between 0 and 100:

\[
x \text{ in } [ 1 .. 100 ]
\]

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 Red Hat Decision Manager 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 \text{count}, which uses the list of elements as the parameter.

For example, the following expression returns 4:

\[
\text{count}([ 2, 3, 4, 5 ])\]

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.
The following sections describe each built-in FEEL function, listed in the format `NAME( PARAMETERS )`. For more information about FEEL functions in DMN, see the OMG Decision Model and Notation specification.

### 4.3.2.1. Conversion functions

The following functions support conversion between values of different types. Some of these functions use specific string formats, such as the following examples:

- **date string**: Follows the format defined in the [XML Schema Part 2: Datatypes](https://www.w3.org/TR/xmlschema-2/) document, such as **2020-06-01**

- **time string**: Follows one of the following formats:
  - Format defined in the [XML Schema Part 2: Datatypes](https://www.w3.org/TR/xmlschema-2/) document, such as **23:59:00z**
  - Format for a local time defined by ISO 8601 followed by @ and an IANA Timezone, such as **00:01:00@Etc/UTC**

- **date time string**: Follows the format of a **date string** followed by `T` and a **time string**, such as **2012-12-25T11:00:00Z**

- **duration string**: Follows the format of **days and time duration** and **years and months duration** defined in the [XQuery 1.0 and XPath 2.0 Data Model](https://www.w3.org/TR/xquery/), such as **P1Y2M**

#### date( from ) - using date

Converts **from** to a **date** value.

**Table 4.3. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>string</td>
<td>date string</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
date( "2012-12-25" ) - date( "2012-12-24" ) = duration( "P1D" )
```

#### date( from ) - using date and time

Converts **from** to a **date** value and sets time components to null.

**Table 4.4. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>date and time</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
date(date and time( "2012-12-25T11:00:00Z" )) = date( "2012-12-25" )
```
date( year, month, day )

Produces a date from the specified year, month, and day values.

Table 4.5. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>number</td>
</tr>
<tr>
<td>month</td>
<td>number</td>
</tr>
<tr>
<td>day</td>
<td>number</td>
</tr>
</tbody>
</table>

Example

date( 2012, 12, 25 ) = date( "2012-12-25" )

date and time( date, time )

Produces a date and time from the specified date and ignores any time components and the specified time.

Table 4.6. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date or date and time</td>
</tr>
<tr>
<td>time</td>
<td>time</td>
</tr>
</tbody>
</table>

Example

date and time( "2012-12-24T23:59:00" ) = date and time(date( "2012-12-24" ), time( "23:59:00" ))

date and time( from )

Produces a date and time from the specified string.

Table 4.7. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>string</td>
<td>date time string</td>
</tr>
</tbody>
</table>

Example

date and time( "2012-12-24T23:59:00" ) + duration( "PT1M" ) = date and time( "2012-12-25T00:00:00" )
**time( from )**

Produces a *time* from the specified string.

**Table 4.8. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>string</td>
<td>time string</td>
</tr>
</tbody>
</table>

**Example**

- \( \text{time( "23:59:00z" ) + duration( "PT2M" ) = time( "00:01:00@Etc/UTC" )} \)

**time( from )**

Produces a *time* from the specified parameter and ignores any date components.

**Table 4.9. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td><em>time</em> or <em>date and time</em></td>
</tr>
</tbody>
</table>

**Example**

- \( \text{time(date and time( "2012-12-25T11:00:00Z" )) = time( "11:00:00Z" )} \)

**time( hour, minute, second, offset? )**

Produces a *time* from the specified hour, minute, and second component values.

**Table 4.10. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hour</td>
<td>number</td>
</tr>
<tr>
<td>minute</td>
<td>number</td>
</tr>
<tr>
<td>second</td>
<td>number</td>
</tr>
<tr>
<td>offset (Optional)</td>
<td>days and time duration or null</td>
</tr>
</tbody>
</table>

**Example**

- \( \text{time( "23:59:00z" ) = time(23, 59, 0, duration( "PT0H" ))} \)
number( from, grouping separator, decimal separator )
Converts from to a number using the specified separators.

Table 4.11. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>string representing a valid number</td>
</tr>
<tr>
<td>grouping separator</td>
<td>Space ( ), comma (,), period (.), or null</td>
</tr>
<tr>
<td>decimal separator</td>
<td>Same types as grouping separator, but the values cannot match</td>
</tr>
</tbody>
</table>

Example

number( "1 000,0", ",", "," ) = number( "1,000.0", ",", "." )

string( from )
Provides a string representation of the specified parameter.

Table 4.12. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>Non-null value</td>
</tr>
</tbody>
</table>

Examples

string( 1.1 ) = "1.1"
string( null ) = null

duration( from )
Converts from to a days and time duration value or years and months duration value.

Table 4.13. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>string</td>
<td>duration string</td>
</tr>
</tbody>
</table>

Examples

date and time( "2012-12-24T23:59:00" ) - date and time( "2012-12-22T03:45:00" ) = duration( "P2DT20H14M" )
duration( "P2Y2M" ) = duration( "P26M" )
years and months duration( from, to )
Calculates the years and months duration between the two specified parameters.

Table 4.14. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>date or date and time</td>
</tr>
<tr>
<td>to</td>
<td>date or date and time</td>
</tr>
</tbody>
</table>

Example

years and months duration( date( "2011-12-22" ), date( "2013-08-24" ) ) = duration( "P1Y8M" )

4.3.2.2. Boolean functions

The following functions support Boolean operations.

not( negand )
Performs the logical negation of the negand operand.

Table 4.15. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>negand</td>
<td>boolean</td>
</tr>
</tbody>
</table>

Examples

not( true ) = false
not( null ) = null

4.3.2.3. String functions

The following functions support string operations.

NOTE
In FEEL, Unicode characters are counted based on their code points.

substring( string, start position, length? )
Returns the substring from the start position for the specified length. The first character is at position value 1.

Table 4.16. Parameters
### Table 4.17. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
</tbody>
</table>

#### Examples

```
substring( "testing",3 ) = "sting"
substring( "testing",3,3 ) = "sti"
substring( "testing", -2, 1 ) = "n"
substring( \"\U01F40Eab\", 2 ) = "ab"
```

### NOTE

In FEEL, the string literal "\U01F40Eab" is the ab string (horse symbol followed by a and b).

### string length( string )

Calculates the length of the specified string.

#### Table 4.18. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
</tbody>
</table>

#### Examples

```
string length( "tes" ) = 3
string length( \"\U01F40Eab\" ) = 3
```

### upper case( string )

Produces an uppercase version of the specified string.

#### Example

```
upper case( "aBc4" ) = "ABC4"
```
lower case( string )

Produces a lowercase version of the specified string.

Table 4.19. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
</tbody>
</table>

Example

lower case( "aBc4" ) = "abc4"

substring before( string, match )

Calculates the substring before the match.

Table 4.20. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>match</td>
<td>string</td>
</tr>
</tbody>
</table>

Examples

substring before( "testing", "ing" ) = "test"
substring before( "testing", "xyz" ) = ""

substring after( string, match )

Calculates the substring after the match.

Table 4.21. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>match</td>
<td>string</td>
</tr>
</tbody>
</table>

Examples

substring after( "testing", "test" ) = "ing"
substring after( "", "a" ) = ""
replace( input, pattern, replacement, flags? )
Calculates the regular expression replacement.

**Table 4.22. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>string</td>
</tr>
<tr>
<td>pattern</td>
<td>string</td>
</tr>
<tr>
<td>replacement</td>
<td>string</td>
</tr>
<tr>
<td>flags (Optional)</td>
<td>string</td>
</tr>
</tbody>
</table>

**NOTE**

This function uses regular expression parameters as defined in XQuery 1.0 and XPath 2.0 Functions and Operators.

**Example**

```xml
replace( "abcd", "(ab)|(a)", "[1=$1][2=$2]" ) = "[1=ab][2=cd"
```

contains( string, match )
Returns true if the string contains the match.

**Table 4.23. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>match</td>
<td>string</td>
</tr>
</tbody>
</table>

**Example**

```xml
contains( "testing", "to" ) = false
```

starts with( string, match )
Returns true if the string starts with the match.

**Table 4.24. Parameters**
### Example

```
starts with( "testing", "te" ) = true
```

**ends with( string, match )**  
Returns **true** if the string ends with the match.

### Table 4.25. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>match</td>
<td>string</td>
</tr>
</tbody>
</table>

### Example

```
ends with( "testing", "g" ) = true
```

**matches( input, pattern, flags? )**  
Returns **true** if the input matches the regular expression.

### Table 4.26. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>string</td>
</tr>
<tr>
<td>pattern</td>
<td>string</td>
</tr>
<tr>
<td>flags</td>
<td>string</td>
</tr>
</tbody>
</table>

**NOTE**

This function uses regular expression parameters as defined in *XQuery 1.0 and XPath 2.0 Functions and Operators*.

### Example

```
matches( "teeesting", "^te\*sting" ) = true
```
split( string, delimiter )

Returns a list of the original string and splits it at the delimiter regular expression pattern.

Table 4.27. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>delimiter</td>
<td>string for a regular expression pattern</td>
</tr>
</tbody>
</table>

NOTE

This function uses regular expression parameters as defined in XQuery 1.0 and XPath 2.0 Functions and Operators.

Examples

split( "John Doe", "\s" ) = ["John", "Doe"]
split( "a;b;c;;", ;" ) = ["a","b","c",",",""]

4.3.2.4. List functions

The following functions support list operations.

NOTE

In FEEL, the index of the first element in a list is 1. The index of the last element in a list can be identified as -1.

list contains( list, element )

Returns true if the list contains the element.

Table 4.28. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
<tr>
<td>element</td>
<td>Any type, including null</td>
</tr>
</tbody>
</table>

Example

list contains( [1,2,3], 2 ) = true

count( list )

Counts the elements in the list.
### Table 4.29. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

#### Examples

- `count([1,2,3]) = 3`
- `count([]) = 0`
- `count([1,2,3]) = 2`

#### min( list )

Returns the minimum comparable element in the list.

### Table 4.30. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

#### Alternative signature

- `min(e1, e2, ..., eN)`

#### Examples

- `min([1,2,3]) = 1`
- `min(1) = 1`
- `min([1]) = 1`

#### max( list )

Returns the maximum comparable element in the list.

### Table 4.31. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

#### Alternative signature

- `max(e1, e2, ..., eN)`

#### Examples

- `max(1,2,3) = 3`
- `max([]) = null`
sum( list )
Returns the sum of the numbers in the list.

Table 4.32. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list of number elements</td>
</tr>
</tbody>
</table>

Alternative signature

sum( n1, n2, ..., nN )

Examples

- sum( [1,2,3] ) = 6
- sum( 1,2,3 ) = 6
- sum( 1 ) = 1
- sum( [] ) = null

mean( list )
Calculates the average (arithmetic mean) of the elements in the list.

Table 4.33. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list of number elements</td>
</tr>
</tbody>
</table>

Alternative signature

mean( n1, n2, ..., nN )

Examples

- mean( [1,2,3] ) = 2
- mean( 1,2,3 ) = 2
- mean( 1 ) = 1
- mean( [] ) = null

all( list )
Returns true if all elements in the list are true.

Table 4.34. Parameters
Alternative signature

\[ \text{all( } b_1, b_2, \ldots, b_N \text{ )} \]

Examples

\[
\begin{align*}
\text{all( [false,null,true] )} &= \text{false} \\
\text{all( true )} &= \text{true} \\
\text{all( [true] )} &= \text{true} \\
\text{all( [] )} &= \text{true} \\
\text{all( 0 )} &= \text{null}
\end{align*}
\]

\[ \text{any( list )} \]

Returns \textbf{true} if any element in the list is true.

Table 4.35. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list of \textbf{boolean} elements</td>
</tr>
</tbody>
</table>

Alternative signature

\[ \text{any( } b_1, b_2, \ldots, b_N \text{ )} \]

Examples

\[
\begin{align*}
\text{any( [false,null,true] )} &= \text{true} \\
\text{any( false )} &= \text{false} \\
\text{any( [] )} &= \text{false} \\
\text{any( 0 )} &= \text{null}
\end{align*}
\]

\[ \text{sublist( list, start position, length? )} \]

Returns the sublist from the start position, limited to the length elements.

Table 4.36. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
<tr>
<td>start position</td>
<td>number</td>
</tr>
</tbody>
</table>
### Table 4.37. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>length (Optional)</td>
<td>number</td>
</tr>
</tbody>
</table>

### Example

```python
sublist([4,5,6], 1, 2) = [4,5]
```

**append(list, item)**

Creates a list that is appended to the item or items.

### Table 4.38. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
<tr>
<td>item</td>
<td>Any type</td>
</tr>
</tbody>
</table>

### Example

```python
append([1], 2, 3) = [1,2,3]
```

**concatenate(list)**

Creates a list that is the result of the concatenated lists.

### Table 4.38. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

### Example

```python
concatenate([1],[2],[3]) = [1,2,3]
```

**insert before(list, position, newItem)**

Creates a list with the `newItem` inserted at the specified position.

### Table 4.39. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
<tr>
<td>position</td>
<td>number</td>
</tr>
</tbody>
</table>

sublist([4,5,6], 1, 2) = [4,5]

append([1], 2, 3) = [1,2,3]

concatenate([1],[2],[3]) = [1,2,3]
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>newltem</td>
<td>Any type</td>
</tr>
</tbody>
</table>

**Example**

```
insert before( [1,3], 1,2 ) = [2,1,3]
```

**remove( list, position )**

Creates a list with the removed element excluded from the specified position.

**Table 4.40. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
<tr>
<td>position</td>
<td>number</td>
</tr>
</tbody>
</table>

**Example**

```
remove( [1,2,3], 2 ) = [1,3]
```

**reverse( list )**

Returns a reversed list.

**Table 4.41. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

**Example**

```
reverse( [1,2,3] ) = [3,2,1]
```

**index of( list, match )**

Returns indexes matching the element.

**Parameters**

- list of type list
- match of any type

**Table 4.42. Parameters**
### Example

- `index of([1,2,3,2],2) = [2,4]`

### union( list )

Returns a list of all the elements from multiple lists and excludes duplicates.

**Table 4.43. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

### Example

- `union([1,2],[2,3]) = [1,2,3]`

### distinct values( list )

Returns a list of elements from a single list and excludes duplicates.

**Table 4.44. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

### Example

- `distinct values([1,2,3,2,1]) = [1,2,3]`

### flatten( list )

Returns a flattened list.

**Table 4.45. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
</tbody>
</table>

### Example

- `flatten([1,2,3,2,2]) = [1,2,3]`
product( list )

Returns the product of the numbers in the list.

Table 4.46. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list of number elements</td>
</tr>
</tbody>
</table>

Alternative signature

| product( n1, n2, ..., nN ) |

Examples

| product( [2, 3, 4] ) = 24 |
| product( 2, 3, 4 ) = 24 |

median( list )

Returns the median of the numbers in the list. If the number of elements is odd, the result is the middle element. If the number of elements is even, the result is the average of the two middle elements.

Table 4.47. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list of number elements</td>
</tr>
</tbody>
</table>

Alternative signature

| median( n1, n2, ..., nN ) |

Examples

| median( 8, 2, 5, 3, 4 ) = 4 |
| median( [6, 1, 2, 3] ) = 2.5 |
| median( [] ) = null |

stddev( list )

Returns the standard deviation of the numbers in the list.

Table 4.48. Parameters
### Alternative signature

```
stddev( n1, n2, ..., nN )
```

### Examples

- `stddev( 2, 4, 7, 5 ) = 2.081665999466132735282297706979931`
- `stddev( [47] ) = null`
- `stddev( 47 ) = null`
- `stddev( [] ) = null`

```
mode( list )
```

Returns the mode of the numbers in the list. If multiple elements are returned, the numbers are sorted in ascending order.

### Table 4.49. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list of number elements</td>
</tr>
</tbody>
</table>

### Alternative signature

```
mode( n1, n2, ..., nN )
```

### Examples

- `mode( 6, 3, 9, 6, 6 ) = [6]`
- `mode( [6, 1, 9, 6, 1] ) = [1, 6]`
- `mode( [] ) = []`

### 4.3.2.5. Numeric functions

The following functions support number operations.

#### decimal( n, scale )

Returns a number with the specified scale.

### Table 4.50. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number</td>
</tr>
</tbody>
</table>
Parameter | Type
--- | ---
scale | number in the range [−6111..6176]

Examples

\[
\begin{align*}
decimal( \frac{1}{3}, 2 ) &= .33 \\
decimal( 1.5, 0 ) &= 2 \\
decimal( 2.5, 0 ) &= 2
\end{align*}
\]

floor( n )

Returns the greatest integer that is less than or equal to the specified number.

Table 4.51. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number</td>
</tr>
</tbody>
</table>

Examples

\[
\begin{align*}
floor( 1.5 ) &= 1 \\
floor( -1.5 ) &= -2
\end{align*}
\]

ceiling( n )

Returns the smallest integer that is greater than or equal to the specified number.

Table 4.52. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number</td>
</tr>
</tbody>
</table>

Examples

\[
\begin{align*}
ceiling( 1.5 ) &= 2 \\
ceiling( -1.5 ) &= -1
\end{align*}
\]

abs( n )

Returns the absolute value.

Table 4.53. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number, days and time duration, or years and months duration</td>
</tr>
</tbody>
</table>
Examples

modulo( dividend, divisor )

Returns the remainder of the division of the dividend by the divisor. If either the dividend or divisor is negative, the result is of the same sign as the divisor.

NOTE

This function is also expressed as modulo(dividend, divisor) = dividend - divisor*floor(dividend/divisor).

Table 4.54. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dividend</td>
<td>number</td>
</tr>
<tr>
<td>divisor</td>
<td>number</td>
</tr>
</tbody>
</table>

Examples

modulo( 12, 5 ) = 2
modulo( -12, -5 ) = 3
modulo( 12,-5 ) = -3
modulo( -12,-5 ) = -2
modulo( 10.1, 4.5 ) = 1.1
modulo( -10.1, 4.5 ) = 3.4
modulo( 10.1, -4.5 ) = -3.4
modulo( -10.1, -4.5 ) = -1.1

sqrt( number )

Returns the square root of the specified number.

Table 4.55. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number</td>
</tr>
</tbody>
</table>

Example

sqrt( 16 ) = 4
log( number )
Returns the logarithm of the specified number.

Table 4.56. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number</td>
</tr>
</tbody>
</table>

Example

decimal( log( 10 ), 2 ) = 2.30

exp( number )
Returns Euler’s number $e$ raised to the power of the specified number.

Table 4.57. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number</td>
</tr>
</tbody>
</table>

Example

decimal( exp( 5 ), 2 ) = 148.41

odd( number )
Returns true if the specified number is odd.

Table 4.58. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number</td>
</tr>
</tbody>
</table>

Examples

odd( 5 ) = true
odd( 2 ) = false

even( number )
Returns true if the specified number is even.

Table 4.59. Parameters
4.3.2.6. Date and time functions

The following functions support date and time operations.

\[ \text{is( value1, value2 )} \]

Returns \text{true} if both values are the same element in the FEEL semantic domain.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>Any type</td>
</tr>
<tr>
<td>value2</td>
<td>Any type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>even( 5 ) = false</td>
</tr>
<tr>
<td>even ( 2 ) = true</td>
</tr>
</tbody>
</table>

\[ \text{is( date( "2012-12-25" ), time( "23:00:50" ) )} = \text{false} \]
\[ \text{is( date( "2012-12-25" ), date( "2012-12-25" ) )} = \text{true} \]
\[ \text{is( time( "23:00:50z" ), time( "23:00:50" ) )} = \text{false} \]

4.3.2.7. Range functions

The following functions support temporal ordering operations to establish relationships between single scalar values and ranges of such values. These functions are similar to the components in the Health Level Seven (HL7) International Clinical Quality Language (CQL) 1.4 syntax.

\[ \text{before( )} \]

Returns \text{true} when an element A is before an element B and when the relevant requirements for evaluating to \text{true} are also met.

\text{Signatures}

a. \text{before( point1 point2 )}

b. \text{before( point range )}

c. \text{before( range point )}
Requirements for evaluating to true

a. \( \text{point1} < \text{point2} \)

b. \( \text{point} < \text{range.start} \) or \(( \text{point} = \text{range.start} \) and \( \neg(\text{range.start} \text{ included}) \) \)

c. \( \text{range.end} < \text{point} \) or \(( \text{range.end} = \text{point} \) and \( \neg(\text{range.end} \text{ included}) \) \)

d. \( \text{range1.end} < \text{range2.start} \) or \((( \neg(\text{range1.end} \text{ included}) \) or \( \neg(\text{range2.start} \text{ included}) \) \)

Examples

\[
\begin{align*}
\text{before}(1, 10) &= \text{true} \\
\text{before}(10, 1) &= \text{false} \\
\text{before}(1, [1..10]) &= \text{false} \\
\text{before}(1, (1..10]) &= \text{true} \\
\text{before}(1, [5..10]) &= \text{true} \\
\text{before}([1..10], 10) &= \text{false} \\
\text{before}([1..10], 10) &= \text{true} \\
\text{before}([1..10], 15) &= \text{true} \\
\text{before}([1..10], [15..20]) &= \text{true} \\
\text{before}([1..10], [10..20]) &= \text{false} \\
\text{before}([1..10], [10..20]) &= \text{true} \\
\text{before}([1..10], (10..20]) &= \text{true}
\end{align*}
\]

after( )

Returns \text{true} when an element \( A \) is after an element \( B \) and when the relevant requirements for evaluating to \text{true} are also met.

Signatures

a. \( \text{after}(\text{point1} \text{point2}) \)

b. \( \text{after}(\text{point} \text{range}) \)

c. \( \text{after}(\text{range}, \text{point}) \)

d. \( \text{after}(\text{range1} \text{range2}) \)

Requirements for evaluating to true

a. \( \text{point1} > \text{point2} \)

b. \( \text{point} > \text{range.end} \) or \(( \text{point} = \text{range.end} \) and \( \neg(\text{range.end} \text{ included}) \) \)

c. \( \text{range.start} > \text{point} \) or \(( \text{range.start} = \text{point} \) and \( \neg(\text{range.start} \text{ included}) \) \)

d. \( \text{range1.start} > \text{range2.end} \) or \((( \neg(\text{range1.start} \text{ included}) \) or \( \neg(\text{range2.end} \text{ included}) \) \)

Examples
meets( )

Returns true when an element A meets an element B and when the relevant requirements for evaluating to true are also met.

Signatures

a. meets( range1, range2 )

Requirements for evaluating to true

a. range1.end included and range2.start included and range1.end = range2.start

Examples

meets( [1..5], [5..10] ) = true
meets( [1..5), [5..10] ) = false
meets( [1..5], (5..10] ) = false
meets( [1..5], [6..10] ) = false

met by( )

Returns true when an element A is met by an element B and when the relevant requirements for evaluating to true are also met.

Signatures

a. met by( range1, range2 )

Requirements for evaluating to true

a. range1.start included and range2.end included and range1.start = range2.end

Examples

met by( [5..10], [1..5] ) = true
met by( [5..10], [1..5) ) = false
met by( (5..10], [1..5] ) = false
met by( [6..10], [1..5] ) = false
overlaps( )

Returns true when an element A overlaps an element B and when the relevant requirements for evaluating to true are also met.

Signatures

a. overlaps( range1, range2 )

Requirements for evaluating to true

a. ( range1.end > range2.start or (range1.end = range2.start and (range1.end included or range2.end included)) ) and ( range1.start < range2.end or (range1.start = range2.end and range1.start included and range2.end included) )

Examples

overlaps( [1..5], [3..8] ) = true
overlaps( [3..8], [1..5] ) = true
overlaps( [1..8], [3..5] ) = true
overlaps( [3..5], [1..8] ) = true
overlaps( [1..5], [6..8] ) = false
overlaps( [6..8], [1..5] ) = false
overlaps( [1..5], [5..8] ) = true
overlaps( [1..5], [5..8] ) = false
overlaps( [1..5], [5..8] ) = false
overlaps( [1..5], [5..8] ) = false
overlaps( [5..8], [1..5] ) = true
overlaps( [5..8], [1..5] ) = false
overlaps( [5..8], [1..5] ) = false
overlaps( [5..8], [1..5] ) = false

overlaps before( )

Returns true when an element A overlaps before an element B and when the relevant requirements for evaluating to true are also met.

Signatures

a. overlaps before( range1 range2 )

Requirements for evaluating to true

a. ( range1.start < range2.start or (range1.start = range2.start and range1.start included and range2.start included) ) and ( range1.end > range2.start or (range1.end = range2.start and range1.end included and range2.start included) ) and ( range1.end < range2.end or (range1.end = range2.end and (not(range1.end included) or range2.end included) )

Examples

overlaps before( [1..5], [3..8] ) = true
overlaps before( [1..5], [6..8] ) = false
overlaps before( [1..5], [5..8] ) = true
overlaps before( [1..5], [5..8] ) = false
overlaps after( )

Returns \textbf{true} when an element A overlaps after an element B and when the relevant requirements for evaluating to \textbf{true} are also met.

Signatures

a. \texttt{overlaps after( range1 range2 )}

Requirements for evaluating to \textbf{true}

a. \((\text{range2.start} < \text{range1.start} \text{ or } (\text{range2.start} = \text{range1.start} \text{ and } \text{range2.start included} \text{ and not(\text{range1.start included})}) \text{ and } (\text{range2.end} > \text{range1.start} \text{ or } (\text{range2.end} = \text{range1.start} \text{ and } \text{range2.end included} \text{ and } \text{range1.start included}) \text{ and } (\text{range2.end} < \text{range1.end} \text{ or } (\text{range2.end} = \text{range1.end} \text{ and } (\text{not(range2.end included}) \text{ or } \text{range1.end included})))))

Examples

overlaps after( [3..8], [1..5] ) = true
overlaps after( [6..8], [1..5] ) = false
overlaps after( [5..8], [1..5] ) = true
overlaps after( [5..8], [1..5] ) = false
overlaps after( [1..5], [1..5] ) = true
overlaps after( [1..5], [1..5] ) = true
overlaps after( [1..5], [1..5] ) = false
overlaps after( [1..5], [1..5] ) = false
overlaps after( [1..5], [1..6] ) = false
overlaps after( [1..5], (1..5) ) = false
overlaps after( (1..5], [1..5) ) = false
overlaps after( (1..5), [1..5] ) = false
overlaps after( (1..5], [2..5] ) = false

finishes( )

Returns \textbf{true} when an element A finishes an element B and when the relevant requirements for evaluating to \textbf{true} are also met.

Signatures

a. \texttt{finishes( point, range )}

b. \texttt{finishes( range1, range2 )}

Requirements for evaluating to \textbf{true}

a. \texttt{range.end included and range.end = point}
b. range1.end included = range2.end included and range1.end = range2.end and (range1.start > range2.start or (range1.start = range2.start and (not(range1.start included) or range2.start included)))

Examples

```
finishes( 10, [1..10] ) = true
finishes( 10, [1..10) ) = false
finishes( [5..10], [1..10] ) = true
finishes( [5..10), [1..10] ) = false
finishes( [5..10), [1..10) ) = true
finishes( [1..10], [1..10] ) = true
finishes( (1..10], [1..10) ) = true
```

finished by( )

Returns true when an element A is finished by an element B and when the relevant requirements for evaluating to true are also met.

Signatures

a. finished by( range, point )

b. finished by( range1 range2 )

Requirements for evaluating to true

a. range.end included and range.end = point

b. range1.end included = range2.end included and range1.end = range2.end and (range1.start < range2.start or (range1.start = range2.start and (range1.start included or not(range2.start included))))

Examples

```
finished by( [1..10], 10 ) = true
finished by( [1..10), 10 ) = false
finished by( [1..10], [5..10] ) = true
finished by( [1..10), [5..10] ) = false
finished by( [1..10), [5..10) ) = true
finished by( [1..10], [1..10] ) = true
finished by( [1..10], (1..10] ) = true
```

includes( )

Returns true when an element A includes an element B and when the relevant requirements for evaluating to true are also met.

Signatures

a. includes( range, point )

b. includes( range1, range2 )

Requirements for evaluating to true
Examples of including ranges: 

| Includes( [1..10], 5 ) | true |
| includes([1..10], 12 ) | false |
| includes([1..10], 1 ) | true |
| includes([1..10], 10 ) | true |
| includes([1..10], 1 ) | false |
| includes([1..10], [4..6] ) | true |
| includes([1..10], [1..5] ) | true |
| includes([1..10], (1..5) ) | true |
| includes([1..10], (1..10) ) | true |
| includes([1..10], [5..10] ) | true |
| includes([1..10], [1..10) ) | true |
| includes([1..10], [1..10] ) | true |

Examples of during: 

during( 5, [1..10] ) = true 
during( 12, [1..10] ) = false 
during( 1, [1..10] ) = true 
during( 10, [1..10] ) = true 
during( 1, (1..10) ) = false 
during( 10, [1..10) ) = false 
during( [4..6], [1..10] ) = true 
during( [1..5], [1..10] ) = true 
during( (1..5], (1..10) ) = true
starts( )
Returns true when an element A starts an element B and when the relevant requirements for evaluating to true are also met.

Signatures
a. starts( point, range )
b. starts( range1, range2 )

Requirements for evaluating to true
a. range.start = point and range.start included
b. range1.start = range2.start and range1.start included = range2.start included and (range1.end < range2.end or (range1.end = range2.end and (not(range1.end included) or range2.end included)) )

Examples
starts( 1, [1..10] ) = true
starts( 1, (1..10) ) = false
starts( 2, [1..10] ) = false
starts( [1..5], [1..10] ) = true
starts( [1..5], (1..10) ) = true
starts( [1..5], [1..10] ) = false
starts( [1..5], (1..10) ) = false
starts( [1..10], [1..10] ) = true
starts( [1..10], (1..10) ) = true
starts( (1..10), (1..10) ) = true

started by( )
Returns true when an element A is started by an element B and when the relevant requirements for evaluating to true are also met.

Signatures
a. started by( range, point )
b. started by( range1, range2 )

Requirements for evaluating to true
a. range.start = point and range.start included
b. \[ \text{range1.start} = \text{range2.start} \text{ and } \text{range1.start included} = \text{range2.start included} \text{ and } (\not\text{range2.end < range1.end} \text{ or } (\text{range2.end} = \text{range1.end} \text{ and } (\not\text{range2.end included}) \text{ or } \text{range1.end included})) \]

**Examples**

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>started by( [1..10], 1 )</td>
<td>true</td>
</tr>
<tr>
<td>started by( (1..10], 1 )</td>
<td>false</td>
</tr>
<tr>
<td>started by( [1..10], 2 )</td>
<td>false</td>
</tr>
<tr>
<td>started by( [1..10], [1..5] )</td>
<td>true</td>
</tr>
<tr>
<td>started by( (1..10], (1..5] )</td>
<td>true</td>
</tr>
<tr>
<td>started by( [1..10], (1..5] )</td>
<td>false</td>
</tr>
<tr>
<td>started by( (1..10], [1..5] )</td>
<td>false</td>
</tr>
<tr>
<td>started by( [1..10], [1..10] )</td>
<td>true</td>
</tr>
<tr>
<td>started by( (1..10], (1..10) )</td>
<td>true</td>
</tr>
</tbody>
</table>

**coincides( )**

Returns true when an element \( A \) coincides with an element \( B \) and when the relevant requirements for evaluating to true are also met.

**Signatures**

a. \( \text{coincides( point1, point2 )} \)

b. \( \text{coincides( range1, range2 )} \)

**Requirements for evaluating to true**

a. \( \text{point1 = point2} \)

b. \[ \text{range1.start} = \text{range2.start} \text{ and } \text{range1.start included} = \text{range2.start included} \text{ and } \text{range1.end} = \text{range2.end} \text{ and } \text{range1.end included} = \text{range2.end included} \]

**Examples**

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>coincides( 5, 5 )</td>
<td>true</td>
</tr>
<tr>
<td>coincides( 3, 4 )</td>
<td>false</td>
</tr>
<tr>
<td>coincides( [1..5], [1..5] )</td>
<td>true</td>
</tr>
<tr>
<td>coincides( (1..5], [1..5] )</td>
<td>false</td>
</tr>
<tr>
<td>coincides( [1..5], [2..6] )</td>
<td>false</td>
</tr>
</tbody>
</table>

**4.3.2.8. Temporal functions**

The following functions support general temporal operations.

**day of year( date )**

Returns the Gregorian number of the day of the year.

**Table 4.61. Parameters**
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date or date and time</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
day of year( date(2019, 9, 17) ) = 260
```

**day of week( date )**

Returns the Gregorian day of the week: "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", or "Sunday".

**Table 4.62. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date or date and time</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
day of week( date(2019, 9, 17) ) = "Tuesday"
```

**month of year( date )**


**Table 4.63. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date or date and time</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
month of year( date(2019, 9, 17) ) = "September"
```

**month of year( date )**

Returns the Gregorian week of the year as defined by ISO 8601.

**Table 4.64. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date or date and time</td>
</tr>
</tbody>
</table>

**Examples**
4.3.2.9. Sort functions

The following functions support sorting operations.

sort( list, precedes )

Returns a list of the same elements but ordered according to the sorting function.

Table 4.65. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>list</td>
</tr>
<tr>
<td>precedes</td>
<td>function</td>
</tr>
</tbody>
</table>

Example

sort( list: [3, 1, 4, 5, 2], precedes: function(x, y) x < y ) = [1, 2, 3, 4, 5]

4.3.2.10. Context functions

The following functions support context operations.

get value( m, key )

Returns the value from the context for the specified entry key.

Table 4.66. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>context</td>
</tr>
<tr>
<td>key</td>
<td>string</td>
</tr>
</tbody>
</table>

Examples

get value( {key1 : "value1"}, "key1" ) = "value1"
get value( {key1 : "value1"}, "unexistent-key" ) = null
get entries( m )

| week of year( date(2019, 9, 17) ) = 38 |
| week of year( date(2003, 12, 29) ) = 1 |
| week of year( date(2004, 1, 4) ) = 1 |
| week of year( date(2005, 1, 1) ) = 53 |
| week of year( date(2005, 1, 3) ) = 1 |
| week of year( date(2005, 1, 9) ) = 1 |
Returns a list of key-value pairs for the specified context.

Table 4.67. Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>context</td>
</tr>
</tbody>
</table>

Example

```java
get entries( {key1 : "value1", key2 : "value2"} ) = [ { key : "key1", value : "value1" }, {key : "key2", value : "value2"} ]
```

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, Red Hat Decision Manager removes the limitation on the use of consecutive spaces. Red Hat Decision Manager 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 Red Hat Decision Manager.

Red Hat Decision Manager 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 Red Hat Decision Manager 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

**NOTE**

Red Hat Decision Manager does not provide boxed list expressions in Business Central, 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 Red Hat Decision Manager, see Section 4.3.1, “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:
The following decision table determines the next step in a lending strategy for applicants depending on applicant loan eligibility and the bureau call type:

**Figure 4.4. Decision table for lending strategy**

<table>
<thead>
<tr>
<th>U</th>
<th>Eligibility (String)</th>
<th>BureauCallType (String)</th>
<th>Strategy (Strategy)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;INELIGIBLE&quot;</td>
<td>-</td>
<td>&quot;DECLINE&quot;</td>
<td>Disregard BureauCallType when ineligible.</td>
</tr>
<tr>
<td>2</td>
<td>&quot;ELIGIBLE&quot;</td>
<td>&quot;FULL&quot;, &quot;MINI&quot;</td>
<td>&quot;BUREAU&quot;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&quot;ELIGIBLE&quot;</td>
<td>&quot;NONE&quot;</td>
<td>&quot;THROUGH&quot;</td>
<td></td>
</tr>
</tbody>
</table>

The following decision table determines applicant qualification for a loan as the concluding decision node in a loan prequalification decision model:
Figure 4.5. Decision table for loan prequalification

Red Hat Decision Manager 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 Red Hat Decision Manager, see Designing a decision service using spreadsheet 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, 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.
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 4.6. Boxed literal expression for minimum PITI value**

```feelm
decimal( acceptable rate, 2 )
```

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 4.7. Boxed literal expression for matching online dating candidates**

```feelm
sort( Candidate Souls, function( c1, c2 ) c1.Score >= c2.Score )
```

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 4.8. Boxed context expression for flight passenger waiting list

<table>
<thead>
<tr>
<th>#</th>
<th>Prioritized Waiting List (Context)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Flight List[ Status = &quot;cancelled&quot; ].Flight Number</td>
</tr>
<tr>
<td>2</td>
<td>Passenger List[ list contains( Cancelled Flights, Flight Number ) ]</td>
</tr>
<tr>
<td>&lt;result&gt;</td>
<td>sort( Waiting List, Passenger Priority )</td>
</tr>
</tbody>
</table>

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:

Figure 4.9. Boxed context expression for front-end client PITI ratio

<table>
<thead>
<tr>
<th>#</th>
<th>Front End Ratio (Context)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PITI</td>
</tr>
<tr>
<td>1</td>
<td>pmt (number)</td>
</tr>
<tr>
<td>2</td>
<td>tax (number)</td>
</tr>
<tr>
<td>3</td>
<td>insurance (number)</td>
</tr>
<tr>
<td>4</td>
<td>income (number)</td>
</tr>
<tr>
<td>&lt;result&gt;</td>
<td>if client PITI &lt;= Lender Acceptable PITI[] then &quot;Sufficient&quot; else &quot;Insufficient&quot;</td>
</tr>
</tbody>
</table>

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:

**Figure 4.10. Boxed relation expression with employee information**

### Employee Information *(Relation)*

<table>
<thead>
<tr>
<th>#</th>
<th>Name <em>(string)</em></th>
<th>Dept <em>(string)</em></th>
<th>Salary <em>(number)</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;John&quot;</td>
<td>&quot;Sales&quot;</td>
<td>100000</td>
</tr>
<tr>
<td>2</td>
<td>&quot;Mary&quot;</td>
<td>&quot;Finances&quot;</td>
<td>120000</td>
</tr>
</tbody>
</table>

**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:

**Figure 4.11. Boxed function expression for flight capacity**

### Flight Capacity *(Function)*

```plaintext
Flight Capacity *(boolean)*

(flight, rebooked list)

flight.Capacity > count( rebooked list[ Flight Number = flight.Flight Number ] )
```

The following boxed function expression contains a basic Java function as a context expression for determining absolute value in a decision model calculation:
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 4.13. Boxed function expression for installment calculation in business knowledge model**

**InstallmentCalculation** (*Function* )

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:
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:
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.

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 4.17. Boxed list expression for approved credit score agencies**

<table>
<thead>
<tr>
<th></th>
<th>Approved credit score agencies (List)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Acme Agency, Inc.&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;Top Scores, Inc.&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;Global Scoring, Inc.&quot;</td>
</tr>
</tbody>
</table>

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 4.18. Boxed list expression using FEEL logic for approved credit score agency status**

<table>
<thead>
<tr>
<th></th>
<th>Approved credit score agencies (List)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Acme Agency&quot; + suffix</td>
</tr>
<tr>
<td>2</td>
<td>&quot;Top Scores&quot; + suffix</td>
</tr>
<tr>
<td>3</td>
<td>&quot;Global Scoring&quot; + suffix</td>
</tr>
</tbody>
</table>

### 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 4.19. 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:

**Figure 4.20. Boxed context expression example for prioritized wait list**

### Prioritized Waiting List (Context)

<table>
<thead>
<tr>
<th>#</th>
<th>Cancelled Flights (FlightNumberList)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Flight List[ Status = &quot;cancelled&quot; ].Flight Number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>Waiting List (PassengerTable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Passenger List[ list contains( Cancelled Flights, Flight Number ) ]</td>
</tr>
</tbody>
</table>

| <result> | sort( Waiting List, Passenger Priority ) |

Some boxed expressions are more complex with greater detail and calculation, such as the following business knowledge model for reassigning the next delayed passenger:

**Figure 4.21. Boxed function expression for passenger reassignment**

### Reassign Next Passenger (Function)

<table>
<thead>
<tr>
<th>F</th>
<th>Next Passenger (Passenger)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Waiting List[3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>Original Flight (Flight)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Best Alternate Flight (Flight)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>Reassigned Passenger (Passenger)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Name (string) = Next Passenger.Name</td>
</tr>
<tr>
<td>2</td>
<td>Status (string) = Next Passenger.Status</td>
</tr>
<tr>
<td>3</td>
<td>Miles (number) = Next Passenger.Miles</td>
</tr>
<tr>
<td>4</td>
<td>Flight Number (string) = Best Alternate Flight.Flight Number</td>
</tr>
</tbody>
</table>

| <result> | Select expression |

<table>
<thead>
<tr>
<th>5</th>
<th>Remaining Waiting List (PassengerTable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Updated Reassigned Passengers List (PassengerTable)</td>
</tr>
</tbody>
</table>

| <result> | if count( Remaining Waiting List ) > 0 then Reassign Next Passenger( Remaining Waiting List, Updated Reassigned Passengers List, Flights ) else Updated Reassigned Passengers List |

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"
xmlns:dmn:definitions id="_0019_flight_rebooking" name="0019-flight-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:definitions>
CHAPTER 4. DECISION MODEL AND NOTATION (DMN)

<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:decision>

<dmn:context>
  <dmn:variable name="Cancelled Flights" typeRef="tFlightNumberList"/>
  <dmn:literalExpression>
    Flight List[ Status = "cancelled "].Flight Number</dmn:literalExpression>
</dmn:contextEntry>

<dmn:variable name="Waiting List" typeRef="tPassengerTable"/>
<dmn:literalExpression>
  Passenger List[ list contains( Cancelled Flights, Flight Number ) ]</dmn:literalExpression>
</dmn:contextEntry>

<dmn:literalExpression>
  sort( Waiting List, passenger priority )</dmn:literalExpression>
</dmn:contextEntry>

<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:decision>

<dmn:invocation>
  <dmn:literalExpression>
    reassign next passenger</dmn:literalExpression>
</dmn:invocation>
</dmn:binding>
<dmn:binding>
  <dmn:parameter name="Waiting List"/>
  <dmn:literalExpression>
    Prioritized Waiting List</dmn:literalExpression>
</dmn:binding>
<dmn:binding>
  <dmn:parameter name="Reassigned Passengers List"/>
  <dmn:literalExpression>
    []</dmn:literalExpression>
</dmn:binding>
<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>
CHAPTER 4. DECISION MODEL AND NOTATION (DMN)

```xml
<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:outputEntry>
<dmn:rule>
<dmn:decisionTable>
<dmn:encapsulatedLogic>
<dmn:variable name="passenger priority" typeRef="feel:boolean"/>
</dmn:businessKnowledgeModel>
</dmn:rule>
</dmn:decisionTable>
<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:literalExpression>
Best Alternate Flight.

**Flight Number**

```xml
dmn:text>Best Alternate Flight. Flight Number</dmn:text>
<dmn:context>
<dmn:text><dmn:variable name="Remaining Waiting List" typeRef="tPassengerTable"/>
<dmn:literalExpression>
<dmn:text>remove( Waiting List, 1 )</dmn:text>
</dmn:literalExpression>
<dmn:contextEntry>
<dmn:text><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: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:contextEntry>
</dmn:context>
</dmn:encapsulatedLogic>
<dmn:variable name="reassign next passenger" typeRef="tPassengerTable"/>
<dmn:knowledgeRequirement>
<dmn:requiredKnowledge href="#b_HasCapacity"/>
</dmn:knowledgeRequirement>
<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 RED HAT DECISION MANAGER

Red Hat Decision Manager provides design and runtime support for DMN 1.2 models at conformance level 3, and runtime-only support for DMN 1.1 and 1.3 models at conformance level 3. You can integrate DMN models with your Red Hat Decision Manager decision services in several ways:

- Design your DMN models directly in Business Central using the DMN designer.
- Import DMN files into your project in Business Central (Menu → Design → Projects → Import Asset). Any DMN 1.1 and 1.3 models (do not contain DMN 1.3 features) that you import into Business Central, open in the DMN designer, and save are converted to DMN 1.2 models.
- Package DMN files as part of your project knowledge JAR (KJAR) file without Business Central.

The following table summarizes the design and runtime support for each DMN version in Red Hat Decision Manager:

Table 5.1. DMN support in Red Hat Decision Manager

<table>
<thead>
<tr>
<th>DMN version</th>
<th>DMN engine support</th>
<th>DMN modeler support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Execution</td>
<td>Open</td>
</tr>
<tr>
<td>DMN 1.1</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>DMN 1.2</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>DMN 1.3</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

In addition to all DMN conformance level 3 requirements, Red Hat Decision Manager also includes enhancements and fixes to FEEL and DMN model components to optimize the experience of implementing DMN decision services with Red Hat Decision Manager. From a platform perspective, DMN models are like any other business asset in Red Hat Decision Manager, such as DRL files or spreadsheet decision tables, that you can include in your Red Hat Decision Manager project and deploy to KIE Server in order to start your DMN decision services.

For more information about including external DMN files with your Red Hat Decision Manager project packaging and deployment method, see Packaging and deploying a Red Hat Decision Manager project.

5.1. CONFIGURABLE DMN PROPERTIES IN RED HAT DECISION MANAGER

Red Hat Decision Manager provides the following DMN properties that you can configure when you execute your DMN models on KIE Server or on your client application. You can configure some of these properties using the kmodule.xml file in your Red Hat Decision Manager project when you deploy your project on KIE Server.
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 decision 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 decision 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 decision 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 when deploying your project on KIE Server, modify this property in the kmodule.xml file of your project. This approach is helpful when the listener is specific to your project and when the configuration must be applied in KIE Server only to your deployed project.

<kmodule xmlns="http://www.drools.org/xsd/kmodule">
  <configuration>
    <property key="org.kie.dmn.runtime.listeners.mylistener" value="org.acme.MyDMNListener"/>
  </configuration>
</kmodule>
To configure this property globally for your Red Hat Decision Manager environment, modify this property using a command terminal or any other global application configuration mechanism. This approach is helpful when the decision engine is embedded as part of your Java application.

- 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 decision engine, but subsequent compilations are more efficient. Default value: false

- Dorg.kie.dmn.compiler.execmodel=true

5.2. CONFIGURABLE DMN VALIDATION IN RED HAT DECISION MANAGER

By default, the kie-maven-plugin component in the pom.xml file of your Red Hat Decision Manager 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

```xml
<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>VALIDATE_SCHEMA,VALIDATE_MODEL</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>ANALYZE_DECISION_TABLE</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_SCHEMA</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>
```

**NOTE**

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 BUSINESS CENTRAL

You can use the DMN designer in Business Central to design DMN decision requirements diagrams (DRDs) and define decision logic for a complete and functional DMN decision model. Red Hat Decision Manager provides design and runtime 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 Red Hat Decision Manager. Red Hat Decision Manager also provides runtime-only support for DMN 1.1 and 1.3 models at conformance level 3, but any DMN 1.1 and 1.3 models (do not contain DMN 1.3 features) that you import into Business Central, open in the DMN designer, and save are converted to DMN 1.2 models.

Procedure

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

2. Create or import a DMN file in your Business Central project.
   To create a DMN file, click Add Asset → DMN, enter an informative DMN model name, select the appropriate Package, and click Ok.

   To import an existing DMN file, click Import Asset, enter the DMN model name, select the appropriate Package, select the DMN file to upload, and click Ok.

   The new DMN file is now listed in the DMN panel of the Project Explorer, and the DMN decision requirements diagram (DRD) canvas appears.

NOTE

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.

3. 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:
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.

4. In the DMN designer canvas, double-click the new DRD node to enter an informative node name.
5. 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 6.2. Opening a new decision node 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 6.4. 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.
6. 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 6.6. Decision node decision table for credit score rating**

<table>
<thead>
<tr>
<th>U</th>
<th>Credit Score.FICO (number)</th>
<th>Credit Score Rating (Credit_Score_Rating)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&gt;= 750</td>
<td>&quot;Excellent&quot;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[700..750)</td>
<td>&quot;Good&quot;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[650..700)</td>
<td>&quot;Fair&quot;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[500..650)</td>
<td>&quot;Poor&quot;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt; 600</td>
<td>&quot;Bad&quot;</td>
<td></td>
</tr>
</tbody>
</table>

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:
7. After you define the decision logic for the selected node, click Back to "<MODEL_NAME>" to return to the DRD view.

8. 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.
9. Continue adding and defining the remaining DRD components of your decision model. Periodically click **Save** in the DMN designer to save your work.

**NOTE**

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.

10. 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:
The following is an example DRD for a phone call handling decision model using a reusable 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 BUSINESS CENTRAL

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 DMN designer in Business Central to define decision logic for your DRD components using built-in boxed expressions.

Prerequisites

- A DMN file is created or imported in Business Central.

Procedure

1. In Business Central, go to **Menu → Design → Projects**, click the project name, and select the DMN file you want to modify.

2. 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 6.11. Opening a new decision node boxed expression**

   - **Figure 6.12. Opening a new business knowledge model boxed expression**

   ![Credit Score Rating (Undefined)](image)
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.

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.
3. 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.

4. 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.

5. 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 6.15. Managing data types for a column header value

6. 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 6.16. Adding a new data type

7. Click Add Constraints, select Enumeration from the drop-down options, and add the following constraints:
   - "Excellent"
   - "Good"
   - "Fair"
- "Poor"
- "Bad"

Figure 6.17. 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:
For information about constraint types and syntax requirements for the specified data type, see the Decision Model and Notation specification.

8. Click OK to save the constraints and click the check mark to the right of the data type to save the data type.

9. 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.

10. 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)

<table>
<thead>
<tr>
<th>U</th>
<th>Credit Score FICO (number)</th>
<th>Credit Score Rating (Credit Score Rating)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&gt;= 750</td>
<td>&quot;Excellent&quot;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[700..750)</td>
<td>&quot;Good&quot;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[650..700)</td>
<td>&quot;Fair&quot;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[500..650)</td>
<td>&quot;Poor&quot;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt; 600</td>
<td>&quot;Bad&quot;</td>
<td></td>
</tr>
</tbody>
</table>

11. 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.
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:

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:

**Figure 6.22. Boxed context expression for front-end client PITI ratio**

Front End Ratio (Context)

<table>
<thead>
<tr>
<th>#</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Client PITI (number)</td>
</tr>
<tr>
<td>2</td>
<td>Tax (number)</td>
</tr>
<tr>
<td>3</td>
<td>Insurance (number)</td>
</tr>
<tr>
<td>4</td>
<td>Income (number)</td>
</tr>
</tbody>
</table>

**PITI**

\[
\text{PITI} = \frac{(\text{Requested Product Amount} \times (\text{Requested Product Rate}/100)/12)}{1-\left(1+(\text{Requested Product Rate}/100)/12\right)^{-\text{Requested Product Term}}}
\]

Aplicant Data. Monthly. Tax

Aplicant Data. Monthly. Insurance

Aplicant Data. Monthly. Income

\[\text{if Client PITI <= Lender Acceptable PITI} \text{ then "Sufficient" else "Insufficient"}\]

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 6.23. Boxed function expression for installment calculation in business knowledge model**

InstallmentCalculation (Function)

<table>
<thead>
<tr>
<th>F</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MonthlyFee (number)</td>
</tr>
<tr>
<td>2</td>
<td>MonthlyRepayment (number)</td>
</tr>
</tbody>
</table>

\[(\text{ProductType, Rate, Term, Amount})
\]

\[\text{if ProductType = "STANDARD LOAN" then 20.00 else if ProductType = "SPECIAL LOAN" then 25.00 else null}
\]

\[\text{(Amount } \times \text{Rate}/12) / (1 - (1 + \text{Rate}/12)^{-\text{Term}})
\]

\[\text{MonthlyRepayment} - \text{MonthlyFee}
\]

For more information and examples of each boxed expression type, see Section 4.4, “DMN decision logic in boxed expressions”.

### 6.2. CREATING CUSTOM DATA TYPES FOR DMN BOXED EXPRESSIONS IN BUSINESS CENTRAL

In DMN boxed expressions in Business Central, 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 Business Central.

**Procedure**

1. In Business Central, go to **Menu → Design → Projects**, click the project name, and select the DMN file you want to modify.

2. 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.

3. 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 6.24. Selecting the logic type for a decision node**

4. 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.
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:
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.

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 6.27. Adding a new data type

If the data type requires a list of items, enable the **List** setting.

6. Click **Add Constraints**, select **Enumeration** from the drop-down options, and add the following constraints:

- "Excellent"
- "Good"
- "Fair"
- "Poor"
- "Bad"

Figure 6.28. 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 6.29. 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, 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.
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:

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:

<table>
<thead>
<tr>
<th>U</th>
<th>Credit Score.FICO (number)</th>
<th>Credit Score Rating (Credit.Score.Rating)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&gt;= 750</td>
<td>&quot;Excellent&quot;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[700..750)</td>
<td>&quot;Good&quot;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[650..700)</td>
<td>&quot;Fair&quot;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[500..550)</td>
<td>&quot;Poor&quot;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt; 600</td>
<td>&quot;Bad&quot;</td>
<td></td>
</tr>
</tbody>
</table>

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:
10. 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.

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:
11. 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.

**Figure 6.35. Decision table for loan prequalification**

<table>
<thead>
<tr>
<th></th>
<th>Credit Score Rating (Credit_Score_Rating)</th>
<th>Back_End_Ratio (Back_End_Ratio)</th>
<th>Front_End_Ratio (Front_End_Ratio)</th>
<th>Loan Pre-Qualification (Loan_Qualification)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Poor”, “Bad”</td>
<td>-</td>
<td>-</td>
<td>&quot;Not qualified&quot;</td>
<td>“Credit Score too low.”</td>
</tr>
<tr>
<td>2</td>
<td>“Insufficient”</td>
<td>&quot;Sufficient&quot;</td>
<td>-</td>
<td>&quot;Not qualified&quot;</td>
<td>“Debt to income ratio is too high.”</td>
</tr>
<tr>
<td>3</td>
<td>&quot;Sufficient&quot;</td>
<td>&quot;Insufficient&quot;</td>
<td>-</td>
<td>&quot;Not qualified&quot;</td>
<td>“Mortgage payment to income ratio is too high.”</td>
</tr>
<tr>
<td>4</td>
<td>“Insufficient”</td>
<td>“Insufficient”</td>
<td>-</td>
<td>&quot;Not qualified&quot;</td>
<td>“Debt to income ratio is too high AND mortgage payment to income ratio is too high.”</td>
</tr>
<tr>
<td>5</td>
<td>“Fair”, “Good”, “Excellent”</td>
<td>&quot;Sufficient&quot;</td>
<td>&quot;Sufficient&quot;</td>
<td>&quot;Qualified&quot;</td>
<td>“The borrower has been successfully prequalified for the requested loan.”</td>
</tr>
</tbody>
</table>

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:
Figure 6.36. Boxed function expression for online dating compatibility

Evaluate Match (Function)

<table>
<thead>
<tr>
<th>F</th>
<th>Evaluated Match (tCandidate)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Lonely Soul, Candidate)</td>
</tr>
<tr>
<td>1</td>
<td>Profile1 (tProfile)</td>
</tr>
<tr>
<td>2</td>
<td>Profile2 (tProfile)</td>
</tr>
<tr>
<td>3</td>
<td>Is Match (boolean)</td>
</tr>
<tr>
<td>4</td>
<td>Score (number)</td>
</tr>
</tbody>
</table>

Select expression

---

Figure 6.37. Custom data type definitions for online dating compatibility

Custom Data Types

- **tProfiles (tProfile)**: List, Yes
- **tCandidate (Structure)**
  - **Profile1 (tProfile)**
    - **Name (string)**
    - **Gender (tGender)**
    - **City (string)**
6.3. INCLUDED MODELS IN DMN FILES IN BUSINESS CENTRAL

In the DMN designer in Business Central, 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 Business Central.

6.3.1. Including other DMN models within a DMN file in Business Central

In Business Central, 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 Business Central.

Prerequisites

- The DMN models are created or imported (as .dmn files) in the same project in Business Central as the DMN file in which you want to include the models.

Procedure

1. In Business Central, go to Menu → Design → Projects, click the project name, and select the DMN file you want to modify.

2. In the DMN designer, click the Included Models tab.
3. 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 6.39. 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 6.40. 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:
4. 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 6.42. 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.
IMPORTANT

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 Business Central

In Business Central, 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 Business Central.

Prerequisites

- The PMML models are imported (as .pmml files) in the same project in Business Central 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</artifactId>
  <version>${rhdm.version}</version>
  <scope>provided</scope>
</dependency>

<!-- Alternative dependencies for JPMML Evaluator, override `kie-pmml` dependency -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-dmn-jpmml</artifactId>
  <version>${rhdm.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>
```
To access the project pom.xml file in Business Central, you can select any existing asset in the project and then in the Project Explorer menu on the left side of the screen, click the Customize View gear icon and select Repository View → pom.xml.

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 dependency are both present, the kie-pmml dependency is disabled. For information about JPMML licensing terms, see Openscoring.io.

**IMPORTANT**

The legacy kie-pmml dependency is deprecated with Red Hat Decision Manager 7.10.0 and will be replaced by kie-pmml-trusty dependency in a future Red Hat Decision Manager release.

**NOTE**

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

Example BOM dependency:

```xml
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.10.0.redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see What is the mapping between RHDM product and maven library version?.

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 ~/kie-server.war/WEB-INF/lib and ~/business-central.war/WEB-INF/lib directories in your Red Hat Decision Manager distribution:

- 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 in KIE Server and Business Central.
IMPORTANT

Red Hat supports integration with the Java Evaluator API for PMML (JPMML) for PMML execution in Red Hat Decision Manager. However, Red Hat does not support the JPMML libraries directly. If you include JPMML libraries in your Red Hat Decision Manager distribution, see the Openscoring.io licensing terms for JPMML.

3. In Business Central, go to Menu → Design → Projects, click the project name, and select the DMN file you want to modify.

4. In the DMN designer, click the Included Models tab.

5. 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 6.43. Including a PMML model

The PMML model is added to this DMN file:
6. 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 6.45. Opening a new decision node boxed expression**
7. Set the expression type to **Function** (default for business knowledge model nodes), click the top-left function cell, and select **PMML**.

8. 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:
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 BUSINESS CENTRAL

For complex DMN models, you can use the DMN designer in Business Central 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 Business Central. For information about creating DRDs, see Chapter 6, Creating and editing DMN models in Business Central.

Procedure

1. In Business Central, 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 6.49. 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.
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 6.51. 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 6.52. Multiple nested DRDs for loan prequalification

Figure 6.53. Overview of front end ratio DRD
Figure 6.54. DRD for front end ratio

Figure 6.55. Overview of credit score rating DRD
Figure 6.56. DRD for credit score rating

Figure 6.57. Overview of back end ratio DRD
6.5. DMN MODEL DOCUMENTATION IN BUSINESS CENTRAL

In the DMN designer in Business Central, 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.
6.6. DMN DESIGNER NAVIGATION AND PROPERTIES IN BUSINESS CENTRAL

The DMN designer in Business Central 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 6.60. Project Explorer view
Figure 6.61. Decision Navigator view

- **Loan Pre-Qualification**
  - Applicant Data
- **Back End Ratio**
  - Context
- **Credit Score Rating**
  - Decision Table
  - Credit Score
- **DTI**
  - $f()$ Function
- **Front End Ratio**
  - Context
- **Lender Acceptable DTI**
  - $f()$ Function
- **Lender Acceptable PITI**
  - $f()$ Function
- **Loan Pre-Qualification**
  - Decision Table
- **Passenger Priority**
  - $f()$ Function
- **PITI**
  - $f()$ Function
  - Requested Product
NOTE

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:
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:
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:
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 6.65. Decision Service details

- **Description**: [Input Data](#), [Encapsulated Decisions](#), [Output Decisions](#)
- **Documentation Links**: None
- **Name**: Call can be handled
- **Decision Service details**
  - **Input Data**
    - Phone number (undefined)
    - Office (undefined)
  - **Encapsulated Decisions**
    - Banned phone numbers (undefined)
    - Is banned (undefined)
    - Is office open (undefined)
  - **Output Decisions**
    - Call conditions satisfied (undefined)
CHAPTER 7. DMN MODEL EXECUTION

You can create or import DMN files in your Red Hat Decision Manager project using Business Central or package the DMN files as part of your project knowledge JAR (KJAR) file without Business Central. After you implement your DMN files in your Red Hat Decision Manager project, you can execute the DMN decision service by deploying the KIE container that contains it to KIE Server for remote access or by manipulating the KIE container 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.

For information about including external DMN assets with your project packaging and deployment method, see Packaging and deploying a Red Hat Decision Manager project.

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 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. Ideally, you have built the DMN project as an executable model for more efficient execution:

  mvn clean install -DgenerateDMNModel=yes

For more information about project packaging and deployment and executable models, see Packaging and deploying a Red Hat Decision Manager project.

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>${rhdm.version}</version>
   </dependency>

   <!-- Required if not using classpath KIE container -->
   <dependency>
     <!-- Required for the DMN runtime API -->
     <dependency>
The `<version>` is the Maven artifact version for Red Hat Decision Manager currently used in your project (for example, 7.48.0.Final-redhat-00004).

**NOTE**

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

Example BOM dependency:

```xml
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.10.0.redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see What is the mapping between RHDM product and maven library version?.

2. Create a KIE container from `classpath` or `Releaseld`:

```java
KieServices kieServices = KieServices.Factory.get();
ReleaseId releaseId = kieServices.newReleaseId("org.acme", "my-kjar", "1.0.0");
KieContainer kieContainer = kieServices.newKieContainer(releaseId);
```

Alternative option:

```java
KieServices kieServices = KieServices.Factory.get();
KieContainer kieContainer = kieServices.getClasspathContainer();
```

3. Obtain `DMNRuntime` from the KIE container and a reference to the DMN model to be evaluated, by using the model `namespace` and `modelName`:

```java
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:

```java
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

If the DMN model was not previously compiled as an executable model for more efficient execution, you can enable the following property when you execute your DMN models:

```java
-Dorg.kie.dmn.compiler.execmodel=true
```

### 7.2. Executing a DMN Service Using the KIE Server Java Client API

The KIE Server Java client API provides a lightweight approach to invoking a remote DMN service either through the REST or JMS interfaces of KIE Server. This approach reduces the number of runtime dependencies necessary to interact with a KIE base. Decoupling the calling code from the decision definition also increases flexibility by enabling them to iterate independently at the appropriate pace.

For more information about the KIE Server Java client API, see *Interacting with Red Hat Decision Manager using KIE APIs*.

**Prerequisites**
KIE Server is installed and configured, including a known user name and credentials for a user with the `kie-server` role. For installation options, see *Planning a Red Hat Decision Manager installation*.

- You have built the DMN project as a KJAR artifact and deployed it to KIE Server. Ideally, you have built the DMN project as an executable model for more efficient execution:

  ```
mvn clean install -DgenerateDMNModel=yes
  ```

  For more information about project packaging and deployment and executable models, see *Packaging and deploying a Red Hat Decision Manager project*.

- You have the ID of the KIE container containing the DMN model. If more than one model is present, you must also know the model namespace and model name of the relevant model.

Procedure

1. In your client application, add the following dependency to the relevant classpath of your Java project:

   ```
   <!-- Required for the KIE Server Java client API -->
   <dependency>
     <groupId>org.kie.server</groupId>
     <artifactId>kie-server-client</artifactId>
     <version>${rhdm.version}</version>
   </dependency>
   ```

   The `<version>` is the Maven artifact version for Red Hat Decision Manager currently used in your project (for example, 7.48.0.Final-redhat-00004).

   **NOTE**

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

   Example BOM dependency:

   ```
   <dependency>
     <groupId>com.redhat.ba</groupId>
     <artifactId>ba-platform-bom</artifactId>
     <version>7.10.0.redhat-00004</version>
     <scope>import</scope>
     <type>pom</type>
   </dependency>
   ```

   For more information about the Red Hat Business Automation BOM, see *What is the mapping between RHDM product and maven library version*?

2. Instantiate a `KieServicesClient` instance with the appropriate connection information.
Example:

```java
KieServicesConfiguration conf =
    KieServicesFactory.newRestConfiguration(URL, USER, PASSWORD); 1
conf.setMarshallingFormat(MarshallingFormat.JSON); 2
KieServicesClient kieServicesClient = KieServicesFactory.newKieServicesClient(conf);
```

1. The connection information:
   - Example URL: `http://localhost:8080/kie-server/services/rest/server`
   - The credentials should reference a user with the `kie-server` role.

2. The Marshalling format is an instance of `org.kie.server.api.marshalling.MarshallingFormat`. It controls whether the messages will be JSON or XML. Options for Marshalling format are JSON, JAXB, or XSTREAM.

3. Obtain a `DMNServicesClient` from the KIE server Java client connected to the related KIE Server by invoking the method `getServicesClient()` on the KIE server Java client instance:

   ```java
   DMNServicesClient dmnClient =
       kieServicesClient.getServicesClient(DMNServicesClient.class);
   ```

   The `dmnClient` can now execute decision services on KIE Server.

4. Execute the decision services for the desired model.
   Example:

   ```java
   for (Integer age : Arrays.asList(1, 12, 13, 64, 65, 66)) {
       DMNContext dmnContext = dmnClient.newContext(); 1
       dmnContext.set("Age", age); 2
       ServiceResponse<DMNResult> serverResp =
           dmnClient.evaluateAll($kieContainerId,
                           $modelNamespace,
                           $modelName,
                           dmnContext);

       DMNResult dmnResult = serverResp.getResult(); 4
       for (DMNDecisionResult dr : dmnResult.getDecisionResults()) {
           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 the DMN Decisions defined in the DMN model:
- `$kieContainerId` is the ID of the container where the KJAR containing the DMN model is deployed.
- `$modelNamespace` is the namespace for the model.
- `$modelName` is the name for the model.

The DMN Result object is available from the server response.

At this point, the `dmnResult` contains all the decision results from the evaluated DMN model.

You can also execute only a specific DMN decision in the model by using alternative methods of the `DMNServicesClient`.

**NOTE**

If the KIE container only contains one DMN model, you can omit `$modelNamespace` and `$modelName` because the KIE Server API selects it by default.

### 7.3. Executing a DMN Service Using the KIE Server REST API

Directly interacting with the REST endpoints of KIE Server 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. The examples in this section demonstrate Nix-style curl commands but provide relevant information to adapt to any REST client.

For more information about the KIE Server REST API, see *Interacting with Red Hat Decision Manager using KIE APIs*.

**Prerequisites**

- KIE Server is installed and configured, including a known user name and credentials for a user with the `kie-server` role. For installation options, see *Planning a Red Hat Decision Manager installation*.
- You have built the DMN project as a KJAR artifact and deployed it to KIE Server. Ideally, you have built the DMN project as an executable model for more efficient execution:

  ```
  mvn clean install -DgenerateDMNModel=yes
  ```

  For more information about project packaging and deployment and executable models, see *Packaging and deploying a Red Hat Decision Manager project*.
- You have the ID of the KIE container containing the DMN model. If more than one model is present, you must also know the model namespace and model name of the relevant model.

**Procedure**

1. Determine the base URL for accessing the KIE Server REST API endpoints. This requires knowing the following values (with the default local deployment values as an example):
   - Host ([localhost](http://localhost))
Port (8080)

Root context (kie-server)

Base REST path (services/rest/)

Example base URL in local deployment:

http://localhost:8080/kie-server/services/rest/

2. Determine user authentication requirements.

When users are defined directly in the KIE Server configuration, HTTP Basic authentication is used and requires the user name and password. Successful requests require that the user have the kie-server role.

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

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

If KIE Server is configured with Red Hat Single Sign-On, the request must include a bearer token:

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

3. Specify the format of the request and response. The REST API endpoints work with both JSON and XML formats and are set using request headers:

**JSON**

```
curl -H "accept: application/json" -H "content-type: application/json"
```

**XML**

```
curl -H "accept: application/xml" -H "content-type: application/xml"
```

4. Optional: Query the container for a list of deployed decision models:

[GET] server/containers/(containerId)/dmn

Example curl request:

```
```

Sample XML output:

```xml
<response type="SUCCESS" msg="OK models successfully retrieved from container 'MovieDMNContainer'">
  <dmn-model-info-list>
    <model>
      <model-namespace>http://www.redhat.com/_c7328033-c355-43cd-b616-0aceef80e52a</model-namespace>
      <model-name>dmn-movieticket-ageclassification</model-name>
      <model-id>_99</model-id>
  </model>
</dmn-model-info-list>
</response>
```
<decisions>
  <dmn-decision-info>
    <decision-id>_3</decision-id>
    <decision-name>AgeClassification</decision-name>
  </dmn-decision-info>
</decisions>
</model>
</dmn-model-info-list>
</response>

Sample JSON output:

```json
{
  "type": "SUCCESS",
  "msg": "OK models successfully retrieved from container 'MovieDMNContainer'",
  "result": {
    "dmn-model-info-list": {
      "models": [
        {
          "model-namespace": "http://www.redhat.com/_c7328033-c355-43cd-b616-0aceef80e52a",
          "model-name": "dmn-movieticket-ageclassification",
          "model-id": "_99",
          "decisions": [
            {
              "decision-id": "_3",
              "decision-name": "AgeClassification"
            }
          ]
        }
      ]
    }
  }
}
```

5. Execute the model:

[POST] server/containers/{containerId}/dmn

Example curl request:

```bash
  "model-name": "dmn-movieticket-ageclassification",
  "decision-name": [],
  "decision-id": [],
  "dmn-context": {
    "Age": 66
  }}'
```

Example JSON request:

```json
{
  "model-namespace": "http://www.redhat.com/_c7328033-c355-43cd-b616-0aceef80e52a",
  "model-name": "dmn-movieticket-ageclassification",
  "decision-name": [],
  "decision-id": [],
  "dmn-context": {
    "Age": 66
  }
}
```

Example XML request (JAXB format):
NOTE

Regardless of the request format, the request requires the following elements:

- Model namespace
- Model name
- Context object containing input values

Example JSON response:

```json
{
    "type": "SUCCESS",
    "msg": "OK from container 'MovieDMNContainer'",
    "result": {
        "dmn-evaluation-result": {
            "messages": [],
            "model-namespace": "http://www.redhat.com/_c7328033-c355-43cd-b616-0aceef80e52a",
            "model-name": "dmn-movieticket-ageclassification",
            "decision-name": [],
            "dmn-context": {
                "Age": 66,
                "AgeClassification": "Senior"
            },
            "decision-results": {
                "_3": {
                    "messages": [],
                    "decision-id": "_3",
                    "decision-name": "AgeClassification",
                    "result": "Senior",
                    "status": "SUCCEEDED"
                }
            }
        }
    }
}
```

Example XML (JAXB format) response:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dmn-evaluation-context>
  <model-namespace>http://www.redhat.com/_c7328033-c355-43cd-b616-0aceef80e52a</model-namespace>
  <model-name>dmn-movieticket-ageclassification</model-name>
  <dmn-context xsi:type="jaxbListWrapper" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <type>MAP</type>
    <element xsi:type="jaxbStringObjectPair" key="Age">
      <value xsi:type="xs:int" xmlns:xs="http://www.w3.org/2001/XMLSchema">66</value>
    </element>
  </dmn-context>
</dmn-evaluation-context>
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response type="SUCCESS" msg="OK from container 'MovieDMNContainer'">
    <dmn-evaluation-result>
        <model-namespace>http://www.redhat.com/_c7328033-c355-43cd-b616-0acef80e52a</model-namespace>
        <model-name>dmn-movieticket-ageclassification</model-name>
        <dmn-context xsi:type="jaxbListWrapper"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <type>MAP</type>
            <element xsi:type="jaxbStringObjectPair" key="Age">
                <value xsi:type="xs:int"
                    xmlns:xs="http://www.w3.org/2001/XMLSchema">66</value>
            </element>
            <element xsi:type="jaxbStringObjectPair" key="AgeClassification">
                <value xsi:type="xs:string"
                    xmlns:xs="http://www.w3.org/2001/XMLSchema">Senior</value>
            </element>
        </dmn-context>
        <messages/>
        <decisionResults>
            <entry>
                <key>_3</key>
                <value>
                    <decision-id>_3</decision-id>
                    <decision-name>AgeClassification</decision-name>
                    <result xsi:type="xs:string"
                    <messages/>
                    <status>SUCCEEDED</status>
                </value>
            </entry>
        </decisionResults>
    </dmn-evaluation-result>
</response>
CHAPTER 8. ADDITIONAL RESOURCES

- Decision Model and Notation specification
- DMN Technology Compatibility Kit
- Packaging and deploying a Red Hat Decision Manager project
- Interacting with Red Hat Decision Manager using KIE APIs
PART II. DESIGNING A DECISION SERVICE USING PMML MODELS

As a business rules developer, you can use Predictive Model Markup Language (PMML) to define statistical or data-mining models that you can integrate with your decision services in Red Hat Decision Manager. Red Hat Decision Manager includes consumer conformance support of PMML 4.2.1 for Regression, Scorecard, Tree, and Mining models. Red Hat Decision Manager does not include a built-in PMML model editor, but you can use an XML or PMML-specific authoring tool to create PMML models and then integrate them with your Red Hat Decision Manager projects.

For more information about PMML, see the DMG PMML specification.

NOTE

You can also design your decision service using Decision Model and Notation (DMN) models and include your PMML models as part of your DMN service. For information about DMN support in Red Hat Decision Manager 7.10, see the following resources:

- Getting started with decision services (step-by-step tutorial with a DMN decision service example)
- Designing a decision service using DMN models (overview of DMN support and capabilities in Red Hat Decision Manager)
CHAPTER 9. DECISION-AUTHORING ASSETS IN RED HAT DECISION MANAGER

Red Hat Decision Manager supports several assets that you can use to define business decisions for your decision service. Each decision-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights the main decision-authoring assets supported in Red Hat Decision Manager projects to help you decide or confirm the best method for defining decisions in your decision service.

Table 9.1. Decision-authoring assets supported in Red Hat Decision Manager

<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision Model and Notation (DMN) models</td>
<td>• Are decision models based on a notation standard defined by the Object Management Group (OMG)</td>
<td>Business Central or other DMN-compliant editor</td>
<td>Designing a decision service using DMN models</td>
</tr>
<tr>
<td></td>
<td>• Use graphical decision requirements diagrams (DRDs) that represent part or all of the overall decision requirements graph (DRG) to trace business decision flows</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Use an XML schema that allows the DMN models to be shared between DMN-compliant platforms</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Support Friendly Enough Expression Language (FEEL) to define decision logic in DMN decision tables and other DMN boxed expressions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Are optimal for creating comprehensive, illustrative, and stable decision flows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asset</td>
<td>Highlights</td>
<td>Authoring tools</td>
<td>Documentation</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Guided decision tables       | • Are tables of rules that you create in a UI-based table designer in Business Central  
• Are a wizard-led alternative to spreadsheet decision tables  
• Provide fields and options for acceptable input  
• Support template keys and values for creating rule templates  
• Support hit policies, real-time validation, and other additional features not supported in other assets  
• Are optimal for creating rules in a controlled tabular format to minimize compilation errors | Business Central | Designing a decision service using guided decision tables                     |
| Spreadsheet decision tables  | • Are XLS or XLSX spreadsheet decision tables that you can upload into Business Central  
• Support template keys and values for creating rule templates  
• Are optimal for creating rules in decision tables already managed outside of Business Central  
• Have strict syntax requirements for rules to be compiled properly when uploaded | Spreadsheet editor | Designing a decision service using spreadsheet decision tables                 |
| Guided rules                 | • Are individual rules that you create in a UI-based rule designer in Business Central  
• Provide fields and options for acceptable input  
• Are optimal for creating single rules in a controlled format to minimize compilation errors | Business Central | Designing a decision service using guided rules                                |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided rule templates  | • Are reusable rule structures that you create in a UI-based template designer in Business Central  
• Provide fields and options for acceptable input  
• Support template keys and values for creating rule templates (fundamental to the purpose of this asset)  
• Are optimal for creating many rules with the same rule structure but with different defined field values | Business Central                           | Designing a decision service using guided rule templates                     |
| DRL rules              | • Are individual rules that you define directly in `.drl` text files  
• Provide the most flexibility for defining rules and other technicalities of rule behavior  
• Can be created in certain standalone environments and integrated with Red Hat Decision Manager  
• Are optimal for creating rules that require advanced DRL options  
• Have strict syntax requirements for rules to be compiled properly | Business Central or integrated development environment (IDE) | Designing a decision service using DRL rules                                     |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Predictive Model Markup Language (PMML) models | - Are predictive data-analytic models based on a notation standard defined by the Data Mining Group (DMG)  
- Use an XML schema that allows the PMML models to be shared between PMML-compliant platforms  
- Support Regression, Scorecard, Tree, Mining, and other model types  
- Can be included with a standalone Red Hat Decision Manager project or imported into a project in Business Central  
- Are optimal for incorporating predictive data into decision services in Red Hat Decision Manager | PMML or XML editor | *Designing a decision service using PMML models* |
CHAPTER 10. PREDICTIVE MODEL MARKUP LANGUAGE (PMML)

Predictive Model Markup Language (PMML) is an XML-based standard established by the Data Mining Group (DMG) for defining statistical and data-mining models. PMML models can be shared between PMML-compliant platforms and across organizations so that business analysts and developers are unified in designing, analyzing, and implementing PMML-based assets and services.

For more information about the background and applications of PMML, see the DMG PMML specification.

10.1. PMML CONFORMANCE LEVELS

The PMML specification defines producer and consumer conformance levels in a software implementation to ensure that PMML models are created and integrated reliably. For the formal definitions of each conformance level, see the DMG PMML conformance page.

The following list summarizes the PMML conformance levels:

**Producer conformance**
A tool or application is producer conforming if it generates valid PMML documents for at least one type of model. Satisfying PMML producer conformance requirements ensures that a model definition document is syntactically correct and defines a model instance that is consistent with semantic criteria that are defined in model specifications.

**Consumer conformance**
An application is consumer conforming if it accepts valid PMML documents for at least one type of model. Satisfying consumer conformance requirements ensures that a PMML model created according to producer conformance can be integrated and used as defined. For example, if an application is consumer conforming for Regression model types, then valid PMML documents defining models of this type produced by different conforming producers would be interchangeable in the application.

Red Hat Decision Manager includes consumer conformance support for the following PMML 4.2.1 model types:

- **Regression models**
- **Scorecard models**
- **Tree models**
- **Mining models** (with sub-types `modelChain`, `selectAll`, and `selectFirst`)

For a list of all PMML model types, including those not supported in Red Hat Decision Manager, see the DMG PMML specification.
CHAPTER 11. PMML MODEL EXAMPLES

PMML defines an XML schema that enables PMML models to be used between different PMML-compliant platforms. The PMML specification enables multiple software platforms to work with the same file for authoring, testing, and production execution, assuming producer and consumer conformance are met.

The following are examples of PMML Regression, Scorecard, Tree, and Mining models. These examples illustrate the supported types of models that you can integrate with your decision services in Red Hat Decision Manager.

For more PMML examples, see the DMG PMML Sample Files page.

Example PMML Regression model

```xml
  <Header copyright="JBoss"/>
  <DataDictionary numberOfFields="5">
    <DataField dataType="double" name="fld1" optype="continuous"/>
    <DataField dataType="double" name="fld2" optype="continuous"/>
    <DataField dataType="string" name="fld3" optype="categorical">
      <Value value="x"/>
      <Value value="y"/>
    </DataField>
    <DataField dataType="double" name="fld4" optype="continuous"/>
    <DataField dataType="double" name="fld5" optype="continuous"/>
  </DataDictionary>
  <RegressionModel algorithmName="linearRegression" functionName="regression"modelName="LinReg" normalizationMethod="logit" targetFieldName="fld4">
    <MiningSchema>
      <MiningField name="fld1"/>
      <MiningField name="fld2"/>
      <MiningField name="fld3"/>
      <MiningField name="fld4" usageType="predicted"/>
      <MiningField name="fld5" usageType="target"/>
    </MiningSchema>
    <RegressionTable intercept="0.5">
      <NumericPredictor coefficient="5" exponent="2" name="fld1"/>
      <NumericPredictor coefficient="2" exponent="1" name="fld2"/>
      <CategoricalPredictor coefficient="-3" name="fld3" value="x"/>
      <CategoricalPredictor coefficient="3" name="fld3" value="y"/>
      <PredictorTerm coefficient="0.4">
        <FieldRef field="fld1"/>
        <FieldRef field="fld2"/>
      </PredictorTerm>
    </RegressionTable>
  </RegressionModel>
</PMML>
```

Example PMML Scorecard model

```xml
  <Header copyright="JBoss"/>
  <DataDictionary numberOfFields="5">
    <DataField dataType="double" name="fld1" optype="continuous"/>
    <DataField dataType="double" name="fld2" optype="continuous"/>
    <DataField dataType="string" name="fld3" optype="categorical">
      <Value value="x"/>
      <Value value="y"/>
    </DataField>
    <DataField dataType="double" name="fld4" optype="continuous"/>
    <DataField dataType="double" name="fld5" optype="continuous"/>
  </DataDictionary>
  <ScorecardModel>
    <MiningSchema>
      <MiningField name="fld1"/>
      <MiningField name="fld2"/>
      <MiningField name="fld3"/>
      <MiningField name="fld4" usageType="predicted"/>
      <MiningField name="fld5" usageType="target"/>
    </MiningSchema>
    <ScorecardTable>
      <ScorecardName name="Scorecard1"/>
      <ScorecardRule name="Rule1">
        <Condition expression="fld1 = x && fld2 = y"/>
        <Score value="1"/>
      </ScorecardRule>
      <ScorecardRule name="Rule2">
        <Condition expression="fld1 = y && fld2 = x"/>
        <Score value="2"/>
      </ScorecardRule>
    </ScorecardTable>
  </ScorecardModel>
</PMML>
```
Example PMML Tree model
Example PMML Mining model (modelChain)

```xml
  <Header>
    <Application name="Drools-PMML" version="7.0.0-SNAPSHOT" />
  </Header>
  <DataDictionary numberOfFields="7">
    <DataField name="age" optype="continuous" dataType="double" />
    <DataField name="occupation" optype="categorical" dataType="string">
      <Value value="SKYDIVER" />
      <Value value="ASTRONAUT" />
      <Value value="PROGRAMMER" />
      <Value value="TEACHER" />
      <Value value="INSTRUCTOR" />
    </DataField>
    <DataField name="residenceState" optype="categorical" dataType="string">
      <Value value="AP" />
      <Value value="KN" />
      <Value value="TN" />
    </DataField>
    <DataField name="validLicense" optype="categorical" dataType="boolean" />
    <DataField name="overallScore" optype="continuous" dataType="double" />
    <DataField name="grade" optype="categorical" dataType="string">
      <Value value="A" />
      <Value value="B" />
      <Value value="C" />
      <Value value="D" />
  </DataDictionary>
  <TreeModel>
    <Node score="tgtY">
      <CompoundPredicate booleanOperator="or">
        <SimplePredicate field="fld4" operator="equal" value="optA" />
        <SimplePredicate field="fld4" operator="equal" value="optC" />
      </CompoundPredicate>
    </Node>
    <Node score="tgtY">
      <CompoundPredicate booleanOperator="and">
        <SimplePredicate field="fld1" operator="greaterThan" value="10.0" />
        <SimplePredicate field="fld1" operator="lessThan" value="50.0" />
        <SimplePredicate field="fld4" operator="equal" value="optA" />
        <SimplePredicate field="fld2" operator="lessThan" value="100.0" />
        <SimplePredicate field="fld3" operator="equal" value="false" />
      </CompoundPredicate>
    </Node>
    <Node score="tgtZ">
      <CompoundPredicate booleanOperator="and">
        <SimplePredicate field="fld4" operator="equal" value="optC" />
        <SimplePredicate field="fld2" operator="lessThan" value="30.0" />
      </CompoundPredicate>
    </Node>
  </TreeModel>
</PMML>
```
<Value value="F" />
</DataField>
<DataField name="qualificationLevel" otype="categorical" dataType="string">
  <Value value="Unqualified" />
  <Value value="Barely" />
  <Value value="Well" />
  <Value value="Over" />
</DataField>
</DataDictionary>

<MiningModel modelName="SampleModelChainMine" functionName="classification">
  <MiningSchema>
    <MiningField name="age" />
    <MiningField name="occupation" />
    <MiningField name="residenceState" />
    <MiningField name="validLicense" />
    <MiningField name="overallScore" />
    <MiningField name="qualificationLevel" usageType="target" />
  </MiningSchema>
  <Segmentation multipleModelMethod="modelChain">
    <Segment id="1">
      <True />
      <Scorecard modelName="Sample Score 1" useReasonCodes="true" isScorable="true" functionName="regression" baselineScore="0.0" initialScore="0.345">
        <MiningSchema>
          <MiningField name="age" usageType="active" invalidValueTreatement="asMissing" />
          <MiningField name="occupation" usageType="active" invalidValueTreatement="asMissing" />
          <MiningField name="residenceState" usageType="active" invalidValueTreatement="asMissing" />
          <MiningField name="validLicense" usageType="active" invalidValueTreatement="asMissing" />
          <MiningField name="overallScore" usageType="predicted" />
        </MiningSchema>
        <Output feature="predictedValue" targetField="overallScore" />
      </Scorecard>
    </Segment>
  </Segmentation>
</MiningModel>
<Attribute partialScore="25.0">
  <Extension name="cellRef" value="$C$13"/>
  <SimplePredicate field="age" operator="greaterOrEqual" value="45"/>
</Attribute>
</Characteristic>

<Attribute partialScore="10.0"/>
  <Extension name="cellRef" value="$C$19"/>
  <SimplePredicate field="occupation" booleanOperator="isIn">
    <Array n="2" type="string">SKYDIVER ASTRONAUT</Array>
  </SimplePredicate>
</Attribute>
</Characteristic>

<Attribute partialScore="5.0">
  <Extension name="cellRef" value="$C$20"/>
  <SimplePredicate field="occupation" operator="equal" value="PROGRAMMER"/>
</Attribute>
</Characteristic>

<Attribute partialScore="0.0" reasonCode="RES">
  <Extension name="cellRef" value="$B$22"/>
  <Attribute partialScore="-10.0"/>
  <Extension name="cellRef" value="$C$24"/>
  <SimplePredicate field="residenceState" operator="equal" value="AP"/>
</Attribute>
</Characteristic>

<Attribute partialScore="10.0">
  <Extension name="cellRef" value="$C$25"/>
  <SimplePredicate field="residenceState" operator="equal" value="KN"/>
</Attribute>
</Characteristic>

<Attribute partialScore="5.0">
  <Extension name="cellRef" value="$C$26"/>
  <SimplePredicate field="residenceState" operator="equal" value="TN"/>
</Attribute>
</Characteristic>

<Attribute partialScore="0.0" reasonCode="LX00">
  <Extension name="cellRef" value="$B$28"/>
  <Attribute partialScore="1.0"/>
  <Extension name="cellRef" value="$C$30"/>
  <SimplePredicate field="validLicense" operator="equal" value="true"/>
</Attribute>
</Characteristic>

<Attribute partialScore="-1.0" reasonCode="LX00">
  <Extension name="cellRef" value="$C$31"/>
  <SimplePredicate field="validLicense" operator="equal" value="false"/>
</Attribute>
</Characteristic>

</Scorecard>
</Segment>

<Segment id="2"> TRUE </Segment>
<TreeModel modelName="SampleTree" functionName="classification"
    missingValueStrategy="lastPrediction" noTrueChildStrategy="returnLastPrediction">
    <MiningSchema>
        <MiningField name="age" usageType="active"/>
        <MiningField name="validLicense" usageType="active"/>
        <MiningField name="calculatedScore" usageType="active"/>
        <MiningField name="qualificationLevel" usageType="predicted"/>
    </MiningSchema>
    <Output>
        <OutputField name="qualification" displayName="Qualification Level"
            dataType="string" feature="predictedValue" targetField="qualificationLevel"/>
    </Output>
    <Node score="Well" id="1">
        <True/>
        <Node score="Barely" id="2">
            <CompoundPredicate booleanOperator="and">
                <SimplePredicate field="age" operator="greaterOrEqual" value="16"/>
                <SimplePredicate field="validLicense" operator="equal" value="true"/>
            </CompoundPredicate>
        </Node>
        <Node score="Well" id="4">
            <CompoundPredicate booleanOperator="and">
                <SimplePredicate field="calculatedScore" operator="greaterThan" value="50.0"/>
                <SimplePredicate field="calculatedScore" operator="lessOrEqual" value="60.0"/>
            </CompoundPredicate>
        </Node>
        <Node score="Over" id="5">
            <SimplePredicate field="calculatedScore" operator="greaterThan" value="60.0"/>
        </Node>
        <Node score="Unqualified" id="6">
            <CompoundPredicate booleanOperator="surrogate">
                <SimplePredicate field="age" operator="lessThan" value="16"/>
                <SimplePredicate field="calculatedScore" operator="lessOrEqual" value="40.0"/>
                <True/>
            </CompoundPredicate>
        </Node>
    </Node>
</TreeModel>
CHAPTER 12. PMML SUPPORT IN RED HAT DECISION MANAGER

Red Hat Decision Manager includes consumer conformance support for the following PMML 4.2.1 model types:

- Regression models
- Scorecard models
- Tree models
- Mining models (with sub-types modelChain, selectAll, and selectFirst)

For a list of all PMML model types, including those not supported in Red Hat Decision Manager, see the DMG PMML specification.

Red Hat Decision Manager does not include a built-in PMML model editor, but you can use an XML or PMML-specific authoring tool to create PMML models and then integrate the PMML models in your decision services in Red Hat Decision Manager. You can import PMML files into your project in Business Central (Menu → Design → Projects → Import Asset) or package the PMML files as part of your project knowledge JAR (KJAR) file without Business Central.

When you add a PMML file to a project in Red Hat Decision Manager, multiple assets are generated. Each type of PMML model generates a different set of assets, but all PMML model types generate at least the following set of assets:

- A DRL file that contains all of the rules associated with your PMML model
- At least two Java classes:
  - A data class that is used as the default object type for the model type
  - A RuleUnit class that is used to manage data sources and rule execution

If a PMML file has MiningModel as the root model, multiple instances of each of these files are generated.

For more information about including assets such as PMML files with your project packaging and deployment method, see Packaging and deploying a Red Hat Decision Manager project.

12.1. PMML NAMING CONVENTIONS IN RED HAT DECISION MANAGER

The following are naming conventions for generated PMML packages, classes, and rules:

- If no package name is given in a PMML model file, then the default package name org.kie.pmml.pmml_4_2 is prefixed to the model name for the generated rules in the format "org.kie.pmml.pmml_4_2"+modelName.
- The package name for the generated RuleUnit Java class is the same as the package name for the generated rules.
- The name of the generated RuleUnit Java class is the model name with RuleUnit added to it in the format modelName+"RuleUnit".
Each PMML model has at least one data class that is generated. The package name for these classes is `org.kie.pmml.pmml_4_2.model`.

The names of generated data classes are determined by the model type, prefixed with the model name:

- **Regression models**: One data class named `modelName+"RegressionData"`
- **Scorecard models**: One data class named `modelName+"ScoreCardData"`
- **Tree models**: Two data classes, the first named `modelName+"TreeNode"` and the second named `modelName+"TreeToken"`
- **Mining models**: One data class named `modelName+"MiningModelData"`

**NOTE**

The mining model also generates all of the rules and classes that are within each of its segments.

### 12.2. PMML EXTENSIONS IN RED HAT DECISION MANAGER

The PMML specification supports **Extension** elements that extend the content of a PMML model. You can use extensions at almost every level of a PMML model definition, and as the first and last child in the main element of a model for maximum flexibility. For more information about PMML extensions, see the DMG PMML Extension Mechanism.

To optimize PMML integration, Red Hat Decision Manager supports the following additional PMML extensions:

- **modelPackage**: Designates a package name for the generated rules and Java classes. Include this extension in the **Header** section of the PMML model file.

- **adapter**: Designates the type of construct (**bean** or **trait**) that is used to contain input and output data for rules. Insert this extension in the **MiningSchema** or **Output** section (or both) of the PMML model file.

- **externalClass**: Used in conjunction with the **adapter** extension in defining a **MiningField** or **OutputField**. This extension contains a class with an attribute name that matches the name of the **MiningField** or **OutputField** element.
You can import PMML files into your Red Hat Decision Manager project using Business Central (Menu → Design → Projects → Import Asset) or package the PMML files as part of your project knowledge JAR (KJAR) file without Business Central. After you implement your PMML files in your Red Hat Decision Manager project, you can execute the PMML-based decision service by embedding PMML calls directly in your Java application or by sending an `ApplyPmmlModelCommand` command to a configured KIE Server.

For more information about including PMML assets with your project packaging and deployment method, see Packaging and deploying a Red Hat Decision Manager project.

**NOTE**

You can also include a PMML model as part of a Decision Model and Notation (DMN) service in Business Central. 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. For more information about including PMML models in a DMN service, see Designing a decision service using DMN models.

### 13.1. EMBEDDING A PMML 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 PMML 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 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**

- A KJAR containing the PMML model to execute has been created. For more information about project packaging, see Packaging and deploying a Red Hat Decision Manager project.

**Procedure**

1. In your client application, add the following dependencies to the relevant classpath of your Java project:

   ```xml
   <!-- Required for the PMML compiler -->
   <dependency>
     <groupId>org.drools</groupId>
     <artifactId>kie-pmml</artifactId>
     <version>${rhdm.version}</version>
   </dependency>
   </dependency>

   <!-- Required for the KIE public API -->
   ```
The `<version>` is the Maven artifact version for Red Hat Decision Manager currently used in your project (for example, 7.48.0.Final-redhat-00004).

**IMPORTANT**

The legacy `kie-pmml` dependency is deprecated with Red Hat Decision Manager 7.10.0 and will be replaced by `kie-pmml-trusty` dependency in a future Red Hat Decision Manager release.

**NOTE**

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

Example BOM dependency:

```xml
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-api</artifactId>
  <version>${rhdm.version}</version>
</dependency>

<!-- Required if not using classpath KIE container -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>${rhdm.version}</version>
</dependency>
```

KieServices kieServices = KieServices.Factory.get();

Releaseld releaseld = kieServices.new releaseld("org.acme", "my-kjar", "1.0.0");
KieContainer kieContainer = kieServices.new KieContainer( releaseld );

Alternative option:

```xml
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.10.0.redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

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

2. Create a KIE container from classpath or Releaseld:
3. Create an instance of the `PMMLRequestData` class, which applies your PMML model to a set of data:

```java
public class PMMLRequestData {
    private String correlationId; // Identifies data that is associated with a particular request or result
    private String modelName; // The name of the model that should be applied to the request data
    private String source; // Used by internally generated PMMLRequestData objects to identify the segment that generated the request
    private List<ParameterInfo<?>> requestParams; // The default mechanism for sending input data points
}
```

1. Identifies data that is associated with a particular request or result
2. The name of the model that should be applied to the request data
3. Used by internally generated PMMLRequestData objects to identify the segment that generated the request
4. The default mechanism for sending input data points

4. Create an instance of the `PMML4Result` class, which holds the output information that is the result of applying the PMML-based rules to the input data:

```java
public class PMML4Result {
    private String correlationId; // Used when the model type is MiningModel. The segmentationId is used to differentiate between multiple segmentations.
    private String segmentationId; // Used in conjunction with the segmentationId to identify which segment generated the results.
    private String segmentId; // Used to maintain the order of segments.
    private int segmentIndex; // Used to determine whether the model was successfully applied, where OK indicates success.
    private String resultCode; // Contains the name of a resultant variable and its associated value.
    private Map<String, Object> resultVariables; // Contains the name of a resultant variable and its associated value.
}
```

1. Used when the model type is `MiningModel`. The `segmentationId` is used to differentiate between multiple segmentations.
2. Used in conjunction with the `segmentationId` to identify which segment generated the results.
3. Used to maintain the order of segments.
4. Used to determine whether the model was successfully applied, where `OK` indicates success.
5. Contains the name of a resultant variable and its associated value.

In addition to the normal getter methods, the `PMML4Result` class also supports the following methods for directly retrieving the values for result variables:
5. Create an instance of the **ParameterInfo** class, which serves as a wrapper for basic data type objects used as part of the **PMMLRequestData** class:

```java
public class ParameterInfo<
    T
> {
    private String correlationId;
    private String name;
    private String capitalizedName;
    private Class<T> type;
    private T value;
}
```

1. The parameterized class to handle many different types
2. The name of the variable that is expected as input for the model
3. The class that is the actual type of the variable
4. The actual value of the variable

6. Execute the PMML model based on the required PMML class instances that you have created:

```java
public void executeModel(KieBase kbase,
    Map<String, Object> variables,
    String modelName,
    String correlationId,
    String modelPkgName)
{
    RuleUnitExecutor executor = RuleUnitExecutor.create().bind(kbase);
    PMMLRequestData request = new PMMLRequestData(correlationId, modelName);
    PMML4Result resultHolder = new PMML4Result(correlationId);
    variables.entrySet().forEach(es -> {
        request.addRequestParam(es.getKey(), es.getValue());
    });
    DataSource<PMMLRequestData> requestData = executor.newDataSource("request");
    DataSource<PMML4Result> resultData = executor.newDataSource("results");
    DataSource<PMMLData> internalData = executor.newDataSource("pmmlData");
    requestData.insert(request);
    resultData.insert(resultHolder);
    List<String> possiblePackageNames = calculatePossiblePackageNames(modelName,
        modelPkgName);
    Class<? extends RuleUnit> ruleUnitClass = getStartingRuleUnit("RuleUnitIndicator",
        (InternalKnowledgeBase)kbase,
        possiblePackageNames);
    if (ruleUnitClass != null) {
        executor.run(ruleUnitClass);
    }
}
```
Rules are executed by the RuleUnitExecutor class. The RuleUnitExecutor class creates KIE sessions and adds the required DataSource objects to those sessions, and then executes the rules based on the RuleUnit that is passed as a parameter to the run() method. The calculatePossiblePackageNames and the getStartingRuleUnit methods determine the fully qualified name of the RuleUnit class that is passed to the run() method.

To facilitate your PMML model execution, you can also use a PMML4ExecutionHelper class supported in Red Hat Decision Manager. For more information about the PMML helper class, see Section 13.11, “PMML execution helper class”.

13.1.1. PMML execution helper class

Red Hat Decision Manager provides a PMML4ExecutionHelper class that helps create the PMMLRequestData class required for PMML model execution and that helps execute rules using the RuleUnitExecutor class.
The following are examples of a PMML model execution without and with the **PMML4ExecutionHelper** class, as a comparison:

### Example PMML model execution without using **PMML4ExecutionHelper**

```java
public void executeModel(KieBase kbase,
                Map<String, Object> variables,
                String modelName,
                String correlationId,
                String modelPkgName) {
    RuleUnitExecutor executor = RuleUnitExecutor.create().bind(kbase);
    PMMLRequestData request = new PMMLRequestData(correlationId, modelName);
    PMML4Result resultHolder = new PMML4Result(correlationId);
    variables.entrySet().forEach(es -> {
        request.addRequestParam(es.getKey(), es.getValue());
    });
    DataSource<PMMLRequestData> requestData = executor.newDataSource("request");
    DataSource<PMML4Result> resultData = executor.newDataSource("results");
    DataSource<PMMLData> internalData = executor.newDataSource("pmmlData");
    requestData.insert(request);
    resultData.insert(resultHolder);
    List<String> possiblePackageNames = calculatePossiblePackageNames(modelName,
                                                                  modelPkgName);
    Class<? extends RuleUnit> ruleUnitClass = getStartingRuleUnit("RuleUnitIndicator",
                                                                  (InternalKnowledgeBase) kbase,
                                                                  possiblePackageNames);
    if (ruleUnitClass != null) {
        executor.run(ruleUnitClass);
        if ("OK".equals(resultHolder.getResultCode())) {
            // extract result variables here
        }
    }
    protected Class<? extends RuleUnit> getStartingRuleUnit(String startingRule,
                                                              InternalKnowledgeBase ikb,
                                                              List<String> possiblePackages) {
        RuleUnitRegistry unitRegistry = ikb.getRuleUnitRegistry();
        Map<String, InternalKnowledgePackage> pkgs = ikb.getPackagesMap();
        RuleImpl ruleImpl = null;
        for (String pkgName: possiblePackages) {
            if (pkgs.containsKey(pkgName)) {
                InternalKnowledgePackage pkg = pkgs.get(pkgName);
                ruleImpl = pkg.getRule(startingRule);
                if (ruleImpl != null) {
                    RuleUnitDescr descr = unitRegistry.getRuleUnitFor(ruleImpl).orElse(null);
                    if (descr != null) {
                        return descr.getRuleUnitClass();
                    }
                }
            }
        }
        return null;
    }
```

Red Hat Decision Manager 7.10 Developing decision services in Red Hat Decision Manager
When you use the `PMML4ExecutionHelper`, you do not need to specify the possible package names nor the `RuleUnit` class as you would in a typical PMML model execution.

To construct a `PMML4ExecutionHelper` class, you use the `PMML4ExecutionHelperFactory` class to determine how instances of `PMML4ExecutionHelper` are retrieved.

The following are the available `PMML4ExecutionHelperFactory` class methods for constructing a `PMML4ExecutionHelper` class:

**PMML4ExecutionHelperFactory methods for PMML assets in a KIE base**

Use these methods when PMML assets have already been compiled and are being used from an existing KIE base:

```java
public static PMML4ExecutionHelper getExecutionHelper(String modelName, KieBase kbase)
public static PMML4ExecutionHelper getExecutionHelper(String modelName, KieBase kbase,
boolean includeMiningDataSources)
```
PMML4ExecutionHelperFactory methods for PMML assets on the project classpath
Use these methods when PMML assets are on the project classpath. The `classPath` argument is the project classpath location of the PMML file:

```java
public static PMML4ExecutionHelper getExecutionHelper(String modelName, String classPath, KieBaseConfiguration kieBaseConf)
```

```java
public static PMML4ExecutionHelper getExecutionHelper(String modelName, String classPath, KieBaseConfiguration kieBaseConf, boolean includeMiningDataSources)
```

PMML4ExecutionHelperFactory methods for PMML assets in a byte array
Use these methods when PMML assets are in the form of a byte array:

```java
public static PMML4ExecutionHelper getExecutionHelper(String modelName, byte[] content, KieBaseConfiguration kieBaseConf)
```

```java
public static PMML4ExecutionHelper getExecutionHelper(String modelName, byte[] content, KieBaseConfiguration kieBaseConf, boolean includeMiningDataSources)
```

PMML4ExecutionHelperFactory methods for PMML assets in a Resource
Use these methods when PMML assets are in the form of an `org.kie.api.io.Resource` object:

```java
public static PMML4ExecutionHelper getExecutionHelper(String modelName, Resource resource, KieBaseConfiguration kieBaseConf)
```

```java
public static PMML4ExecutionHelper getExecutionHelper(String modelName, Resource resource, KieBaseConfiguration kieBaseConf, boolean includeMiningDataSources)
```

**NOTE**

The classpath, byte array, and resource `PMML4ExecutionHelperFactory` methods create a KIE container for the generated rules and Java classes. The container is used as the source of the KIE base that the `RuleUnitExecutor` uses. The container is not persisted. The `PMML4ExecutionHelperFactory` method for PMML assets that are already in a KIE base does not create a KIE container in this way.

### 13.2. Executing a PMML Model Using KIE Server

You can execute PMML models that have been deployed to KIE Server by sending the `ApplyPmmlModelCommand` command to the configured KIE Server. When you use this command, a `PMMLRequestData` object is sent to the KIE Server and a `PMML4Result` result object is received as a reply. You can send PMML requests to KIE Server through the KIE Server REST API from a configured Java class or directly from a REST client.

**Prerequisites**

- KIE Server is installed and configured, including a known user name and credentials for a user with the `kie-server` role. For installation options, see [Planning a Red Hat Decision Manager installation](#).
A KIE container is deployed in KIE Server in the form of a KJAR that includes the PMML model. For more information about project packaging, see Packaging and deploying a Red Hat Decision Manager project.

You have the container ID of the KIE container containing the PMML model.

Procedure

1. In your client application, add the following dependencies to the relevant classpath of your Java project:

```xml
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>kie-pmml</artifactId>
  <version>${rhdm.version}</version>
</dependency>

<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-api</artifactId>
  <version>${rhdm.version}</version>
</dependency>

<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>${rhdm.version}</version>
</dependency>

<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>${rhdm.version}</version>
</dependency>
```

The `<version>` is the Maven artifact version for Red Hat Decision Manager currently used in your project (for example, 7.48.0.Final-redhat-00004).

**IMPORTANT**

The legacy `kie-pmml` dependency is deprecated with Red Hat Decision Manager 7.10.0 and will be replaced by `kie-pmml-trusty` dependency in a future Red Hat Decision Manager release.
NOTE

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

Example BOM dependency:

```xml
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.10.0.redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see What is the mapping between RHDM product and maven library version?.

2. Create a KIE container from classpath or `ReleaseId`:

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

ReleaseId releaselId = kieServices.newReleaseId("org.acme", "my-kjar", "1.0.0");
KieContainer kieContainer = kieServices.newKieContainer(releaselId);
```

Alternative option:

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

3. Create a class for sending requests to KIE Server and receiving responses:

```java
public class ApplyScorecardModel {
  private static final ReleaseId releaselId =
    new ReleaseId("org.acme", "my-kjar", "1.0.0");
  private static final String containerId = "SampleModelContainer";
  private static ClassLoader kjarClassLoader;
  private RuleServicesClient serviceClient;

  // Attributes specific to your class instance
  private String rankedFirstCode;
  private Double score;

  // Initialization of non-final static attributes
  static {
    commandFactory = KieServices.Factory.get().getCommands();
  }
}
// Specifications for kjarClassLoader, if used
KieMavenRepository kmp = KieMavenRepository.getMavenRepository();
File artifactFile = kmp.resolveArtifact(releaseId).getFile();
if (artifactFile != null) {
    URL urls[] = new URL[1];
    try {
        urls[0] = artifactFile.toURI().toURL();
        classLoader = new KieURLClassLoader(urls,PMML4Result.class.getClassLoader());
    } catch (MalformedURLException e) {
        logger.error("Error getting classLoader for "+containerId);
        logger.error(e.getMessage());
    }
} else {
    logger.warn("Did not find the artifact file for "+releaseId.toString());
}

public ApplyScorecardModel(KieServicesConfiguration kieConfig) {
    KieServicesClient clientFactory = KieServicesFactory.newKieServicesClient(kieConfig);
    serviceClient = clientFactory.getServicesClient(RuleServicesClient.class);
}

// Method for executing the PMML model on KIE Server
public void applyModel(String occupation, int age) {
    PMMLRequestData input = new PMMLRequestData("1234","SampleModelName");
    input.addRequestParam(new ParameterInfo("1234","occupation",String.class,occupation));
    input.addRequestParam(new ParameterInfo("1234","age",Integer.class,age));

    CommandFactoryServiceImpl cf = (CommandFactoryServiceImpl)commandFactory;
    ApplyPmmlModelCommand command = (ApplyPmmlModelCommand)
    cf.newApplyPmmlModel(request);

    ServiceResponse<ExecutionResults> results =
    ruleClient.executeCommandsWithResults(CONTAINER_ID, command);

    if (results != null) {
        PMML4Result resultHolder = (PMML4Result)results.getResult().getValue("results");
        if (resultHolder != null && "OK".equals(resultHolder.getResultCode())) {
            this.score = resultHolder.getResultValue("ScoreCard","score",Double.class).get();
            Map<String, Object> rankingMap =
            (Map<String, Object>)resultHolder.getResultValue("ScoreCard","ranking");
            if (rankingMap != null && !rankingMap.isEmpty()) {
                this.rankedFirstCode = rankingMap.keySet().iterator().next();
            }
        }
    }
}

1. Defines the class loader if you did not include the KJAR in your client project dependencies
2. Identifies the service client as defined in the configuration settings, including KIE Server REST API access credentials
3. Initializes a `PMMLRequestData` object
4. Creates an instance of the `ApplyPmmlModelCommand`
5. Sends the command using the service client
6. Retrieves the results of the executed PMML model

4. Execute the class instance to send the PMML invocation request to KIE Server. Alternatively, you can use JMS and REST interfaces to send the `ApplyPmmlModelCommand` command to KIE Server. For REST requests, you use the `ApplyPmmlModelCommand` command as a **POST** request to `http://SERVER:PORT/kie-server/services/rest/server/containers/instances/{containerId}` in JSON, JAXB, or XStream request format.

**Example POST endpoint**

```java
curl http://localhost:8080/kie-server/services/rest/server/containers/instances/SampleModelContainer
```

**Example JSON request body**

```json
{
   "commands": [
      {
         "apply-pmml-model-command": {
            "outIdentifier": null,
            "packageName": null,
            "hasMining": false,
            "requestData": {
               "correlationId": "123",
               "modelName": "SimpleScorecard",
               "source": null,
               "requestParams": [
                  {
                     "correlationId": "123",
                     "name": "param1",
                     "type": "java.lang.Double",
                     "value": "10.0"
                  },
                  {
                     "correlationId": "123",
                     "name": "param2",
                     "type": "java.lang.Double",
                     "value": "15.0"
                  }
               ]
            }
         }
      }
   ]
}
```

**Example curl request with endpoint and body**
curl -X POST "http://localhost:8080/kie-server/services/rest/server/containers/instances/ SampleModelContainer" -H "accept: application/json" -H "content-type: application/json" -d "{ "commands": [ { "apply-pmml-model-command": { "outIdentifier": null, "packageName": null, "hasMining": false, "requestData": { "correlationId": "123", "modelName": "SimpleScorecard", "source": null, "requestParams": [ { "correlationId": "123", "name": "param1", "type": \"java.lang.Double\", "value": \"10.0\", } ] } } ] }"

Example JSON response

```json
{"results": [ { "value": {"org.kie.api.pmml.DoubleFieldOutput":{ "value": 40.8, "correlationId": "123", "segmentationId": null, "segmentId": null, "name": "OverallScore", "displayValue": "OverallScore", "weight": 1.0 }}, "key": "OverallScore" }, { "value": {"org.kie.api.pmml.PMML4Result":{ "resultVariables": { "OverallScore": { "value": 40.8, "correlationId": "123", "segmentationId": null, "segmentId": null, "name": "OverallScore", "displayValue": "OverallScore", "weight": 1.0 }}, "ScoreCard": { "modelName": "SimpleScorecard", "score": 40.8, "holder": { "modelName": "SimpleScorecard", "correlationId": "123", "voverallScore": null, "moverallScore": true, "vparam1": 10.0, "mparam1": false, "vparam2": 15.0, "mparam2": false }}, "enableRC": true, "pointsBelow": true, "ranking": { "reasonCh1": 5.0, "reasonCh2": -6.0 } } ] }
```
{},
"correlationId" : "123",
"segmentationId" : null,
"segmentId" : null,
"segmentIndex" : 0,
"resultCode" : "OK",
"resultObjectName" : null
}}
"key" : "results"
} },
"facts" : [ ]
}
CHAPTER 14. ADDITIONAL RESOURCES

- PMML specification
- Packaging and deploying a Red Hat Decision Manager project
- Interacting with Red Hat Decision Manager using KIE APIs
PART III. DESIGNING A DECISION SERVICE USING DRL RULES

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

NOTE

You can also design your decision service using Decision Model and Notation (DMN) models instead of rule-based or table-based assets. For information about DMN support in Red Hat Decision Manager 7.10, see the following resources:

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

Prerequisites

- The space and project for the DRL rules have been created in Business Central. Each asset is associated with a project assigned to a space. For details, see Getting started with decision services.
CHAPTER 15. DECISION-AUTHORING ASSETS IN RED HAT DECISION MANAGER

Red Hat Decision Manager supports several assets that you can use to define business decisions for your decision service. Each decision-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights the main decision-authoring assets supported in Red Hat Decision Manager projects to help you decide or confirm the best method for defining decisions in your decision service.

Table 15.1. Decision-authoring assets supported in Red Hat Decision Manager

<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Decision Model and Notation (DMN) models | - Are decision models based on a notation standard defined by the Object Management Group (OMG)  
  - Use graphical decision requirements diagrams (DRDs) that represent part or all of the overall decision requirements graph (DRG) to trace business decision flows  
  - Use an XML schema that allows the DMN models to be shared between DMN-compliant platforms  
  - Support Friendly Enough Expression Language (FEEL) to define decision logic in DMN decision tables and other DMN boxed expressions  
  - Are optimal for creating comprehensive, illustrative, and stable decision flows | Business Central or other DMN-compliant editor | Designing a decision service using DMN models |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided decision tables      | • Are tables of rules that you create in a UI-based table designer in Business Central  
• Are a wizard-led alternative to spreadsheet decision tables  
• Provide fields and options for acceptable input  
• Support template keys and values for creating rule templates  
• Support hit policies, real-time validation, and other additional features not supported in other assets  
• Are optimal for creating rules in a controlled tabular format to minimize compilation errors | Business Central | Designing a decision service using guided decision tables                     |
| Spreadsheet decision tables | • Are XLS or XLSX spreadsheet decision tables that you can upload into Business Central  
• Support template keys and values for creating rule templates  
• Are optimal for creating rules in decision tables already managed outside of Business Central  
• Have strict syntax requirements for rules to be compiled properly when uploaded | Spreadsheet editor | Designing a decision service using spreadsheet decision tables                 |
| Guided rules                | • Are individual rules that you create in a UI-based rule designer in Business Central  
• Provide fields and options for acceptable input  
• Are optimal for creating single rules in a controlled format to minimize compilation errors | Business Central | Designing a decision service using guided rules                               |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided rule templates | • Are reusable rule structures that you create in a UI-based template designer in Business Central  
• Provide fields and options for acceptable input  
• Support template keys and values for creating rule templates (fundamental to the purpose of this asset)  
• Are optimal for creating many rules with the same rule structure but with different defined field values | Business Central                     | Designing a decision service using guided rule templates |
| DRL rules             | • Are individual rules that you define directly in .drl text files  
• Provide the most flexibility for defining rules and other technicalities of rule behavior  
• Can be created in certain standalone environments and integrated with Red Hat Decision Manager  
• Are optimal for creating rules that require advanced DRL options  
• Have strict syntax requirements for rules to be compiled properly | Business Central or integrated development environment (IDE) | Designing a decision service using DRL rules            |
### Predictive Model Markup Language (PMML) models

- Are predictive data-analytic models based on a notation standard defined by the Data Mining Group (DMG)
- Use an XML schema that allows the PMML models to be shared between PMML-compliant platforms
- Support Regression, Scorecard, Tree, Mining, and other model types
- Can be included with a standalone Red Hat Decision Manager project or imported into a project in Business Central
- Are optimal for incorporating predictive data into decision services in Red Hat Decision Manager

<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMML or XML editor</td>
<td>Designing a decision service using PMML models</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DRL (Drools Rule Language) rules are business rules that you define directly in .drl text files. These DRL files are the source in which all other rule assets in Business Central are ultimately rendered. You can create and manage DRL files within the Business Central interface, or create them externally as part of a Maven or Java project using Red Hat CodeReady Studio or another integrated development environment (IDE). A DRL file can contain one or more rules that define at a minimum the rule conditions (when) and actions (then). The DRL designer in Business Central provides syntax highlighting for Java, DRL, and XML.

DRL files consist of the following components:

**Components in a DRL file**

- `package`
- `import`
- `function` // Optional
- `query` // Optional
- `declare` // Optional
- `global` // 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"
  salience 15
  agenda-group "applicationGroup"
  when
    $application : LoanApplication()
    Applicant( age < 21 )
  then
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
  end
```

A DRL file can contain single or multiple rules, queries, and functions, and can define resource declarations such as imports, globals, and attributes that are assigned and used by your rules and
queries. The DRL package 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 package. If you use the same rule name more than once in any DRL file in the package, the rules fail to compile. Always enclose rule names with double quotation marks (rule "rule name") to prevent possible compilation errors, especially if you use spaces in rule names.

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

16.1. PACKAGES IN DRL

A package is a folder of related assets in Red Hat Decision Manager, 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

```java
package org.mortgages;
```

16.2. IMPORT STATEMENTS IN DRL

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 decision 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

```java
import org.mortgages.LoanApplication;
```

16.3. FUNCTIONS IN DRL

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

16.4. QUERIES IN DRL

Queries in DRL files search the working memory of the decision 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 global to the KIE base and therefore must be unique among all other rule queries in the project. To return the results of a query, you construct a QueryResults definition using ksession.getQueryResults("name"), where "name" is the query name. This returns a list of query results, which enable you to retrieve the objects that matched the query. You define the query and query results parameters above the rules in the DRL file.

The following example is a query definition in a DRL file for underage applicants in a mortgage application decision service, with the accompanying application code:

Example query definition in a DRL file

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

Example application code to obtain 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" );

You can also iterate over the returned QueryResults using a standard for loop. Each element is a QueryResultsRow that you can use to access each of the columns in the tuple.

Example application code to obtain and iterate over query results

QueryResults results = ksession.getQueryResults( "people under the age of 21" );
16.5. TYPE DECLARATIONS AND METADATA IN DRL

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 Red Hat Decision Manager is `Object`, but you can declare other types in DRL files as needed. Declaring fact types in DRL files enables you to define a new fact model directly in the decision 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 decision engine and used in the reasoning process.

16.5.1. 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 decision engine searches for an existing fact class in the classpath and raises an error if the class is missing.

The following example is a declaration of a new fact type `Person` with no metadata in a DRL file:

**Example declaration of a new fact type with a rule**

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

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

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`. 

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" );
}
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**

```java
import java.util.Date

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

When you declare a new fact type, the decision 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:

**Example rule that uses the declared Person fact type**

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

DRL supports the declaration of enumerative types in the format `declare enum <factType>`, 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("Friday"), SAT("Saturday");

fullName : String
end

rule "Using a declared Enum"
when
  $emp : Employee( dayOff == DaysOfWeek.MONDAY )
then
  ...
end
```

16.5.3. Extended type declarations in DRL

DRL supports type declaration inheritance in the format `declare <factType1> extends <factType2>`. 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
```

16.5.4. 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
decision 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 and @maxLength are declared for the name
attribute. The @key metadata attribute has no required value, so the parentheses and the value are
omitted.

Example metadata declaration for fact types and attributes

```java
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

```java
import org.drools.examples.Person

declare Person
  @author( Bob )
  @dateOfCreation( 01-Feb-2009 )
end
```

Example metadata declaration for a declared type

```java
declare org.drools.examples.Person
  @author( Bob )
  @dateOfCreation( 01-Feb-2009 )
end
```

16.5.5. Metadata tags for fact type and attribute declarations in DRL

Although you can define custom metadata attributes in DRL declarations, the decision engine also
supports the following predefined metadata tags for declarations of fact types or fact type attributes.
NOTE

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**

```java
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 decision engine during complex event processing.

Default parameter: `fact`

Supported parameters: `fact`, `event`

@role( fact | event )

Example: Declare VoiceCall as event type

```java
declare VoiceCall
    @role( event )
end
```

@timestamp

This tag is automatically assigned to every event in the decision 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 decision 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 decision engine session clock

Supported parameters: Session clock time or custom time stamp attribute

@timestamp( <attributeName> )

Example: Declare VoiceCall timestamp attribute

```java
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
end
```

@duration

This tag determines the duration time for events in the decision 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 decision 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 decision 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

```plaintext
@duration( <attributeName> )
```

**Example: Declare VoiceCall duration attribute**

```plaintext
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 decision 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 decision 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]`

```plaintext
@expires( <timeOffset> )
```

**Example: Declare expiration offset for VoiceCall events**

```plaintext
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`

```plaintext
@typesafe( <boolean> )
```
Example: Declare VoiceCall for type-unsafe evaluation

```drl
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

```drl
@serialVersionUID( <integer> )
```

Example: Declare serialVersionUID for a VoiceCall class

```drl
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 decision engine can also generate a constructor using all the key attributes as parameters.

Default parameter: None

Supported parameters: None

```drl
<attributeDefinition> @key
```

Example: Declare Person type attributes as keys

```drl
declare Person
  firstName : String @key
  lastName : String @key
  age : int
end
```

For this example, the decision 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 decision 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**

```java
Person person = new Person( "John", "Doe" );
```

@position

This tag determines the position of a declared fact type attribute or field in a positional argument, overriding the default declared order of attributes. You can use this tag to modify positional constraints in patterns while maintaining a consistent format in your type declarations and positional arguments. You can use this tag only for fields in classes on the classpath. If some fields in a single class use this tag and some do not, the attributes without this tag are positioned last, in the declared order. Inheritance of classes is supported, but not interfaces of methods.

Default parameter: None

Supported parameters: Any integer

```java
<attributeDefinition> @position ( <integer> )
```

**Example: Declare a fact type and override declared order**

```java
declare Person
   firstName : String @position( 1 )
   lastName : String @position( 0 )
   age : int @position( 2 )
   occupation: String
end
```

In this example, the attributes are prioritized in positional arguments in the following order:

1. lastName
2. firstName
3. age
4. occupation

In positional arguments, you do not need to specify the field name because the position maps to a known named field. For example, the argument `Person( lastName == "Doe" )` is the same as `Person( "Doe" ; )`, where the `lastName` field has the highest position annotation in the DRL declaration. The semicolon `;` indicates that everything before it is a positional argument. You can mix positional and named arguments on a pattern by using the semicolon to separate them. Any variables in a positional argument that have not yet been bound are bound to the field that maps to that position.

The following example patterns illustrate different ways of constructing positional and named...
arguments. The patterns have two constraints and a binding, and the semicolon differentiates the positional section from the named argument section. Variables and literals and expressions using only literals are supported in positional arguments, but not variables alone.

Example patterns with positional and named arguments

Person( "Doe", "John", $a; )
Person( "Doe", "John"; $a : age )
Person( "Doe"; firstName == "John", $a : age )
Person( lastName == "Doe"; firstName == "John", $a : age )

Positional arguments can be classified as input arguments or output arguments. Input arguments contain a previously declared binding and constrain against that binding using unification. Output arguments generate the declaration and bind it to the field represented by the positional argument when the binding does not yet exist.

In extended type declarations, use caution when defining @position annotations because the attribute positions are inherited in subtypes. This inheritance can result in a mixed attribute order that can be confusing in some cases. Two fields can have the same @position value and consecutive values do not need to be declared. If a position is repeated, the conflict is solved using inheritance, where position values in the parent type have precedence, and then using the declaration order from the first to last declaration.

For example, the following extended type declarations result in mixed positional priorities:

Example extended fact type with mixed position annotations

```
declare Person
    firstName : String @position( 1 )
    lastName : String @position( 0 )
    age : int @position( 2 )
    occupation: String
end

declare Student extends Person
    degree : String @position( 1 )
    school : String @position( 0 )
    graduationDate : Date
end
```

In this example, the attributes are prioritized in positional arguments in the following order:

1. lastName (position 0 in the parent type)
2. school (position 0 in the subtype)
3. firstName (position 1 in the parent type)
4. degree (position 1 in the subtype)
5. age (position 2 in the parent type)
6. occupation (first field with no position annotation)
7. graduationDate (second field with no position annotation)

16.5.6. Property-change settings and listeners for fact types

By default, the decision 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 decision 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**

```java
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 Red Hat Decision Manager distribution:

**Example property reactivity setting in system properties**

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

The decision 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 decision 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 decision 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 decision 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 decision 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 decision 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 decision 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 decision 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 decision engine to monitor changes in the fact properties.

Example: Declare property change support in JavaBeans object
decclare Person
   @propertyChangeSupport
end

16.5.7. Access to DRL declared types in application code

Declared types in DRL are typically used within the DRL files while Java models are typically used when the model is shared between rules and applications. Because declared types are generated at KIE base compile time, an application cannot access them until application run time. In some cases, an application needs to access and handle facts directly from the declared types, especially when the application wraps the decision engine and provides higher-level, domain-specific user interfaces for rules management.

To handle declared types directly from the application code, you can use the `org.drools.definition.type.FactType` API in Red Hat Decision Manager. Through this API, you can instantiate, read, and write fields in the declared fact types.

The following example code modifies a `Person` fact type directly from an application:

Example application code to handle a declared fact type through the FactType API

```java
import java.util.Date;
import org.kie.api.definition.type.FactType;
import org.kie.api.KieBase;
import org.kie.api.runtime.KieSession;
```
The API also includes other helpful methods, such as setting all the attributes at once, reading values from a `Map` collection, or reading all attributes at once into a `Map` collection.

Although the API behavior is similar to Java reflection, the API does not use reflection and relies on more performant accessors that are implemented with generated bytecode.

### 16.6. GLOBAL VARIABLES IN DRL

Global variables in DRL files typically provide data or services for the rules, such as application services used in rule consequences, and return data from rules, such as logs or values added in rule consequences. You set the global value in the working memory of the decision engine through a KIE session configuration or REST operation, declare the global variable above the rules in the DRL file, and then use it in an action (then) part of the rule. For multiple global variables, use separate lines in the DRL file.

The following example illustrates a global variable list configuration for the decision engine and the corresponding global variable definition in the DRL file:

**Example global list configuration for the decision engine**

```java
List<String> list = new ArrayList<>();
KieSession kieSession = kiebase.newKieSession();
kieSession.setGlobal( "myGlobalList", list );
```

**Example global variable definition with a rule**

```java
global java.util.List myGlobalList;
rule "Using a global"
```
when
  // Empty
then
  myGlobalList.add("My global list");
end

WARNING

Do not use global variables to establish conditions in rules unless a global variable has a constant immutable value. Global variables are not inserted into the working memory of the decision engine, so the decision engine cannot track value changes of variables.

Do not use global variables to share data between rules. Rules always reason and react to the working memory state, so if you want to pass data from rule to rule, assert the data as facts into the working memory of the decision engine.

A use case for a global variable might be an instance of an email service. In your integration code that is calling the decision engine, you obtain your `emailService` object and then set it in the working memory of the decision engine. In the DRL file, you declare that you have a global of type `emailService` and give it the name "email", and then in your rule consequences, you can use actions such as `email.sendSMS(number, message)`.

If you declare global variables with the same identifier in multiple packages, then you must set all the packages with the same type so that they all reference the same global value.

16.7. RULE ATTRIBUTES IN DRL

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:

```plaintext
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>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
</table>

Table 16.1. Rule attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
</table>
| salience      | An integer defining the priority of the rule. Rules with a higher salience value are given higher priority when ordered in the activation queue.  
Example: **salience 10** |
| enabled       | A Boolean value. When the option is selected, the rule is enabled. When the option is not selected, the rule is disabled.  
Example: **enabled true** |
| date-effective | A string containing a date and time definition. The rule can be activated only if the current date and time is after a **date-effective** attribute.  
Example: **date-effective "4-Sep-2018"** |
| date-expires  | A string containing a date and time definition. The rule cannot be activated if the current date and time is after the **date-expires** attribute.  
Example: **date-expires "4-Oct-2018"** |
| no-loop       | A Boolean value. When the option is selected, the rule cannot be reactivated (looped) if a consequence of the rule re-triggers a previously met condition. When the condition is not selected, the rule can be looped in these circumstances.  
Example: **no-loop true** |
| agenda-group  | A string identifying an agenda group to which you want to assign the rule. Agenda groups allow you to partition the agenda to provide more execution control over groups of rules. Only rules in an agenda group that has acquired a focus are able to be activated.  
Example: **agenda-group "GroupName"** |
| activation-group | A string identifying an activation (or XOR) group to which you want to assign the rule. In activation groups, only one rule can be activated. The first rule to fire will cancel all pending activations of all rules in the activation group.  
Example: **activation-group "GroupName"** |
| duration      | A long integer value defining the duration of time in milliseconds after which the rule can be activated, if the rule conditions are still met.  
Example: **duration 10000** |
| timer         | A string identifying either **int** (interval) or **cron** timer definitions for scheduling the rule.  
Example: **timer ( cron:* 0/15 * * * ? ) (every 15 minutes)** |
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>calendar</td>
<td>A Quartz calendar definition for scheduling the rule. Example: <code>calendars &quot;* * 0-7,18-23 ? * *&quot;</code> (exclude non-business hours)</td>
</tr>
<tr>
<td>auto-focus</td>
<td>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: <code>auto-focus true</code></td>
</tr>
<tr>
<td>lock-on-active</td>
<td>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: <code>lock-on-active true</code></td>
</tr>
<tr>
<td>ruleflow-group</td>
<td>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: <code>ruleflow-group &quot;GroupName&quot;</code></td>
</tr>
<tr>
<td>dialect</td>
<td>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: <code>dialect &quot;JAVA&quot;</code></td>
</tr>
</tbody>
</table>
The timer attribute in DRL rules is a string identifying either int (interval) or cron timer definitions for scheduling a rule and supports the following formats:

**Timer attribute formats**

- `timer ( int: <initial delay> <repeat interval> )`
- `timer ( cron: <cron expression> )`

**Example interval timer attributes**

- Run after a 30-second delay
  ```
  timer ( int: 30s )
  ```
- Run every 5 minutes after a 30-second delay each time
  ```
  timer ( int: 30s 5m )
  ```

**Example cron timer attribute**

- Run every 15 minutes
  ```
  timer ( cron: * 0/15 * * * ? )
  ```

Interval timers follow the semantics of `java.util.Timer` objects, with an initial delay and an optional repeat interval. Cron timers follow standard Unix cron expressions.

The following example DRL rule uses a cron timer to send an SMS text message every 15 minutes:

**Example DRL rule with a cron timer**

```
rule "Send SMS message every 15 minutes"
  timer ( cron: * 0/15 * * * ? )
  when
    $a : Alarm( on == true )
  then
    channels[ "sms" ].insert( new Sms( $a.mobileNumber, "The alarm is still on." ) );
end
```

Generally, a rule that is controlled by a timer becomes active when the rule is triggered and the rule consequence is executed repeatedly, according to the timer settings. The execution stops when the rule condition no longer matches incoming facts. However, the way the decision engine handles rules with timers depends on whether the decision engine is in active mode or in passive mode.

By default, the decision engine runs in passive mode and evaluates rules, according to the defined timer settings, when a user or an application explicitly calls `fireAllRules()`. Conversely, if a user or application calls `fireUntilHalt()`, the decision engine starts in active mode and evaluates rules continually until the user or application explicitly calls `halt()`.

When the decision engine is in active mode, rule consequences are executed even after control returns from a call to `fireUntilHalt()` and the decision engine remains reactive to any changes made to the working memory. For example, removing a fact that was involved in triggering the timer rule execution causes the repeated execution to terminate, and inserting a fact so that some rule matches causes that rule to be executed. However, the decision engine is not continually active, but is active only after a rule is executed. Therefore, the decision engine does not react to asynchronous fact insertions until the next execution of a timer-controlled rule. Disposing a KIE session terminates all timer activity.
When the decision engine is in passive mode, rule consequences of timed rules are evaluated only when `fireAllRules()` is invoked again. However, you can change the default timer-execution behavior in passive mode by configuring the KIE session with a `TimedRuleExecutionOption` option, as shown in the following example:

**KIE session configuration to automatically execute timed rules in passive mode**

```java
KieSessionConfiguration ksconf = KieServices.Factory.get().newKieSessionConfiguration();
ksconf.setOption( TimedRuleExecutionOption.YES );
KSession ksession = kbase.newKieSession(ksconf, null);
```

You can additionally set a `FILTERED` specification on the `TimedRuleExecutionOption` option that enables you to define a callback to filter those rules, as shown in the following example:

**KIE session configuration to filter which timed rules are automatically executed**

```java
KieSessionConfiguration ksconf = KieServices.Factory.get().newKieSessionConfiguration();
conf.setOption( new TimedRuleExecutionOption.FILTERED( new TimedRuleExecutionFilter() { 
    public boolean accept(Rule[] rules) {
        return rules[0].getName().equals("MyRule");
    }
} ));
```

For interval timers, you can also use an expression timer with `expr` instead of `int` to define both the delay and interval as an expression instead of a fixed value.

The following example DRL file declares a fact type with a delay and period that are then used in the subsequent rule with an expression timer:

**Example rule with an expression timer**

```drl
declare Bean
    delay   : String = "30s"
    period  : long = 60000
end

rule "Expression timer"
    timer ( expr: $d, $p )
    when
        Bean( $d : delay, $p : period )
    then
        // Actions
end
```

The expressions, such as `$d` and `$p` in this example, can use any variable defined in the pattern-matching part of the rule. The variable can be any `String` value that can be parsed into a time duration or any numeric value that is internally converted in a `long` value for a duration in milliseconds.

Both interval and expression timers can use the following optional parameters:

- `start` and `end`: A `Date` or a `String` representing a `Date` or a `long` value. The value can also be a `Number` that is transformed into a Java `Date` in the format `new Date( ((Number) n).longValue() )`.


repeat-limit: An integer that defines the maximum number of repetitions allowed by the timer. If both the end and the repeat-limit parameters are set, the timer stops when the first of the two is reached.

Example timer attribute with optional start, end, and repeat-limit parameters

```
timer (int: 30s 1h; start=3-JAN-2020, end=4-JAN-2020, repeat-limit=50)
```

In this example, the rule is scheduled for every hour, after a delay of 30 seconds each hour, beginning on 3 January 2020 and ending either on 4 January 2020 or when the cycle repeats 50 times.

If the system is paused (for example, the session is serialized and then later deserialized), the rule is scheduled only one time to recover from missing activations regardless of how many activations were missed during the pause, and then the rule is subsequently scheduled again to continue in sync with the timer setting.

The calendar attribute in DRL rules is a Quartz calendar definition for scheduling a rule and supports the following format:

**Calendar attribute format**

```
calendar "<definition or registered name>"
```

**Example calendar attributes**

```
// Exclude non-business hours
calendars "* * 0-7,18-23 ? * *"

// Weekdays only, as registered in the KIE session
calendars "weekday"
```

You can adapt a Quartz calendar based on the Quartz calendar API and then register the calendar in the KIE session, as shown in the following example:

**Adapting a Quartz Calendar**

```
Calendar weekDayCal = QuartzHelper.quartzCalendarAdapter(org.quartz.Calendar quartzCal)
ksession.getCalendars().set( "weekday", weekDayCal );
```

You can use calendars with standard rules and with rules that use timers. The calendar attribute can contain one or more comma-separated calendar names written as `String` literals.

The following example rules use both calendars and timers to schedule the rules:

**Example rules with calendars and timers**

```
rule "Weekdays are high priority"
calendars "weekday"
timer ( int:0 1h )
when
```
Alarm()
then
send( "priority high - we have an alarm" );
end

rule "Weekends are low priority"
calendar "weekend"
timer ( int:0 4h )
when
  Alarm()
then
  send( "priority low - we have an alarm" );
end

16.8. RULE CONDITIONS IN DRL (WHEN)

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 **patterns** and **constraints**, with optional **bindings** and supported rule condition elements (keywords), based on the available data objects in the package. For example, if a bank requires loan applicants to have over 21 years of age, then the **when** condition of an "Underage" rule would be `Applicant( age < 21 )`.

**NOTE**

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 a `fireAllRules()` call is made in the decision engine. This is useful if you want to use rules to set up the decision engine state.

The following example rule uses empty conditions to insert a fact every time the rule is executed:

**Example rule without conditions**

```java
rule "Always insert applicant"
  when
    // Empty
  then  // Actions to be executed once
    insert( new Applicant() );
end

// The rule is internally rewritten in the following way:

rule "Always insert applicant"
  when
    eval( true )
  then
    insert( new Applicant() );
end
```
If rule conditions use multiple patterns with no defined keyword conjunctions (such as **and**, **or**, or **not**), the default conjunction is **and**:

**Example rule without keyword conjunctions**

```java
rule "Underage"
  when
    application : LoanApplication()
    Applicant( age < 21 )
  then
    // Actions
end

// The rule is internally rewritten in the following way:

rule "Underage"
  when
    application : LoanApplication()
    and Applicant( age < 21 )
  then
    // Actions
end
```

### 16.8.1. Patterns and constraints

A **pattern** in a DRL rule condition is the segment to be matched by the decision engine. A pattern can potentially match each fact that is inserted into the working memory of the decision engine. A pattern can also contain **constraints** to further define the facts to be matched.

In the simplest form, with no constraints, a pattern matches a fact of the given type. In the following example, the type is **Person**, so the pattern will match against all **Person** objects in the working memory of the decision engine:

**Example pattern for a single fact type**

```java
Person()
```

The type does not need to be the actual class of some fact object. Patterns can refer to superclasses or even interfaces, potentially matching facts from many different classes. For example, the following pattern matches all objects in the working memory of the decision engine:

**Example pattern for all objects**

```java
Object() // Matches all objects in the working memory
```

The parentheses of a pattern enclose the constraints, such as the following constraint on the person's age:

**Example pattern with a constraint**

```java
Person( age == 50 )
```
A constraint is an expression that returns true or false. Pattern 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 bean 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

```
Person( age == 50 )
// This is the same as the following getter format:
Person( getAge() == 50 )
```

Red Hat Decision Manager uses the standard JDK Introspector class to achieve this mapping, so it follows the standard JavaBeans specification. For optimal decision engine performance, use the property access format, such as age, instead of using getters explicitly, such as getAge().

**WARNING**

Do not use property accessors to change the state of the object in a way that might affect the rules because the decision 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 fireAllRules() 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**

```
Person( age == 50 )
// If `Person.getAge()` does not exist, the compiler uses the following syntax:
```

---

CHAPTER 16. DRL (DROOLS RULE LANGUAGE) RULES

209
Person( age() == 50 )

You can also nest access properties in patterns, as shown in the following example. Nested properties are indexed by the decision engine.

Example pattern with nested property access

Person( address.houseNumber == 50 )

// This is the same as the following format:

Person( getAddress().getHouseNumber() == 50 )

WARNING

In stateful KIE sessions, use nested accessors carefully because the working memory of the decision engine is not aware of any of the nested values and does not detect when they change. Either consider the nested values immutable while any of their parent references are inserted into the working memory, or, if you want to modify a nested value, mark all of the outer facts as updated. In the previous example, when the houseNumber property changes, any Person with that Address must be marked as updated.

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

Person( age == 50 )

You can change the evaluation priority by using parentheses, as in any logical or mathematical expression:

Example evaluation order of constraints

Person( 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

Person( Math.round( weight / ( height * height ) ) < 25.0 )
WARNING

Do not use constraints to change the state of the object in a way that might affect the rules because the decision 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:

- Person( incrementAndGetAge() == 10 ) // Do not do this.
- Person( 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 `Person( 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 `Person( 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 decision 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**

- Person( 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:
  Person( age > 50, weight > 80 )

- // Person is at least 50 years old, weighs at least 80 kilograms, and is taller than 2 meters:
  Person( age > 50, weight > 80, height > 2 )
NOTE

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 decision 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:
Person( ( age > 50, weight > 80 ) || height > 2 )

// Use the following format instead:
Person( ( age > 50 && weight > 80 ) || height > 2 )
```

### 16.8.2. Bound variables in patterns and constraints

You can bind variables to 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. To differentiate more easily between variables and fields in a rule, use the standard format $variable for variables, especially in complex rules. This convention is helpful but not required in DRL.

For example, the following DRL rule uses the variable $p for a pattern with the Person fact:

**Pattern with a bound variable**

```
rule "simple rule"
  when
    $p : Person()
  then
    System.out.println("Person " + $p);
end
```

Similarly, you can also bind variables to properties in pattern constraints, as shown in the following example:

```
// Two persons of the same age:
Person($firstAge : age) // Binding
Person( age == $firstAge ) // Constraint expression
```
NOTE

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:
Person( $age : age * 2 < 100 )

// Use the following format instead:
Person( age * 2 < 100, $age : age )

The decision engine does not support bindings to the same declaration, but does support unification of arguments across several properties. While positional arguments are always processed with unification, the unification symbol := exists for named arguments.

The following example patterns unify the age property across two Person facts:

Example pattern with unification

Person( $age := age )
Person( $age := age )

Unification declares a binding for the first occurrence and constrains to the same value of the bound field for sequence occurrences.

16.8.3. 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

Person( name == "mark", address.city == "london", address.country == "uk" )

You can group these property accessors to nested objects with the syntax .( <constraints> ) for more readable rules, as shown in the following example:

Example pattern with grouped constraints

Person( name == "mark", address.( city == "london", country == "uk") )

NOTE

The period prefix . differentiates the nested object constraints from a method call.

When you work with nested objects in patterns, you can use the syntax <type)#<subtype> 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:
Person( name == "mark", address#LongAddress.country == "uk" )

// Inline casting with fully qualified class name:
Person( name == "mark", address#org.domain.LongAddress.country == "uk" )

// Multiple inline casts:
Person( name == "mark", address#LongAddress.country#DetailedCountry.population > 1000000 )

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.

You can use the instanceof operator to infer the results of the specified type in subsequent uses of that field with the pattern, as shown in the following example:

Person( name == "mark", address instanceof LongAddress, address.country == "uk" )

If an inline cast is not possible (for example, if instanceof returns false), the evaluation is considered false.

16.8.4. Date literal in constraints

By default, the decision 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

Person( bornBefore < "27-Oct-2009" )

16.8.5. Supported operators in DRL pattern constraints

DRL supports standard Java semantics for operators in pattern 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 patterns with nested objects

// Ungrouped property accessors:
Person( name == "mark", address.city == "london", address.country == "uk" )

// Grouped property accessors:
Person( name == "mark", address.( city == "london", country == "uk" ) )
The period prefix . differentiates the nested object constraints from a method call.

**Example patterns with inline casting to a subtype**

// Inline casting with subtype name:
Person( name == "mark", address#LongAddress.country == "uk" )

// Inline casting with fully qualified class name:
Person( name == "mark", address#org.domain.LongAddress.country == "uk" )

// Multiple inline casts:
Person( name == "mark", address#LongAddress.country#DetailedCountry.population > 1000000 )

!.

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**

Person( $streetName : address!.street )

// This is internally rewritten in the following way:
Person( 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`:
Person(childList[0].age == 18)

// The following format is the same as `credentialMap.get("jdoe").isValid()`:
Person(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**

Person( birthDate < $otherBirthDate )

Person( 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**

```java
Person( firstName == "John" )
```

// This is similar to the following formats:

```java
java.util.Objects.equals(person.getFirstName(), "John")
"John".equals(person.getFirstName())
```

**Example constraint with null-safe not equality**

```java
Person( firstName != "John" )
```

// This is similar to the following format:

```java
!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 `&&`:
```java
Person(age > 30 && < 40)
```

// Complex abbreviated combined relation using groupings:
```java
Person(age ((> 30 && < 40) || (> 20 && < 25)))
```

// Mixing abbreviated combined relation with constraint connectives:
```java
Person(age > 30 && < 40 || location == "london")
```

**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**

```java
Person(country matches "(USA)?\S*UK")
```

```java
Person(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**

```java
// 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**

```java
// Sting literal with a specified field:
Person( fullName contains "Jr" )

Person( fullName not contains "Jr" )

// String literal with a variable:
Person( fullName contains $var )

Person( fullName not contains $var )
```

**NOTE**

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**

```java
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":
Person( 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:
Message( routingValue str[startsWith] "R1" )

// Verify what the String ends with:
Message( routingValue str[endsWith] "R2" )

// Verify the length of the String:
Message( 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 !=.

Example constraints with in and notin

Person( $color : favoriteColor )
Color( type in ( "red", "blue", $color ) )

Person( $color : favoriteColor )
Color( type notin ( "red", "blue", $color ) )

16.8.6. 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 16.2. Operator precedence in DRL pattern constraints

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Operators</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested or null-safe property access</td>
<td>..().!</td>
<td>Not standard Java semantics</td>
</tr>
<tr>
<td>List or Map access</td>
<td>[]</td>
<td>Not standard Java semantics</td>
</tr>
<tr>
<td>Constraint binding</td>
<td>:</td>
<td>Not standard Java semantics</td>
</tr>
</tbody>
</table>
16.8.7. 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.

### Example patterns with and

//Infix `and`
Color( colorType : type ) and Person( favoriteColor == colorType )

//Infix `and` with grouping:
(Color( colorType : type ) and Person( favoriteColor == colorType )) or Person( favoriteColor == colorType )
NOTE

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 : (Person( name == "Romeo" ) and Person( 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.

Example patterns with *or*

```
//Infix `or`:
Color( colorType : type ) or Person( favoriteColor == colorType )

//Infix `or` with grouping:
(Color( colorType : type ) or (Person( favoriteColor == colorType ) and Person( favoriteColor ==
colorType )))

// Prefix `or`:
(or Color( colorType : type ) Person( favoriteColor == colorType ))
```

Example patterns with *or* and pattern binding

```
pensioner : (Person( sex == "f", age > 60 ) or Person( sex == "m", age > 65 ))

(or pensioner : Person( sex == "f", age > 60 )
pensioner : Person( 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 decision 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 ()..

Example patterns with exists

exists Person( firstName == "John")
exists (Person( firstName == "John", age == 42 ))
exists (Person( firstName == "John" ) and
       Person( 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 ()..

Example patterns with not

not Person( firstName == "John")
not (Person( firstName == "John", age == 42 ))
not (Person( firstName == "John" ) and
       Person( 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.

Example rule with forall

rule "All full-time employees have red ID badges"
when
   forall( $emp : Employee( type == "fulltime" )
             Employee( 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 decision 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( Employee( 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 : Employee()
       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 : Employee()
       HealthCare( employee == $emp )
       DentalCare( employee == $emp )
   )
then
   // True, not all employees have health and dental care.
end

NOTE

The format forall( p1 p2 p3 …) is equivalent to not( p1 and not( and p2 p3 … )).

from

Use this to specify a data source for a pattern. This enables the decision 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.

Example rule with from and pattern binding

rule "Validate zipcode"
when
   Person( $personAddress : address )
   Address( zipcode == "23920W" ) from $personAddress
Example rule with **from** and a graph notation

```drl
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

```drl
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
```

**NOTE**

For large collections of objects, instead of adding an object with a large graph that the decision 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:

```drl
when
  $order : Order()
  OrderItem( value > 100, order == $order )
```

Example rule with **from** and **lock-on-active** rule attribute

```drl
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
```

```drl
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

**IMPORTANT**

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 decision 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**

```drl
// 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**

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 decision 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**
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

Example Java application code with EntryPoint object and inserted facts

```java
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);
```

collect

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 decision 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.

Example rule with `collect`

```java
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 decision 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`

```java
import java.util.List
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);
```
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

Use this to iterate over a collection of objects, execute custom actions for each of the elements, and return one or more result objects (if the constraints evaluate to true). This element is a more flexible and powerful form of the collect condition element. 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( <source pattern>; <functions> [;<constraints>] )

NOTE

Although the decision engine supports alternate formats for the accumulate element for backward compatibility, this format is preferred for optimal performance in rules and applications.

The decision 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 : Sensor()
    accumulate( Reading( 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 \texttt{average} function with \texttt{accumulate} to calculate the average profit for all items in an order:

\textbf{Example rule with \texttt{accumulate} to calculate average profit}

rule "Average profit"
when
$order : Order()
    accumulate( OrderItem( 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 \texttt{accumulate} conditions, create a Java class that implements the \texttt{org.kie.api.runtime.rule.AccumulateFunction} interface. For example, the following Java class defines a custom implementation of an \texttt{AverageData} function:

\textbf{Example Java class with custom implementation of \texttt{average} function}

\begin{verbatim}
// 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;
        public double total = 0;

        public AverageData() {}  

        public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  
            count = in.readInt();
            total = in.readDouble();  
        }
    }
}
\end{verbatim}
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;
}
To use the custom function in a DRL rule, import the function using the `import accumulate` statement:

**Format to import a custom function**

```plaintext
import accumulate <class_name> <function_name>
```

**Example rule with the imported average function**

```plaintext
import accumulate AverageAccumulateFunction.AverageData average

rule "Average profit"
when
  $order : Order()
  accumulate( OrderItem( order == $order, $cost : cost, $price : price );
            $avgProfit : average( 1 - $cost / $price ) )
then
  // Average profit for `$order` is `$avgProfit`.
end
```

16.8.8. **OOPath syntax with graphs of objects in DRL rule conditions**

OOPath is an object-oriented syntax extension of XPath that is designed for browsing graphs of objects in DRL rule condition constraints. OOPath uses the compact notation from XPath for navigating through related elements while handling collections and filtering constraints, and is specifically useful for graphs of objects.

When the field of a fact is a collection, you can use the `from` condition element (keyword) to bind and reason over all the items in that collection one by one. If you need to browse a graph of objects in the rule condition constraints, the extensive use of the `from` condition element results in a verbose and repetitive syntax, as shown in the following example:

**Example rule that browses a graph of objects with from**

```plaintext
rule "Find all grades for Big Data exam"
when
  $student: Student( $plan: plan )
  $exam: Exam( course == "Big Data" ) from $plan.exams
  $grade: Grade() from $exam.grades
then
  // Actions
end
```

In this example, the domain model contains a **Student** object with a **Plan** of study. The **Plan** can have
zero or more Exam instances and an Exam can have zero or more Grade instances. Only the root object of the graph, the Student in this case, needs to be in the working memory of the decision engine for this rule setup to function.

As a more efficient alternative to using extensive from statements, you can use the abbreviated OOPPath syntax, as shown in the following example:

**Example rule that browses a graph of objects with OOPPath syntax**

```
rule "Find all grades for Big Data exam"
  when
  Student( $grade: /plan/exams[course == "Big Data"]/grades )
  then
    // Actions
  end
```

Formally, the core grammar of an OOPPath expression is defined in extended Backus-Naur form (EBNF) notation in the following way:

**EBNF notation for OOPPath expressions**

```
OOPExpr = [ID ( ":" | ":=" )] ( "/" | "?/" ) OOPSegment { ( "/" | "?/" | "." ) OOPSegment } ;
OOPSegment = ID ["#" ID] [["[" ( Number | Constraints ) "]"]
```

In practice, an OOPPath expression has the following features and capabilities:

- Starts with a forward slash / or with a question mark and forward slash ?/ if it is a non-reactive OOPPath expression (described later in this section).
- Can dereference a single property of an object with the period . operator.
- Can dereference multiple properties of an object with the forward slash / operator. If a collection is returned, the expression iterates over the values in the collection.
- Can filter out traversed objects that do not satisfy one or more constraints. The constraints are written as predicate expressions between square brackets, as shown in the following example:

**Constraints as a predicate expression**

```
Student( $grade: /plan/exams[ course == "Big Data" ]/grades )
```

- Can downcast a traversed object to a subclass of the class declared in the generic collection. Subsequent constraints can also safely access the properties declared only in that subclass, as shown in the following example. Objects that are not instances of the class specified in this inline cast are automatically filtered out.

**Constraints with downcast objects**

```
Student( $grade: /plan/exams#AdvancedExam[ course == "Big Data", level > 3 ]/grades )
```

- Can backreference an object of the graph that was traversed before the currently iterated graph. For example, the following OOPPath expression matches only the grades that are above the average for the passed exam:
Constraints with backreferenced object

- Student( $grade: /plan/exams/grades[ result > ../averageResult ] )

- Can recursively be another OOPath expression, as shown in the following example:

  Recursive constraint expression

- Student( $exam: /plan/exams[ /grades[ result > 20 ] ] )

- Can access objects by their index between square brackets [], as shown in the following example. To adhere to Java convention, OOPath indexes are 0-based, while XPath indexes are 1-based.

Constraints with access to objects by index

- Student( $grade: /plan/exams[0]/grades )

OOPath expressions can be reactive or non-reactive. The decision engine does not react to updates involving a deeply nested object that is traversed during the evaluation of an OOPath expression.

To make these objects reactive to changes, modify the objects to extend the class org.drools.core.phreak.ReactiveObject. After you modify an object to extend the ReactiveObject class, the domain object invokes the inherited method notifyModification to notify the decision engine when one of the fields has been updated, as shown in the following example:

Example object method to notify the decision engine that an exam has been moved to a different course

```java
public void setCourse(String course) {
    this.course = course;
    notifyModification(this);
}
```

With the following corresponding OOPath expression, when an exam is moved to a different course, the rule is re-executed and the list of grades matching the rule is recomputed:

Example OOPath expression from "Big Data" rule

- Student( $grade: /plan/exams[ course == "Big Data" ]/grades )

You can also use the ?/ separator instead of the / separator to disable reactivity in only one sub-portion of an OOPath expression, as shown in the following example:

Example OOPath expression that is partially non-reactive

- Student( $grade: /plan/exams[ course == "Big Data" ]?/grades )

With this example, the decision engine reacts to a change made to an exam or if an exam is added to the plan, but not if a new grade is added to an existing exam.

If an OOPath portion is non-reactive, all remaining portions of the OOPath expression also become non-reactive. For example, the following OOPath expression is completely non-reactive:
Example OOPath expression that is completely non-reactive

Student( $grade: ?/plan/exams[ course == "Big Data" ]/grades )

For this reason, you cannot use the ?/ separator more than once in the same OOPath expression. For example, the following expression causes a compilation error:

Example OOPath expression with duplicate non-reactivity markers

Student( $grade: /plan?/exams[ course == "Big Data" ]?/grades )

Another alternative for enabling OOPath expression reactivity is to use the dedicated implementations for List and Set interfaces in Red Hat Decision Manager. These implementations are the ReactiveList and ReactiveSet classes. A ReactiveCollection class is also available. The implementations also provide reactive support for performing mutable operations through the Iterator and ListIterator classes.

The following example class uses these classes to configure OOPath expression reactivity:

Example Java class to configure OOPath expression reactivity

```java
public class School extends AbstractReactiveObject {
    private String name;
    private final List<Child> children = new ReactiveList<Child>();

    public void setName(String name) {
        this.name = name;
        notifyModification();
    }

    public void addChild(Child child) {
        children.add(child);
        // No need to call `notifyModification()` here
    }
}
```

1. Uses the ReactiveList instance for reactive support over the standard Java List instance.
2. Uses the required notifyModification() method for when a field is changed in reactive support.
3. The children field is a ReactiveList instance, so the notifyModification() method call is not required. The notification is handled automatically, like all other mutating operations performed over the children field.

16.9. RULE ACTIONS IN DRL (THEN)

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. Actions consist of one or more methods that execute consequences based on the rule conditions and on available data objects in the package. For example, if a bank requires loan applicants to have over 21 years of age (with a rule condition Applicant( age < 21 )) and a loan applicant is under 21 years old, the then action of an "Underage" rule would be setApproved( false ), declining the loan because the applicant is under age.
The main purpose of rule actions is to insert, delete, or modify data in the working memory of the decision 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**

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

**16.9.1. Supported rule action methods in DRL**

DRL supports the following rule action methods that you can use in DRL rule actions. You can use these methods to modify the working memory of the decision engine without having to first reference a working memory instance. These methods act as shortcuts to the methods provided by the KnowledgeHelper class in your Red Hat Decision Manager distribution.

For all rule action methods, download the Red Hat Decision Manager 7.10.0 Source Distribution ZIP file from the Red Hat Customer Portal and navigate to `~/rhdm-7.10.0-sources/src/drools-$VERSION/drools-core/src/main/java/org/drools/core/spi/KnowledgeHelper.java`.

**set**

Use this to set the value of a field.

```drl
set<field> ( <value> )
```

**Example rule action to set the values of a loan application approval**

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

**modify**

Use this to specify fields to be modified for a fact and to notify the decision engine of the change. This method provides a structured approach to fact updates. It combines the `update` operation with setter calls to change object fields.

```drl
modify ( <fact-expression> ) {
    <expression>,
    <expression>,
    ...
}
```

**Example rule action to modify a loan application amount and approval**

```drl
modify( LoanApplication ) {
    setAmount( 100 ),
    setApproved( true )
}
```
update

Use this to specify fields and the entire related fact to be updated and to notify the decision engine of the change. After a fact has changed, you must call update before changing another fact that might be affected by the updated values. To avoid this added step, use the modify method instead.

```
update ( <object, <handle> )  // Informs the decision engine that an object has changed

update ( <object> )  // Causes `KieSession` to search for a fact handle of the object
```

Example rule action to update a loan application amount and approval

```
LoanApplication.setAmount( 100 );
update( LoanApplication );
```

NOTE

If you provide property-change listeners, you do not need to call this method when an object changes. For more information about property-change listeners, see Decision engine in Red Hat Decision Manager.

insert

Use this to insert a new fact into the working memory of the decision engine and to define resulting fields and values as needed for the fact.

```
insert( new <object> );
```

Example rule action to insert a new loan applicant object

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

insertLogical

Use this to insert a new fact logically into the decision engine. The decision engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts must be retracted explicitly. 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.

```
insertLogical( new <object> );
```

Example rule action to logically insert a new loan applicant object

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

delete

Use this to remove an object from the decision engine. The keyword retract is also supported in DRL and executes the same action, but delete is typically preferred in DRL code for consistency with the keyword insert.

```
delete( <object> );
```
Example rule action to delete a loan applicant object

```java
delete( Applicant );
```

16.9.2. Other rule action methods from drools and kcontext variables

In addition to the standard rule action methods, the decision engine supports methods in conjunction with the predefined `drools` and `kcontext` variables that you can also use in rule actions.

You can use the `drools` variable to call methods from the `KnowledgeHelper` class in your Red Hat Decision Manager distribution, which is also the class that the standard rule action methods are based on. For all `drools` rule action options, download the Red Hat Decision Manager 7.10.0 Source Distribution ZIP file from the Red Hat Customer Portal and navigate to `~/rhdm-7.10.0-sources/src/drools-$VERSION/drools-core/src/main/java/org/drools/core/spi/KnowledgeHelper.java`.

The following examples are common methods that you can use with the `drools` variable:

- `drools.halt()`: Terminates rule execution if a user or application has previously called `fireUntilHalt()`. When a user or application calls `fireUntilHalt()`, the decision engine starts in **active mode** and evaluates rules continually until the user or application explicitly calls `halt()`. Otherwise, by default, the decision engine runs in **passive mode** and evaluates rules only when a user or an application explicitly calls `fireAllRules()`.

- `drools.getWorkingMemory()`: Returns the `WorkingMemory` object.

- `drools.setFocus( "<agenda_group>" )`: Sets the focus to a specified agenda group to which the rule belongs.

- `drools.getRule().getName()`: Returns the name of the rule.

- `drools.getTuple(), drools.getActivation()`: Returns the `Tuple` that matches the currently executing rule and then delivers the corresponding `Activation`. These calls are useful for logging and debugging purposes.

You can use the `kcontext` variable with the `getKieRuntime()` method to call other methods from the `KieContext` class and, by extension, the `RuleContext` class in your Red Hat Decision Manager distribution. The full Knowledge Runtime API is exposed through the `kcontext` variable and provides extensive rule action methods. For all `kcontext` rule action options, download the Red Hat Decision Manager 7.10.0 Source Distribution ZIP file from the Red Hat Customer Portal and navigate to `~/rhdm-7.10.0-sources/src/kie-api-parent-$VERSION/kie-api/src/main/java/org/kie/api/runtime/rule/RuleContext.java`.

The following examples are common methods that you can use with the `kcontext.getKieRuntime()` variable-method combination:

- `kcontext.getKieRuntime().halt()`: Terminates rule execution if a user or application has previously called `fireUntilHalt()`. This method is equivalent to the `drools.halt()` method. When a user or application calls `fireUntilHalt()`, the decision engine starts in **active mode** and evaluates rules continually until the user or application explicitly calls `halt()`. Otherwise, by default, the decision engine runs in **passive mode** and evaluates rules only when a user or an application explicitly calls `fireAllRules()`.

- `kcontext.getKieRuntime().getAgenda()`: Returns a reference to the KIE session `Agenda`, and in turn provides access to rule activation groups, rule agenda groups, and ruleflow groups.
Example call to access agenda group "CleanUp" and set the focus

```java
kcontext.getKieRuntime().getAgenda().getAgendaGroup("CleanUp").setFocus();
```

This example is equivalent to `drools.setFocus("CleanUp").

- `kcontext.getKieRuntime().getQueryResults(<string> query)`: Runs a query and returns the results. This method is equivalent to `drools.getKieRuntime().getQueryResults()`.

- `kcontext.getKieRuntime().getKieBase()`: Returns the `KieBase` object. The KIE base is the source of all the knowledge in your rule system and the originator of the current KIE session.

- `kcontext.getKieRuntime().setGlobal(~.getGlobal(~.getGlobals())`: Sets or retrieves global variables.

- `kcontext.getKieRuntime().getEnvironment()`: Returns the runtime `Environment`, similar to your operating system environment.

16.9.3. Advanced rule actions with conditional and named consequences

In general, effective rule actions are small, declarative, and readable. However, in some cases, the limitation of having a single consequence for each rule can be challenging and lead to verbose and repetitive rule syntax, as shown in the following example rules:

**Example rules with verbose and repetitive syntax**

```java
rule "Give 10% discount to customers older than 60"
when
  $customer : Customer( age > 60 )
then
  modify($customer) { setDiscount( 0.1 ) };
end

rule "Give free parking to customers older than 60"
when
  $customer : Customer( age > 60 )
  $car : Car( owner == $customer )
then
  modify($car) { setFreeParking( true ) };
end
```

A partial solution to the repetition is to make the second rule extend the first rule, as shown in the following modified example:

**Partially enhanced example rules with an extended condition**

```java
rule "Give 10% discount to customers older than 60"
when
  $customer : Customer( age > 60 )
then
  modify($customer) { setDiscount( 0.1 ) };
end

rule "Give free parking to customers older than 60"
extends "Give 10% discount to customers older than 60"
```
As a more efficient alternative, you can consolidate the two rules into a single rule with modified conditions and labelled corresponding rule actions, as shown in the following consolidated example:

**Consolidated example rule with conditional and named consequences**

```
rule "Give 10% discount and free parking to customers older than 60"
when
  $customer : Customer( age > 60 )
  do[giveDiscount]
  $car : Car( owner == $customer )
then
  modify($car) { setFreeParking( true ) };
  then[giveDiscount]
  modify($customer) { setDiscount( 0.1 ) };
end
```

This example rule uses two actions: the usual default action and another action named `giveDiscount`. The `giveDiscount` action is activated in the condition with the keyword `do` when a customer older than 60 years old is found in the KIE base, regardless of whether or not the customer owns a car.

You can configure the activation of a named consequence with an additional condition, such as the `if` statement in the following example. The condition in the `if` statement is always evaluated on the pattern that immediately precedes it.

**Consolidated example rule with an additional condition**

```
rule "Give free parking and 10% discount to over 60 Golden customer and 5% to Silver ones"
when
  $customer : Customer( age > 60 )
  if ( type == "Golden" ) do[giveDiscount]
  $car : Car( owner == $customer )
then
  modify($car) { setFreeParking( true ) };
  then[giveDiscount]
  modify($customer) { setDiscount( 0.1 ) };
end
```

You can also evaluate different rule conditions using a nested `if` and `else if` construct, as shown in the following more complex example:

**Consolidated example rule with more complex conditions**

```
rule "Give free parking and 10% discount to over 60 Golden customer and 5% to Silver ones"
when
  $customer : Customer( age > 60 )
  if ( type == "Golden" ) do[giveDiscount10]
  else if ( type == "Silver" ) break[giveDiscount5]
  $car : Car( owner == $customer )
then
```

---

CHAPTER 16. DRL (DROOLS RULE LANGUAGE) RULES
modify($car) { setFreeParking( true ) };
then[giveDiscount10]
modify($customer) { setDiscount( 0.1 ) };
then[giveDiscount5]
modify($customer) { setDiscount( 0.05 ) };
end

This example rule gives a 10% discount and free parking to Golden customers over 60, but only a 5% discount without free parking to Silver customers. The rule activates the consequence named giveDiscount5 with the keyword break instead of do. The keyword do schedules a consequence in the decision engine agenda, enabling the remaining part of the rule conditions to continue being evaluated, while break blocks any further condition evaluation. If a named consequence does not correspond to any condition with do but is activated with break, the rule fails to compile because the conditional part of the rule is never reached.

16.10. 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 decision engine when the DRL file is processed.

Example rule with comments

rule "Underage"
  // This is a single-line comment.
  when
    $application : LoanApplication() // This is an in-line comment.
    Applicant( age < 21 )
  then
    /* This is a multi-line comment
        in the rule actions. */
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
end

IMPORTANT
The hash symbol # is not supported for DRL comments.

16.11. ERROR MESSAGES FOR DRL TROUBLESHOOTING

Red Hat Decision Manager 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 16.1. Error message format for DRL file problems

- **1st Block**: Error code
- **2nd Block**: Line and column in the DRL source where the error occurred
Red Hat Decision Manager 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

```java
1: rule "simple rule"
2:   when
3:     exists Person()
4:     exits Student()  // 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

```java
1: package org.drools.examples;
2: rule    // Must be `rule "rule name"` (or `rule rule_name` if no spacing)
3:     Object()
4:   then
5:     System.out.println("A RHS");
6: 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

```java
1: rule "simple rule"
2:   when
3:     Student( 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
```
NOTE

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 : Person(
4:       // Must be a complete rule statement
```

Error message

```
[ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule "simple rule" in pattern Person
```

NOTE

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( Car( ( type == "tesla", price == 10000 ) || ( type == "kia", price == 1000 ) ) from $carList )
6:       // Must use `&&` operators instead of commas ,
7:     then
8:     System.out.println("OK");
9: 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: Person( name=="Michael" )
11: then
12: $p.name = "other";
13: System.out.println(p.name);
14: end
```

Error message

```
[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.

**NOTE**

This error is similar to 102: mismatched input, but usually involves DRL keywords.

104: trailing semi-colon not allowed

Indicates that an eval() clause in a rule condition uses a semicolon ; but must not use one.

Example rule with eval() and trailing semicolon

```
1: rule "simple rule"
2: when
3: eval( abc(); ) // Must not use semicolon `;`
4: then
5: end
```

Error message

```
[ERR 104] Line 3:4 trailing semi-colon not allowed in rule "simple rule"
```

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

```plaintext
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.

**NOTE**

If you encounter other DRL error messages that you cannot resolve, contact your Red Hat Technical Account Manager.

### 16.12. RULE UNITS IN DRL RULE SETS

Rule units are groups of data sources, global variables, and DRL rules that function together for a specific purpose. You can use rule units to partition a rule set into smaller units, bind different data sources to those units, and then execute the individual unit. Rule units are an enhanced alternative to rule-grouping DRL attributes such as rule agenda groups or activation groups for execution control.

Rule units are helpful when you want to coordinate rule execution so that the complete execution of one rule unit triggers the start of another rule unit and so on. For example, assume that you have a set of rules for data enrichment, another set of rules that processes that data, and another set of rules that extract the output from the processed data. If you add these rule sets into three distinct rule units, you can coordinate those rule units so that complete execution of the first unit triggers the start of the second unit and the complete execution of the second unit triggers the start of third unit.

To define a rule unit, implement the RuleUnit interface as shown in the following example:

**Example rule unit class**

```java
package org.mypackage.myunit;

public static class AdultUnit implements RuleUnit {
    private int adultAge;
    private DataSource<Person> persons;

    public AdultUnit() {}  // default constructor

    public AdultUnit( DataSource<Person> persons, int age ) {
        this.persons = persons;
        this.age = age;
    }
}
```
In this example, `persons` is a source of facts of type `Person`. A rule unit data source is a source of the data processed by a given rule unit and represents the entry point that the decision engine uses to evaluate the rule unit. The `adultAge` global variable is accessible from all the rules belonging to this rule unit. The last two methods are part of the rule unit life cycle and are invoked by the decision engine.

The decision engine supports the following optional life-cycle methods for rule units:

<table>
<thead>
<tr>
<th>Method</th>
<th>Invoked when</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>onStart()</code></td>
<td>Rule unit execution starts</td>
</tr>
<tr>
<td><code>onEnd()</code></td>
<td>Rule unit execution ends</td>
</tr>
<tr>
<td><code>onSuspend()</code></td>
<td>Rule unit execution is suspended (used only with <code>runUntilHalt()</code>)</td>
</tr>
<tr>
<td><code>onResume()</code></td>
<td>Rule unit execution is resumed (used only with <code>runUntilHalt()</code>)</td>
</tr>
<tr>
<td><code>onYield(RuleUnit other)</code></td>
<td>The consequence of a rule in the rule unit triggers the execution of a different rule unit</td>
</tr>
</tbody>
</table>

You can add one or more rules to a rule unit. By default, all the rules in a DRL file are automatically associated with a rule unit that follows the naming convention of the DRL file name. If the DRL file is in the same package and has the same name as a class that implements the `RuleUnit` interface, then all of the rules in that DRL file implicitly belong to that rule unit. For example, all the rules in the `AdultUnit.drl` file in the `org.mypackage.myunit` package are automatically part of the rule unit `org.mypackage.myunit.AdultUnit`. 

```
// A data source of `Persons` in this rule unit:
public DataSource<Person> getPersons() {
    return persons;
}

// A global variable in this rule unit:
public int getAdultAge() {
    return adultAge;
}

// Life-cycle methods:
@Override
public void onStart() {
    System.out.println("AdultUnit started.");
}

@Override
public void onEnd() {
    System.out.println("AdultUnit ended.");
}
```
To override this naming convention and explicitly declare the rule unit that the rules in a DRL file belong to, use the `unit` keyword in the DRL file. The `unit` declaration must immediately follow the package declaration and contain the name of the class in that package that the rules in the DRL file are part of.

**Example rule unit declaration in a DRL file**

```
package org.mypackage.myunit
unit AdultUnit

rule Adult
  when
    $p : Person(age >= adultAge) from persons
  then
    System.out.println($p.getName() + " is adult and greater than " + adultAge);
end
```

**WARNING**

Do not mix rules with and without a rule unit in the same KIE base. Mixing two rule paradigms in a KIE base results in a compilation error.

You can also rewrite the same pattern in a more convenient way using OOPath notation, as shown in the following example:

**Example rule unit declaration in a DRL file that uses OOPath notation**

```
package org.mypackage.myunit
unit AdultUnit

rule Adult
  when
    $p : /persons[age >= adultAge]
  then
    System.out.println($p.getName() + " is adult and greater than " + adultAge);
end
```

**NOTE**

OOPath is an object-oriented syntax extension of XPath that is designed for browsing graphs of objects in DRL rule condition constraints. OOPath uses the compact notation from XPath for navigating through related elements while handling collections and filtering constraints, and is specifically useful for graphs of objects.

In this example, any matching facts in the rule conditions are retrieved from the `persons` data source defined in the `DataSource` definition in the rule unit class. The rule condition and action use the `adultAge` variable in the same way that a global variable is defined at the DRL file level.

To execute one or more rule units defined in a KIE base, create a new `RuleUnitExecutor` class bound to the KIE base, create the rule unit from the relevant data source, and run the rule unit executor:
Example rule unit execution

```java
// Create a `RuleUnitExecutor` class and bind it to the KIE base:
KieBase kbase = kieContainer.getKieBase();
RuleUnitExecutor executor = RuleUnitExecutor.create().bind(kbase);

// Create the `AdultUnit` rule unit using the `persons` data source and run the executor:
RuleUnit adultUnit = new AdultUnit(persons, 18);
executor.run(adultUnit);
```

Rules are executed by the `RuleUnitExecutor` class. The `RuleUnitExecutor` class creates KIE sessions and adds the required `DataSource` objects to those sessions, and then executes the rules based on the `RuleUnit` that is passed as a parameter to the `run()` method.

The example execution code produces the following output when the relevant `Person` facts are inserted in the `persons` data source:

Example rule unit execution output

```
org.mypackage.myunit.AdultUnit started.
Jane is adult and greater than 18
John is adult and greater than 18
org.mypackage.myunit.AdultUnit ended.
```

Instead of explicitly creating the rule unit instance, you can register the rule unit variables in the executor and pass to the executor the rule unit class that you want to run, and then the executor creates an instance of the rule unit. You can then set the `DataSource` definition and other variables as needed before running the rule unit.

Alternate rule unit execution option with registered variables

```
executor.bindVariable( "persons", persons );
.bindVariable( "adultAge", 18 );
executor.run( AdultUnit.class );
```

The name that you pass to the `RuleUnitExecutor.bindVariable()` method is used at run time to bind the variable to the field of the rule unit class with the same name. In the previous example, the `RuleUnitExecutor` inserts into the new rule unit the data source bound to the "persons" name and inserts the value 18 bound to the String "adultAge" into the fields with the corresponding names inside the `AdultUnit` class.

To override this default variable-binding behavior, use the `@UnitVar` annotation to explicitly define a logical binding name for each field of the rule unit class. For example, the field bindings in the following class are redefined with alternative names:

Example code to modify variable binding names with `@UnitVar`

```java
package org.mypackage.myunit;

public static class AdultUnit implements RuleUnit {
    @UnitVar("minAge")
    private int adultAge = 18;
```
You can then bind the variables to the executor using those alternative names and run the rule unit:

Example rule unit execution with modified variable names

```java
executor.bindVariable("data", persons);
    .bindVariable("minAge", 18);
executor.run(AdultUnit.class);
```

You can execute a rule unit in passive mode by using the `run()` method (equivalent to invoking `fireAllRules()` on a KIE session) or in active mode using the `runUntilHalt()` method (equivalent to invoking `fireUntilHalt()` on a KIE session). By default, the decision engine runs in passive mode and evaluates rule units only when a user or an application explicitly calls `run()` (or `fireAllRules()` for standard rules). If a user or application calls `runUntilHalt()` for rule units (or `fireUntilHalt()` for standard rules), the decision engine starts in active mode and evaluates rule units continually until the user or application explicitly calls `halt()`.

If you use the `runUntilHalt()` method, invoke the method on a separate execution thread to avoid blocking the main thread:

Example rule unit execution with `runUntilHalt()` on a separate thread

```java
new Thread( () -> executor.runUntilHalt( adultUnit ) ).start();
```

16.12.1. Data sources for rule units

A rule unit data source is a source of the data processed by a given rule unit and represents the entry point that the decision engine uses to evaluate the rule unit. A rule unit can have zero or more data sources and each `DataSource` definition declared inside a rule unit can correspond to a different entry point into the rule unit executor. Multiple rule units can share a single data source, but each rule unit must use different entry points through which the same objects are inserted.

You can create a `DataSource` definition with a fixed set of data in a rule unit class, as shown in the following example:

Example data source definition

```java
DataSource<Person> persons = DataSource.create( new Person( "John", 42 ),
              new Person( "Jane", 44 ),
              new Person( "Sally", 4 ) );
```

Because a data source represents the entry point of the rule unit, you can insert, update, or delete facts in a rule unit:

Example code to insert, modify, and delete a fact in a rule unit

```java
// Insert a fact:
Person john = new Person( "John", 42);
FactHandle johnFh = persons.insert( john );

// Modify the fact and optionally specify modified properties (for property reactivity):
```
Rule units are helpful when you want to coordinate rule execution so that the execution of one rule unit triggers the start of another rule unit and so on.

To facilitate rule unit execution control, the decision engine supports the following rule unit methods that you can use in DRL rule actions to coordinate the execution of rule units:

- **drools.run()**: Triggers the execution of a specified rule unit class. This method imperatively interrupts the execution of the rule unit and activates the other specified rule unit.

- **drools.guard()**: Prevents (guards) a specified rule unit class from being executed until the associated rule condition is met. This method declaratively schedules the execution of the other specified rule unit. When the decision engine produces at least one match for the condition in the guarding rule, the guarded rule unit is considered active. A rule unit can contain multiple guarding rules.

As an example of the **drools.run()** method, consider the following DRL rules that each belong to a specified rule unit. The **NotAdult** rule uses the **drools.run(AdultUnit.class)** method to trigger the execution of the **AdultUnit** rule unit:

**Example DRL rules with controlled execution using drools.run()**

```java
package org.mypackage.myunit
unit AdultUnit

rule Adult
  when
    Person(age >= 18, $name : name) from persons
  then
    System.out.println($name + " is adult");
end

package org.mypackage.myunit
unit NotAdultUnit

rule NotAdult
  when
    $p : Person(age < 18, $name : name) from persons
  then
    System.out.println($name + " is NOT adult");
    modify($p) { setAge(18); }
    drools.run( AdultUnit.class );
end
```

The example also uses a **RuleUnitExecutor** class created from the KIE base that was built from these rules and a **DataSource** definition of **persons** bound to it:

**Example rule executor and data source definitions**

```java
john.setAge( 43 );
persons.update( johnFh, john, "age" );

// Delete the fact:
persons.delete( johnFh );
```
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );
DataSource<Person> persons = executor.newDataSource( "persons",
    new Person( "John", 42 ),
    new Person( "Jane", 44 ),
    new Person( "Sally", 4 ) );

In this case, the example creates the DataSource definition directly from the RuleUnitExecutor class and binds it to the "persons" variable in a single statement.

The example execution code produces the following output when the relevant Person facts are inserted in the persons data source:

**Example rule unit execution output**

Sally is NOT adult
John is adult
Jane is adult
Sally is adult

The NotAdult rule detects a match when evaluating the person "Sally", who is under 18 years old. The rule then modifies her age to 18 and uses the drools.run( AdultUnit.class ) method to trigger the execution of the AdultUnit rule unit. The AdultUnit rule unit contains a rule that can now be executed for all of the 3 persons in the DataSource definition.

As an example of the drools.guard() method, consider the following BoxOffice class and BoxOfficeUnit rule unit class:

**Example BoxOffice class**

```java
public class BoxOffice {
    private boolean open;

    public BoxOffice( boolean open ) {
        this.open = open;
    }

    public boolean isOpen() {
        return open;
    }

    public void setOpen( boolean open ) {
        this.open = open;
    }
}
```

**Example BoxOfficeUnit rule unit class**

```java
public class BoxOfficeUnit implements RuleUnit {
    private DataSource<BoxOffice> boxOffices;

    public DataSource<BoxOffice> getBoxOffices() {
        return boxOffices;
    }
}
```
The example also uses the following **TicketIssuerUnit** rule unit class to keep selling box office tickets for the event as long as at least one box office is open. This rule unit uses **DataSource** definitions of **persons** and **tickets**:

**Example TicketIssuerUnit rule unit class**

```java
public class TicketIssuerUnit implements RuleUnit {
    private DataSource<Person> persons;
    private DataSource<AdultTicket> tickets;

    private List<String> results;

    public TicketIssuerUnit() {
    }

    public TicketIssuerUnit( DataSource<Person> persons, DataSource<AdultTicket> tickets ) {
        this.persons = persons;
        this.tickets = tickets;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public DataSource<AdultTicket> getTickets() {
        return tickets;
    }

    public List<String> getResults() {
        return results;
    }
}
```

The **BoxOfficeUnit** rule unit contains a **BoxOfficeIsOpen** DRL rule that uses the **drools.guard**(TicketIssuerUnit.class) method to guard the execution of the **TicketIssuerUnit** rule unit that distributes the event tickets, as shown in the following DRL rule examples:

**Example DRL rules with controlled execution using drools.guard()**

```java
package org.mypackage.myunit;
unit TicketIssuerUnit;

rule IssueAdultTicket when
    $p: /persons[ age >= 18 ]
then
    tickets.insert(new AdultTicket($p));
end
rule RegisterAdultTicket when
    $t: /tickets
then
    results.add( $t.getPerson().getName() );
end

package org.mypackage.myunit;
unit BoxOfficeUnit;
```

```java
package org.mypackage.myunit;
unit TicketIssuerUnit;
```
rule BoxOfficesOpen
  when
  $box: /boxOffices[ open ]
  then
    drools.guard( TicketIssuerUnit.class );
end

In this example, so long as at least one box office is open, the guarded TicketIssuerUnit rule unit is active and distributes event tickets. When no more box offices are in open state, the guarded TicketIssuerUnit rule unit is prevented from being executed.

The following example class illustrates a more complete box office scenario:

Example class for the box office scenario

```java
DataSource<Person> persons = executor.newDataSource( "persons" );
DataSource<BoxOffice> boxOffices = executor.newDataSource( "boxOffices" );
DataSource<AdultTicket> tickets = executor.newDataSource( "tickets" );

List<String> list = new ArrayList<>();
executor.bindVariable( "results", list );

// Two box offices are open:
BoxOffice office1 = new BoxOffice( true );
FactHandle officeFH1 = boxOffices.insert( office1 );
BoxOffice office2 = new BoxOffice( true );
FactHandle officeFH2 = boxOffices.insert( office2 );

persons.insert( new Person("John", 40));

// Execute `BoxOfficesOpen` rule, run `TicketIssuerUnit` rule unit, and execute `RegisterAdultTicket` rule:
executor.run(BoxOfficeUnit.class);
assertEquals( 1, list.size() );
assertEquals( "John", list.get(0) );
list.clear();

persons.insert( new Person("Matteo", 30));

// Execute `RegisterAdultTicket` rule:
executor.run(BoxOfficeUnit.class);
assertEquals( 1, list.size() );
assertEquals( "Matteo", list.get(0) );
list.clear();

// One box office is closed, the other is open:
office1.setOpen( false );
boxOffices.update(officeFH1, office1);
persons.insert( new Person("Mark", 35));
executor.run(BoxOfficeUnit.class);
assertEquals( 1, list.size() );
assertEquals( "Mark", list.get(0) );
```
16.12.3. Rule unit identity conflicts

In rule unit execution scenarios with guarded rule units, a rule can guard multiple rule units and at the same time a rule unit can be guarded and then activated by multiple rules. For these two-way guarding scenarios, rule units must have a clearly defined identity to avoid identity conflicts.

By default, the identity of a rule unit is the rule unit class name and is treated as a singleton class by the RuleUnitExecutor. This identification behavior is encoded in the getUnitIdentity() default method of the RuleUnit interface:

**Default identity method in the RuleUnit interface**

```java
default Identity getUnitIdentity() {
    return new Identity( getClass() );
}
```

In some cases, you may need to override this default identification behavior to avoid conflicting identities between rule units.

For example, the following RuleUnit class contains a DataSource definition that accepts any kind of object:

**Example Unit0 rule unit class**

```java
public class Unit0 implements RuleUnit {
    private DataSource<Object> input;

    public DataSource<Object> getInput() {
        return input;
    }
}
```

This rule unit contains the following DRL rule that guards another rule unit based on two conditions (in OOPath notation):

**Example GuardAgeCheck DRL rule in the rule unit**

```java
package org.mypackage.myunit
unit Unit0

rule GuardAgeCheck
    when
    $i: /input#Integer
    $s: /input#String
    list.clear();
    office2.setOpen(false);
    boxOffices.update(officeFH2, office2); // Guarding rule is no longer true.
    persons.insert(new Person("Edson", 35));
    executor.run(BoxOfficeUnit.class); // No execution
    assertEquals(0, list.size());
```
then
drools.guard( new AgeCheckUnit($i) );
drools.guard( new AgeCheckUnit($s.length()) );
end

The guarded AgeCheckUnit rule unit verifies the age of a set of persons. The AgeCheckUnit contains a DataSource definition of the persons to check, a minAge variable that it verifies against, and a List for gathering the results:

Example AgeCheckUnit rule unit

```java
public class AgeCheckUnit implements RuleUnit {
    private final int minAge;
    private DataSource<Person> persons;
    private List<String> results;

    public AgeCheckUnit( int minAge ) {
        this.minAge = minAge;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public int getMinAge() {
        return minAge;
    }

    public List<String> getResults() {
        return results;
    }
}
```

The AgeCheckUnit rule unit contains the following DRL rule that performs the verification of the persons in the data source:

Example CheckAge DRL rule in the rule unit

```drl
package org.mypackage.myunit
unit AgeCheckUnit

rule CheckAge
    when
        $p : /persons{ age > minAge }
    then
        results.add($p.getName() + " > " + minAge);
end
```

This example creates a RuleUnitExecutor class, binds the class to the KIE base that contains these two rule units, and creates the two DataSource definitions for the same rule units:

Example executor and data source definitions

```java
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );
```
You can now insert some objects into the input data source and execute the `Unit0` rule unit:

**Example rule unit execution with inserted objects**

```java
DataSource<Object> input = executor.newDataSource("input");
DataSource<Person> persons = executor.newDataSource("persons",
    new Person("John", 42),
    new Person("Sally", 4));

List<String> results = new ArrayList<>();
executor.bindVariable("results", results);

ds.insert("test");
ds.insert(3);
ds.insert(4);
executor.run(Unit0.class);
```

**Example results list from the execution**

```
[Sally>3, John>3]
```

In this example, the rule unit named `AgeCheckUnit` is considered a singleton class and then executed only once, with the `minAge` variable set to 3. Both the String "test" and the Integer 4 inserted into the input data source can also trigger a second execution with the `minAge` variable set to 4. However, the second execution does not occur because another rule unit with the same identity has already been evaluated.

To resolve this rule unit identity conflict, override the `getUnitIdentity()` method in the `AgeCheckUnit` class to include also the `minAge` variable in the rule unit identity:

**Modified AgeCheckUnit rule unit to override the getUnitIdentity() method**

```java
public class AgeCheckUnit implements RuleUnit {
    ...

    @Override
    public Identity getUnitIdentity() {
        return new Identity(getClass(), minAge);
    }
}
```

With this override in place, the previous example rule unit execution produces the following output:

**Example results list from executing the modified rule unit**

```
[John>4, Sally>3, John>3]
```

The rule units with `minAge` set to 3 and 4 are now considered two different rule units and both are executed.
CHAPTER 17. DATA OBJECTS

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

17.1. CREATING DATA OBJECTS

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

Procedure

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

2. Click **Add Asset → Data Object**.

3. Enter a unique **Data Object** name and select the **Package** where you want the data object to be available for other rule assets. Data objects with the same name cannot exist in the same package. In the specified DRL file, you can import a data object from any package.

   **IMPORTING DATA OBJECTS FROM OTHER PACKAGES**

   You can import an existing data object from another package directly into the asset designers like guided rules or guided decision table designers. Select the relevant rule asset within the project and in the asset designer, go to **Data Objects → New item** to select the object to be imported.

4. To make your data object persistable, select the **Persistable** checkbox. Persistable data objects are able to be stored in a database according to the JPA specification. The default JPA is Hibernate.

5. Click **Ok**.

6. In the data object designer, click **add field** to add a field to the object with the attributes **Id**, **Label**, and **Type**. Required attributes are marked with an asterisk (*).

   - **Id**: Enter the unique ID of the field.
   - **Label**: (Optional) Enter a label for the field.
   - **Type**: Enter the data type of the field.
   - **List**: (Optional) Select this check box to enable the field to hold multiple items for the specified type.
### Figure 17.1. Add data fields to a data object

<table>
<thead>
<tr>
<th>New Field</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Id</strong></td>
<td>salary</td>
</tr>
<tr>
<td><strong>Label</strong></td>
<td>Salary</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>BigInteger</td>
</tr>
<tr>
<td><strong>List</strong></td>
<td></td>
</tr>
</tbody>
</table>

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

**NOTE**

To edit a field, select the field row and use the **general properties** on the right side of the screen.
CHAPTER 18. CREATING DRL RULES IN BUSINESS CENTRAL

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

Procedure

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

Components in a DRL file

```plaintext
package
import
function  // Optional
query  // Optional
declare  // Optional
global  // Optional

rule "rule name"
  // Attributes
  when
    // Conditions
  then
    // Actions
end
```
**rule** "rule2 name"

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

- **import**: Use this to identify the data objects from either this package or another package that you want to use in the DRL file. Specify the package and data object in the format `packageName.objectName`, with multiple imports on separate lines.

**Importing data objects**

```java
import org.mortgages.LoanApplication;
```

- **function**: (optional) Use this to include a function to be used by rules in the DRL file. 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.

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

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

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

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

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

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

- **query**: (optional) Use this to search the decision 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 global to the KIE base and therefore must be unique among all other rule queries in the project. To return the results of a query, construct a traditional `QueryResults` definition using `ksession.getQueryResults("name")`, where
"name" is the query name. This returns a list of query results, which enable you to retrieve the objects that matched the query. Define the query and query results parameters above the rules in the DRL file.

Example query definition in a DRL file

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

Example application code to obtain 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" );

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

Declaring and using a new fact type

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

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

• **global**: (optional) Use this to include a global variable to be used by rules in the DRL file. Global variables typically provide data or services for the rules, such as application services used in rule consequences, and return data from rules, such as logs or values added in rule consequences. Set the global value in the working memory of the decision engine through a KIE session configuration or REST operation, declare the global variable above the rules in the DRL file, and then use it in an action (**then**) part of the rule. For multiple global variables, use separate lines in the DRL file.

Setting the global list configuration for the decision engine

```
List<String> list = new ArrayList<>();
KieSession kieSession = kiebase.newKieSession();
kieSession.setGlobal( "myGlobalList", list );
```

Defining the global list in a rule
global java.util.List myGlobalList;

rule "Using a global"
  when
    // Empty
  then
    myGlobalList.add( "My global list" );
end

WARNING
Do not use global variables to establish conditions in rules unless a global variable has a constant immutable value. Global variables are not inserted into the working memory of the decision engine, so the decision engine cannot track value changes of variables.

Do not use global variables to share data between rules. Rules always reason and react to the working memory state, so if you want to pass data from rule to rule, assert the data as facts into the working memory of the decision engine.

- **rule**: Use this to define each rule in the DRL file. Rules consist of a rule name in the format `rule "name"`, followed by optional attributes that define rule behavior (such as `salience` or `no-loop`), followed by `when` and `then` definitions. Each rule must have a unique name within the rule package. The `when` part of the rule contains the conditions that must be met to execute an action. For example, if a bank requires loan applicants to have over 21 years of age, then the `when` condition for an "Underage" rule would be `Applicant( age < 21 )`. The `then` part of the rule contains the actions to be performed when the conditional part of the rule has been met. For example, when the loan applicant is under 21 years old, the `then` action would be `setApproved( false )`, declining the loan because the applicant is under age.

**Rule for loan application age limit**

```drl
rule "Underage"
  salience 15
  when
    $application : LoanApplication()
    Applicant( age < 21 )
  then
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
end
```

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

The following is an example DRL file in a loan application decision service:

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

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

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

**Prerequisites**

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

- The **import** list of data objects used in the rule is defined below the **package** line of the DRL file. Data objects can be from this package or from another package in Business Central.

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

**Procedure**

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

   **Example rule without conditions**

   ```drl
   rule "Always insert applicant"
   when
      // Empty
   then   // Actions to be executed once
      insert( new Applicant() );
   end

   // The rule is internally rewritten in the following way:

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

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

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

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

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

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

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

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

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

```
  // All of the following examples are interpreted the same way:
  $a : LoanApplication() and Applicant( age < 21 )

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

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

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

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

```
  // All of the following examples are interpreted the same way:
  Bankruptcy( amountOwed == 100000 ) or IncomeSource( amount == 20000 )

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

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

• **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 `( )`.

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

• **not:** Use this to specify facts and constraints that must not exist.
not ( Applicant( age < 21 ) )

- **forall**: Use this to 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**.

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

- **from**: Use this to specify a data source for a pattern.

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

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

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

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

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

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

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

**NOTE**

For more information about DRL rule conditions, see Section 16.8, “Rule conditions in DRL (WHEN)”.

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

5. Click **Save** in the DRL designer to save your work.
18.2. ADDING THEN ACTIONS IN DRL RULES

The *then* part of the rule contains the actions to be performed when the conditional part of the rule has been met. For example, when a loan applicant is under 21 years old, the *then* action of an "Underage" rule would be `setApproved( false )`, declining the loan because the applicant is under age. Actions consist of one or more methods that execute consequences based on the rule conditions and on available data objects in the package. The main purpose of rule actions is to insert, delete, or modify data in the working memory of the decision engine.

Prerequisites

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

Procedure

1. In the DRL designer, enter **then** after the **when** section of the rule to begin adding action statements.
2. Enter one or more actions to be executed on fact patterns based on the conditions for the rule. The following are some of the keyword options for defining DRL actions:
   - **set**: Use this to set the value of a field.
     ```java
     $application.setApproved( false );
     $application.setExplanation( "has been bankrupt" );
     ```
   - **modify**: Use this to specify fields to be modified for a fact and to notify the decision engine of the change. This method provides a structured approach to fact updates. It combines the **update** operation with setter calls to change object fields.
     ```java
     modify( LoanApplication ) {
       setAmount( 100 ),
       setApproved( true )
     }
     ```
   - **update**: Use this to specify fields and the entire related fact to be updated and to notify the decision engine of the change. After a fact has changed, you must call **update** before changing another fact that might be affected by the updated values. To avoid this added step, use the **modify** method instead.
     ```java
     LoanApplication.setAmount( 100 );
     update( LoanApplication );
     ```
   - **insert**: Use this to insert a **new** fact into the decision engine.
- **insertLogical**: Use this to insert a new fact logically into the decision engine. The decision engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts must be retracted explicitly. 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.

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

**NOTE**

For more information about DRL rule actions, see Section 16.9, “Rule actions in DRL (THEN)”.

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

4. Click **Save** in the DRL designer to save your work.
CHAPTER 19. EXECUTING RULES

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

Prerequisites

- Business Central and KIE Server are installed and running. For installation options, see Planning a Red Hat Decision Manager installation.

Procedure

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

2. In the upper-right corner of the project Assets page, click **Deploy** to build the project and deploy it to KIE Server. If the build fails, address any problems described in the **Alerts** panel at the bottom of the screen.
   For more information about project deployment options, see Packaging and deploying a Red Hat Decision Manager project.

   **NOTE**

   If the rule assets in your project are not built from an executable rule model by default, verify that the following dependency is in the pom.xml file of your project and rebuild the project:

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

   This dependency is required for rule assets in Red Hat Decision Manager to be built from executable rule models by default. This dependency is included as part of the Red Hat Decision Manager core packaging, but depending on your Red Hat Decision Manager upgrade history, you may need to manually add this dependency to enable the executable rule model behavior.

   For more information about executable rule models, see Packaging and deploying a Red Hat Decision Manager project.

3. Create a Maven or Java project outside of Business Central, if not created already, that you can use for executing rules locally or that you can use as a client application for executing rules on KIE Server. The project must contain a pom.xml file and any other required components for executing the project resources.
   For example test projects, see “Other methods for creating and executing DRL rules”.

4. Open the pom.xml file of your test project or client application and add the following dependencies, if not added already:

   - **kie-ci**: Enables your client application to load Business Central project data locally using **ReleaseId**
• **kie-server-client**: Enables your client application to interact remotely with assets on KIE Server

• **slf4j**: (Optional) Enables your client application to use Simple Logging Facade for Java (SLF4J) to return debug logging information after you interact with KIE Server

Example dependencies for Red Hat Decision Manager 7.10 in a client application pom.xml file:

```xml
<dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-ci</artifactId>
    <version>7.48.0.Final-redhat-00004</version>
</dependency>

<dependency>
    <groupId>org.kie.server</groupId>
    <artifactId>kie-server-client</artifactId>
    <version>7.48.0.Final-redhat-00004</version>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.25</version>
</dependency>

<!-- For debug logging (optional) -->
<dependency>
    <groupId>com.redhat.ba</groupId>
    <artifactId>ba-platform-bom</artifactId>
    <version>7.10.0.redhat-00004</version>
    <scope>import</scope>
    <type>pom</type>
</dependency>
```

For available versions of these artifacts, search the group ID and artifact ID in the Nexus Repository Manager online.

**NOTE**

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

Example BOM dependency:

```xml
<dependency>
    <groupId>com.redhat.ba</groupId>
    <artifactId>ba-platform-bom</artifactId>
    <version>7.10.0.redhat-00004</version>
    <scope>import</scope>
    <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see What is the mapping between Red Hat Decision Manager and the Maven library version?.
5. Ensure that the dependencies for artifacts containing model classes are defined in the client application `pom.xml` file exactly as they appear in the `pom.xml` file of the deployed project. If dependencies for model classes differ between the client application and your projects, execution errors can occur.

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

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

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

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

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

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

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

```java
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
when
  Person(hourlyRate * wage > 100)
  Person(name : firstName, surname : lastName)
then
  System.out.println("Hello " + " + name + " + surname + ")
  System.out.println("You are rich!");
end
```

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

**Executing rules locally**

```java
import org.kie.api.KieServices;
import org.kie.api.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;
```
To test this rule on KIE Server, configure the .java class with the imports and rule execution information similarly to the local example, and additionally specify KIE services configuration and KIE services client details:

**Executing rules on KIE Server**

```java
import org.kie.api.runtime.KieSession;
import org.drools.compiler.kproject.ReleaseIdImpl;

public class RulesTest {

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

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

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

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

            // Fire all rules:
            kSession.fireAllRules();
            kSession.dispose();
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

To test this rule on KIE Server, configure the .java class with the imports and rule execution information similarly to the local example, and additionally specify KIE services configuration and KIE services client details:
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;

import com.sample.Person;

public class RulesTest {

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

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

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

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

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

            // Use rule services client to send request:
            RuleServicesClient ruleClient = kieServicesClient.getServicesClient(RuleServicesClient.class);
            ServiceResponse<ExecutionResults> executeResponse = ruleClient.executeCommandsWithResults(containerName, batch);
            System.out.println("number of fired rules:" + executeResponse.getResult().getValue("numberOfFiredRules"));
        }
    }
}
Run the configured .java class from your project directory. You can run the file in your development platform (such as Red Hat CodeReady Studio) or in the command line. Example Maven execution (within project directory):

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

Example Java execution (within project directory)

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

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

```
catch (Throwable t) {
    t.printStackTrace();
}
```
CHAPTER 20. OTHER METHODS FOR CREATING AND EXECUTING DRL RULES

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

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

20.1. CREATING AND EXECUTING DRL RULES IN RED HAT CODEREADY STUDIO

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

Prerequisites

- Red Hat CodeReady Studio has been installed from the Red Hat Customer Portal.

Procedure

1. In the Red Hat CodeReady Studio, click File → New → Project.

2. In the New Project window that opens, select Drools → Drools Project and click Next.

3. Click the second icon to Create a project and populate it with some example files to help you get started quickly. Click Next.

4. Enter a Project name and select the Maven radio button as the project building option. The GAV values are generated automatically. You can update these values as needed for your project:
   - Group ID: com.sample
   - Artifact ID: my-project
   - Version: 1.0.0-SNAPSHOT

5. Click Finish to create the project.
   This configuration sets up a basic project structure, class path, and sample rules. The following is an overview of the project structure:
Notice the following elements:

- A `Sample.drl` rule file in the `src/main/resources` directory, containing an example Hello World and GoodBye rules.

- A `DroolsTest.java` file under the `src/main/java` directory in the `com.sample` package. The `DroolsTest` class can be used to execute the `Sample.drl` rule.

- The Drools Library directory, which acts as a custom class path containing JAR files necessary for execution.

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

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

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

```java
public class Person {
    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;

    public String getFirstName() {
        return firstName;
    }
```
7. Click **File → Save** to save the file.

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

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

```java
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"

when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)

then
    System.out.println("Hello" + " "+ name + " "+ surname + "!");
    System.out.println("You are rich!");
end
```

9. Click **File → Save** to save the file.
10. Create a main class and save it to the same directory as the Java object that you created. The main class will load the KIE base and execute rules.

**NOTE**

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

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

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

```java
package com.sample;

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

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

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

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

            // Fire all rules:
            kSession.fireAllRules();
            kSession.dispose();
        }
        catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

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

13. After you create and save all DRL assets in your project, right-click your project folder and select **Run As → Java Application** to build the project. If the project build fails, address any problems described in the **Problems** tab of the lower window in CodeReady Studio, and try again to validate the project until the project builds.
If the Run As → Java Application option is not available

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

To integrate the new rule assets with an existing project in Red Hat Decision Manager, you can compile the new project as a knowledge JAR (KJAR) and add it as a dependency in the pom.xml file of the project in Business Central. To access the project pom.xml file in Business Central, you can select any existing asset in the project and then in the Project Explorer menu on the left side of the screen, click the Customize View gear icon and select Repository View → pom.xml.

20.2. Creating and Executing DRL Rules Using Java

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

Procedure

1. Create a Java object on which the rule or rules will operate.
   In this example, a Person.java file is created in a directory my-project. The Person class contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person:

```java
public class Person {
    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Integer getHourlyRate() {
        return hourlyRate;
    }
}
```
2. Create a rule file in .drl format under the my-project directory. The DRL file must contain at a minimum a package specification (if applicable), an import list of data objects to be used by the rule or rules, and one or more rules with when conditions and then actions. The following Wage.drl file contains a Wage rule that calculates the wage and hourly rate values and displays a message based on the result:

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

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

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

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

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

public class RulesTest {
  public static final void main(String[] args) {
    try {
      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.getKieClasspathContainer();
      KieSession kSession = kContainer.newKieSession();
      // Insert facts:
      Person person = new Person(firstName, lastName);
      person.setHourlyRate(200);
      // Execute rule:
      kSession.insert(person);
      kSession.fireAllRules();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```
5. Download the Red Hat Decision Manager 7.10.0 Source Distribution ZIP file from the Red Hat Customer Portal and extract it under `my-project/dm-engine-jars/`.

6. In the `my-project/META-INF` directory, create a `kmodule.xml` metadata file with the following content:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud" equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg1">
    <ksession name="KSession1_1" type="stateful" default="true"></ksession>
    <ksession name="KSession1_2" type="stateful" default="true" beliefSystem="jtms"></ksession>
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream" equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg2,org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateless" default="true" clockType="realtime">
      <fileLogger file="debugInfo" threaded="true" interval="10"></fileLogger>
      <workItemHandlers>
        <workItemHandler name="name" type="new org.domain.WorkItemHandler()"></workItemHandler>
      </workItemHandlers>
    </ksession>
  </kbase>
</kmodule>
```

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

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

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

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

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

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

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

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

To integrate the new rule assets with an existing project in Red Hat Decision Manager, you can compile the new Java project as a knowledge JAR (KJAR) and add it as a dependency in the pom.xml file of the project in Business Central. To access the project pom.xml file in Business Central, you can select any existing asset in the project and then in the Project Explorer menu on the left side of the screen, click the Customize View gear icon and select Repository View → pom.xml.

20.3. CREATING AND EXECUTING DRL RULES USING MAVEN

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

Procedure

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

```bash
mvn archetype:generate -DgroupId=com.sample.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```
This creates a directory `my-app` with the following structure:

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

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

- A `src/main` directory for storing the application sources
- A `src/test` directory for storing the test sources
- A `pom.xml` file with the project configuration

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

```java
package com.sample.app;

public class Person {

    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getWage() {
        return wage.toString();
    }

    public void setWage(Integer wage) {
        this.wage = wage;
    }

    public String getHourlyRate() {
        return hourlyRate.toString();
    }

    public void setHourlyRate(Integer hourlyRate) {
        this.hourlyRate = hourlyRate;
    }

    // Add other methods as needed...
}
```
public Integer getHourlyRate() {
    return hourlyRate;
}

public void setHourlyRate(Integer hourlyRate) {
    this.hourlyRate = hourlyRate;
}

public Integer getWage() {
    return wage;
}

public void setWage(Integer wage) {
    this.wage = wage;
}

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

```java
package com.sample.app;

import com.sample.app.Person;

dialect "java"

rule "Wage"
    when
        Person(hourlyRate * wage > 100)
        Person(name : firstName, surname : lastName)
    then
        System.out.println("Hello " + name + " " + surname + "!");
        System.out.println("You are rich!");
end
```

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

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

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

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
</kmodule>
```
This example defines two KIE bases. Two KIE sessions are instantiated from the 
**KBase1** KIE base, and one KIE session from **KBase2**. The KIE session from 
**KBase2** is a **stateless** KIE session, which means that data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations. Specific **packages** of rule assets are 
included with both KIE bases. When you specify packages in this way, you must organize your 
DRL files in a folder structure that reflects the specified packages.

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sample.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0.0</version>
  <repositories>
    <repository>
      <id>jboss-ga-repository</id>
      <url>http://maven.repository.redhat.com/ga/</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>drools-compiler</artifactId>
      <version>VERSION</version>
    </dependency>
  </dependencies>
</project>
```
For information about Maven dependencies and the BOM (Bill of Materials) in Red Hat Decision Manager, see What is the mapping between Red Hat Decision Manager and Maven library version?.

6. Use the `testApp` method in `my-app/src/test/java/com/sample/app/AppTest.java` to test the rule. The `AppTest.java` file is created by Maven by default.

7. In the `AppTest.java` file, add the required `import` statements to import KIE services, a KIE container, and a KIE session. Then load the KIE base, insert facts, and execute the rule from the `testApp()` method that passes the fact model to the rule.

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

public void testApp() {

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

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

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

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

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

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

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

\begin{verbatim}
mvn exec:java -Dexec.mainClass="com.sample.app"
\end{verbatim}

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

To integrate the new rule assets with an existing project in Red Hat Decision Manager, you can compile the new Maven project as a knowledge JAR (KJAR) and add it as a dependency in the \texttt{pom.xml} file of the project in Business Central. To access the project \texttt{pom.xml} file in Business Central, you can select any existing asset in the project and then in the \texttt{Project Explorer} menu on the left side of the screen, click the \texttt{Customize View} gear icon and select \texttt{Repository View \rightarrow pom.xml}. 


CHAPTER 21. EXAMPLE DECISIONS IN RED HAT DECISION MANAGER FOR AN IDE

Red Hat Decision Manager provides example decisions distributed as Java classes that you can import into your integrated development environment (IDE). You can use these examples to better understand decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

The following example decision sets are some of the examples available in Red Hat Decision Manager:

- **Hello World example**: Demonstrates basic rule execution and use of debug output
- **State example**: Demonstrates forward chaining and conflict resolution through rule salience and agenda groups
- **Fibonacci example**: Demonstrates recursion and conflict resolution through rule salience
- **Banking example**: Demonstrates pattern matching, basic sorting, and calculation
- **Pet Store example**: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration
- **Sudoku example**: Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration
- **House of Doom example**: Demonstrates backward chaining and recursion

**NOTE**

For optimization examples provided with Red Hat Business Optimizer, see *Getting started with Red Hat Business Optimizer*.

21.1. IMPORTING AND EXECUTING RED HAT DECISION MANAGER EXAMPLE DECISIONS IN AN IDE

You can import Red Hat Decision Manager example decisions into your integrated development environment (IDE) and execute them to explore how the rules and code function. You can use these examples to better understand decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

**Prerequisites**

- Java 8 or later is installed.
- Maven 3.5.x or later is installed.
- An IDE is installed, such as Red Hat CodeReady Studio.

**Procedure**

1. Download and unzip the Red Hat Decision Manager 7.10.0 Source Distribution from the Red Hat Customer Portal to a temporary directory, such as /rhdm-7.10.0-sources.
2. Open your IDE and select **File → Import → Maven → Existing Maven Projects** or the equivalent option for importing a Maven project.

3. Click **Browse**, navigate to `~/rhdm-7.10.0-sources/src/drools-$VERSION/drools-examples` (or, for the Conway’s Game of Life example, `~/rhdm-7.10.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples`), and import the project.

4. Navigate to the example package that you want to run and find the Java class with the **main** method.

5. Right-click the Java class and select **Run As → Java Application** to run the example. To run all examples through a basic user interface, run the **DroolsExamplesApp.java** class (or, for Conway’s Game of Life, the **DroolsJbpmIntegrationExamplesApp.java** class) in the **org.drools.examples** main class.
Figure 21.1. Interface for all examples in drools-examples (DroolsExamplesApp.java)
21.2. HELLO WORLD EXAMPLE DECISIONS (BASIC RULES AND DEBUGGING)

The Hello World example decision set demonstrates how to insert objects into the decision engine working memory, how to match the objects using rules, and how to configure logging to trace the internal activity of the decision engine.

The following is an overview of the Hello World example:

- **Name**: helloworld
- **Main class**: org.drools.examples.helloworld.HelloWorldExample (in src/main/java)
- **Module**: drools-examples
- **Type**: Java application
- **Rule file**: org.drools.examples.helloworld.HelloWorld.drl (in src/main/resources)
- **Objective**: Demonstrates basic rule execution and use of debug output

In the Hello World example, a KIE session is generated to enable rule execution. All rules require a KIE session for execution.

**KIE session for rule execution**

```java
KieServices ks = KieServices.Factory.get();  // 1
KieContainer kc = ks.getKieClasspathContainer();  // 2
KieSession ksession = kc.newKieSession("HelloWorldKS");  // 3
```

1. Obtains the **KieServices** factory. This is the main interface that applications use to interact with the decision engine.

2. Creates a **KieContainer** from the project class path. This detects a `/META-INF/kmodule.xml` file from which it configures and instantiates a **KieContainer** with a **KieModule**.

3. Creates a **KieSession** based on the "HelloWorldKS" KIE session configuration defined in the `/META-INF/kmodule.xml` file.
NOTE

For more information about Red Hat Decision Manager project packaging, see Packaging and deploying a Red Hat Decision Manager project.

Red Hat Decision Manager has an event model that exposes internal engine activity. Two default debug listeners, `DebugAgendaEventListener` and `DebugRuleRuntimeEventListener`, print debug event information to the `System.err` output. The `KieRuntimeLogger` provides execution auditing, the result of which you can view in a graphical viewer.

Debug listeners and audit loggers

```java
// Set up listeners.
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugRuleRuntimeEventListener() );

// Set up a file-based audit logger.
KieRuntimeLogger logger = KieServices.get().getLoggers().newFileLogger( ksession, "/target/helloworld" );

// Set up a ThreadedFileLogger so that the audit view reflects events while debugging.
KieRuntimeLogger logger = ks.getLoggers().newThreadedFileLogger( ksession, "/target/helloworld", 1000 );
```

The logger is a specialized implementation built on the `Agenda` and `RuleRuntime` listeners. When the decision engine has finished executing, `logger.close()` is called.

The example creates a single `Message` object with the message "Hello World", inserts the status `HELLO` into the `KieSession`, executes rules with `fireAllRules()`.

Data insertion and execution

```java
// Insert facts into the KIE session.
final Message message = new Message();
message.setMessage( "Hello World" );
message.setStatus( Message.HELLO );
ksession.insert( message );

// Fire the rules.
ksession.fireAllRules();
```

Rule execution uses a data model to pass data as inputs and outputs to the `KieSession`. The data model in this example has two fields: the `message`, which is a `String`, and the `status`, which can be `HELLO` or `GOODBYE`.

Data model class

```java
public static class Message {
    public static final int HELLO   = 0;
    public static final int GOODBYE = 1;

    private String     message;
}
The two rules are located in the file
src/main/resources/org/drools/examples/helloworld/HelloWorld.drl.

The `when` condition of the "Hello World" rule states that the rule is activated for each `Message` object inserted into the KIE session that has the status `Message.HELLO`. Additionally, two variable bindings are created: the variable `message` is bound to the `message` attribute and the variable `m` is bound to the matched `Message` object itself.

The `then` action of the rule specifies to print the content of the bound variable `message` to `System.out`, and then changes the values of the `message` and `status` attributes of the `Message` object bound to `m`. The rule uses the `modify` statement to apply a block of assignments in one statement and to notify the decision engine of the changes at the end of the block.

"Hello World" rule

```java
rule "Hello World"
  when
    m : Message( status == Message.HELLO, message : message )
  then
    System.out.println( message );
    modify ( m ) { message = "Goodbye cruel world", status = Message.GOODBYE };
end
```

The "Good Bye" rule is similar to the "Hello World" rule except that it matches `Message` objects that have the status `Message.GOODBYE`.

"Good Bye" rule

```java
rule "Good Bye"
  when
    Message( status == Message.GOODBYE, message : message )
  then
    System.out.println( message );
end
```

To execute the example, run the `org.drools.examples.helloworld.HelloWorldExample` class as a Java application in your IDE. The rule writes to `System.out`, the debug listener writes to `System.err`, and the audit logger creates a log file in `target/helloworld.log`.

System.out output in the IDE console

```
Hello World
Goodbye cruel world
```

System.err output in the IDE console

```
==>[ActivationCreated(0): rule=Hello World;
   tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[ObjectInserted: handle=
```
To better understand the execution flow of this example, you can load the audit log file from `target/helloworld.log` into your IDE debug view or Audit View, if available (for example, in Window → Show View in some IDEs).

In this example, the Audit view shows that the object is inserted, which creates an activation for the "Hello World" rule. The activation is then executed, which updates the Message object and causes the "Good Bye" rule to activate. Finally, the "Good Bye" rule is executed. When you select an event in the Audit View, the origin event, which is the "Activation created" event in this example, is highlighted in green.

Figure 21.3. Hello World example Audit View

### 21.3. STATE EXAMPLE DECISIONS (FORWARD CHAINING AND CONFLICT RESOLUTION)

The State example decision set demonstrates how the decision engine uses forward chaining and any changes to facts in the working memory to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values or through agenda groups that you can define in rules.

The following is an overview of the State example:

- **Name**: state
- **Main classes**: `org.drools.examples.state.StateExampleUsingSalience`, `org.drools.examples.state.StateExampleUsingAgendaGroup` (in `src/main/java`)
- **Module**: drools-examples
- **Type**: Java application
- **Rule files:** `org.drools.examples.state.*.drl` *(in src/main/resources)*

- **Objective:** Demonstrates forward chaining and conflict resolution through rule salience and agenda groups

A forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision 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 decision 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 decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:
In the State example, each `State` class has fields for its name and its current state (see the class `org.drools.examples.state.State`). The following states are the two possible states for each object:

- **NOTRUN**
- **FINISHED**

**State class**

```java
public class State {
    public static final int NOTRUN = 0;
}
The State example contains two versions of the same example to resolve rule execution conflicts:

- A **StateExampleUsingSalience** version that resolves conflicts by using rule salience
- A **StateExampleUsingAgendaGroups** version that resolves conflicts by using rule agenda groups

Both versions of the state example involve four State objects: A, B, C, and D. Initially, their states are set to **NOTRUN**, which is the default value for the constructor that the example uses.

**State example using salience**
The **StateExampleUsingSalience** version of the State example uses salience values in rules to resolve rule execution conflicts. Rules with a higher salience value are given higher priority when ordered in the activation queue.

The example inserts each State instance into the KIE session and then calls **fireAllRules()**.

**Salience State example execution**

```java
public static final int FINISHED = 1;

private final PropertyChangeSupport changes =
    new PropertyChangeSupport( this );

private String name;
private int state;

... setters and getters go here...
}
```

```java
final State a = new State( "A" );
final State b = new State( "B" );
final State c = new State( "C" );
final State d = new State( "D" );

ksession.insert( a );
ksession.insert( b );
ksession.insert( c );
ksession.insert( d );

ksession.fireAllRules();
```

// Dispose KIE session if stateful (not required if stateless).
ksession.dispose();

To execute the example, run the **org.drools.examples.state.StateExampleUsingSalience** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

**Salience State example output in the IDE console**

A finished
B finished
C finished
D finished

Four rules are present.

First, the "Bootstrap" rule fires, setting A to state FINISHED, which then causes B to change its state to FINISHED. Objects C and D are both dependent on B, causing a conflict that is resolved by the salience values.

To better understand the execution flow of this example, you can load the audit log file from target/state.log into your IDE debug view or Audit View, if available (for example, in Window → Show View in some IDEs).

In this example, the Audit View shows that the assertion of the object A in the state NOTRUN activates the "Bootstrap" rule, while the assertions of the other objects have no immediate effect.

Figure 21.5. Salience State example Audit View

Rule "Bootstrap" in salience State example

```
rule "Bootstrap"
   when
      a : State(name == "A", state == State.NOTRUN )
   then
      System.out.println(a.getName() + " finished");
      a.setState(State.FINISHED);
   end
```

The execution of the "Bootstrap" rule changes the state of A to FINISHED, which activates rule "A to B".

Rule "A to B" in salience State example
The execution of rule "A to B" changes the state of B to FINISHED, which activates both rules "B to C" and "B to D", placing their activations onto the decision engine agenda.

Rules "B to C" and "B to D" in salience State example

```
rule "B to C"
    salience 10
    when
        State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
    then
        System.out.println(c.getName() + " finished");
    c.setState( State.FINISHED );
end

rule "B to D"
    when
        State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
    then
        System.out.println(d.getName() + " finished");
    d.setState( State.FINISHED );
end
```

From this point on, both rules may fire and, therefore, the rules are in conflict. The conflict resolution strategy enables the decision engine agenda to decide which rule to fire. Rule "B to C" has the higher salience value (10 versus the default salience value of 0), so it fires first, modifying object C to state FINISHED.

The Audit View in your IDE shows the modification of the State object in the rule "A to B", which results in two activations being in conflict.

You can also use the Agenda View in your IDE to investigate the state of the decision engine agenda. In this example, the Agenda View shows the breakpoint in the rule "A to B" and the state of the agenda with the two conflicting rules. Rule "B to D" fires last, modifying object D to state FINISHED.
The **StateExampleUsingAgendaGroups** version of the State example uses agenda groups in rules to resolve rule execution conflicts. Agenda groups enable you to partition the decision engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.
Initially, a working memory has its focus on the agenda group **MAIN**. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method `setFocus()` or the rule attribute `auto-focus`. The `auto-focus` attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

In this example, the `auto-focus` attribute enables rule "B to C" to fire before "B to D".

**Rule "B to C" in agenda group State example**

```java
rule "B to C"
    agenda-group "B to C"
    auto-focus true
    when
        State(name == "B", state == State.FINISHED )
        c : State(name == "C", state == State.NOTRUN )
    then
        System.out.println(c.getName() + " finished");
        c.setState( State.FINISHED );
        kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to D" ).setFocus();
end
```

The rule "B to C" calls `setFocus()` on the agenda group "B to D", enabling its active rules to fire, which then enables the rule "B to D" to fire.

**Rule "B to D" in agenda group State example**

```java
rule "B to D"
    agenda-group "B to D"
    when
        State(name == "B", state == State.FINISHED )
        d : State(name == "D", state == State.NOTRUN )
    then
        System.out.println(d.getName() + " finished");
        d.setState( State.FINISHED );
end
```

To execute the example, run the `org.drools.examples.state.StateExampleUsingAgendaGroups` class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window (same as the salience version of the State example):

**Agenda group State example output in the IDE console**

```
A finished
B finished
C finished
D finished
```

**Dynamic facts in the State example**

Another notable concept in this State example is the use of **dynamic facts**, based on objects that implement a `PropertyChangeListener` object. In order for the decision engine to see and react to changes of fact properties, the application must notify the decision engine that changes occurred. You
can configure this communication explicitly in the rules by using the modify statement, or implicitly by specifying that the facts implement the PropertyChangeSupport interface as defined by the JavaBeans specification.

This example demonstrates how to use the PropertyChangeSupport interface to avoid the need for explicit modify statements in the rules. To make use of this interface, ensure that your facts implement PropertyChangeSupport in the same way that the class org.drools.example.State implements it, and then use the following code in the DRL rule file to configure the decision engine to listen for property changes on those facts:

Declaring a dynamic fact

```
declare type State
  @propertyChangeSupport
end
```

When you use PropertyChangeListener objects, each setter must implement additional code for the notification. For example, the following setter for state is in the class org.drools.examples:

Setter example with PropertyChangeSupport

```java
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",
                                      oldState,
                                      newState );
}
```

21.4. FIBONACCI EXAMPLE DECISIONS (RECURSION AND CONFLICT RESOLUTION)

The Fibonacci example decision set demonstrates how the decision engine uses recursion to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values that you can define in rules.

The following is an overview of the Fibonacci example:

- **Name**: fibonacci
- **Main class**: org.drools.examples.fibonacci.FibonacciExample (in src/main/java)
- **Module**: drools-examples
- **Type**: Java application
- **Rule file**: org.drools.examples.fibonacci.Fibonacci.drl (in src/main/resources)
- **Objective**: Demonstrates recursion and conflict resolution through rule salience

The Fibonacci Numbers form a sequence starting with 0 and 1. The next Fibonacci number is obtained by adding the two preceding Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, and so on.

The Fibonacci example uses the single fact class Fibonacci with the following two fields:
• sequence

• value

The **sequence** field indicates the position of the object in the Fibonacci number sequence. The **value** field shows the value of that Fibonacci object for that sequence position, where `-1` indicates a value that still needs to be computed.

**Fibonacci class**

```java
public static class Fibonacci {
    private int sequence;
    private long value;

    public Fibonacci( final int sequence ) {
        this.sequence = sequence;
        this.value = -1;
    }

    // ... setters and getters go here...
}
```

To execute the example, run the `org.drools.examples.fibonacci.FibonacciExample` class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

**Fibonacci example output in the IDE console**

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
6 == 8
...
47 == 2971215073
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

To achieve this behavior in Java, the example inserts a single `Fibonacci` object with a sequence field of **50**. The example then uses a recursive rule to insert the other 49 `Fibonacci` objects.
Instead of implementing the `PropertyChangeSupport` interface to use dynamic facts, this example uses the MVEL dialect `modify` keyword to enable a block setter action and notify the decision engine of changes.

**Fibonacci example execution**

```java
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

This example uses the following three rules:

- "Recurse"
- "Bootstrap"
- "Calculate"

The rule "Recurse" matches each asserted `Fibonacci` object with a value of -1, creating and asserting a new `Fibonacci` object with a sequence of one less than the currently matched object. Each time a Fibonacci object is added while the one with a sequence field equal to 1 does not exist, the rule re-matches and fires again. The not conditional element is used to stop the rule matching once you have all 50 Fibonacci objects in memory. The rule also has a salience value because you need to have all 50 `Fibonacci` objects asserted before you execute the "Bootstrap" rule.

**Rule "Recurse"**

```java
rule "Recurse"
  salience 10
  when
    f : Fibonacci ( value == -1 )
    not ( Fibonacci ( sequence == 1 ) )
  then
    insert( new Fibonacci( f.sequence - 1 ) );
    System.out.println( "recurse for " + f.sequence );
end
```

To better understand the execution flow of this example, you can load the audit log file from `target/fibonacci.log` into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit View** shows the original assertion of the `Fibonacci` object with a sequence field of 50, done from Java code. From there on, the **Audit View** shows the continual recursion of the rule, where each asserted `Fibonacci` object causes the "Recurse" rule to become activated and to fire again.
When a `Fibonacci` object with a `sequence` field of 2 is asserted, the "Bootstrap" rule is matched and activated along with the "Recurse" rule. Notice the multiple restrictions on field `sequence` that test for equality with 1 or 2:

**Rule "Bootstrap"**

```java
rule "Bootstrap"
when
  f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-restriction
then
  modify ( f ){ value = 1 };
  System.out.println( f.sequence + " == " + f.value );
end
```

You can also use the Agenda View in your IDE to investigate the state of the decision engine agenda. The "Bootstrap" rule does not fire yet because the "Recurse" rule has a higher salience value.
When a *Fibonacci* object with a sequence of 1 is asserted, the "Bootstrap" rule is matched again, causing two activations for this rule. The "Recurse" rule does not match and activate because the not conditional element stops the rule matching as soon as a *Fibonacci* object with a sequence of 1 exists.
The "Bootstrap" rule sets the objects with a sequence of 1 and 2 to a value of 1. Now that you have two Fibonacci objects with values not equal to -1, the "Calculate" rule is able to match.

At this point in the example, nearly 50 Fibonacci objects exist in the working memory. You need to select a suitable triple to calculate each of their values in turn. If you use three Fibonacci patterns in a rule without field constraints to confine the possible cross products, the result would be 50x49x48 possible combinations, leading to about 125,000 possible rule firings, most of them incorrect.

The "Calculate" rule uses field constraints to evaluate the three Fibonacci patterns in the correct order. This technique is called cross-product matching.

The first pattern finds any Fibonacci object with a value != -1 and binds both the pattern and the field. The second Fibonacci object does the same thing, but adds an additional field constraint to ensure that its sequence is greater by one than the Fibonacci object bound to f1. When this rule fires for the first time, you know that only sequences 1 and 2 have values of 1, and the two constraints ensure that f1 references sequence 1 and that f2 references sequence 2.

The final pattern finds the Fibonacci object with a value equal to -1 and with a sequence one greater than f2.

At this point in the example, three Fibonacci objects are correctly selected from the available cross products, and you can calculate the value for the third Fibonacci object that is bound to f3.

Rule "Calculate"

```java
rule "Calculate"
when
    // Bind f1 and s1.
    f1 : Fibonacci( s1 : sequence, value != -1 )
    // Bind f2 and v2, refer to bound variable s1.
    f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )
    // Bind f3 and s3, alternative reference of f2.sequence.
    f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )
then
    // Note the various referencing techniques.
    modify ( f3 ) { value = f1.value + v2 };
    System.out.println( s3 + " == " + f3.value );
end
```

The modify statement updates the value of the Fibonacci object bound to f3. This means that you now have another new Fibonacci object with a value not equal to -1, which allows the "Calculate" rule to rematch and calculate the next Fibonacci number.

The debug view or Audit View of your IDE shows how the firing of the last "Bootstrap" rule modifies the Fibonacci object, enabling the "Calculate" rule to match, which then modifies another Fibonacci object that enables the "Calculate" rule to match again. This process continues until the value is set for all Fibonacci objects.
21.5. PRICING EXAMPLE DECISIONS (DECISION TABLES)

The Pricing example decision set demonstrates how to use a spreadsheet decision table for calculating the retail cost of an insurance policy in tabular format instead of directly in a DRL file.

The following is an overview of the Pricing example:

- **Name:** decisiontable
- **Main class:** org.drools.examples.decisiontable.PricingRuleDTExample (in src/main/java)
- **Module:** drools-examples
- **Type:** Java application
- **Rule file:** org.drools.examples.decisiontable.ExamplePolicyPricing.xls (in src/main/resources)
- **Objective:** Demonstrates use of spreadsheet decision tables to define rules

Spreadsheet decision tables are XLS or XLSX spreadsheets that contain business rules defined in a tabular format. You can include spreadsheet decision tables with standalone Red Hat Decision Manager projects or upload them to projects in Business Central. Each row in a decision table is a rule, and each column is a condition, an action, or another rule attribute. After you create and upload your decision tables into your Red Hat Decision Manager project, the rules you defined are compiled into Drools Rule Language (DRL) rules as with all other rule assets.
The purpose of the Pricing example is to provide a set of business rules to calculate the base price and a discount for a car driver applying for a specific type of insurance policy. The driver’s age and history and the policy type all contribute to calculate the basic premium, and additional rules calculate potential discounts for which the driver might be eligible.

To execute the example, run the `org.drools.examples.decisiontable.PricingRuleDTExample` class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

```
Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
```

The code to execute the example follows the typical execution pattern: the rules are loaded, the facts are inserted, and a stateless KIE session is created. The difference in this example is that the rules are defined in an `ExamplePolicyPricing.xls` file instead of a DRL file or other source. The spreadsheet file is loaded into the decision engine using templates and DRL rules.

**Spreadsheet decision table setup**

The `ExamplePolicyPricing.xls` spreadsheet contains two decision tables in the first tab:

- **Base pricing rules**
- **Promotional discount rules**

As the example spreadsheet demonstrates, you can use only the first tab of a spreadsheet to create decision tables, but multiple tables can be within a single tab. Decision tables do not necessarily follow top-down logic, but are more of a means to capture data resulting in rules. The evaluation of the rules is not necessarily in the given order, because all of the normal mechanics of the decision engine still apply. This is why you can have multiple decision tables in the same tab of a spreadsheet.

The decision tables are executed through the corresponding rule template files `BasePricing.drt` and `PromotionalPricing.drt`. These template files reference the decision tables through their template parameter and directly reference the various headers for the conditions and actions in the decision tables.

**BasePricing.drt rule template file**

```java
package org.drools.examples.decisiontable;

template "Pricing bracket"
  age
  profile
  priorClaims
  policyType
  base
  reason

rule "Pricing bracket_@{row.rowNumber}"
  when
```
Driver(age >= @{age0}, age <= @{age1}, priorClaims == "@{priorClaims}"
, locationRiskProfile == "@{profile}"
)
policy: Policy(type == "@{policyType}"
then
policy.setBasePrice(@{base});
System.out.println("@{reason}");
end
end template

PromotionalPricing.drt rule template file

template header
age[]
priorClaims
policyType
discount

package org.drools.examples.decisiontable;

template "discounts"
age
priorClaims
policyType
discount

rule "Discounts_@{row.rowNumber}"
when
Driver(age >= @{age0}, age <= @{age1}, priorClaims == "@{priorClaims}"
policy: Policy(type == "@{policyType}"
then
policy.applyDiscount(@{discount});
end
end template

The rules are executed through the kmodule.xml reference of the KIE Session DTableWithTemplateKB, which specifically mentions the ExamplePolicyPricing.xls spreadsheet and is required for successful execution of the rules. This execution method enables you to execute the rules as a standalone unit (as in this example) or to include the rules in a packaged knowledge JAR (KJAR) file, so that the spreadsheet is packaged along with the rules for execution.

The following section of the kmodule.xml file is required for the execution of the rules and spreadsheet to work successfully:

```xml
<kbase name="DecisionTableKB" packages="org.drools.examples.decisiontable">
  <ksession name="DecisionTableKS" type="stateless"/>
</kbase>

<kbase name="DTableWithTemplateKB" packages="org.drools.examples.decisiontable-template">
  <ruleTemplate dtable="org/drools/examples/decisiontable-template/ExamplePolicyPricingTemplateData.xls"
    template="org/drools/examples/decisiontable-template/BasePricing.drt"
    row="3" col="3"/>
  <ruleTemplate dtable="org/drools/examples/decisiontable-template/ExamplePolicyPricingTemplateData.xls"
    template="org/drools/examples/decisiontable-template/BasePricing.drt"
    row="3" col="3"/>
</kbase>
```
As an alternative to executing the decision tables using rule template files, you can use the `DecisionTableConfiguration` object and specify an input spreadsheet as the input type, such as `DecisionTableInputType.xls`:

```java
DecisionTableConfiguration dtableconfiguration = KnowledgeBuilderFactory.newDecisionTableConfiguration();
    dtableconfiguration.setInputType( DecisionTableInputType.XLS );

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

Resource xlsRes = ResourceFactory.newClassPathResource( "ExamplePolicyPricing.xls", getClass() );
    kbuilder.add( xlsRes,
        ResourceType.DTABLE,
        dtableconfiguration );
```

The Pricing example uses two fact types:

- Driver
- Policy.

The example sets the default values for both facts in their respective Java classes `Driver.java` and `Policy.java`. The **Driver** is 30 years old, has had no prior claims, and currently has a risk profile of **LOW**. The **Policy** that the driver is applying for is **COMPREHENSIVE**.

In any decision table, each row is considered a different rule and each column is a condition or an action. Each row is evaluated in a decision table unless the agenda is cleared upon execution.

Decision table spreadsheets (XLS or XLSX) require two key areas that define rule data:

- A **RuleSet** area
- A **RuleTable** area

The **RuleSet** area of the spreadsheet defines elements that you want to apply globally to all rules in the same package (not only the spreadsheet), such as a rule set name or universal rule attributes. The **RuleTable** area defines the actual rules (rows) and the conditions, actions, and other rule attributes (columns) that constitute that rule table within the specified rule set. A decision table spreadsheet can contain multiple **RuleTable** areas, but only one **RuleSet** area.
The RuleTable area also defines the objects to which the rule attributes apply, in this case Driver and Policy, followed by constraints on the objects. For example, the Driver object constraint that defines the Age Bracket column is age >= $1, age <= $2, where the comma-separated range is defined in the table column values, such as 18,24.

**Base pricing rules**

The Base pricing rules decision table in the Pricing example evaluates the age, risk profile, number of claims, and policy type of the driver and produces the base price of the policy based on these conditions.

The Driver attributes are defined in the following table columns:

- **Age Bracket**: The age bracket has a definition for the condition age >= $1, age <= $2, which defines the condition boundaries for the driver’s age. This condition column highlights the use of $1 and $2, which is comma delimited in the spreadsheet. You can write these values as 18,24 or 18, 24 and both formats work in the execution of the business rules.
- **Location risk profile**: The risk profile is a string that the example program passes always as **LOW** but can be changed to reflect **MED** or **HIGH**.

- **Number of prior claims**: The number of claims is defined as an integer that the condition column must exactly equal to trigger the action. The value is not a range, only exact matches.

The **Policy** of the decision table is used in both the conditions and the actions of the rule and has attributes defined in the following table columns:

- **Policy type applying for**: The policy type is a condition that is passed as a string that defines the type of coverage: **COMPREHENSIVE**, **FIRE_THEFT**, or **THIRD_PARTY**.

- **Base $ AUD**: The **basePrice** is defined as an **ACTION** that sets the price through the constraint `policy.setBasePrice($param)`; based on the spreadsheet cells corresponding to this value. When you execute the corresponding DRL rule for this decision table, the **then** portion of the rule executes this action statement on the true conditions matching the facts and sets the base price to the corresponding value.

- **Record Reason**: When the rule successfully executes, this action generates an output message to the **System.out** console reflecting which rule fired. This is later captured in the application and printed.

The example also uses the first column on the left to categorize rules. This column is for annotation only and has no affect on rule execution.

**Promotional discount rules**

The **Promotional discount rules** decision table in the Pricing example evaluates the age, number of prior claims, and policy type of the driver to generate a potential discount on the price of the insurance policy.

**Figure 21.13. Discount calculation**

This decision table contains the conditions for the discount for which the driver might be eligible. Similar to the base price calculation, this table evaluates the **Age**, **Number of prior claims** of the driver, and the **Policy type applying for** to determine a **Discount %** rate to be applied. For example, if the driver is 30 years old, has no prior claims, and is applying for a **COMPREHENSIVE** policy, the driver is given a discount of **20** percent.

### 21.6. PET STORE EXAMPLE DECISIONS (AGENDA GROUPS, GLOBAL VARIABLES, CALLBACKS, AND GUI INTEGRATION)

The Pet Store example decision set demonstrates how to use agenda groups and global variables in rules and how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based desktop application. The example also demonstrates how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.
The following is an overview of the Pet Store example:

- **Name**: `petstore`
- **Main class**: `org.drools.examples.petstore.PetStoreExample` *(in src/main/java)*
- **Module**: `drools-examples`
- **Type**: Java application
- **Rule file**: `org.drools.examples.petstore.PetStore.drl` *(in src/main/resources)*
- **Objective**: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration

In the Pet Store example, the sample `PetStoreExample.java` class defines the following principal classes (in addition to several classes to handle Swing events):

- **Petstore** contains the `main()` method.
- **PetStoreUI** is responsible for creating and displaying the Swing-based GUI. This class contains several smaller classes, mainly for responding to various GUI events, such as user mouse clicks.
- **TableModel** holds the table data. This class is essentially a JavaBean that extends the Swing class `AbstractTableModel`.
- **CheckoutCallback** enables the GUI to interact with the rules.
- **Ordershow** keeps the items that you want to buy.
- **Purchase** stores details of the order and the products that you are buying.
- **Product** is a JavaBean containing details of the product available for purchase and its price.

Much of the Java code in this example is either plain JavaBean or Swing based. For more information about Swing components, see the Java tutorial on Creating a GUI with JFC/Swing.

**Rule execution behavior in the Pet Store example**

Unlike other example decision sets where the facts are asserted and fired immediately, the Pet Store example does not execute the rules until more facts are gathered based on user interaction. The example executes rules through a `PetStoreUI` object, created by a constructor, that accepts the `Vector` object `stock` for collecting the products. The example then uses an instance of the `CheckoutCallback` class containing the rule base that was previously loaded.

**Pet Store KIE container and fact execution setup**

```java
// KieServices is the factory for all KIE services.
KieServices ks = KieServices.Factory.get();

// Create a KIE container on the class path.
KieContainer kc = ks.getKieClasspathContainer();

// Create the stock.
Vector<Product> stock = new Vector<Product>();
stock.add( new Product( "Gold Fish", 5 ) );
stock.add( new Product( "Fish Tank", 25 ) );
stock.add( new Product( "Fish Food", 2 ) );
```
The Java code that fires the rules is in the `CheckoutCallBack.checkout()` method. This method is triggered when the user clicks `Checkout` in the UI.

**Rule execution from CheckoutCallBack.checkout()**

```java
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();

    // Iterate through list and add to cart.
    for (Product p: items) {
        order.addItem(new Purchase(order, p));
    }

    // Add the JFrame to the ApplicationData to allow for user interaction.

    // From the KIE container, a KIE session is created based on
    // its definition and configuration in the META-INF/kmodule.xml file.
    KieSession ksession = kcontainer.newKieSession("PetStoreKS");

    ksession.setGlobal( "frame", frame );
    ksession.setGlobal( "textArea", this.output );

    ksession.insert( new Product( "Gold Fish", 5 ) );
    ksession.insert( new Product( "Fish Tank", 25 ) );
    ksession.insert( new Product( "Fish Food", 2 ) );

    ksession.insert( new Product( "Fish Food Sample", 0 ) );

    ksession.insert( order );

    // Execute rules.
    ksession.fireAllRules();

    // Return the state of the cart
    return order.toString();
}
```

The example code passes two elements into the `CheckoutCallBack.checkout()` method. One element is the handle for the `JFrame` Swing component surrounding the output text frame, found at the bottom of the GUI. The second element is a list of order items, which comes from the `TableModel` that stores the information from the `Table` area at the upper-right section of the GUI.

The `for` loop transforms the list of order items coming from the GUI into the `Order` JavaBean, also contained in the file `PetStoreExample.java`.

In this case, the rule is firing in a stateless KIE session because all of the data is stored in Swing components and is not executed until the user clicks `Checkout` in the UI. Each time the user clicks `Checkout`, the content of the list is moved from the Swing `TableModel` into the KIE session working memory and is then executed with the `ksession.fireAllRules()` method.
Within this code, there are nine calls to `KieSession`. The first of these creates a new `KieSession` from the `KieContainer` (the example passed in this `KieContainer` from the `CheckoutCallBack` class in the `main()` method). The next two calls pass in the two objects that hold the global variables in the rules: the Swing text area and the Swing frame used for writing messages. More inserts put information on products into the `KieSession`, as well as the order list. The final call is the standard `fireAllRules()`.

**Pet Store rule file imports, global variables, and Java functions**

The `PetStore.drl` file contains the standard package and import statements to make various Java classes available to the rules. The rule file also includes `global variables` to be used within the rules, defined as `frame` and `textArea`. The global variables hold references to the Swing components `JFrame` and `JTextArea` components that were previously passed on by the Java code that called the `setGlobal()` method. Unlike standard variables in rules, which expire as soon as the rule has fired, global variables retain their value for the lifetime of the KIE session. This means the contents of these global variables are available for evaluation on all subsequent rules.

**PetStore.drl package, imports, and global variables**

```java
package org.drools.examples;

import org.kie.api.runtime.KieRuntime;
import org.drools.examples.petstore.PetStoreExample.Order;
import org.drools.examples.petstore.PetStoreExample.Purchase;
import org.drools.examples.petstore.PetStoreExample.Product;
import java.util.ArrayList;
import javax.swing.JOptionPane;
import javax.swing.JFrame;

global JFrame frame

global javax.swing.JTextArea textArea
```

The `PetStore.drl` file also contains two functions that the rules in the file use:

**PetStore.drl Java functions**

```java
function void doCheckout(JFrame frame, KieRuntime krt) {
    Object[] options = {"Yes", "No"};

    int n = JOptionPane.showOptionDialog(frame,
        "Would you like to checkout?", "",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null,
        options,
        options[0]);

    if (n == 0) {
        krt.getAgenda().getAgendaGroup("checkout").setFocus();
    }
}

function boolean requireTank(JFrame frame, KieRuntime krt, Order order, Product fishTank, int total) {
```

313
The two functions perform the following actions:

- **doCheckout()** displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the `checkout` agenda group, enabling rules in that group to (potentially) fire.

- **requireTank()** displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank `Product` is added to the order list in the working memory.

**NOTE**
For this example, all rules and functions are within the same rule file for efficiency. In a production environment, you typically separate the rules and functions in different files or build a static Java method and import the files using the `import` function, such as `import function my.package.name.hello`.

**Pet Store rules with agenda groups**
Most of the rules in the Pet Store example use agenda groups to control rule execution. Agenda groups allow you to partition the decision engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group `MAIN`. You can use the `agenda-group` attribute to specify a different agenda group for the rule.

Initially, a working memory has its focus on the agenda group `MAIN`. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method `setFocus()` or the rule attribute `auto-focus`. The `auto-focus` attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

The Pet Store example uses the following agenda groups for rules:
For example, the sample rule "Explode Cart" uses the "init" agenda group to ensure that it has the option to fire and insert shopping cart items into the KIE session working memory:

**Rule "Explode Cart"**

```java
// Insert each item in the shopping cart into the working memory.
rule "Explode Cart"
    agenda-group "init"
    auto-focus true
    salience 10
    when
    $order : Order( grossTotal == -1 )
    $item : Purchase() from $order.items
    then
        insert( $item );
        kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show items" ).setFocus();
        kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate" ).setFocus();
end
```

This rule matches against all orders that do not yet have their `grossTotal` calculated. The execution loops for each purchase item in that order.

The rule uses the following features related to its agenda group:

- **agenda-group "init"** defines the name of the agenda group. In this case, only one rule is in the group. However, neither the Java code nor a rule consequence sets the focus to this group, and therefore it relies on the `auto-focus` attribute for its chance to fire.

- **auto-focus true** ensures that this rule, while being the only rule in the agenda group, gets a chance to fire when `fireAllRules()` is called from the Java code.

- **kcontext...setFocus()** sets the focus to the "show items" and "evaluate" agenda groups, enabling their rules to fire. In practice, you loop through all items in the order, insert them into memory, and then fire the other rules after each insertion.

The "show items" agenda group contains only one rule, "Show Items". For each purchase in the order currently in the KIE session working memory, the rule logs details to the text area at the bottom of the GUI, based on the `textarea` variable defined in the rule file.

**Rule "Show Items"**

```java
rule "Show Items"
    agenda-group "show items"
    when
    $order : Order()
    $p : Purchase( order == $order )
```
The "evaluate" agenda group also gains focus from the "Explode Cart" rule. This agenda group contains two rules, "Free Fish Food Sample" and "Suggest Tank", which are executed in that order.

Rule "Free Fish Food Sample"

// Free fish food sample when users buy a goldfish if they did not already buy
// fish food and do not already have a fish food sample.

rule "Free Fish Food Sample"
  agenda-group "evaluate"
  when
    $order : Order()
    not ( $p : Product( name == "Fish Food") && Purchase( product == $p ) )
    not ( $p : Product( name == "Fish Food Sample") && Purchase( product == $p ) )
    exists ( $p : Product( name == "Gold Fish") && Purchase( product == $p ) )
    $fishFoodSample : Product( name == "Fish Food Sample" );
  then
    System.out.println( "Adding free Fish Food Sample to cart" );
    purchase = new Purchase($order, $fishFoodSample);
    insert( purchase );
    $order.addItem( purchase );
  end

The rule "Free Fish Food Sample" fires only if all of the following conditions are true:

1. The agenda group "evaluate" is being evaluated in the rules execution.
2. User does not already have fish food.
3. User does not already have a free fish food sample.
4. User has a goldfish in the order.

If the order facts meet all of these requirements, then a new product is created (Fish Food Sample) and is added to the order in working memory.

Rule "Suggest Tank"

// Suggest a fish tank if users buy more than five goldfish and
// do not already have a tank.

rule "Suggest Tank"
  agenda-group "evaluate"
  when
    $order : Order()
    not ( $p : Product( name == "Fish Tank") && Purchase( product == $p ) )
    ArrayList( $total : size > 5 ) from collect( Purchase( product.name == "Gold Fish" ) )
    $fishTank : Product( name == "Fish Tank" )
  then
    requireTank(frame, kcontext.getKieRuntime(), $order, $fishTank, $total);
  end
The rule "Suggest Tank" fires only if the following conditions are true:

1. User does not have a fish tank in the order.
2. User has more than five fish in the order.

When the rule fires, it calls the requireTank() function defined in the rule file. This function displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank Product is added to the order list in the working memory. When the rule calls the requireTank() function, the rule passes the frame global variable so that the function has a handle for the Swing GUI.

The "do checkout" rule in the Pet Store example has no agenda group and no when conditions, so the rule is always executed and considered part of the default MAIN agenda group.

**Rule "do checkout"**

```java
rule "do checkout"
  when
  then
    doCheckout(frame, kcontext.getKieRuntime());
end
```

When the rule fires, it calls the doCheckout() function defined in the rule file. This function displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the checkout agenda group, enabling rules in that group to (potentially) fire. When the rule calls the doCheckout() function, the rule passes the frame global variable so that the function has a handle for the Swing GUI.

**NOTE**

This example also demonstrates a troubleshooting technique if results are not executing as you expect: You can remove the conditions from the when statement of a rule and test the action in the then statement to verify that the action is performed correctly.

The "checkout" agenda group contains three rules for processing the order checkout and applying any discounts: "Gross Total", "Apply 5% Discount", and "Apply 10% Discount".

**Rules "Gross Total", "Apply 5% Discount", and "Apply 10% Discount"**

```java
rule "Gross Total"
  agenda-group "checkout"
  when
    $order : Order( grossTotal == -1)
    Number( total : doubleValue ) from accumulate( Purchase( $price : product.price ), sum( $price ) )
  then
    modify( $order ) { grossTotal = total }
    textArea.append( "gross total=\n" + total + "\n" );
end

rule "Apply 5% Discount"
  agenda-group "checkout"
  when
    $order : Order( grossTotal >= 10 && < 20 )
end
```
If the user has not already calculated the gross total, the *Gross Total* accumulates the product prices into a total, puts this total into the KIE session, and displays it through the Swing *JTextArea* using the *textArea* global variable.

If the gross total is between 10 and 20 (currency units), the "*Apply 5% Discount*" rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

If the gross total is not less than 20, the "*Apply 10% Discount*" rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

**Pet Store example execution**

Similar to other Red Hat Decision Manager decision examples, you execute the Pet Store example by running the `org.drools.examples.petstore.PetStoreExample` class as a Java application in your IDE.

When you execute the Pet Store example, the *Pet Store Demo* GUI window appears. This window displays a list of available products (upper left), an empty list of selected products (upper right), *Checkout* and *Reset* buttons (middle), and an empty system messages area (bottom).
Figure 21.14. Pet Store example GUI after launch

The following events occurred in this example to establish this execution behavior:

1. The `main()` method has run and loaded the rule base but has not yet fired the rules. So far, this is the only code in connection with rules that has been run.

2. A new `PetStoreUI` object has been created and given a handle for the rule base, for later use.

3. Various Swing components have performed their functions, and the initial UI screen is displayed and waits for user input.

You can click various products from the list to explore the UI setup:
No rules code has been fired yet. The UI uses Swing code to detect user mouse clicks and add selected products to the `TableModel` object for display in the upper-right corner of the UI. This example illustrates the Model-View-Controller design pattern.

When you click **Checkout**, the rules are then fired in the following way:

1. Method `CheckOutCallBack.checkout()` is called (eventually) by the Swing class waiting for a user to click **Checkout**. This inserts the data from the `TableModel` object (upper-right corner of the UI) into the KIE session working memory. The method then fires the rules.
2. The "Explode Cart" rule is the first to fire, with the auto-focus attribute set to true. The rule loops through all of the products in the cart, ensures that the products are in the working memory, and then gives the "show Items" and "evaluate" agenda groups the option to fire. The rules in these groups add the contents of the cart to the text area (bottom of the UI), evaluate if you are eligible for free fish food, and determine whether to ask if you want to buy a fish tank.

Figure 21.16. Fish tank qualification

3. The "do checkout" rule is the next to fire because no other agenda group currently has focus and because it is part of the default MAIN agenda group. This rule always calls the doCheckout() function, which asks you if you want to check out.

4. The doCheckout() function sets the focus to the "checkout" agenda group, giving the rules in that group the option to fire.

5. The rules in the "checkout" agenda group display the contents of the cart and apply the appropriate discount.

6. Swing then waits for user input to either select more products (and cause the rules to fire again) or to close the UI.
Figure 21.17. Pet Store example GUI after all rules have fired

You can add more `System.out` calls to demonstrate this flow of events in your IDE console:

**System.out output in the IDE console**

- Adding free Fish Food Sample to cart
- SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes

**21.7. HONEST POLITICIAN EXAMPLE DECISIONS (TRUTH MAINTENANCE AND SALIENCE)**
The Honest Politician example decision set demonstrates the concept of truth maintenance with logical insertions and the use of salience in rules.

The following is an overview of the Honest Politician example:

- **Name**: honestpolitician
- **Main class**: org.drools.examples.honestpolitician.HonestPoliticianExample (in src/main/java)
- **Module**: drools-examples
- **Type**: Java application
- **Rule file**: org.drools.examples.honestpolitician.HonestPolitician.drl (in src/main/resources)
- **Objective**: Demonstrates the concept of truth maintenance based on the logical insertion of facts and the use of salience in rules

The basic premise of the Honest Politician example is that an object can only exist while a statement is true. A rule consequence can logically insert an object with the `insertLogical()` method. This means the object remains in the KIE session working memory as long as the rule that logically inserted it remains true. When the rule is no longer true, the object is automatically retracted.

In this example, rule execution causes a group of politicians to change from being honest to being dishonest as a result of a corrupt corporation. As each politician is evaluated, they start out with their honesty attribute being set to `true`, but a rule fires that makes the politicians no longer honest. As they switch their state from being honest to dishonest, they are then removed from the working memory. The rule salience notifies the decision engine how to prioritize any rules that have a salience defined for them, otherwise utilizing the default salience value of 0. Rules with a higher salience value are given higher priority when ordered in the activation queue.

**Politician and Hope classes**
The sample class Politician in the example is configured for an honest politician. The Politician class is made up of a String item name and a Boolean item honest:

**Politician class**

```java
public class Politician {
    private String name;
    private boolean honest;
    ...
}
```

The Hope class determines if a Hope object exists. This class has no meaningful members, but is present in the working memory as long as society has hope.

**Hope class**

```java
public class Hope {
    public Hope() {
    }
}
```
Rule definitions for politician honesty
In the Honest Politician example, when at least one honest politician exists in the working memory, the
"We have an honest Politician" rule logically inserts a new Hope object. As soon as all politicians
become dishonest, the Hope object is automatically retracted. This rule has a salience attribute with a
value of 10 to ensure that it fires before any other rule, because at that stage the "Hope is Dead" rule
is true.

Rule "We have an honest politician"

```
rule "We have an honest Politician"
salience 10
when
   exists( Politician( honest == true ) )
then
   insertLogical( new Hope() );
end
```

As soon as a Hope object exists, the "Hope Lives" rule matches and fires. This rule also has a salience
value of 10 so that it takes priority over the "Corrupt the Honest" rule.

Rule "Hope Lives"

```
rule "Hope Lives"
salience 10
when
   exists( Hope() )
then
   System.out.println("Hurrah!!! Democracy Lives");
end
```

Initially, four honest politicians exist so this rule has four activations, all in conflict. Each rule fires in turn,
corrupting each politician so that they are no longer honest. When all four politicians have been
corrupted, no politicians have the property honest == true. The rule "We have an honest Politician" is
no longer true and the object it logically inserted (due to the last execution of new Hope()) is
automatically retracted.

Rule "Corrupt the Honest"

```
rule "Corrupt the Honest"
when
   politician : Politician( honest == true )
   exists( Hope() )
then
   System.out.println( "I'm an evil corporation and I have corrupted " + politician.getName() );
   modify ( politician ) { honest = false };
end
```

With the Hope object automatically retracted through the truth maintenance system, the conditional
element not applied to Hope is no longer true so that the "Hope is Dead" rule matches and fires.

Rule "Hope is Dead"

```
rule "Hope is Dead"
when
```
not( Hope() )
then
  System.out.println( "We are all Doomed!!! Democracy is Dead" );
end

Example execution and audit trail
In the HonestPoliticianExample.java class, the four politicians with the honest state set to true are inserted for evaluation against the defined business rules:

HonestPoliticianExample.java class execution

```java
public static void execute( KieContainer kc ) {
    KieSession ksession = kc.newKieSession("HonestPoliticianKS");

    final Politician p1 = new Politician( "President of Umpa Lumpa", true );
    final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
    final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
    final Politician p4 = new Politician( "Omnipotence Om", true );

    ksession.insert( p1 );
    ksession.insert( p2 );
    ksession.insert( p3 );
    ksession.insert( p4 );

    ksession.fireAllRules();
    ksession.dispose();
}
```

To execute the example, run the org.drools.examples.honestpolitician.HonestPoliticianExample class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Execution output in the IDE console

```
Hurrah!!! Democracy Lives
I'm an evil corporation and I have corrupted President of Umpa Lumpa
I'm an evil corporation and I have corrupted Prime Minster of Cheeseland
I'm an evil corporation and I have corrupted Tsar of Pringapopaloo
I'm an evil corporation and I have corrupted Omnipotence Om
We are all Doomed!!! Democracy is Dead
```

The output shows that, while there is at least one honest politician, democracy lives. However, as each politician is corrupted by some corporation, all politicians become dishonest, and democracy is dead.

To better understand the execution flow of this example, you can modify the HonestPoliticianExample.java class to include a DebugRuleRuntimeEventListener listener and an audit logger to view execution details:

HonestPoliticianExample.java class with an audit logger

```java
package org.drools.examples.honestpolitician;
import org.kie.api.KieServices;
```
import org.kie.api.event.rule.DebugAgendaEventListener;
import org.kie.api.event.rule.DebugRuleRuntimeEventListener;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class HonestPoliticianExample {

/**
 * @param args
 */

public static void main(final String[] args) {
   KieServices ks = KieServices.Factory.get();  //ks = KieServices.Factory.get();
   KieContainer kc = KieServices.Factory.get().getKieClasspathContainer();
   System.out.println(kc.verify().getMessages().toString());
   //execute( kc );
   execute( ks, kc);  
}

public static void execute( KieServices ks, KieContainer kc ) {

   KieSession ksession = kc.newKieSession("HonestPoliticianKS");

   final Politician p1 = new Politician( "President of Umpa Lumpa", true );
   final Politician p2 = new Politician( "Prime Minister of Cheeseland", true );
   final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
   final Politician p4 = new Politician( "Omnipotence Om", true );

   ksession.insert( p1 );
   ksession.insert( p2 );
   ksession.insert( p3 );
   ksession.insert( p4 );

   // The application can also setup listeners
   ksession.addEventListener( new DebugAgendaEventListener() );
   ksession.addEventListener( new DebugRuleRuntimeEventListener() );

   // Set up a file-based audit logger.
   ks.getLoggers().newFileLogger( ksession, ".target/honestpolitician" );

   ksession.fireAllRules();
   ksession.dispose();
}

Adds to your imports the packages that handle the DebugAgendaEventListener and DebugRuleRuntimeEventListener

Creates a KieServices Factory and a ks element to produce the logs because this audit log is not available at the KieContainer level

Modifies the execute method to use both KieServices and KieContainer

Modifies the execute method to pass in KieServices in addition to the KieContainer
When you run the Honest Politician with this modified logging capability, you can load the audit log file from `target/honestpolitician.log` into your IDE debug view or Audit View, if available (for example, in Window → Show View in some IDEs).

In this example, the Audit View shows the flow of executions, insertions, and retractions as defined in the example classes and rules:

Figure 21.18. Honest Politician example Audit View

When the first politician is inserted, two activations occur. The rule "We have an honest Politician" is activated only once time for the first inserted politician because it uses an exists conditional element, which matches when at least one politician is inserted. The rule "Hope is Dead" is also activated at this stage because the Hope object is not yet inserted. The rule "We have an honest Politician" fires first because it has a higher salience value than the rule "Hope is Dead", and inserts the Hope object (highlighted in green). The insertion of the Hope object activates the rule "Hope Lives" and deactivates the rule "Hope is Dead". The insertion also activates the rule "Corrupt the Honest" for each inserted honest politician. The rule "Hope Lives" is executed and prints "Hurrah!!! Democracy Lives".

Next, for each politician, the rule "Corrupt the Honest" fires, printing "I'm an evil corporation and I have corrupted X", where X is the name of the politician, and modifies the politician honesty value to false. When the last honest politician is corrupted, Hope is automatically retracted by the truth maintenance system (highlighted in blue). The green highlighted area shows the origin of the currently selected blue highlighted area. After the Hope fact is retracted, the rule "Hope is dead" fires, printing "We are all Doomed!!! Democracy is Dead".
21.8. SUDOKU EXAMPLE DECISIONS (COMPLEX PATTERN MATCHING, CALLBACKS, AND GUI INTEGRATION)

The Sudoku example decision set, based on the popular number puzzle Sudoku, demonstrates how to use rules in Red Hat Decision Manager to find a solution in a large potential solution space based on various constraints. This example also shows how to integrate Red Hat Decision Manager rules into a graphical user interface (GUI), in this case a Swing-based desktop application, and how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.

The following is an overview of the Sudoku example:

- **Name**: sudoku
- **Main class**: `org.drools.examples.sudoku.SudokuExample` (in `src/main/java`)
- **Module**: drools-examples
- **Type**: Java application
- **Rule files**: `org.drools.examples.sudoku.*.drl` (in `src/main/resources`)
- **Objective**: Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration

Sudoku is a logic-based number placement puzzle. The objective is to fill a 9x9 grid so that each column, each row, and each of the nine 3x3 zones contains the digits from 1 to 9 only one time. The puzzle setter provides a partially completed grid and the puzzle solver’s task is to complete the grid with these constraints.

The general strategy to solve the problem is to ensure that when you insert a new number, it must be unique in its particular 3x3 zone, row, and column. This Sudoku example decision set uses Red Hat Decision Manager rules to solve Sudoku puzzles from a range of difficulty levels, and to attempt to resolve flawed puzzles that contain invalid entries.

**Sudoku example execution and interaction**

Similar to other Red Hat Decision Manager decision examples, you execute the Sudoku example by running the `org.drools.examples.sudoku.SudokuExample` class as a Java application in your IDE.

When you execute the Sudoku example, the Drools Sudoku Example GUI window appears. This window contains an empty grid, but the program comes with various grids stored internally that you can load and solve.

Click File → Samples → Simple to load one of the examples. Notice that all buttons are disabled until a grid is loaded.
When you load the **Simple** example, the grid is filled according to the puzzle’s initial state.
Choose from the following options:

- Click **Solve** to fire the rules defined in the Sudoku example that fill out the remaining values and that make the buttons inactive again.
Figure 21.21. Simple sample solved

- Click **Step** to see the next digit found by the rule set. The console window in your IDE displays detailed information about the rules that are executing to solve the step.

**Step execution output in the IDE console**

- single 8 at [0,1]
- column elimination due to [1,2]: remove 9 from [4,2]
- hidden single 9 at [1,2]
- row elimination due to [2,8]: remove 7 from [2,4]
- remove 6 from [3,8] due to naked pair at [3,2] and [3,7]
- hidden pair in row at [4,6] and [4,4]

- Click **Dump** to see the state of the grid, with cells showing either the established value or the remaining possibilities.

**Dump execution output in the IDE console**

```
Row 0: 123456789 --- 5 --- 6 --- 8 --- 123456789 --- 1 --- 9 --- 4 ---
123456789
Row 1: 123456789 123456789 6 --- 8 --- 123456789 --- 5 --- 123456789
123456789 --- 3 ---
Row 2: 123456789 123456789 4 --- 9 --- 3 --- 123456789 123456789
--- 8 ---
```
The Sudoku example includes a deliberately broken sample file that the rules defined in the example can resolve.

Click **File → Samples → !DELIBERATELY BROKEN!** to load the broken sample. The grid starts with some issues, for example, the value **5** appears two times in the first row, which is not allowed.

Figure 21.22. Broken Sudoku example initial state

![Drools Sudoku Example](image)

Click **Solve** to apply the solving rules to this invalid grid. The associated solving rules in the Sudoku example detect the issues in the sample and attempts to solve the puzzle as far as possible. This process does not complete and leaves some cells empty.

The solving rule activity is displayed in the IDE console window:

**Detected issues in the broken sample**
cell [0,8]: 5 has a duplicate in row 0
cell [0,0]: 5 has a duplicate in row 0

cell [6,0]: 8 has a duplicate in col 0

cell [4,0]: 8 has a duplicate in col 0

Validation complete.

Figure 21.23. Broken sample solution attempt

The sample Sudoku files labeled **Hard** are more complex and the solving rules might not be able to solve them. The unsuccessful solution attempt is displayed in the IDE console window:

**Hard sample unresolved**

Validation complete.

... 

Sorry - can't solve this grid.

The rules that work to solve the broken sample implement standard solving techniques based on the sets of values that are still candidates for a cell. For example, if a set contains a single value, then this is the value for the cell. For a single occurrence of a value in one of the groups of nine cells, the rules insert a fact of type **Setting** with the solution value for some specific cell. This fact causes the elimination of this value from all other cells in any of the groups the cell belongs to and the value is retracted.

Other rules in the example reduce the permissible values for some cells. The rules "**naked pair**", "**hidden pair in row**", "**hidden pair in column**", and "**hidden pair in square**" eliminate possibilities but do not establish solutions. The rules "**X-wings in rows**", "**X-wings in columns**", "**intersection removal**"
row”, and “intersection removal column” perform more sophisticated eliminations.

Sudoku example classes
The package org.drools.examples.sudoku.swing contains the following core set of classes that implement a framework for Sudoku puzzles:

- The SudokuGridModel class defines an interface that is implemented to store a Sudoku puzzle as a 9x9 grid of Cell objects.
- The SudokuGridView class is a Swing component that can visualize any implementation of the SudokuGridModel class.
- The SudokuGridEvent and SudokuGridListener classes communicate state changes between the model and the view. Events are fired when a cell value is resolved or changed.
- The SudokuGridSamples class provides partially filled Sudoku puzzles for demonstration purposes.

NOTE
This package does not have any dependencies on Red Hat Decision Manager libraries.

The package org.drools.examples.sudoku contains the following core set of classes that implement the elementary Cell object and its various aggregations:

- The CellFile class, with subtypes CellRow, CellCol, and CellSqr, all of which are subtypes of the CellGroup class.
- The Cell and CellGroup subclasses of SetOfNine, which provides a property free with the type Set<Integer>. For a Cell class, the set represents the individual candidate set. For a CellGroup class, the set is the union of all candidate sets of its cells (the set of digits that still need to be allocated).
  In the Sudoku example are 81 Cell and 27 CellGroup objects and a linkage provided by the Cell properties cellRow, cellCol, and cellSqr, and by the CellGroup property cells (a list of Cell objects). With these components, you can write rules that detect the specific situations that permit the allocation of a value to a cell or the elimination of a value from some candidate set.
- The Setting class is used to trigger the operations that accompany the allocation of a value. The presence of a Setting fact is used in all rules that detect a new situation in order to avoid reactions to inconsistent intermediary states.
- The Stepping class is used in a low priority rule to execute an emergency halt when a “Step” does not terminate regularly. This behavior indicates that the program cannot solve the puzzle.
- The main class org.drools.examples.sudoku.SudokuExample implements a Java application combining all of these components.

Sudoku validation rules (validate.drl)
The validate.drl file in the Sudoku example contains validation rules that detect duplicate numbers in cell groups. They are combined in a “validate” agenda group that enables the rules to be explicitly activated after a user loads the puzzle.

The when conditions of the three rules "duplicate in cell ..." all function in the following ways:

- The first condition in the rule locates a cell with an allocated value.
The second condition in the rule pulls in any of the three cell groups to which the cell belongs.

- The final condition finds a cell (other than the first one) with the same value as the first cell and in the same row, column, or square, depending on the rule.

Rules "duplicate in cell ..."

```
rule "duplicate in cell row"
  when
  $c: Cell( $v: value != null )
  $cr: CellRow( cells contains $c )
  exists Cell( this != $c, value == $v, cellRow == $cr )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in row " + $cr.getNumber() );
  end

rule "duplicate in cell col"
  when
  $c: Cell( $v: value != null )
  $cc: CellCol( cells contains $c )
  exists Cell( this != $c, value == $v, cellCol == $cc )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in col " + $cc.getNumber() );
  end

rule "duplicate in cell sqr"
  when
  $c: Cell( $v: value != null )
  $cs: CellSqr( cells contains $c )
  exists Cell( this != $c, value == $v, cellSqr == $cs )
  then
    System.out.println( "cell " + $c.toString() + " has duplicate in its square of nine." );
  end
```

The rule "terminate group" is the last to fire. This rule prints a message and stops the sequence.

Rule "terminate group"

```
rule "terminate group"
  salience -100
  when
  then
    System.out.println( "Validation complete." );
    drools.halt();
  end
```

Sudoku solving rules (sudoku.drl)
The `sudoku.drl` file in the Sudoku example contains three types of rules: one group handles the allocation of a number to a cell, another group detects feasible allocations, and the third group eliminates values from candidate sets.

The rules "set a value", "eliminate a value from Cell", and "retract setting" depend on the presence of a Setting object. The first rule handles the assignment to the cell and the operations for removing the value from the free sets of the three groups of the cell. This group also reduces a counter that, when zero, returns control to the Java application that has called `fireUntilHalt()`.
The purpose of the rule "eliminate a value from Cell" is to reduce the candidate lists of all cells that are related to the newly assigned cell. Finally, when all eliminations have been made, the rule "retract setting" retracts the triggering Setting fact.

Rules "set a value", "eliminate a value from a Cell", and "retract setting"

// A Setting object is inserted to define the value of a Cell.  
// Rule for updating the cell and all cell groups that contain it  
rule "set a value"  
  when  
  // A Setting with row and column number, and a value  
  $s: Setting( $rn: rowNo, $cn: colNo, $v: value )  
  // A matching Cell, with no value set  
  $c: Cell( rowNo == $rn, colNo == $cn, value == null,  
    $cr: cellRow, $cc: cellCol, $cs: cellSqr )  
  // Count down  
  $ctr: Counter( $count: count )  
  then  
  // Modify the Cell by setting its value.  
  modify( $c ){ setValue( $v ) }  
  // System.out.println( "set cell " + $c.toString() );  
  modify( $cr ){ blockValue( $v ) }  
  modify( $cc ){ blockValue( $v ) }  
  modify( $cs ){ blockValue( $v ) }  
  modify( $ctr ){ setCount( $count - 1 ) }  
end  

// Rule for removing a value from all cells that are siblings in one of the three cell groups  
rule "eliminate a value from Cell"  
  when  
  // A Setting with row and column number, and a value  
  $s: Setting( $rn: rowNo, $cn: colNo, $v: value )  
  // The matching Cell, with the value already set  
  $c: Cell( rowNo == $rn, colNo == $cn, value == $v, $exCells: exCells )  
  // For all Cells that are associated with the updated cell  
  $c: Cell( free contains $v ) from $exCells  
  then  
  // System.out.println( "clear " + $v + " from cell " + $c.posAsString() );  
  // Modify a related Cell by blocking the assigned value.  
  modify( $c ){ blockValue( $v ) }  
end  

// Rule for eliminating the Setting fact  
rule "retract setting"  
  when  
  // A Setting with row and column number, and a value  
  $s: Setting( $rn: rowNo, $cn: colNo, $v: value )  
  // The matching Cell, with the value already set  
  $c: Cell( rowNo == $rn, colNo == $cn, value == $v )
Two solving rules detect a situation where an allocation of a number to a cell is possible. The rule "single" fires for a Cell with a candidate set containing a single number. The rule "hidden single" fires when no cell exists with a single candidate, but when a cell exists containing a candidate, this candidate is absent from all other cells in one of the three groups to which the cell belongs. Both rules create and insert a Setting fact.

Rules "single" and "hidden single"

```java
// Detect a set of candidate values with cardinality 1 for some Cell. // This is the value to be set. rule "single" when // Currently no setting underway not Setting() // One element in the "free" set $c: Cell( $rn: rowNo, $cn: colNo, freeCount == 1 ) then Integer i = $c.getFreeValue(); if (explain) System.out.println( "single " + i + " at " + $c.posAsString() ); // Insert another Setter fact. insert( new Setting( $rn, $cn, i ) ); end

// Detect a set of candidate values with a value that is the only one // in one of its groups. This is the value to be set. rule "hidden single" when // Currently no setting underway not Setting() not Cell( freeCount == 1 ) // Some integer $i: Integer() // The "free" set contains this number $c: Cell( $rn: rowNo, $cn: colNo, freeCount > 1, free contains $i ) // A cell group contains this cell $c. $cg: CellGroup( cells contains $c ) // No other cell from that group contains $i. not ( Cell( this != $c, free contains $i ) from $cg.getCells() ) then if (explain) System.out.println( "hidden single " + $i + " at " + $c.posAsString() );
```
Rules from the largest group, either individually or in groups of two or three, implement various solving techniques used for solving Sudoku puzzles manually.

The rule "naked pair" detects identical candidate sets of size 2 in two cells of a group. These two values may be removed from all other candidate sets of that group.

**Rule "naked pair"**

```
// A "naked pair" is two cells in some cell group with their sets of
// permissible values being equal with cardinality 2. These two values
// can be removed from all other candidate lists in the group.
rule "naked pair"
  when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // One cell with two candidates
    $c1: Cell( freeCount == 2, $f1: free, $r1: cellRow, $rn1: rowNo, $cn1: colNo, $b1: cellSqr )

    // The containing cell group
    $cg: CellGroup( freeCount > 2, cells contains $c1 )

    // Another cell with two candidates, not the one we already have
    $c2: Cell( this != $c1, free == $f1 /***, rowNo >= $rn1, colNo >= $cn1 */ ) from $cg.cells

    // Get one of the "naked pair".
    Integer( $v: intValue ) from $c1.getFree()

    // Get some other cell with a candidate equal to one from the pair.
    $c3: Cell( this != $c1 && != $c2, freeCount > 1, free contains $v ) from $cg.cells
  then
    if (explain) System.out.println( "remove " + $v + " from " + $c3.posAsString() + " due to naked pair at " + $c1.posAsString() + " and " + $c2.posAsString() );
    // Remove the value.
    modify( $c3 ){ blockValue( $v ) }
end
```

The three rules "hidden pair in ..." functions similarly to the rule "naked pair". These rules detect a subset of two numbers in exactly two cells of a group, with neither value occurring in any of the other cells of the group. This means that all other candidates can be eliminated from the two cells harboring the hidden pair.

**Rules "hidden pair in ..."**

```
// If two cells within the same cell group contain candidate sets with more than
// two values, with two values being in both of them but in none of the other
// cells, then we have a "hidden pair". We can remove all other candidates from
// these two cells.
rule "hidden pair in row"
  when
```
// Currently no setting underway
not Setting()
not Cell( freeCount == 1 )

// Establish a pair of Integer facts.
$i1: Integer()
$i2: Integer( this > $i1 )

// Look for a Cell with these two among its candidates. (The upper bound on
// the number of candidates avoids a lot of useless work during startup.)
$\text{c1}: \text{Cell( } \text{\$rn1: rowNo, } \text{\$cn1: colNo, freeCount > 2 \&\& < 9, free contains } \$i1 \&\& \text{ contains } \$i2, \$
\text{cellRow: cellRow } )$

// Get another one from the same row, with the same pair among its candidates.
$\text{c2}: \text{Cell( this != } \text{\$c1, cellRow == } \text{\$cellRow, freeCount > 2, free contains } \$i1 \&\& \text{ contains } \$i2 )$

// Ascertain that no other cell in the group has one of these two values.
not( Cell( this != $\text{\$c1}$ \&\& != $\text{\$c2}$, free contains $\$i1$ \&\& \text{ contains } $\$i2$ ) from $\text{cellRow.getCells()} )
then
if( explain) System.out.println( "hidden pair in row at " + $\text{\$c1}.posAsString() + " and " +
$\text{\$c2}.posAsString() );
// Set the candidate lists of these two Cells to the "hidden pair".
modify( $\text{\$c1} )$ { blockExcept( $\$i1, $\$i2 ) }
modify( $\text{\$c2} )$ { blockExcept( $\$i1, $\$i2 ) }
end

rule "hidden pair in column"
when
not Setting()
not Cell( freeCount == 1 )

$\$i1: Integer()
$\$i2: Integer( this > $\$i1 )
$\text{c1}: \text{Cell( } \text{\$rn1: rowNo, } \text{\$cn1: colNo, freeCount > 2 \&\& < 9, free contains } \$i1 \&\& \text{ contains } \$i2, \$
\text{cellCol: cellCol } )$

$\text{c2}: \text{Cell( this != } \text{\$c1, cellCol == } \text{\$cellCol, freeCount > 2, free contains } \$i1 \&\& \text{ contains } \$i2 )
not( Cell( this != $\text{\$c1}$ \&\& != $\text{\$c2}$, free contains $\$i1$ \&\& \text{ contains } $\$i2$ ) from $\text{cellCol.getCells()} )
then
if (explain) System.out.println( "hidden pair in column at " + $\text{\$c1}.posAsString() + " and " +
$\text{\$c2}.posAsString() );
// Set the candidate lists of these two Cells to the "hidden pair".
modify( $\text{\$c1} )$ { blockExcept( $\$i1, $\$i2 ) }
modify( $\text{\$c2} )$ { blockExcept( $\$i1, $\$i2 ) }
end

rule "hidden pair in square"
when
not Setting()
not Cell( freeCount == 1 )

$\$i1: Integer()
$\$i2: Integer( this > $\$i1 )
$\text{c1}: \text{Cell( } \text{\$rn1: rowNo, } \text{\$cn1: colNo, freeCount > 2 \&\& < 9, free contains } \$i1 \&\& \text{ contains } \$i2, \$
\text{cellSqr: cellSqr } )$

$\text{c2}: \text{Cell( this != } \text{\$c1, cellSqr == } \text{\$cellSqr, freeCount > 2, free contains } \$i1 \&\& \text{ contains } \$i2 )
not( Cell( this != $\text{\$c1}$ \&\& != $\text{\$c2}$, free contains $\$i1$ \&\& \text{ contains } $\$i2$ ) from $\text{cellSqr.getCells()} )
then
if (explain) System.out.println( "hidden pair in square " + $c1.posAsString() + " and " + $c2.posAsString() );
modify( $c1 ){ blockExcept( $i1, $i2 ) }
modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

Two rules deal with "X-wings" in rows and columns. When only two possible cells for a value exist in each of two different rows (or columns) and these candidates lie also in the same columns (or rows), then all other candidates for this value in the columns (or rows) can be eliminated. When you follow the pattern sequence in one of these rules, notice how the conditions that are conveniently expressed by words such as same or only result in patterns with suitable constraints or that are prefixed with not.

Rules "X-wings in ..."

rule "X-wings in rows"
when
not Setting()
not Cell( freeCount == 1 )

$i: Integer()

$ca1: Cell( freeCount > 1, free contains $i, $ra: cellRow, $rano: rowNo, $c1: cellCol, $c1no: colNo )
$cb1: Cell( freeCount > 1, free contains $i, $rb: cellRow, $rbno: rowNo > $rano, cellCol == $c1 )

not( Cell( this != $ca1 && != $cb1, free contains $i ) from $c1.getCells() )

$ca2: Cell( freeCount > 1, free contains $i, cellRow == $ra, $c2: cellCol, $c2no: colNo > $c1no )
$cb2: Cell( freeCount > 1, free contains $i, cellRow == $rb, cellCol == $c2 )

not( Cell( this != $ca2 && != $cb2, free contains $i ) from $c2.getCells() )

$cx: Cell( rowNo == $rano || == $rbno, colNo != $c1no && != $c2no, freeCount > 1, free contains $i )
then
if (explain) {
    System.out.println( "X-wing with " + $i + " in rows " + $ca1.posAsString() + " - " + $cb1.posAsString() + $ca2.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString() );
}
modify( $cx ){ blockValue( $i ) }
end

rule "X-wings in columns"
when
not Setting()
not Cell( freeCount == 1 )

$i: Integer()

$ca1: Cell( freeCount > 1, free contains $i, $c1: cellCol, $c1no: colNo, $ra: cellRow, $rano: rowNo )
$ca2: Cell( freeCount > 1, free contains $i, $c2: cellCol, $c2no: colNo > $c1no, cellRow == $ra )

not( Cell( this != $ca1 && != $ca2, free contains $i ) from $ra.getCells() )

$cb1: Cell( freeCount > 1, free contains $i,
The two rules "intersection removal ..." are based on the restricted occurrence of some number within one square, either in a single row or in a single column. This means that this number must be in one of those two or three cells of the row or column and can be removed from the candidate sets of all other cells of the group. The pattern establishes the restricted occurrence and then fires for each cell outside of the square and within the same cell file.

Rules "intersection removal ...

rule "intersection removal column"
when
not Setting()
not Cell( freeCount == 1 )

$i: Integer()
// Occurs in a Cell
$c: Cell( free contains $i, $cs: cellSqr, $cc: cellCol )
// Does not occur in another cell of the same square and a different column
not Cell( this != $c, free contains $i, cellSqr == $cs, cellCol != $cc )

// A cell exists in the same column and another square containing this value.
$cx: Cell( freeCount > 1, free contains $i, cellCol == $cc, cellSqr != $cs )
then
// Remove the value from that other cell.
if (explain) {
    System.out.println( "column elimination due to " + $c.posAsString() + ": remove " + $i + " from " + $cx.posAsString() );
}
modify( $cx ){ blockValue( $i ) }
end

rule "intersection removal row"
when
not Setting()
not Cell( freeCount == 1 )

$i: Integer()
// Occurs in a Cell
$c: Cell( free contains $i, $cs: cellSqr, $cr: cellRow )
// Does not occur in another cell of the same square and a different row.
not Cell( this != $c, free contains $i, cellSqr == $cs, cellRow != $cr )

These rules are sufficient for many but not all Sudoku puzzles. To solve very difficult grids, the rule set requires more complex rules. (Ultimately, some puzzles can be solved only by trial and error.)

21.9. CONWAY’S GAME OF LIFE EXAMPLE DECISIONS (RULEFLOW GROUPS AND GUI INTEGRATION)

The Conway’s Game of Life example decision set, based on the famous cellular automaton by John Conway, demonstrates how to use ruleflow groups in rules to control rule execution. The example also demonstrates how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based implementation of Conway’s Game of Life.

The following is an overview of the Conway’s Game of Life (Conway) example:

- **Name**: conway
- **Main classes**: org.drools.examples.conway.ConwayRuleFlowGroupRun, org.drools.examples.conway.ConwayAgendaGroupRun (in src/main/java)
- **Module**: droolsjbpm-integration-examples
- **Type**: Java application
- **Rule files**: org.drools.examples.conway.*.drl (in src/main/resources)
- **Objective**: Demonstrates ruleflow groups and GUI integration

**NOTE**

The Conway’s Game of Life example is separate from most of the other example decision sets in Red Hat Decision Manager and is located in ~/rhdm-7.10.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples of the Red Hat Decision Manager 7.10.0 Source Distribution from the Red Hat Customer Portal.

In Conway’s Game of Life, a user interacts with the game by creating an initial configuration or an advanced pattern with defined properties and then observing how the initial state evolves. The objective of the game is to show the development of a population, generation by generation. Each generation results from the preceding one, based on the simultaneous evaluation of all cells.

The following basic rules govern what the next generation looks like:

- If a live cell has fewer than two live neighbors, it dies of loneliness.
• If a live cell has more than three live neighbors, it dies from overcrowding.
• If a dead cell has exactly three live neighbors, it comes to life.

Any cell that does not meet any of those criteria is left as is for the next generation.

The Conway’s Game of Life example uses Red Hat Decision Manager rules with `ruleflow-group` attributes to define the pattern implemented in the game. The example also contains a version of the decision set that achieves the same behavior using agenda groups. Agenda groups enable you to partition the decision engine agenda to provide execution control over groups of rules. By default, all rules are in the agenda group `MAIN`. You can use the `agenda-group` attribute to specify a different agenda group for the rule.

This overview does not explore the version of the Conway example using agenda groups. For more information about agenda groups, see the Red Hat Decision Manager example decision sets that specifically address agenda groups.

**Conway example execution and interaction**

Similar to other Red Hat Decision Manager decision examples, you execute the Conway ruleflow example by running the `org.drools.examples.conway.ConwayRuleFlowGroupRun` class as a Java application in your IDE.

When you execute the Conway example, the **Conway’s Game of Life** GUI window appears. This window contains an empty grid, or “arena” where the life simulation takes place. Initially the grid is empty because no live cells are in the system yet.

**Figure 21.24. Conway example GUI after launch**

Select a predefined pattern from the **Pattern** drop-down menu and click **Next Generation** to click through each population generation. Each cell is either alive or dead, where live cells contain a green ball. As the population evolves from the initial pattern, cells live or die relative to neighboring cells, according to the rules of the game.
Neighbors include not only cells to the left, right, top, and bottom but also cells that are connected diagonally, so that each cell has a total of eight neighbors. Exceptions are the corner cells, which have only three neighbors, and the cells along the four borders, with five neighbors each.

You can manually intervene to create or kill cells by clicking the cell.

To run through an evolution automatically from the initial pattern, click **Start**.

**Conway example rules with ruleflow groups**

The rules in the **ConwayRuleFlowGroupRun** example use ruleflow groups to control rule execution. A ruleflow group is a group of rules associated by the **ruleflow-group** rule attribute. These rules can only fire when the group is activated. The group itself can only become active when the elaboration of the ruleflow diagram reaches the node representing the group.

The Conway example uses the following ruleflow groups for rules:

- "register neighbor"
- "evaluate"
- "calculate"
- "reset calculate"
- "birth"
- "kill"
- "kill all"

All of the **Cell** objects are inserted into the KIE session and the "register ..." rules in the ruleflow group "register neighbor" are allowed to execute by the ruleflow process. This group of four rules creates **Neighbor** relations between some cell and its northeastern, northern, northwestern, and western neighbors.
This relation is bidirectional and handles the other four directions. Border cells do not require any special treatment. These cells are not paired with neighboring cells where there is not any.

By the time all activations have fired for these rules, all cells are related to all their neighboring cells.

Rules "register ..."

```java
rule "register north east"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northEast : Cell( row == ($row - 1), col == ($col + 1) )
  then
    insert( new Neighbor( $cell, $northEast ) );
    insert( new Neighbor( $northEast, $cell ) );
  end

rule "register north"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $north : Cell( row == ($row - 1), col == $col )
  then
    insert( new Neighbor( $cell, $north ) );
    insert( new Neighbor( $north, $cell ) );
  end

rule "register north west"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northWest : Cell( row == ($row - 1), col == ($col - 1) )
  then
    insert( new Neighbor( $cell, $northWest ) );
    insert( new Neighbor( $northWest, $cell ) );
  end

rule "register west"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $west : Cell( row == $row, col == ($col - 1) )
  then
    insert( new Neighbor( $cell, $west ) );
    insert( new Neighbor( $west, $cell ) );
  end
```

After all the cells are inserted, some Java code applies the pattern to the grid, setting certain cells to Live. Then, when the user clicks Start or Next Generation, the example executes the Generation ruleflow. This ruleflow manages all changes of cells in each generation cycle.
The ruleflow process enters the "evaluate" ruleflow group and any active rules in the group can fire. The rules "Kill the ..." and "Give Birth" in this group apply the game rules to birth or kill cells. The example uses the phase attribute to drive the reasoning of the Cell object by specific groups of rules. Typically, the phase is tied to a ruleflow group in the ruleflow process definition.

Notice that the example does not change the state of any Cell objects at this point because it must complete the full evaluation before those changes can be applied. The example sets the cell to a phase that is either Phase.KILL or Phase.BIRTH, which is used later to control actions applied to the Cell object.

Rules "Kill the ..." and "Give Birth"

```plaintext
rule "Kill The Lonely"
    ruleflow-group "evaluate"
    no-loop
    when
        // A live cell has fewer than 2 live neighbors.
        theCell: Cell( liveNeighbors < 2, cellState == CellState.LIVE, phase == Phase.EVALUATE )
    then
        modify( theCell ){
            setPhase( Phase.KILL );
        }
end

rule "Kill The Overcrowded"
    ruleflow-group "evaluate"
    no-loop
    when
        // A live cell has more than 3 live neighbors.
        theCell: Cell( liveNeighbors > 3, cellState == CellState.LIVE, phase == Phase.EVALUATE )
    then
```
modify( theCell ){
    setPhase( Phase.KILL );
}
end

rule "Give Birth"
    ruleflow-group "evaluate"
    no-loop
    when
        // A dead cell has 3 live neighbors.
        theCell: Cell( liveNeighbors == 3, cellState == CellState.DEAD,
                        phase == Phase.EVALUATE )
    then
        modify( theCell ){
            theCell.setPhase( Phase.BIRTH );
        }
    end

After all Cell objects in the grid have been evaluated, the example uses the "reset calculate" rule to clear any activations in the "calculate" ruleflow group. The example then enters a split in the ruleflow that enables the rules "kill" and "birth" to fire, if the ruleflow group is activated. These rules apply the state change.

Rules "reset calculate", "kill", and "birth"

rule "reset calculate"
    ruleflow-group "reset calculate"
    when
    then
        WorkingMemory wm = drools.getWorkingMemory();
        wm.clearRuleFlowGroup( "calculate" );
    end

rule "kill"
    ruleflow-group "kill"
    no-loop
    when
        theCell: Cell( phase == Phase.KILL )
    then
        modify( theCell ){
            setCellState( CellState.DEAD ),
            setPhase( Phase.DONE );
        }
    end

rule "birth"
    ruleflow-group "birth"
    no-loop
    when
        theCell: Cell( phase == Phase.BIRTH )
    then
        modify( theCell ){
            setCellState( CellState.LIVE ),
            setPhase( Phase.DONE );
        }
    end
At this stage, several Cell objects have been modified with the state changed to either LIVE or DEAD. When a cell becomes live or dead, the example uses the Neighbor relation in the rules "Calculate ..." to iterate over all surrounding cells, increasing or decreasing the liveNeighbor count. Any cell that has its count changed is also set to the EVALUATE phase to make sure it is included in the reasoning during the evaluation stage of the ruleflow process.

After the live count has been determined and set for all cells, the ruleflow process ends. If the user initially clicked Start, the decision engine restarts the ruleflow at that point. If the user initially clicked Next Generation, the user can request another generation.

Rules "Calculate ..."

```java
rule "Calculate Live"
    ruleflow-group "calculate"
    lock-on-active
    when
        theCell: Cell( cellState == CellState.LIVE )
        Neighbor( cell == theCell, $neighbor : neighbor )
    then
        modify( $neighbor ){
            setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 ),
            setPhase( Phase.EVALUATE );
        }
end

rule "Calculate Dead"
    ruleflow-group "calculate"
    lock-on-active
    when
        theCell: Cell( cellState == CellState.DEAD )
        Neighbor( cell == theCell, $neighbor : neighbor )
    then
        modify( $neighbor ){
            setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 ),
            setPhase( Phase.EVALUATE );
        }
end
```

21.10. HOUSE OF DOOM EXAMPLE DECISIONS (BACKWARD CHAINING AND RECURSION)

The House of Doom example decision set demonstrates how the decision engine uses backward chaining and recursion to reach defined goals or subgoals in a hierarchical system.

The following is an overview of the House of Doom example:

- **Name:** backwardchaining
- **Main class:** org.drools.examples.backwardchaining.HouseOfDoomMain (in src/main/java)
- **Module:** drools-examples
- **Type:** Java application
- **Rule file**: `org.drools.examples.backwardchaining.BC-Example.drl` (in `src/main/resources`)
- **Objective**: Demonstrates backward chaining and recursion

A backward-chaining rule system is a goal-driven system that starts with a conclusion that the decision 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.

In contrast, a forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision 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.

The decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:
Figure 21.27. Rule evaluation logic using forward and backward chaining

The House of Doom example uses rules with various types of queries to find the location of rooms and items within the house. The sample class `Location.java` contains the `item` and `location` elements used in the example. The sample class `HouseOfDoomMain.java` inserts the items or rooms in their respective locations in the house and executes the rules.

**Items and locations in HouseOfDoomMain.java class**

```java
ksession.insert( new Location("Office", "House") );
ksession.insert( new Location("Kitchen", "House") );
ksession.insert( new Location("Knife", "Kitchen") );
ksession.insert( new Location("Cheese", "Kitchen") );
```
The example rules rely on backward chaining and recursion to determine the location of all items and rooms in the house structure.

The following diagram illustrates the structure of the House of Doom and the items and rooms within it:

**Figure 21.28. House of Doom structure**

```java
ksession.insert( new Location("Desk", "Office") );
ksession.insert( new Location("Chair", "Office") );
ksession.insert( new Location("Computer", "Desk") );
ksession.insert( new Location("Drawer", "Desk") );
```

To execute the example, run the `org.drools.examples.backwardchaining.HouseOfDoomMain` class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

**Execution output in the IDE console**

```plaintext
go1
Office is in the House
---
go2
Drawer is in the House
---
go3
---
Key is in the Office
---

go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
---

go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
```
All rules in the example have fired to detect the location of all items in the house and to print the location of each in the output.

**Recursive query and related rules**

A recursive query repeatedly searches through the hierarchy of a data structure for relationships between elements.

In the House of Doom example, the BC-Example.drl file contains an `isContainedIn` query that most of the rules in the example use to recursively evaluate the house data structure for data inserted into the decision engine:

**Recursive query in BC-Example.drl**

```drl
query isContainedIn( String x, String y )
    Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

The rule "go" prints every string inserted into the system to determine how items are implemented, and the rule "go1" calls the query `isContainedIn`:

**Rules "go" and "go1"**

```drl
rule "go" salience 10
  when
  $s : String()
  then
  System.out.println( $s );
end

rule "go1"
  when
  String( this == "go1" )
  isContainedIn("Office", "House";)
  then
  System.out.println( "Office is in the House" );
end
```
The example inserts the "go1" string into the decision engine and activates the "go1" rule to detect that item Office is in the location House:

**Insert string and fire rules**

```
ksession.insert("go1");
ksession.fireAllRules();
```

**Rule "go1" output in the IDE console**

```
go1
Office is in the House
```

**Transitive closure rule**

Transitive closure is a relationship between an element contained in a parent element that is multiple levels higher in a hierarchical structure.

The rule "go2" identifies the transitive closure relationship of the Drawer and the House: The Drawer is in the Desk in the Office in the House.

```
rule "go2"
when
  String( this == "go2" )
  isContainedIn("Drawer", "House");
then
  System.out.println( "Drawer is in the House" );
end
```

The example inserts the "go2" string into the decision engine and activates the "go2" rule to detect that item Drawer is ultimately within the location House:

**Insert string and fire rules**

```
ksession.insert("go2");
ksession.fireAllRules();
```

**Rule "go2" output in the IDE console**

```
go2
Drawer is in the House
```

The decision engine determines this outcome based on the following logic:

1. The query recursively searches through several levels in the house to detect the transitive closure between Drawer and House.
2. Instead of using `Location( x, y; )`, the query uses the value of `(z, y; )` because Drawer is not directly in House.
3. The `z` argument is currently unbound, which means it has no value and returns everything that is in the argument.
4. The `y` argument is currently bound to House, so `z` returns Office and Kitchen.
5. The query gathers information from the *Office* and checks recursively if the *Drawer* is in the *Office*. The query line `isContainedIn( x, z; )` is called for these parameters.

6. No instance of *Drawer* exists directly in *Office*, so no match is found.

7. With *z* unbound, the query returns data within the *Office* and determines that *z == Desk*

   ```
   isContainedIn(x==drawer, z==desk)
   ```

8. The *isContainedIn* query recursively searches three times, and on the third time, the query detects an instance of *Drawer* in *Desk*.

   ```
   Location(x==drawer, y==desk)
   ```

9. After this match on the first location, the query recursively searches back up the structure to determine that the *Drawer* is in the *Desk*, the *Desk* is in the *Office*, and the *Office* is in the *House*. Therefore, the *Drawer* is in the *House* and the rule is satisfied.

**Reactive query rule**

A reactive query searches through the hierarchy of a data structure for relationships between elements and is dynamically updated when elements in the structure are modified.

The rule "*go3*" functions as a reactive query that detects if a new item *Key* ever becomes present in the *Office* by transitive closure: A *Key* in the *Drawer* in the *Office*.

**Rule "go3"**

```java
rule "go3"
when
  String( this == "go3" )
  isContainedIn("Key", "Office"; )
then
  System.out.println("Key is in the Office");
end
```

The example inserts the "*go3*" string into the decision engine and activates the "*go3*" rule. Initially, this rule is not satisfied because no item *Key* exists in the house structure, so the rule produces no output.

**Insert string and fire rules**

```java
ksession.insert( "go3" );
ksession.fireAllRules();
```

**Rule "go3" output in the IDE console (unsatisfied)**

```
go3
```

The example then inserts a new item *Key* in the location *Drawer*, which is in *Office*. This change satisfies the transitive closure in the "*go3*" rule and the output is populated accordingly.

**Insert new item location and fire rules**
ksession.insert( new Location("Key", "Drawer") );
ksession.fireAllRules();

Rule "go3" output in the IDE console (satisfied)

Key is in the Office

This change also adds another level in the structure that the query includes in subsequent recursive searches.

Queries with unbound arguments in rules
A query with one or more unbound arguments returns all undefined (unbound) items within a defined (bound) argument of the query. If all arguments in a query are unbound, then the query returns all items within the scope of the query.

The rule "go4" uses an unbound argument thing to search for all items within the bound argument Office, instead of using a bound argument to search for a specific item in the Office:

Rule "go4"

```java
rule "go4"
when
  String( this == "go4" )
  isContainedIn(thing, "Office"; )
then
  System.out.println( thing + "is in the Office" );
end
```

The example inserts the "go4" string into the decision engine and activates the "go4" rule to return all items in the Office:

Insert string and fire rules

```java
ksession.insert( "go4" );
ksession.fireAllRules();
```

Rule "go4" output in the IDE console

```
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
```

The rule "go5" uses both unbound arguments thing and location to search for all items and their locations in the entire House data structure:

Rule "go5"

```java
rule "go5"
when
  String( this == "go5" )
```
isContainedIn(thing, location; )
then
    System.out.println(thing + " is in " + location);
end

The example inserts the "go5" string into the decision engine and activates the "go5" rule to return all items and their locations in the House data structure:

Insert string and fire rules
ksession.insert("go5");
ksession.fireAllRules();

Rule "go5" output in the IDE console
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk
CHAPTER 22. PERFORMANCE TUNING CONSIDERATIONS WITH DRL

The following key concepts or suggested practices can help you optimize DRL rules and decision 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 Red Hat Decision Manager.

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 decision engine performance.

For more information about DRL patterns and constraints, see Section 16.8, “Rule conditions in DRL (WHEN)”.

Use equality operators more than other operator types in pattern constraints when possible

Although the decision 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 decision 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 decision 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 decision 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 Section 16.8, “Rule conditions in DRL (WHEN)”.

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**

```drl
when
$c: Company()
$e : Employee ( salary > 100000.00) from $c.employees
```

In such cases, the decision 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 decision 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**

```drl
when
$c: Company();
Employee (salary > 100000.00, company == $c)
```

In this example, the decision 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 Section 16.8.7, "Supported rule condition elements in DRL (keywords)".

**Use decision 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 decision engine event listeners when possible. If these listeners do not meet your requirements, use a system logging utility supported by the decision engine, such as Logback, Apache Commons Logging, or Apache Log4j.

For more information about supported decision engine event listeners and logging utilities, see Decision engine in Red Hat Decision Manager.

**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 decision 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 decision engine performance using `drools-metric`, add `drools-metric` to your project dependencies and enable trace logging for `org.drools.metric.util.MetricLogUtils`, as shown in the following example:

**Example project dependency for drools-metric**

```xml
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-metric</artifactId>
</dependency>
```
Example logback.xml configuration file

```xml
<configuration>
  <logger name="org.drools.metric.util.MetricLogUtils" level="trace"/>
  ...
</configuration>
```

Also, enable MetricLogUtils by setting the system property `drools.metric.logger.enabled` to `true`. Optionally, you can change the microseconds threshold of metric logging by setting the `drools.metric.logger.threshold` system property.

**NOTE**
Only node executions exceeding the threshold are logged. The default value is 500.

After you complete the configuration, rule execution produces logs as shown in the following example:

**Example rule execution output**

```plaintext
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@ee2a6922, 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.
- **elapsedMicro** is the elapsed time of the node execution in microseconds.

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**

```java
ReteDumper.dumpAssociatedRulesRete(kbase);
```

**Example ReteDumper output**

```plaintext
[ AccumulateNode(8) ] : [Collect expensive orders combination] ...
```
CHAPTER 23. NEXT STEPS

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

**NOTE**

You can also design your decision service using Decision Model and Notation (DMN) models instead of rule-based or table-based assets. For information about DMN support in Red Hat Decision Manager 7.10, see the following resources:

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

**Prerequisites**

- The space and project for the guided decision tables have been created in Business Central. Each asset is associated with a project assigned to a space. For details, see *Getting started with decision services*. 
CHAPTER 24. DECISION-AUTHORING ASSETS IN RED HAT DECISION MANAGER

Red Hat Decision Manager supports several assets that you can use to define business decisions for your decision service. Each decision-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights the main decision-authoring assets supported in Red Hat Decision Manager projects to help you decide or confirm the best method for defining decisions in your decision service.

Table 24.1. Decision-authoring assets supported in Red Hat Decision Manager

<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Decision Model and Notation (DMN) models | • Are decision models based on a notation standard defined by the Object Management Group (OMG)  
• Use graphical decision requirements diagrams (DRDs) that represent part or all of the overall decision requirements graph (DRG) to trace business decision flows  
• Use an XML schema that allows the DMN models to be shared between DMN-compliant platforms  
• Support Friendly Enough Expression Language (FEEL) to define decision logic in DMN decision tables and other DMN boxed expressions  
• Are optimal for creating comprehensive, illustrative, and stable decision flows | Business Central or other DMN-compliant editor | Designing a decision service using DMN models |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided decision tables   | - Are tables of rules that you create in a UI-based table designer in Business Central  
- Are a wizard-led alternative to spreadsheet decision tables  
- Provide fields and options for acceptable input  
- Support template keys and values for creating rule templates  
- Support hit policies, real-time validation, and other additional features not supported in other assets  
- Are optimal for creating rules in a controlled tabular format to minimize compilation errors | Business Central | Designing a decision service using guided decision tables |
| Spreadsheet decision tables | - Are XLS or XLSX spreadsheet decision tables that you can upload into Business Central  
- Support template keys and values for creating rule templates  
- Are optimal for creating rules in decision tables already managed outside of Business Central  
- Have strict syntax requirements for rules to be compiled properly when uploaded | Spreadsheet editor | Designing a decision service using spreadsheet decision tables |
| Guided rules             | - Are individual rules that you create in a UI-based rule designer in Business Central  
- Provide fields and options for acceptable input  
- Are optimal for creating single rules in a controlled format to minimize compilation errors | Business Central | Designing a decision service using guided rules          |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided rule templates         | • Are reusable rule structures that you create in a UI-based template designer in Business Central  
• Provide fields and options for acceptable input  
• Support template keys and values for creating rule templates (fundamental to the purpose of this asset)  
• Are optimal for creating many rules with the same rule structure but with different defined field values | Business Central                        | Designing a decision service using guided rule templates                   |
| DRL rules                     | • Are individual rules that you define directly in `.drl` text files  
• Provide the most flexibility for defining rules and other technicalities of rule behavior  
• Can be created in certain standalone environments and integrated with Red Hat Decision Manager  
• Are optimal for creating rules that require advanced DRL options  
• Have strict syntax requirements for rules to be compiled properly | Business Central or integrated development environment (IDE) | Designing a decision service using DRL rules                               |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictive Model Markup Language (PMML) models</td>
<td>• Are predictive data-analytic models based on a notation standard defined by the Data Mining Group (DMG)</td>
<td>PMML or XML editor</td>
<td>Designing a decision service using PMML models</td>
</tr>
<tr>
<td></td>
<td>• Use an XML schema that allows the PMML models to be shared between PMML-compliant platforms</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Support Regression, Scorecard, Tree, Mining, and other model types</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Can be included with a standalone Red Hat Decision Manager project or imported into a project in Business Central</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Are optimal for incorporating predictive data into decision services in Red Hat Decision Manager</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 25. GUIDED DECISION TABLES

Guided decision tables are a wizard-led alternative to spreadsheet decision tables for defining business rules in a tabular format. With guided decision tables, you are led by a UI-based wizard in Business Central that helps you define rule attributes, metadata, conditions, and actions based on specified data objects in your project. After you create your guided decision tables, the rules you defined are compiled into Drools Rule Language (DRL) rules as with all other rule assets.

All data objects related to a guided decision table must be in the same project package as the guided decision table. Assets in the same package are imported by default. After you create the necessary data objects and the guided decision table, you can use the Data Objects tab of the guided decision tables designer to verify that all required data objects are listed or to import other existing data objects by adding a New item.
CHAPTER 26. DATA OBJECTS

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

26.1. CREATING DATA OBJECTS

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

Procedure

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

   **IMPORTING DATA OBJECTS FROM OTHER PACKAGES**

   You can import an existing data object from another package directly into the asset designers like guided rules or guided decision table designers. Select the relevant rule asset within the project and in the asset designer, go to `Data Objects → New item` to select the object to be imported.

4. To make your data object persistable, select the `Persistable` checkbox. Persistable data objects are able to be stored in a database according to the JPA specification. The default JPA is Hibernate.
5. Click `Ok`.
6. In the data object designer, click `add field` to add a field to the object with the attributes `Id`, `Label`, and `Type`. Required attributes are marked with an asterisk (*).
   - **Id**: Enter the unique ID of the field.
   - **Label**: (Optional) Enter a label for the field.
   - **Type**: Enter the data type of the field.
   - **List**: (Optional) Select this check box to enable the field to hold multiple items for the specified type.
Figure 26.1. Add data fields to a data object

<table>
<thead>
<tr>
<th>New Field</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Id</strong></td>
</tr>
<tr>
<td><strong>Label</strong></td>
</tr>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><strong>List</strong></td>
</tr>
</tbody>
</table>

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

**NOTE**

To edit a field, select the field row and use the general properties on the right side of the screen.
You can use guided decision tables to define rule attributes, metadata, conditions, and actions in a tabular format that can be added to your business rules project.

Procedure

1. In Business Central, go to \textbf{Menu $\rightarrow$ Design $\rightarrow$ Projects} and click the project name.

2. Click \textbf{Add Asset $\rightarrow$ Guided Decision Table}.

3. Enter an informative \textbf{Guided Decision Table} name and select the appropriate \textbf{Package}. The package that you specify must be the same package where the required data objects have been assigned or will be assigned.

4. Select \textbf{Use Wizard} to finish setting up the table in the wizard, or leave this option unselected to finish creating the table and specify remaining configurations in the guided decision tables designer.

5. Select the hit policy that you want your rows of rules in the table to conform to. For details, see Chapter 28, \textit{Hit policies for guided decision tables}.

6. Specify whether you want the \textbf{Extended entry} or \textbf{Limited entry} table. For details, see Section 28.1.1, “Types of guided decision tables”.

7. Click \textbf{Ok} to complete the setup. If you have selected \textbf{Use Wizard}, the Guided Decision Table wizard is displayed. If you did not select the \textbf{Use Wizard} option, this prompt does not appear and you are taken directly to the table designer.
   For example, the following wizard setup is for a guided decision table in a loan application decision service:
8. If you are using the wizard, add any available imports, fact patterns, constraints, and actions, and select whether table columns should expand. Click **Finish** to close the wizard and view the table designer.

In the guided decision tables designer, you can add or edit columns and rows, and make other final adjustments.
CHAPTER 28. HIT POLICIES FOR GUIDED DECISION TABLES

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

The following hit policies are available:

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

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

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

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

- **Rule Order**: Multiple rows can be executed and verification does not report conflicts between the rows since they are expected to happen.
Figure 28.1. Available hit policies

28.1. HIT POLICY EXAMPLES: DECISION TABLE FOR DISCOUNTS ON MOVIE TICKETS

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

Table 28.1. Example decision table for available discounts on movie tickets

<table>
<thead>
<tr>
<th>Row Number</th>
<th>Discount Type</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Senior citizen (age 60+)</td>
<td>10%</td>
</tr>
<tr>
<td>2</td>
<td>Student</td>
<td>10%</td>
</tr>
<tr>
<td>3</td>
<td>Military</td>
<td>10%</td>
</tr>
</tbody>
</table>

In this example, the total discount to be applied in the end will vary depending on the hit policy specified for the table:

None

This is the normal hit mode. Old decision tables will use this by default, but since 7.0 uses PHREAK the row order now matters. There is no migration tooling needed for the old tables. Multiple rows can fire. Verification warns about rows that conflict.
- **None/Rule Order:** With both None and Rule Order hit policies, all applicable rules are incorporated, in this case allowing discounts to be stacked for each customer. Example: A senior citizen who is also a student and a military veteran will receive all three discounts, totaling 30%.

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

- **First Hit/Resolved Hit:** With both First Hit and Resolved Hit policies, only one of the discounts can be applied.

  For **First Hit**, the discount that is satisfied first in the list is applied and the others are ignored.

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

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

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

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

**Table 28.2. Modified decision table that accommodates a Resolved Hit policy**

<table>
<thead>
<tr>
<th>Row Number</th>
<th>Discount Type</th>
<th>Has Priority over Row</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Senior citizen (age 60+)</td>
<td></td>
<td>10%</td>
</tr>
<tr>
<td>2</td>
<td>Student</td>
<td></td>
<td>10%</td>
</tr>
<tr>
<td>3</td>
<td>Military</td>
<td>1</td>
<td>10%</td>
</tr>
</tbody>
</table>

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

**NOTE**

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

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

- **Unique Hit:** A modified table is required. With a Unique Hit policy, rows must be created in a way that it is impossible to satisfy multiple rules at one time. However, you can still specify row-by-row whether to apply one rule or multiple. In this way, with a Unique Hit policy you can make...
decision tables more granular and prevent overlap warnings. Consider the following modified decision table that accommodates a **Unique Hit** policy:

**Table 28.3. Modified decision table that accommodates a** **Unique Hit** **policy**

<table>
<thead>
<tr>
<th>Row Number</th>
<th>Is Senior Citizen (age 65+)</th>
<th>Is Student</th>
<th>Is Military</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>10%</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>10%</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>10%</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>20%</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>20%</td>
</tr>
<tr>
<td>6</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>20%</td>
</tr>
<tr>
<td>7</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>30%</td>
</tr>
</tbody>
</table>

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

### 28.1.1. Types of guided decision tables

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

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

<table>
<thead>
<tr>
<th>Pricing loans</th>
<th>#</th>
<th>Description</th>
<th>application : LoanApplication</th>
<th>fate : IncomeSource</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>amount min</td>
<td>amount max</td>
<td>period</td>
<td>deposit max</td>
<td>income</td>
</tr>
<tr>
<td>1</td>
<td>12,1000</td>
<td>20,000</td>
<td>30</td>
<td>20,000</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>10,000</td>
<td>10,000</td>
<td>20</td>
<td>20,000</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>10,000</td>
<td>13,000</td>
<td>20</td>
<td>30,000</td>
<td>true</td>
</tr>
</tbody>
</table>

- **Limited entry:** A Limited Entry decision table is one for which the column definitions specify value in addition to Pattern, Field, and Operator. The decision table states, held in the body of the table, are boolean where a positive value (a marked check box) has the effect of meaning the column should apply, or be matched. A negative value (a cleared check box) means the column does not apply.
<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>CR = AA</th>
<th>CR = OK</th>
<th>CR = Sub prime</th>
<th>Approve</th>
<th>Decline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
CHAPTER 29. ADDING COLUMNS TO GUIDED DECISION TABLES

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

Prerequisites

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

For descriptions of these column parameters, see the "Required column parameters" segments for each column type in Chapter 30, Types of columns in guided decision tables.

For details about creating data objects, see Section 26.1, "Creating data objects".

Procedure

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

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

3. Select the column type that you want to add, click Next, and follow the steps in the wizard to specify the data required to add the column.

   For descriptions of each column type and required parameters for setup, see Chapter 30, Types of columns in guided decision tables.

4. Click Finish to add the configured column.
After all columns are added, you can begin adding rows of rules correlating to your columns to complete the decision table. For details, see Chapter 33, *Adding rows and defining rules in guided decision tables*.

The following is an example decision table for a loan application decision service:

**Figure 29.2. Example of complete guided decision table**

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>application : LoanApplication</th>
<th>income : IncomeSource</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>amount min</td>
<td>amount max</td>
<td>period</td>
</tr>
<tr>
<td>1</td>
<td>121000</td>
<td>200000</td>
<td>20</td>
<td>20000</td>
</tr>
<tr>
<td>2</td>
<td>10000</td>
<td>100000</td>
<td>20</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>100000</td>
<td>130000</td>
<td>20</td>
<td>3000</td>
</tr>
</tbody>
</table>
CHAPTER 30. TYPES OF COLUMNS IN GUIDED DECISION TABLES

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

- Add a Condition
- Add a Condition BRL fragment
- Add a Metadata column
- Add an Action BRL fragment
- Add an Attribute column
- Delete an existing fact
- Execute a Work Item
- Set the value of a field
- Set the value of a field with a Work Item result

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

IMPORTANT: REQUIRED DATA OBJECTS FOR COLUMN PARAMETERS

Some of the column parameters described in this section, such as Fact Patterns and Fields, provide drop-down options consisting only of data objects already defined within the same package where the guided decision table is found. Available data objects for the package are listed in the Data Objects panel of the Project Explorer and in the Data Objects tab of the guided decision tables designer. You can create additional data objects within the package as needed, or import them from another package in Data Objects → New item of the guided decision tables designer. For details about creating data objects, see Section 26.1, “Creating data objects”.

30.1. "ADD A CONDITION"

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

Example rule conditions

```
when
  $i : IncomeSource( type == "Asset" ) // Binds the IncomeSource object to the $i variable
then
  ...
end
```
when not IncomeSource( type == "Asset" ) // Negates matching pattern
then
...
en
After a binding is specified, you can define field constraints. If two or more columns are defined using the same fact pattern binding, the field constraints become composite field constraints on the same pattern. If you define multiple bindings for a single model class, each binding becomes a separate model class in the condition ("WHEN") side of the rule.

Required column parameters

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

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

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

- **Calculation type**: Select one of the following calculation types:
  - **Literal value**: The value in the cell will be compared with the field using the operator.
  - **Formula**: The expression in the cell will be evaluated and then compared with the field.
  - **Predicate**: No field is needed; the expression will be evaluated to true or false.

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

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

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

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

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

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

- **Calculation type** for the condition column has been set to **Literal value**.
- **Operator** has been set as equality `==` or inequality `!=`.

The *any other* value enables a rule to be defined for any other field values not explicitly defined in the rules already in the table. In the DRL source, *any other* is notated as `not in`.

**Example rule condition with not in used for any other**

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

then
    ...

end
```

**Procedure**

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

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

30.2. "ADD A CONDITION BRL FRAGMENT"

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

The following example is a condition BRL fragment for a loan application:
You can also select Free form DRL from the list of condition options to define the condition BRL fragment without the embedded guided rules designer.
Figure 30.2. Add a condition BRL fragment with free form DRL

A Business Rule Language (BRL) fragment is a section of a rule created using the Guided Rule Editor. The condition BRL fragment is the "WHEN" portion of the rule, and the Action BRL fragment is the "THEN" portion of the rule. With this column option, you can define a Condition BRL fragment to be used in the left ("WHEN") side of a rule. In the embedded Guided Rule Editor, field values defined as "Template Keys" form columns in the decision table. Simpler column types can refer to Facts and Fact fields bound in the BRL fragment and vice-versa.
TEMPLATE KEYS

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

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

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

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

Required column parameters

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

- **Rule Modeller**: Define the condition BRL fragment ("WHEN" portion) for the rule.
- **Header (description)**: Add header text for the column.
- **Hide column**: Select this to hide the column, or clear this to display the column.

30.3. "ADD A METADATA COLUMN"

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

Required column parameter

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

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

30.4. "ADD AN ACTION BRL FRAGMENT"

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

The following example is an action BRL fragment for a loan application:
You can also select **Add free form DRL** from the list of action options to define the action BRL fragment without the embedded guided rules designer.
Figure 30.4. Add an action BRL fragment with free form DRL
When you add a field for an action BRL fragment, one of the value options is **Template key** (as opposed to **Literal** or **Formula**). Template keys are placeholder variables that are interchanged with a specified value when the guided decision table is generated, and form separate columns in the table for each template key value specified. You can specify the default value of Template key in the **Value options** page. While Literal and Formula values are static in a decision table, Template key values can be modified as needed.

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

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

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

**Required column parameters**

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

- **Rule Modeller**: Define the action BRL fragment (“THEN” portion) for the rule.
- **Header (description)**: Add header text for the column.
- **Hide column**: Select this to hide the column, or clear this to display the column.

30.5. "ADD AN ATTRIBUTE COLUMN"

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

For example, the following guided decision table uses the **salience** attribute to specify rule priority and the **enabled** attribute to enable or disable rules for evaluation. Rules with a higher **salience** value are evaluated first, and rules with the **enabled** attribute are evaluated only if the check box is selected.

*Figure 30.5. Example rules with salience and enabled attributes to define evaluation behavior*

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>salience</th>
<th>enabled</th>
<th>amount min</th>
<th>amount max</th>
<th>period</th>
<th>deposit max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>100</td>
<td>✔</td>
<td>131000</td>
<td>200000</td>
<td>30</td>
<td>20000</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>✔</td>
<td>10000</td>
<td>100000</td>
<td>20</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>100000</td>
<td>130000</td>
<td>20</td>
<td>3000</td>
</tr>
</tbody>
</table>

*Example rule source with the rule attributes*

```
rule "Row 1 Pricing loans"
salience 100
enabled true
when
... then
```
... end
...

rule "Row 3 Pricing loans"
    enabled false
    when
        ...
        then
        ...
    end

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

HIT POLICIES AND ATTRIBUTES

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

Required Column Parameter

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

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

30.6. "DELETE AN EXISTING FACT"

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

Required column parameters

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

- **Header (description)**: Add header text for the column.
- **Hide column**: Select this to hide the column, or clear this to display the column.

30.7. "EXECUTE A WORK ITEM"

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

Required column parameters

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

- **Work Item**: Select from the list of your predefined work items.
- **Header (description)**: Add header text for the column.
• **Hide column:** Select this to hide the column, or clear this to display the column.

### 30.8. "SET THE VALUE OF A FIELD"

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

#### Required column parameters

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

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

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

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

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

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

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

- **Logically insert:** This option appears when the selected Fact Pattern is not currently used in another column in the guided decision table (see the next field description). Select this to insert the fact pattern logically into the decision engine, or clear this to insert it regularly. The decision engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts have to be retracted explicitly. After logical insertions, facts are automatically retracted when the conditions that asserted the facts in the first place are no longer true.

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

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

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

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

Required column parameters

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

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

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

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

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

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

- **Logically insert**: This option appears when the selected Fact Pattern is not currently used in another column in the guided decision table (see the next field description). Select this to insert the fact pattern logically into the decision engine, or clear this to insert it regularly. The decision engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts have to be retracted explicitly. After logical insertions, facts are automatically retracted when the conditions that asserted the facts in the first place are no longer true.

- **Update engine with changes**: This option appears when the selected Fact Pattern is already used in another column in the guided decision table. Select this to update the decision engine with the modified field values, or clear this to not update the decision engine.
CHAPTER 31. VIEWING RULE NAME COLUMN IN GUIDED DECISION TABLES

You can view the Rule Name column in the guided decision table if needed.

Procedure

1. In the guided decision tables designer, click Columns.
2. Select the Show rule name column check box.
3. Click Finish to save.

The default rule name format is Row (row_number)(table_name). The Source contains the default value if you do not specify a rule name. In the guided decision table, you can add a rule name in the Rule Name column and override the default value.
CHAPTER 32. EDITING OR DELETING COLUMNS IN GUIDED DECISION TABLES

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

Procedure

1. In the guided decision tables designer, click **Columns**.
2. Expand the appropriate section and click **Edit** or **Delete** next to the column name.

![Figure 32.1. Edit or delete columns](image)

**NOTE**

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

3. After any column changes, click **Finish** in the wizard to save.
CHAPTER 33. ADDING ROWS AND DEFINING RULES IN GUIDED DECISION TABLES

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

Prerequisites

- Columns for the guided decision table have been added as described in Chapter 29, Adding columns to guided decision tables.

Procedure

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

   ![](Figure 33.1. Add Rows)

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

   ![](Figure 33.2. Enter input data in each cell)

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

   ![NOTE]

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

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

   After you define your guided decision table contents, in the upper-right corner of the guided decision tables designer, you can use the search bar if needed to search for text that appears in your guided decision table. The search feature is especially helpful in complex guided decision tables with many values:
## Figure 33.3. Search guided decision table contents

<table>
<thead>
<tr>
<th>IncomeSource</th>
<th>application</th>
<th>deposit max</th>
<th>Loan approved</th>
<th>LMI</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asset</td>
<td>true</td>
<td>20000</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Job</td>
<td>true</td>
<td>2000</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Job</td>
<td>true</td>
<td>3000</td>
<td>10</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 34. DEFINING ENUMERATIONS FOR DROP-DOWN LISTS IN RULE ASSETS

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

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

Procedure

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

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

**Figure 34.1. Example enumeration for applicant credit rating in Business Central**

<table>
<thead>
<tr>
<th>Model</th>
<th>Overview</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add enum</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fact</th>
<th>Field</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant</td>
<td>creditRating</td>
<td>['AA', 'OK', 'Sub prime']</td>
</tr>
</tbody>
</table>

**Example enumeration for applicant credit rating in the DRL source**

394
In this example, for any guided rule, guided rule template, or guided decision table that is in the same package of the project and that uses the Applicant data object and the creditRating field, the configured values are available as drop-down options:

Figure 34.2. Example enumeration drop-down options in a guided rule or guided rule template

Figure 34.3. Example enumeration drop-down options in a guided decision table

34.1. ADVANCED ENUMERATION OPTIONS FOR RULE ASSETS

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

Mapping between DRL values and values in Business Central

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

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

Enumeration value dependencies

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

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

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

External data sources in enumerations

If you want to retrieve a list of enumeration values from an external data source instead of defining the values directly in the enumeration definition, on the class path of your project, add a helper class that returns a `java.util.List` list of strings. In the enumeration definition, instead of specifying a list of values, identify the helper class that you configured to retrieve the values externally.

For example, in the following enumeration definition for loan applicant region, instead of defining applicant regions explicitly in the format `'Applicant.region' : ['country1', 'country2', ... ]`, the enumeration uses a helper class that returns the list of values defined externally:

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

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

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

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

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

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

```java
public class SampleDataSource {

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

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

        return data;
    }
}
```

The following enumeration definition correlates to this example Java class. The enumeration contains no references to fact or field names because they are defined in the Java class:
The `=` operator enables Business Central to load all enumeration data from the helper class. The helper methods are statically evaluated when the enumeration definition is requested for use in an editor.

**NOTE**

Defining an enumeration without a fact and field definition is currently not supported in Business Central. To define the enumeration for the associated Java class in this way, use the DRL source in your Red Hat Decision Manager project.
CHAPTER 35. REAL-TIME VERIFICATION AND VALIDATION OF GUIDED DECISION TABLES

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

35.1. TYPES OF PROBLEMS IN GUIDED DECISION TABLES

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

Redundancy

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

Subsumption

Subsumption is similar to redundancy and occurs when two rules execute the same consequences, but one executes on a subset of facts of the other. For example, consider these two rules:

- \( \text{when Person age} > 10 \text{ then Increase Counter} \)
- \( \text{when Person age} > 20 \text{ then Increase Counter} \)

In this case, if a person is 15 years old, only one rule fires and if a person is 20 years old, both rules fire. Such cases cause similar trouble during runtime as redundancy.

Conflicts

A conflicting situation occurs when two similar conditions have different consequences. Conflicts can occur between two rows (rules) or two cells in a decision table.

The following example illustrates conflict between two rows in a decision table:

- \( \text{when Deposit} > 20000 \text{ then Approve Loan} \)
- \( \text{when Deposit} > 20000 \text{ then Refuse Loan} \)

In this case, there is no way to know if the loan will be approved or not.

The following example illustrates conflict between two cells in a decision table:

- \( \text{when Age} > 25 \)
- \( \text{when Age} < 25 \)

A row with conflicting cells never executes.

Broken Unique Hit Policy

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

- \( \text{when Is Student} = \text{true} \)
when Is Military = true

If a customer is both a student and in the military, both conditions apply and break the **Unique Hit** policy. Rows in this type of table must therefore be created in a way that does not allow multiple rules to fire at one time. For details about hit policies, see *Chapter 28, Hit policies for guided decision tables*.

**Deficiency**

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

- when Age > 20 then Approve Loan
- when Deposit < 20000 then Refuse Loan

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

**Missing Columns**

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

**Incomplete Ranges**

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

### 35.2. TYPES OF NOTIFICATIONS

The verification and validation feature uses three types of notifications:

- **>Error:** A serious problem that may lead to the guided decision table failing to work as designed at run time. Conflicts, for example, are reported as errors.
- **>Warning:** Likely a serious problem that may not prevent the guided decision table from working but requires attention. Subsumptions, for example, are reported as warnings.
- **>Information:** A moderate or minor problem that may not prevent the guided decision table from working but requires attention. Missing columns, for example, are reported as information.

**NOTE**

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

### 35.3. DISABLING VERIFICATION AND VALIDATION OF GUIDED DECISION TABLES

The decision table verification and validation feature of Business Central is enabled by default. This feature helps you validate your guided decision tables, but with complex guided decision tables, this
feature can hinder decision engine performance. You can disable this feature by setting the `org.kie.verification.disable-dtable-realtime-verification` system property value to `true` in your Red Hat Decision Manager distribution.

Procedure

Navigate to `~/standalone-full.xml` and add the following system property:

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

For example, on Red Hat JBoss EAP, you add this system property in `$EAP_HOME/standalone/configuration/standalone-full.xml`. 
CHAPTER 36. CONVERTING A GUIDED DECISION TABLE TO A SPREADSHEET DECISION TABLE

After you define a guided decision table in Business Central, you can convert the guided decision table to an XLS spreadsheet decision table file for offline reference and file sharing. The guided decision table must be an extended entry guided decision table in order to be converted. The conversion tool does not support limited entry guided decision tables.

For more information about spreadsheet decision tables, see Designing a decision service using spreadsheet decision tables.

WARNING

Guided decision tables and spreadsheet decision tables are different table formats that support different features. Any supported features that differ between the two decision table formats (Hit policy, for example) are modified or lost when you convert one decision table format to the other.

Procedure

In Business Central, navigate to the guided decision table asset that you want to convert and in the upper-right toolbar of the decision tables designer, click Convert to XLS:

Figure 36.1. Convert an uploaded decision table

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>application</th>
<th>Income</th>
<th>Loan approved</th>
<th>LMI</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>131000</td>
<td>200000</td>
<td>30</td>
<td>20000</td>
<td>Asset</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>100000</td>
<td>100000</td>
<td>20</td>
<td>20000</td>
<td>Job</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>100001</td>
<td>130000</td>
<td>20</td>
<td>30000</td>
<td>Job</td>
<td>true</td>
</tr>
</tbody>
</table>

After the conversion, the converted decision table is then available as a spreadsheet decision table asset in your project that you can download for offline reference.
CHAPTER 37. EXECUTING RULES

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

Prerequisites

- Business Central and KIE Server are installed and running. For installation options, see Planning a Red Hat Decision Manager installation.

Procedure

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

2. In the upper-right corner of the project Assets page, click Deploy to build the project and deploy it to KIE Server. If the build fails, address any problems described in the Alerts panel at the bottom of the screen.
   For more information about project deployment options, see Packaging and deploying a Red Hat Decision Manager project.

   NOTE

   If the rule assets in your project are not built from an executable rule model by default, verify that the following dependency is in the pom.xml file of your project and rebuild the project:

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

   This dependency is required for rule assets in Red Hat Decision Manager to be built from executable rule models by default. This dependency is included as part of the Red Hat Decision Manager core packaging, but depending on your Red Hat Decision Manager upgrade history, you may need to manually add this dependency to enable the executable rule model behavior.

   For more information about executable rule models, see Packaging and deploying a Red Hat Decision Manager project.

3. Create a Maven or Java project outside of Business Central, if not created already, that you can use for executing rules locally or that you can use as a client application for executing rules on KIE Server. The project must contain a pom.xml file and any other required components for executing the project resources.
   For example test projects, see “Other methods for creating and executing DRL rules”.

4. Open the pom.xml file of your test project or client application and add the following dependencies, if not added already:

   - kie-ci: Enables your client application to load Business Central project data locally using Releaseld

Red Hat Decision Manager 7.10 Developing decision services in Red Hat Decision Manager
• **kie-server-client**: Enables your client application to interact remotely with assets on KIE Server

• **slf4j**: (Optional) Enables your client application to use Simple Logging Facade for Java (SLF4J) to return debug logging information after you interact with KIE Server

Example dependencies for Red Hat Decision Manager 7.10 in a client application **pom.xml** file:

```xml
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.48.0.Final-redhat-00004</version>
</dependency>

<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.48.0.Final-redhat-00004</version>
</dependency>

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.25</version>
</dependency>
```

For available versions of these artifacts, search the group ID and artifact ID in the **Nexus Repository Manager** online.

**NOTE**

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

Example BOM dependency:

```xml
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.10.0.redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see [What is the mapping between Red Hat Decision Manager and the Maven library version?](#).
5. Ensure that the dependencies for artifacts containing model classes are defined in the client application `pom.xml` file exactly as they appear in the `pom.xml` file of the deployed project. If dependencies for model classes differ between the client application and your projects, execution errors can occur.

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

For example, the following `Person` class dependency appears in both the client and deployed project `pom.xml` files:

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

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

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

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

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

```java
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
when
  Person(hourlyRate * wage > 100)
  Person(name : firstName, surname : lastName)
then
  System.out.println("Hello" + " " + name + " " + surname + "!");
  System.out.println("You are rich!");
end
```

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

**Executing rules locally**

```java
import org.kie.api.KieServices;
import org.kie.api.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;
```
public class RulesTest {

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

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

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

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

// Fire all rules:
        kSession.fireAllRules();
        kSession.dispose();
    } catch (Throwable t) {
        t.printStackTrace();
    }
}
}

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

### Executing rules on KIE Server

package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;
import com.sample.Person;

public class RulesTest {

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

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

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

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

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

            // Use rule services client to send request:
            RuleServicesClient ruleClient =
                kieServicesClient.getServicesClient(RuleServicesClient.class);
            ServiceResponse<ExecutionResults> executeResponse =
                ruleClient.executeCommandsWithResults(containerName, batch);
            System.out.println("number of fired rules:" +
                executeResponse.getResult().getValue("numberOfFiredRules"));
        }
    }
}
8. Run the configured .java class from your project directory. You can run the file in your development platform (such as Red Hat CodeReady Studio) or in the command line. Example Maven execution (within project directory):

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

Example Java execution (within project directory)

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

9. Review the rule execution status in the command line and in the server log. If any rules do not execute as expected, review the configured rules in the project and the main class configuration to validate the data provided.
CHAPTER 38. NEXT STEPS

- Testing a decision service using test scenarios
- Packaging and deploying a Red Hat Decision Manager project
PART V. DESIGNING A DECISION SERVICE USING SPREADSHEET DECISION TABLES

As a business analyst or business rules developer, you can define business rules in a tabular format in spreadsheet decision tables and then upload the spreadsheets to your project in Business Central. These rules are compiled into Drools Rule Language (DRL) and form the core of the decision service for your project.

NOTE

You can also design your decision service using Decision Model and Notation (DMN) models instead of rule-based or table-based assets. For information about DMN support in Red Hat Decision Manager 7.10, see the following resources:

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

Prerequisites

- The space and project for the decision tables have been created in Business Central. Each asset is associated with a project assigned to a space. For details, see *Getting started with decision services*. 
CHAPTER 39. DECISION-AUTHORING ASSETS IN RED HAT DECISION MANAGER

Red Hat Decision Manager supports several assets that you can use to define business decisions for your decision service. Each decision-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights the main decision-authoring assets supported in Red Hat Decision Manager projects to help you decide or confirm the best method for defining decisions in your decision service.

Table 39.1. Decision-authoring assets supported in Red Hat Decision Manager

<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision Model and Notation (DMN) models</td>
<td>- Are decision models based on a notation standard defined by the Object Management Group (OMG)</td>
<td>Business Central or other DMN-compliant editor</td>
<td>Designing a decision service using DMN models</td>
</tr>
<tr>
<td></td>
<td>- Use graphical decision requirements diagrams (DRDs) that represent part or all of the overall decision requirements graph (DRG) to trace business decision flows</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Use an XML schema that allows the DMN models to be shared between DMN-compliant platforms</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Support Friendly Enough Expression Language (FEEL) to define decision logic in DMN decision tables and other DMN boxed expressions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Are optimal for creating comprehensive, illustrative, and stable decision flows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asset</td>
<td>Highlights</td>
<td>Authoring tools</td>
<td>Documentation</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Guided decision tables   | - Are tables of rules that you create in a UI-based table designer in Business Central  
- Are a wizard-led alternative to spreadsheet decision tables  
- Provide fields and options for acceptable input  
- Support template keys and values for creating rule templates  
- Support hit policies, real-time validation, and other additional features not supported in other assets  
- Are optimal for creating rules in a controlled tabular format to minimize compilation errors | Business Central | Designing a decision service using guided decision tables |
| Spreadsheet decision tables | - Are XLS or XLSX spreadsheet decision tables that you can upload into Business Central  
- Support template keys and values for creating rule templates  
- Are optimal for creating rules in decision tables already managed outside of Business Central  
- Have strict syntax requirements for rules to be compiled properly when uploaded | Spreadsheet editor | Designing a decision service using spreadsheet decision tables |
| Guided rules             | - Are individual rules that you create in a UI-based rule designer in Business Central  
- Provide fields and options for acceptable input  
- Are optimal for creating single rules in a controlled format to minimize compilation errors | Business Central | Designing a decision service using guided rules |
### Guided rule templates

- Are reusable rule structures that you create in a UI-based template designer in Business Central
- Provide fields and options for acceptable input
- Support template keys and values for creating rule templates (fundamental to the purpose of this asset)
- Are optimal for creating many rules with the same rule structure but with different defined field values

**Authoring tools**: Business Central

**Documentation**: Designing a decision service using guided rule templates

### DRL rules

- Are individual rules that you define directly in `.drl` text files
- Provide the most flexibility for defining rules and other technicalities of rule behavior
- Can be created in certain standalone environments and integrated with Red Hat Decision Manager
- Are optimal for creating rules that require advanced DRL options
- Have strict syntax requirements for rules to be compiled properly

**Authoring tools**: Business Central or integrated development environment (IDE)

**Documentation**: Designing a decision service using DRL rules
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Predictive Model Markup Language (PMML) models | - Are predictive data-analytic models based on a notation standard defined by the Data Mining Group (DMG)  
- Use an XML schema that allows the PMML models to be shared between PMML-compliant platforms  
- Support Regression, Scorecard, Tree, Mining, and other model types  
- Can be included with a standalone Red Hat Decision Manager project or imported into a project in Business Central  
- Are optimal for incorporating predictive data into decision services in Red Hat Decision Manager | PMML or XML editor | Designing a decision service using PMML models |
CHAPTER 40. SPREADSHEET DECISION TABLES

Spreadsheet decision tables are XLS or XLSX spreadsheets that contain business rules defined in a tabular format. You can include spreadsheet decision tables with standalone Red Hat Decision Manager projects or upload them to projects in Business Central. Each row in a decision table is a rule, and each column is a condition, an action, or another rule attribute. After you create and upload your spreadsheet decision tables, the rules you defined are compiled into Drools Rule Language (DRL) rules as with all other rule assets.

All data objects related to a spreadsheet decision table must be in the same project package as the spreadsheet decision table. Assets in the same package are imported by default. Existing assets in other packages can be imported with the decision table.
CHAPTER 41. DATA OBJECTS

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

41.1. CREATING DATA OBJECTS

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

Procedure

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

   IMPORING DATA OBJECTS FROM OTHER PACKAGES

   You can import an existing data object from another package directly into the asset designers like guided rules or guided decision table designers. Select the relevant rule asset within the project and in the asset designer, go to Data Objects → New item to select the object to be imported.

4. To make your data object persistable, select the Persistable checkbox. Persistable data objects are able to be stored in a database according to the JPA specification. The default JPA is Hibernate.
5. Click Ok.
6. In the data object designer, click add field to add a field to the object with the attributes Id, Label, and Type. Required attributes are marked with an asterisk (*).
   - Id: Enter the unique ID of the field.
   - Label: (Optional) Enter a label for the field.
   - Type: Enter the data type of the field.
   - List: (Optional) Select this check box to enable the field to hold multiple items for the specified type.
Figure 41.1. Add data fields to a data object

<table>
<thead>
<tr>
<th>New Field</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Id</strong></td>
<td>salary</td>
</tr>
<tr>
<td><strong>Label</strong></td>
<td>Salary</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>BigInteger</td>
</tr>
<tr>
<td><strong>List</strong></td>
<td></td>
</tr>
</tbody>
</table>

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

**NOTE**

To edit a field, select the field row and use the general properties on the right side of the screen.
An online shopping site lists the shipping charges for ordered items. The site provides free shipping under the following conditions:

- The number of items ordered is 4 or more and the checkout total is $300 or more.
- Standard shipping is selected (4 or 5 business days from the date of purchase).

The following are the shipping rates under these conditions:

**Table 42.1. For orders less than $300**

<table>
<thead>
<tr>
<th>Number of items</th>
<th>Delivery day</th>
<th>Shipping charge in USD, N = Number of items</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 or fewer</td>
<td>Next day</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>2nd day</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>10</td>
</tr>
<tr>
<td>4 or more</td>
<td>Next day</td>
<td>N*7.50</td>
</tr>
<tr>
<td></td>
<td>2nd day</td>
<td>N*3.50</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>N*2.50</td>
</tr>
</tbody>
</table>

**Table 42.2. For orders more than $300**

<table>
<thead>
<tr>
<th>Number of items</th>
<th>Delivery day</th>
<th>Shipping charge in USD, N = Number of items</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 or fewer</td>
<td>Next day</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>2nd day</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>N*1.50</td>
</tr>
<tr>
<td>4 or more</td>
<td>Next day</td>
<td>N*5</td>
</tr>
<tr>
<td></td>
<td>2nd day</td>
<td>N*2</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>FREE</td>
</tr>
</tbody>
</table>

These conditions and rates are shown in the following example spreadsheet decision table:
Figure 42.1. Decision table for shipping charges

<table>
<thead>
<tr>
<th></th>
<th>RuleSet</th>
<th>Charge Calculator</th>
<th>Condition</th>
<th>Condition</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rule Table</td>
<td>Basic</td>
<td>$order: Order</td>
<td>itemCount &lt;= $1</td>
<td>deliveredDays = $1</td>
<td>Insert(new Charge($1));</td>
</tr>
<tr>
<td>2</td>
<td>import</td>
<td>guvmorph.feature.databases.Order</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Variables</td>
<td>Integer totalCount</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SequentialMaxPriority</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>Min items</td>
<td>Max items</td>
<td>Delivered in days</td>
<td>Pay charge in US dollars</td>
</tr>
<tr>
<td>12</td>
<td>expensive</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td>$order.getItemsCount() * 7.5</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td>$order.getItemsCount() * 3.5</td>
</tr>
<tr>
<td>17</td>
<td>cheap</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$order.getItemsCount() * 2.5</td>
</tr>
<tr>
<td>19</td>
<td>Rule Table</td>
<td>Expensive</td>
<td>$order: Order</td>
<td>itemCount &lt;= $1</td>
<td>deliveredDays = $1</td>
<td>insert(new Charge($1));</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td>Min items</td>
<td>Max items</td>
<td>Delivered in days</td>
<td>More expensive than</td>
</tr>
<tr>
<td>22</td>
<td>expensive</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>300</td>
<td>25</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>300</td>
<td>15</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>300</td>
<td>$order.getItemsCount() * 1.5</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>4</td>
<td>1</td>
<td>300</td>
<td>$order.getItemsCount() * 5</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>4</td>
<td>2</td>
<td>300</td>
<td>$order.getItemsCount() * 2</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>cheap</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In order for a decision table to be uploaded in Business Central, the table must comply with certain structure and syntax requirements, within an XLS or XLSX spreadsheet, as shown in this example. For more information, see Chapter 43, Defining spreadsheet decision tables.
CHAPTER 43. DEFINING SPREADSHEET DECISION TABLES

Spreadsheet decision tables (XLS or XLSX) require two key areas that define rule data: a RuleSet area and a RuleTable area. The RuleSet area of the spreadsheet defines elements that you want to apply globally to all rules in the same package (not only the spreadsheet), such as a rule set name or universal rule attributes. The RuleTable area defines the actual rules (rows) and the conditions, actions, and other rule attributes (columns) that constitute that rule table within the specified rule set. A spreadsheet of decision tables can contain multiple RuleTable areas, but only one RuleSet area.

**IMPORTANT**

You should typically upload only one spreadsheet of decision tables, containing all necessary RuleTable definitions, per rule package in Business Central. You can upload separate decision table spreadsheets for separate packages, but uploading multiple spreadsheets in the same package can cause compilation errors from conflicting RuleSet or RuleTable attributes and is therefore not recommended.

Refer to the following sample spreadsheet as you define your decision table:

**Figure 43.1. Sample spreadsheet decision table for shipping charges**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RuleSet</td>
<td>Charge Calculator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Import</td>
<td>guroro.feature.diables.Order, guroro.feature.diables.Charge</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Variables</td>
<td>Integer totalCount</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SequentialMaxPriority</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>RuleTable Basic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>DESCRIPTION</td>
<td>CONDITION</td>
<td>CONDITION</td>
<td>CONDITION</td>
<td>ACTION</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>$order : Order</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>itemCount &gt; $1</td>
<td>itemCount &lt;= $1</td>
<td>deliveryDays &lt;= $1</td>
<td>insert(new Charge($1));</td>
</tr>
<tr>
<td>11</td>
<td>Min items</td>
<td>Max items</td>
<td>Delivered in days</td>
<td>Pay charge in US dollars</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>expensive</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>35</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>0</td>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>4</td>
<td>1</td>
<td>borderAdorners.Count() * 7.5</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>4</td>
<td>2</td>
<td>borderAdorners.Count() * 3.5</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>4</td>
<td>2.5</td>
<td>borderAdorners.Count()</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>cheap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>RuleTable Expensive</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>DESCRIPTION</td>
<td>CONDITION</td>
<td>CONDITION</td>
<td>CONDITION</td>
<td>ACTION</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>$order : Order</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>itemCount &gt; $1</td>
<td>itemCount &lt;= $1</td>
<td>deliveryDays &lt;= $1</td>
<td>totalPrice &gt; $1</td>
</tr>
<tr>
<td>23</td>
<td>Min items</td>
<td>Max items</td>
<td>Delivered in days</td>
<td>More expensive than</td>
<td>Pay charge in US dollars</td>
</tr>
<tr>
<td>24</td>
<td>expensive</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>300</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>300</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>0</td>
<td>3</td>
<td>300</td>
<td>1.5</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>4</td>
<td>1</td>
<td>300</td>
<td>5</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>4</td>
<td>2</td>
<td>300</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>cheap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Procedure**

1. In a new XLS or XLSX spreadsheet, go to the second or third column and label a cell RuleSet (row 1 in example). Reserve the column or columns to the left for descriptive metadata (optional).
2. In the next cell to the right, enter a name for the **RuleSet**. This named rule set will contain all **RuleTable** rules defined in the rule package.

3. Under the **RuleSet** cell, define any rule attributes (one per cell) that you want to apply globally to all rule tables in the package. Specify attribute values in the cells to the right. For example, you can enter an **Import** label and in the cell to the right, specify relevant data objects from other packages that you want to import into the package for the decision table (in the format `package.name.object.name`). For supported cell labels and values, see [Section 43.1, “RuleSet definitions”](#).

4. Below the **RuleSet** area and in the same column as the **RuleSet** cell, skip a row and label a new cell **RuleTable** (row 7 in example) and enter a table name in the same cell. The name is used as the initial part of the name for all rules derived from this rule table, with the row number appended for distinction. You can override this automatic naming by inserting a **NAME** attribute column.

5. Use the next four rows to define the following elements as needed (rows 8-11 in example):
   - **Rule attributes**: Conditions, actions, or other attributes. For supported cell labels and values, see [Section 43.2, “RuleTable definitions”](#).
   - **Object types**: The data objects to which the rule attributes apply. If the same object type applies to multiple columns, merge the object cells into one cell across multiple columns (as shown in the sample decision table), instead of repeating the object type in multiple cells. When an object type is merged, all columns below the merged range will be combined into one set of constraints within a single pattern for matching a single fact at a time. When an object is repeated in separate columns, the separate columns can create different patterns, potentially matching different or identical facts.
   - **Constraints**: Constraints on the object types.
   - **Column label**: (Optional) Any descriptive label for the column, as a visual aid. Leave blank if unused.

   **NOTE**
   As an alternative to populating both the object type and constraint cells, you can leave the object type cell or cells empty and enter the full expression in the corresponding constraint cell or cells. For example, instead of **Order** as the object type and `itemsCount > $1` as a constraint (separate cells), you can leave the object type cell empty and enter `Order( itemsCount > $1 )` in the constraint cell, and then do the same for other constraint cells.

6. After you have defined all necessary rule attributes (columns), enter values for each column as needed, row by row, to generate rules (rows 12-17 in example). Cells with no data are ignored (such as when a condition or action does not apply).
   If you need to add more rule tables to this decision table spreadsheet, skip a row after the last rule in the previous table, label another **RuleTable** cell in the same column as the previous **RuleTable** and **RuleSet** cells, and create the new table following the same steps in this section (rows 19-29 in example).

7. Save your XLS or XLSX spreadsheet to finish.
NOTE

By default, only the first worksheet in a spreadsheet workbook is processed as a decision table when you upload the spreadsheet in Business Central. Each RuleSet name combined with the RuleTable name must be unique across all decision table files in the same package.

If you want to process multiple worksheet decision tables, then create a .properties file with the same name as the spreadsheet workbook. The .properties file must contain a property sheet with comma-separated values (CSV) for the names of the worksheets, for example:

```
sheets=Sheet1,Sheet2
```

After you upload the decision table in Business Central, the rules are rendered as DRL rules like the following example, from the sample spreadsheet:

```drl
//row 12
rule "Basic_12"
salience 10
when
  $order : Order( itemsCount > 0, itemsCount <= 3, deliverInDays == 1 )
then
  insert( new Charge( 35 ) );
end
```

ENABLING WHITE SPACE USED IN CELL VALUES

By default, any white space before or after values in decision table cells is removed before the decision table is processed by the decision engine. To retain white space that you use intentionally before or after values in cells, set the drools.trimCellsInDTable system property to false in your Red Hat Decision Manager distribution.

For example, if you use Red Hat Decision Manager with Red Hat JBoss EAP, add the following system property to your $EAP_HOME/standalone/configuration/standalone-full.xml file:

```
<property name="drools.trimCellsInDTable" value="false"/>
```

If you use the decision engine embedded in your Java application, add the system property with the following command:

```
java -jar yourApplication.jar -Ddrools.trimCellsInDTable=false
```

43.1. RULESET DEFINITIONS

Entries in the RuleSet area of a decision table define DRL constructs and rule attributes that you want to apply to all rules in a package (not only in the spreadsheet). Entries must be in a vertically stacked sequence of cell pairs, where the first cell contains a label and the cell to the right contains the value. A decision table spreadsheet can have only one RuleSet area.

The following table lists the supported labels and values for RuleSet definitions:
Table 43.1. Supported RuleSet definitions

<table>
<thead>
<tr>
<th>Label</th>
<th>Value</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>RuleSet</td>
<td>The package name for the generated DRL file. Optional, the default is rule_table.</td>
<td>Must be the first entry.</td>
</tr>
<tr>
<td>Sequential</td>
<td>true or false. If true, then salience is used to ensure that rules fire from the top down.</td>
<td>Optional, at most once. If omitted, no firing order is imposed.</td>
</tr>
<tr>
<td>SequentialMaxPriority</td>
<td>Integer numeric value</td>
<td>Optional, at most once. In sequential mode, this option is used to set the start value of the salience. If omitted, the default value is 65535.</td>
</tr>
<tr>
<td>SequentialMinPriority</td>
<td>Integer numeric value</td>
<td>Optional, at most once. In sequential mode, this option is used to check if this minimum salience value is not violated. If omitted, the default value is 0.</td>
</tr>
<tr>
<td>EscapeQuotes</td>
<td>true or false. If true, then quotation marks are escaped so that they appear literally in the DRL.</td>
<td>Optional, at most once. If omitted, quotation marks are escaped.</td>
</tr>
<tr>
<td>Import</td>
<td>A comma-separated list of Java classes to import from another package.</td>
<td>Optional, may be used repeatedly.</td>
</tr>
<tr>
<td>Variables</td>
<td>Declarations of DRL globals (a type followed by a variable name). Multiple global definitions must be separated by commas.</td>
<td>Optional, may be used repeatedly.</td>
</tr>
<tr>
<td>Functions</td>
<td>One or more function definitions, according to DRL syntax.</td>
<td>Optional, may be used repeatedly.</td>
</tr>
<tr>
<td>Queries</td>
<td>One or more query definitions, according to DRL syntax.</td>
<td>Optional, may be used repeatedly.</td>
</tr>
<tr>
<td>Declare</td>
<td>One or more declarative types, according to DRL syntax.</td>
<td>Optional, may be used repeatedly.</td>
</tr>
<tr>
<td>Unit</td>
<td>The rule units that the rules generated from this decision table belong to.</td>
<td>Optional, at most once. If omitted, the rules do not belong to any unit.</td>
</tr>
</tbody>
</table>
In some cases, Microsoft Office, LibreOffice, and OpenOffice might encode a double quotation mark differently, causing a compilation error. For example, “A” will fail, but "A" will pass.

### 43.2. RULETABLE DEFINITIONS

Entries in the RuleTable area of a decision table define conditions, actions, and other rule attributes for the rules in that rule table. A spreadsheet of decision tables can contain multiple RuleTable areas.

The following table lists the supported labels (column headers) and values for RuleTable definitions. For column headers, you can use either the given labels or any custom labels that begin with the letters listed in the table.

<table>
<thead>
<tr>
<th>Label</th>
<th>Or custom label that begins with</th>
<th>Value</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>N</td>
<td>Provides the name for the rule generated from that row. The default is constructed from the text following the RuleTable tag and the row number.</td>
<td>At most one column.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>I</td>
<td>Results in a comment within the generated rule.</td>
<td>At most one column.</td>
</tr>
<tr>
<td>CONDITION</td>
<td>C</td>
<td>Code snippet and interpolated values for constructing a constraint within a pattern in a condition.</td>
<td>At least one per rule table.</td>
</tr>
<tr>
<td>ACTION</td>
<td>A</td>
<td>Code snippet and interpolated values for constructing an action for the consequence of the rule.</td>
<td>At least one per rule table.</td>
</tr>
<tr>
<td>METADATA</td>
<td>@</td>
<td>Code snippet and interpolated values for constructing a metadata entry for the rule.</td>
<td>Optional, any number of columns.</td>
</tr>
</tbody>
</table>
The following sections provide more details about how condition, action, and metadata columns use cell data:

### Conditions

For columns headed **CONDITION**, the cells in consecutive lines result in a conditional element:

- **First cell:** Text in the first cell below **CONDITION** develops into a pattern for the rule condition, and uses the snippet in the next line as a constraint. If the cell is merged with one or more neighboring cells, a single pattern with multiple constraints is formed. All constraints are combined into a parenthesized list and appended to the text in this cell.
  
  If this cell is empty, the code snippet in the cell below it must result in a valid conditional element on its own. For example, instead of **Order** as the object type and **itemsCount > $1** as a constraint (separate cells), you can leave the object type cell empty and enter **Order( itemsCount > $1 )** in the constraint cell, and then do the same for any other constraint cells.

To include a pattern without constraints, you can write the pattern in front of the text of another pattern, with or without an empty pair of parentheses. You can also append a **from** clause to the pattern.

If the pattern ends with **eval**, code snippets produce boolean expressions for inclusion into a pair of parentheses after **eval**.

You can terminate the pattern with **@watch** annotation, which is used to customize the properties that the pattern is reactive on.

- **Second cell:** Text in the second cell below **CONDITION** is processed as a constraint on the object reference in the first cell. The code snippet in this cell is modified by interpolating values from cells farther down in the column. If you want to create a constraint consisting of a comparison using **==** with the value from the cells below, then the field selector alone is sufficient. If you use the field selector alone, but you want to use the condition as it is without appending any **==** comparison, you must terminate the condition with the symbol **?**. Any other comparison operator must be specified as the last item within the snippet, and the value from the cells below is appended. For all other constraint forms, you must mark the position for including the contents of a cell with the symbol **$param**. Multiple insertions are possible if you use the symbols **$1**, **$2**, and so on, and a comma-separated list of values in the cells below. However, do not separate **$1**, **$2**, and so on, by commas, or the table will fail to process.
  
  To expand a text according to the pattern **forall($delimiter){$snippet}**, repeat the **$snippet** once for each of the values of the comma-separated list in each of the cells below, insert the value in place of the symbol **$**, and join these expansions by the given **$delimiter**. Note that the **forall** construct may be surrounded by other text.

If the first cell contains an object, the completed code snippet is added to the conditional element from that cell. A pair of parentheses is provided automatically, as well as a separating comma if multiple constraints are added to a pattern in a merged cell. If the first cell is empty, the code snippet in this cell must result in a valid conditional element on its own.

For example, instead of **Order** as the object type and **itemsCount > $1** as a constraint (separate cells), you can leave the object type cell empty and enter **Order( itemsCount > $1 )** in the constraint cell, and then do the same for any other constraint cells.

- **Third cell:** Text in the third cell below **CONDITION** is a descriptive label that you define for the column, as a visual aid.

- **Fourth cell:** From the fourth row on, non-blank entries provide data for interpolation. A blank cell omits the condition or constraint for this rule.
Actions

For columns headed **ACTION**, the cells in consecutive lines result in an action statement:

- **First cell:** Text in the first cell below **ACTION** is optional. If present, the text is interpreted as an object reference.

- **Second cell:** Text in the second cell below **ACTION** is a code snippet that is modified by interpolating values from cells farther down in the column. For a singular insertion, mark the position for including the contents of a cell with the symbol `$param`. Multiple insertions are possible if you use the symbols `$1`, `$2`, and so on, and a comma-separated list of values in the cells below. However, do not separate `$1`, `$2`, and so on, by commas, or the table will fail to process. A text without any marker symbols can execute a method call without interpolation. In this case, use any non-blank entry in a row below the cell to include the statement. The `forall` construct is supported.

  If the first cell contains an object, then the cell text (followed by a period), the text in the second cell, and a terminating semicolon are strung together, resulting in a method call that is added as an action statement for the consequence. If the first cell is empty, the code snippet in this cell must result in a valid action element on its own.

- **Third cell:** Text in the third cell below **ACTION** is a descriptive label that you define for the column, as a visual aid.

- **Fourth cell:** From the fourth row on, non-blank entries provide data for interpolation. A blank cell omits the condition or constraint for this rule.

Metadata

For columns headed **METADATA**, the cells in consecutive lines result in a metadata annotation for the generated rules:

- **First cell:** Text in the first cell below **METADATA** is ignored.

- **Second cell:** Text in the second cell below **METADATA** is subject to interpolation, using values from the cells in the rule rows. The metadata marker character `@` is prefixed automatically, so you do not need to include that character in the text for this cell.

- **Third cell:** Text in the third cell below **METADATA** is a descriptive label that you define for the column, as a visual aid.

- **Fourth cell:** From the fourth row on, non-blank entries provide data for interpolation. A blank cell results in the omission of the metadata annotation for this rule.

43.3. ADDITIONAL RULE ATTRIBUTES FOR RULESET OR RULETABLE DEFINITIONS

The **RuleSet** and **RuleTable** areas also support labels and values for other rule attributes, such as **PRIORITY** or **NO-LOOP**. Rule attributes specified in a **RuleSet** area will affect all rule assets in the same package (not only in the spreadsheet). Rule attributes specified in a **RuleTable** area will affect only the rules in that rule table. You can use each rule attribute only once in a **RuleSet** area and once in a **RuleTable** area. If the same attribute is used in both **RuleSet** and **RuleTable** areas within the spreadsheet, then **RuleTable** takes priority and the attribute in the **RuleSet** area is overridden.
The following table lists the supported labels (column headers) and values for additional RuleSet or RuleTable definitions. For column headers, you can use either the given labels or any custom labels that begin with the letters listed in the table.

Table 43.3. Additional rule attributes for RuleSet or RuleTable definitions

<table>
<thead>
<tr>
<th>Label</th>
<th>Or custom label that begins with</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIORITY</td>
<td>P</td>
<td>An integer defining the salience value of the rule. Rules with a higher salience value are given higher priority when ordered in the activation queue. Overridden by the Sequential flag. Example: PRIORITY 10</td>
</tr>
<tr>
<td>DATE-EFFECTIVE</td>
<td>V</td>
<td>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 &quot;4-Sep-2018&quot;</td>
</tr>
<tr>
<td>DATE-EXPIRES</td>
<td>Z</td>
<td>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 &quot;4-Oct-2018&quot;</td>
</tr>
<tr>
<td>NO-LOOP</td>
<td>U</td>
<td>A Boolean value. When this option is set to true, the rule cannot be reactivated (looped) if a consequence of the rule re-triggers a previously met condition. Example: NO-LOOP true</td>
</tr>
<tr>
<td>AGENDA-GROUP</td>
<td>G</td>
<td>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 &quot;GroupName&quot;</td>
</tr>
<tr>
<td>ACTIVATION-GROUP</td>
<td>X</td>
<td>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 &quot;GroupName&quot;</td>
</tr>
<tr>
<td>DURATION</td>
<td>D</td>
<td>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</td>
</tr>
<tr>
<td>Label</td>
<td>Or custom label that begins with</td>
<td>Value</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TIMER</td>
<td>T</td>
<td>A string identifying either int (interval) or cron timer definitions for scheduling the rule. Example: TIMER &quot;*/5 * * * *&quot; (every 5 minutes)</td>
</tr>
<tr>
<td>CALENDAR</td>
<td>E</td>
<td>A Quartz calendar definition for scheduling the rule. Example: CALENDAR &quot;** 0-7,18-23 ? * *&quot; (exclude non-business hours)</td>
</tr>
<tr>
<td>AUTO-FOCUS</td>
<td>F</td>
<td>A Boolean value, applicable only to rules within agenda groups. When this option is set to true, 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</td>
</tr>
<tr>
<td>LOCK-ON-ACTIVE</td>
<td>L</td>
<td>A Boolean value, applicable only to rules within rule flow groups or agenda groups. When this option is set to true, 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</td>
</tr>
<tr>
<td>RULEFLOW-GROUP</td>
<td>R</td>
<td>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 &quot;GroupName&quot;</td>
</tr>
</tbody>
</table>
Figure 43.2. Sample decision table spreadsheet with attribute columns

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 44. UPLOADING SPREADSHEET DECISION TABLES TO BUSINESS CENTRAL

After you define your rules in an external XLS or XLSX spreadsheet of decision tables, you can upload the spreadsheet file to your project in Business Central.

IMPORTANT

You should typically upload only one spreadsheet of decision tables, containing all necessary RuleTable definitions, per rule package in Business Central. You can upload separate decision table spreadsheets for separate packages, but uploading multiple spreadsheets in the same package can cause compilation errors from conflicting RuleSet or RuleTable attributes and is therefore not recommended.

Procedure

1. In Business Central, go to Menu → Design → Projects and click the project name.
2. Click Add Asset → Decision Table (Spreadsheet).
3. Enter an informative Decision Table name and select the appropriate Package.
4. Click the Choose File icon, and select the spreadsheet. Click Ok to upload.
5. In the decision tables designer, click Validate in the upper-right toolbar to validate the table. If the table validation fails, open the XLS or XLSX file and address any syntax errors. For syntax help, see Chapter 43, Defining spreadsheet decision tables.

You can upload a new version of the decision table or download the current version:

Figure 44.1. Uploaded decision table options
CHAPTER 45. CONVERTING AN UPLOADED SPREADSHEET DECISION TABLE TO A GUIDED DECISION TABLE IN BUSINESS CENTRAL

After you upload an XLS or XLSX spreadsheet decision table file to your project in Business Central, you can convert the decision table to a guided decision table that you can modify directly in Business Central.

For more information about guided decision tables, see Designing a decision service using guided decision tables.

**WARNING**

Guided decision tables and spreadsheet decision tables are different decision table formats that support different features. Any supported features that differ between the two decision table formats are modified or lost when you convert one decision table format to the other.

**Procedure**

In Business Central, navigate to the uploaded decision table asset that you want to convert and in the upper-right toolbar of the decision tables designer, click **Convert**:

**Figure 45.1. Convert an uploaded decision table**

After the conversion, the converted decision table is then available as a guided decision table asset in your project that you can modify directly in Business Central.
CHAPTER 46. EXECUTING RULES

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

Prerequisites

- Business Central and KIE Server are installed and running. For installation options, see Planning a Red Hat Decision Manager installation.

Procedure

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

2. In the upper-right corner of the project Assets page, click Deploy to build the project and deploy it to KIE Server. If the build fails, address any problems described in the Alerts panel at the bottom of the screen.
   For more information about project deployment options, see Packaging and deploying a Red Hat Decision Manager project.

   **NOTE**

   If the rule assets in your project are not built from an executable rule model by default, verify that the following dependency is in the pom.xml file of your project and rebuild the project:

   ```xml
   <dependency>
     <groupId>org.drools</groupId>
     <artifactId>drools-model-compiler</artifactId>
     <version>${rhdm.version}</version>
   </dependency>
   
   This dependency is required for rule assets in Red Hat Decision Manager to be built from executable rule models by default. This dependency is included as part of the Red Hat Decision Manager core packaging, but depending on your Red Hat Decision Manager upgrade history, you may need to manually add this dependency to enable the executable rule model behavior.

   For more information about executable rule models, see Packaging and deploying a Red Hat Decision Manager project.

3. Create a Maven or Java project outside of Business Central, if not created already, that you can use for executing rules locally or that you can use as a client application for executing rules on KIE Server. The project must contain a pom.xml file and any other required components for executing the project resources.
   For example test projects, see “Other methods for creating and executing DRL rules”.

4. Open the pom.xml file of your test project or client application and add the following dependencies, if not added already:

   - kie-ci: Enables your client application to load Business Central project data locally using ReleasesId
- **kie-server-client**: Enables your client application to interact remotely with assets on KIE Server

- **slf4j**: (Optional) Enables your client application to use Simple Logging Facade for Java (SLF4J) to return debug logging information after you interact with KIE Server

Example dependencies for Red Hat Decision Manager 7.10 in a client application **pom.xml** file:

```xml
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.48.0.Final-redhat-00004</version>
</dependency>

<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.48.0.Final-redhat-00004</version>
</dependency>

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.25</version>
</dependency>
```

For available versions of these artifacts, search the group ID and artifact ID in the **Nexus Repository Manager** online.

**NOTE**

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

Example BOM dependency:

```xml
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.10.0.redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see What is the mapping between Red Hat Decision Manager and the Maven library version?.
5. Ensure that the dependencies for artifacts containing model classes are defined in the client application `pom.xml` file exactly as they appear in the `pom.xml` file of the deployed project. If dependencies for model classes differ between the client application and your projects, execution errors can occur.

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

For example, the following Person class dependency appears in both the client and deployed project `pom.xml` files:

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

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

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

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

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

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

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

**Executing rules locally**

```
import org.kie.api.KieServices;
import org.kie.api.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;
```
To test this rule on KIE Server, configure the `RulesTest.java` class with the imports and rule execution information similarly to the local example, and additionally specify KIE services configuration and KIE services client details:

### Executing rules on KIE Server

```java
package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;

import org.kie.api.runtime.KieSession;
import org.drools.compiler.kproject.ReleaseIdImpl;

public class RulesTest {

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

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

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

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

            // Fire all rules:
            kSession.fireAllRules();
            kSession.dispose();
        }
        catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

To test this rule on KIE Server, configure the `java` class with the imports and rule execution information similarly to the local example, and additionally specify KIE services configuration and KIE services client details:
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;
import com.sample.Person;

class RulesTest {

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

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

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

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

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

            // Use rule services client to send request:
            RuleServicesClient ruleClient = kieServicesClient.getServicesClient(RuleServicesClient.class);
            ServiceResponse<ExecutionResults> executeResponse = ruleClient.executeCommandsWithResults(containerName, batch);
            System.out.println("number of fired rules:" + executeResponse.getResult().getValue("numberOfFiredRules"));
        }
    }
}
8. Run the configured `.java` class from your project directory. You can run the file in your development platform (such as Red Hat CodeReady Studio) or in the command line. Example Maven execution (within project directory):

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

Example Java execution (within project directory)

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

9. Review the rule execution status in the command line and in the server log. If any rules do not execute as expected, review the configured rules in the project and the main class configuration to validate the data provided.
CHAPTER 47. NEXT STEPS

- Testing a decision service using test scenarios
- Packaging and deploying a Red Hat Decision Manager project
PART VI. DESIGNING A DECISION SERVICE USING GUIDED RULES

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

NOTE

You can also design your decision service using Decision Model and Notation (DMN) models instead of rule-based or table-based assets. For information about DMN support in Red Hat Decision Manager 7.10, see the following resources:

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

Prerequisites

- The space and project for the guided rules have been created in Business Central. Each asset is associated with a project assigned to a space. For details, see *Getting started with decision services*.
CHAPTER 48. DECISION-AUTHORING ASSETS IN RED HAT DECISION MANAGER

Red Hat Decision Manager supports several assets that you can use to define business decisions for your decision service. Each decision-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights the main decision-authoring assets supported in Red Hat Decision Manager projects to help you decide or confirm the best method for defining decisions in your decision service.

Table 48.1. Decision-authoring assets supported in Red Hat Decision Manager

<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Decision Model and Notation (DMN) models   | • Are decision models based on a notation standard defined by the Object Management Group (OMG)  
• Use graphical decision requirements diagrams (DRDs) that represent part or all of the overall decision requirements graph (DRG) to trace business decision flows  
• Use an XML schema that allows the DMN models to be shared between DMN-compliant platforms  
• Support Friendly Enough Expression Language (FEEL) to define decision logic in DMN decision tables and other DMN boxed expressions  
• Are optimal for creating comprehensive, illustrative, and stable decision flows | Business Central or other DMN-compliant editor | Designing a decision service using DMN models |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided decision tables| • Are tables of rules that you create in a UI-based table designer in Business Central  
• Are a wizard-led alternative to spreadsheet decision tables  
• Provide fields and options for acceptable input  
• Support template keys and values for creating rule templates  
• Support hit policies, real-time validation, and other additional features not supported in other assets  
• Are optimal for creating rules in a controlled tabular format to minimize compilation errors | Business Central       | Designing a decision service using guided decision tables                         |
| Spreadsheet decision tables| • Are XLS or XLSX spreadsheet decision tables that you can upload into Business Central  
• Support template keys and values for creating rule templates  
• Are optimal for creating rules in decision tables already managed outside of Business Central  
• Have strict syntax requirements for rules to be compiled properly when uploaded | Spreadsheet editor      | Designing a decision service using spreadsheet decision tables                     |
| Guided rules          | • Are individual rules that you create in a UI-based rule designer in Business Central  
• Provide fields and options for acceptable input  
• Are optimal for creating single rules in a controlled format to minimize compilation errors | Business Central       | Designing a decision service using guided rules                                   |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided rule templates        | ● Are reusable rule structures that you create in a UI-based template designer in Business Central  
● Provide fields and options for acceptable input  
● Support template keys and values for creating rule templates (fundamental to the purpose of this asset)  
● Are optimal for creating many rules with the same rule structure but with different defined field values | Business Central | Designing a decision service using guided rule templates                        |
| DRL rules                    | ● Are individual rules that you define directly in .drl text files  
● Provide the most flexibility for defining rules and other technicalities of rule behavior  
● Can be created in certain standalone environments and integrated with Red Hat Decision Manager  
● Are optimal for creating rules that require advanced DRL options  
● Have strict syntax requirements for rules to be compiled properly | Business Central or integrated development environment (IDE) | Designing a decision service using DRL rules                                  |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictive Model Markup Language (PMML) models</td>
<td>- Are predictive data-analytic models based on a notation standard defined by the Data Mining Group (DMG)</td>
<td>PMML or XML editor</td>
<td>Designing a decision service using PMML models</td>
</tr>
<tr>
<td></td>
<td>- Use an XML schema that allows the PMML models to be shared between PMML-compliant platforms</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Support Regression, Scorecard, Tree, Mining, and other model types</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Can be included with a standalone Red Hat Decision Manager project or imported into a project in Business Central</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Are optimal for incorporating predictive data into decision services in Red Hat Decision Manager</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 49. GUIDED RULES

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

All data objects related to a guided rule must be in the same project package as the guided rule. Assets in the same package are imported by default. After you create the necessary data objects and the guided rule, you can use the **Data Objects** tab of the guided rules designer to verify that all required data objects are listed or to import other existing data objects by adding a **New item**.
CHAPTER 50. DATA OBJECTS

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

50.1. CREATING DATA OBJECTS

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

Procedure

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

IMPORTING DATA OBJECTS FROM OTHER PACKAGES

You can import an existing data object from another package directly into the asset designers like guided rules or guided decision table designers. Select the relevant rule asset within the project and in the asset designer, go to Data Objects → New item to select the object to be imported.

4. To make your data object persistable, select the Persistable checkbox. Persistable data objects are able to be stored in a database according to the JPA specification. The default JPA is Hibernate.
5. Click Ok.
6. In the data object designer, click add field to add a field to the object with the attributes Id, Label, and Type. Required attributes are marked with an asterisk (*).
   - Id: Enter the unique ID of the field.
   - Label: (Optional) Enter a label for the field.
   - Type: Enter the data type of the field.
   - List: (Optional) Select this check box to enable the field to hold multiple items for the specified type.
### Figure 50.1. Add data fields to a data object

<table>
<thead>
<tr>
<th>New Field</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Id</strong></td>
<td>salary</td>
</tr>
<tr>
<td><strong>Label</strong></td>
<td>Salary</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>BigInteger</td>
</tr>
<tr>
<td><strong>List</strong></td>
<td>False</td>
</tr>
</tbody>
</table>

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

**NOTE**

To edit a field, select the field row and use the **general properties** on the right side of the screen.
CHAPTER 51. CREATING GUIDED RULES

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

Procedure

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

2. Click Add Asset → Guided Rule.

3. Enter an informative Guided Rule name and select the appropriate Package. The package that you specify must be the same package where the required data objects have been assigned or will be assigned.

   You can also select Show declared DSL sentences if any domain specific language (DSL) assets have been defined in your project. These DSL assets will then become usable objects for conditions and actions that you define in the guided rules designer.

4. Click Ok to create the rule asset.

   The new guided rule is now listed in the Guided Rules panel of the Project Explorer, or in the Guided Rules (with DSL) panel if you selected the Show declared DSL sentences option.

5. Click the Data Objects tab and confirm that all data objects required for your rules are listed. If not, click New item to import data objects from other packages, or create data objects within your package.

6. After all data objects are in place, return to the Model tab of the guided rules designer and use the buttons on the right side of the window to add and define the WHEN (condition) and THEN (action) sections of the rule, based on the available data objects.

   Figure 51.1. The guided rules designer

   The WHEN part of the rule contains the conditions that must be met to execute an action. For example, if a bank requires loan applicants to have over 21 years of age, then the WHEN condition of an Underage rule would be Age | less than | 21.

   The THEN part of the rule contains the actions to be performed when the conditional part of the rule has been met. For example, when the loan applicant is under 21 years old, the THEN action would set approved to false, declining the loan because the applicant is under age.

   You can also specify exceptions for more complex rules, such as if a bank may approve of an under-aged applicant when a guarantor is involved. In that case, you would create or import a guarantor data object and then add the field to the guided rule.
7. After you define all components of the rule, click Validate in the upper-right toolbar of the guided rules designer to validate the guided rule. If the rule validation fails, address any problems described in the error message, review all components in the rule, and try again to validate the rule until the rule passes.

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

51.1. ADDING WHEN CONDITIONS IN GUIDED RULES

The WHEN part of the rule contains the conditions that must be met to execute an action. For example, if a bank requires loan applicants to have over 21 years of age, then the WHEN condition of an Underage rule would be Age | less than | 21. You can set simple or complex conditions to determine how and when your rules are applied.

Prerequisites

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

Procedure

1. In the guided rules designer, click the plus icon (+) on the right side of the WHEN section. The Add a condition to the rule window with the available condition elements opens.
The list includes the data objects from the Data Objects tab of the guided rules designer, any DSL objects defined for the package (if you selected Show declared DSL sentences when you created this guided rule), and the following standard options:

- **The following does not exist**: Use this to specify facts and constraints that must not exist.
- **The following exists**: Use this to specify facts and constraints that must exist. This option is triggered on only the first match, not subsequent matches.
- **Any of the following are true**: Use this to list any facts or constraints that must be true.
- **From**: Use this to define a **From** conditional element for the rule.
- **From Accumulate**: Use this to define an **Accumulate** conditional element for the rule.
- **From Collect**: Use this to define a **Collect** conditional element for the rule.
- **From Entry Point**: Use this to define an **Entry Point** for the pattern.
- **Free form DRL**: Use this to insert a free-form DRL field where you can define condition elements freely, without the guided rules designer.

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

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

**Figure 51.3. Modify a condition**

**NOTE**

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

```
a : LoanApplication()
Bankruptcy( application == a ).
```
After you select a constraint, the window closes automatically.

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

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

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

Figure 51.4. Add multiple field constraints

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

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

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

51.2. ADDING THEN ACTIONS IN GUIDED RULES

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

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

Procedure

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

Figure 51.5. Add a new action to the rule

Add a new action...
Position: Bottom

Filter DSLs and Fact Types...

- Change field values of a...
- Delete a
- Modify a...
- Insert fact Applicant...
- Insert fact Bankruptcy...
- Insert fact IncomeSource...
- Insert fact LoanApplication...
- Logically Insert fact Applicant...
- Logically Insert fact Bankruptcy...
- Logically Insert fact IncomeSource...
- Logically Insert fact LoanApplication...
- Add free form DRL
- Call method on a...
The list includes insertion and modification options based on the data objects in the Data Objects tab of the guided rules designer, and on any DSL objects defined for the package (if you selected Show declared DSL sentences when you created this guided rule):

- **Change field values of**: Use this to set the value of fields on a fact (such as LoanApplication) without notifying the decision engine of the change.

- **Delete**: Use this to delete a fact.

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

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

- **Logically Insert fact**: Use this to insert a fact logically into the decision engine and define resulting fields and values for the fact. The decision engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts have to be retracted explicitly. After logical insertions, facts are automatically retracted when the conditions that originally asserted the facts are no longer true.

- **Add free form DRL**: Use this to insert a free-form DRL field where you can define condition elements freely, without the guided rules designer.

- **Call method on**: Use this to invoke a method from another fact.

2. Choose an action element (for example, **Modify**) and click **Ok**.

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

![Figure 51.6. Add a field](image)

After you select a field, the window closes automatically.

4. Click the edit icon (✏️) to define the field value. The field value can be a literal value or a formula.

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

6. Click **Save** in the guided rules designer to save your work.
51.3. DEFINING ENUMERATIONS FOR DROP-DOWN LISTS IN RULE ASSETS

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

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

Procedure

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

2. Click Add Asset → Enumeration.

3. Enter an informative Enumeration name and select the appropriate Package. The package that you specify must be the same package where the required data objects and relevant rule assets have been assigned or will be assigned.

4. Click Ok to create the enumeration.
   The new enumeration is now listed in the Enumeration Definitions panel of the Project Explorer.

5. In the Model tab of the enumerations designer, click Add enum and define the following values for the enumeration:

   - **Fact**: Specify an existing data object within the same package of your project with which you want to associate this enumeration. Open the Data Objects panel in the Project Explorer to view the available data objects, or create the relevant data object as a new asset if needed.

   - **Field**: Specify an existing field identifier that you defined as part of the data object that you selected for the Fact. Open the Data Objects panel in the Project Explorer to select the relevant data object and view the list of available Identifier options. You can create the relevant identifier for the data object if needed.

   - **Context**: Specify a list of values in the format ['string1','string2','string3'] or [integer1, integer2, integer3] that you want to map to the Fact and Field definitions. These values will be displayed as a drop-down list for the relevant field of the rule asset.

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

**Figure 51.7. Example enumeration for applicant credit rating in Business Central**

<table>
<thead>
<tr>
<th>Model</th>
<th>Overview</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add enum</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fact</th>
<th>Field</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant</td>
<td>creditRating</td>
<td>['AA', 'OK', 'Sub prime']</td>
</tr>
</tbody>
</table>

**Example enumeration for applicant credit rating in the DRL source**
'Applicant.creditRating': ['AA', 'OK', 'Sub prime']

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

**Figure 51.8. Example enumeration drop-down options in a guided rule or guided rule template**

<table>
<thead>
<tr>
<th>WHEN</th>
<th>THEN</th>
</tr>
</thead>
</table>
| 1. There is a LoanApplication [app]  
Any of the following are true:  
There is an Applicant with:  
creditRating equal to | OK |
| 2. There is an Applicant with:  
creditRating equal to | Sub prime |

**Figure 51.9. Example enumeration drop-down options in a guided decision table**

### 51.3.1. Advanced enumeration options for rule assets

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

**Mapping between DRL values and values in Business Central**

If you want the enumeration values to appear differently or more completely in the Business Central interface than they appear in the DRL source, use a mapping in the format 'fact.field': ['sourceValue1=UIValue1', 'sourceValue2=UIValue2', ...] for your enumeration definition values.

For example, in the following enumeration definition for loan status, the options A or D are used in the DRL file but the options Approved or Declined are displayed in Business Central:

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

**Enumeration value dependencies**

If you want the selected value in one drop-down list to determine the available options in a subsequent drop-down list, use the format 'fact.fieldB[fieldA=value1]': ['value2', 'value3', ...] for your enumeration definition.

For example, in the following enumeration definition for insurance policies, the policyType field accepts the values Home or Car. The type of policy that the user selects determines the policy coverage field options that are then available:

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

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

External data sources in enumerations

If you want to retrieve a list of enumeration values from an external data source instead of defining the values directly in the enumeration definition, on the class path of your project, add a helper class that returns a `java.util.List` list of strings. In the enumeration definition, instead of specifying a list of values, identify the helper class that you configured to retrieve the values externally.

For example, in the following enumeration definition for loan applicant region, instead of defining applicant regions explicitly in the format `'Applicant.region' : ['country1', 'country2', ... ]`, the enumeration uses a helper class that returns the list of values defined externally:

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

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

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

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

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

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

```java
public class SampleDataSource {

    public Map<String, List<String>> loadData() {
        Map<String, List<String>> data = new HashMap();
        List<String> d = new ArrayList();
        d.add("AU");
        d.add("DE");
        d.add("ES");
        d.add("UK");
        d.add("US");
        ...
        data.put("Applicant.region", d);

        return data;
    }
}
```

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

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

**NOTE**

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

### 51.4. ADDING OTHER RULE OPTIONS

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

**Procedure**

1. In the rule designer, click (show options...) under the **THEN** section.
2. Click the plus icon (➕) on the right side of the window to add options.
3. Select an option to be added to the rule:

   - **Metadata**: Enter a metadata label and click the plus icon (➕). Then enter any needed data in the field provided in the rule designer.
   - **Attribute**: Select from the list of rule attributes. Then further define the value in the field or option displayed in the rule designer.
   - **Freeze areas for editing**: Select **Conditions** or **Actions** to restrict the area from being modified in the rule designer.
4. Click **Save** in the rule designer to save your work.

### 51.4.1. Rule attributes

Rule attributes are additional specifications that you can add to business rules to modify rule behavior.

The following table lists the names and supported values of the attributes that you can assign to rules:

#### Table 51.1. Rule attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
</table>
| **salience**      | An integer defining the priority of the rule. Rules with a higher salience value are given higher priority when ordered in the activation queue.  
Example: **salience 10** |
| **enabled**       | A Boolean value. When the option is selected, the rule is enabled. When the option is not selected, the rule is disabled.  
Example: **enabled true** |
| **date-effective**| A string containing a date and time definition. The rule can be activated only if the current date and time is after a **date-effective** attribute.  
Example: **date-effective "4-Sep-2018"** |
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>date-expires</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: date-expires &quot;4-Oct-2018&quot;</td>
</tr>
<tr>
<td>no-loop</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: no-loop true</td>
</tr>
<tr>
<td>agenda-group</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: agenda-group &quot;GroupName&quot;</td>
</tr>
<tr>
<td>activation-group</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: activation-group &quot;GroupName&quot;</td>
</tr>
<tr>
<td>duration</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: duration 10000</td>
</tr>
<tr>
<td>timer</td>
<td>A string identifying either int (interval) or cron timer definitions for scheduling the rule.</td>
</tr>
<tr>
<td></td>
<td>Example: timer ( cron:* 0/15 * * ? ) (every 15 minutes)</td>
</tr>
<tr>
<td>calendar</td>
<td>A Quartz calendar definition for scheduling the rule.</td>
</tr>
<tr>
<td></td>
<td>Example: calendars &quot;* * 0-7,18-23 ? * *&quot; (exclude non-business hours)</td>
</tr>
<tr>
<td>auto-focus</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: auto-focus true</td>
</tr>
<tr>
<td>Attribute</td>
<td>Value</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>lock-on-active</strong></td>
<td>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: <strong>lock-on-active true</strong></td>
</tr>
<tr>
<td><strong>ruleflow-group</strong></td>
<td>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: <strong>ruleflow-group &quot;GroupName&quot;</strong></td>
</tr>
<tr>
<td><strong>dialect</strong></td>
<td>A string identifying either <strong>JAVA</strong> or <strong>MVEL</strong> 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: <strong>dialect &quot;JAVA&quot;</strong></td>
</tr>
</tbody>
</table>

**NOTE**

When you use Red Hat Decision Manager without the executable model, the **dialect "JAVA"** rule consequences support only Java 5 syntax. For more information about executable models, see Packaging and deploying a Red Hat Decision Manager project.
CHAPTER 52. EXECUTING RULES

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

Prerequisites

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

Procedure

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

2. In the upper-right corner of the project **Assets** page, click **Deploy** to build the project and deploy it to KIE Server. If the build fails, address any problems described in the **Alerts** panel at the bottom of the screen. For more information about project deployment options, see *Packaging and deploying a Red Hat Decision Manager project*.

   **NOTE**

   If the rule assets in your project are not built from an executable rule model by default, verify that the following dependency is in the **pom.xml** file of your project and rebuild the project:

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

   This dependency is required for rule assets in Red Hat Decision Manager to be built from executable rule models by default. This dependency is included as part of the Red Hat Decision Manager core packaging, but depending on your Red Hat Decision Manager upgrade history, you may need to manually add this dependency to enable the executable rule model behavior.

   For more information about executable rule models, see *Packaging and deploying a Red Hat Decision Manager project*.

3. Create a Maven or Java project outside of Business Central, if not created already, that you can use for executing rules locally or that you can use as a client application for executing rules on KIE Server. The project must contain a **pom.xml** file and any other required components for executing the project resources.

   For example test projects, see “Other methods for creating and executing DRL rules”.

4. Open the **pom.xml** file of your test project or client application and add the following dependencies, if not added already:

   - **kie-ci**: Enables your client application to load Business Central project data locally using **ReleasesId**
- **kie-server-client**: Enables your client application to interact remotely with assets on KIE Server

- **slf4j**: (Optional) Enables your client application to use Simple Logging Facade for Java (SLF4J) to return debug logging information after you interact with KIE Server

Example dependencies for Red Hat Decision Manager 7.10 in a client application **pom.xml** file:

```xml
<!-- For local execution -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.48.0.Final-redhat-00004</version>
</dependency>

<!-- For remote execution on KIE Server -->
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.48.0.Final-redhat-00004</version>
</dependency>

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

For available versions of these artifacts, search the group ID and artifact ID in the [Nexus Repository Manager](https://nexus.redhat.com) online.

**NOTE**

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

Example BOM dependency:

```xml
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.10.0.redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see **What is the mapping between Red Hat Decision Manager and the Maven library version?**.
5. Ensure that the dependencies for artifacts containing model classes are defined in the client application pom.xml file exactly as they appear in the pom.xml file of the deployed project. If dependencies for model classes differ between the client application and your projects, execution errors can occur.

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

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

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

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

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

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

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

```java
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

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

**Executing rules locally**

```java
import org.kie.api.KieServices;
import org.kie.api.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;
```
import org.kie.api.runtime.KieSession;
import org.drools.compiler.kproject.ReleaseIdImpl;

public class RulesTest {

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

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

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

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

            // Fire all rules:
            kSession.fireAllRules();
            kSession.dispose();
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}

package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;

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

**Executing rules on KIE Server**
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;

import com.sample.Person;

public class RulesTest {

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

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

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

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

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

            // Use rule services client to send request:
            RuleServicesClient ruleClient =
                    kieServicesClient.getServicesClient(RuleServicesClient.class);
            ServiceResponse<ExecutionResults> executeResponse =
                    ruleClient.executeCommandsWithResults(containerName, batch);
            System.out.println("number of fired rules:" +
                               executeResponse.getResult().getValue("numberOfFiredRules"));
        }
    }
}
8. Run the configured .java class from your project directory. You can run the file in your development platform (such as Red Hat CodeReady Studio) or in the command line. Example Maven execution (within project directory):

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

Example Java execution (within project directory)

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

9. Review the rule execution status in the command line and in the server log. If any rules do not execute as expected, review the configured rules in the project and the main class configuration to validate the data provided.
CHAPTER 53. NEXT STEPS

- Testing a decision service using test scenarios
- Packaging and deploying a Red Hat Decision Manager project
PART VII. DESIGNING A DECISION SERVICE USING GUIDED RULE TEMPLATES

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

NOTE
You can also design your decision service using Decision Model and Notation (DMN) models instead of rule-based or table-based assets. For information about DMN support in Red Hat Decision Manager 7.10, see the following resources:

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

Prerequisites

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

CHAPTER 54. DECISION-AUTHORING ASSETS IN RED HAT DECISION MANAGER

Red Hat Decision Manager supports several assets that you can use to define business decisions for your decision service. Each decision-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights the main decision-authoring assets supported in Red Hat Decision Manager projects to help you decide or confirm the best method for defining decisions in your decision service.

Table 54.1. Decision-authoring assets supported in Red Hat Decision Manager

<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision Model and Notation (DMN) models</td>
<td>• Are decision models based on a notation standard defined by the Object Management Group (OMG)</td>
<td>Business Central or other DMN-compliant editor</td>
<td>Designing a decision service using DMN models</td>
</tr>
<tr>
<td></td>
<td>• Use graphical decision requirements diagrams (DRDs) that represent part or all of the overall decision requirements graph (DRG) to trace business decision flows</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Use an XML schema that allows the DMN models to be shared between DMN-compliant platforms</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Support Friendly Enough Expression Language (FEEL) to define decision logic in DMN decision tables and other DMN boxed expressions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Are optimal for creating comprehensive, illustrative, and stable decision flows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asset</td>
<td>Highlights</td>
<td>Authoring tools</td>
<td>Documentation</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Guided decision tables| - Are tables of rules that you create in a UI-based table designer in Business Central  
- Are a wizard-led alternative to spreadsheet decision tables  
- Provide fields and options for acceptable input  
- Support template keys and values for creating rule templates  
- Support hit policies, real-time validation, and other additional features not supported in other assets  
- Are optimal for creating rules in a controlled tabular format to minimize compilation errors | Business Central  | Designing a decision service using guided decision tables                     |
| Spreadsheet decision tables| - Are XLS or XLSX spreadsheet decision tables that you can upload into Business Central  
- Support template keys and values for creating rule templates  
- Are optimal for creating rules in decision tables already managed outside of Business Central  
- Have strict syntax requirements for rules to be compiled properly when uploaded | Spreadsheet editor | Designing a decision service using spreadsheet decision tables               |
| Guided rules          | - Are individual rules that you create in a UI-based rule designer in Business Central  
- Provide fields and options for acceptable input  
- Are optimal for creating single rules in a controlled format to minimize compilation errors | Business Central  | Designing a decision service using guided rules                               |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Guided rule templates       | • Are reusable rule structures that you create in a UI-based template designer in Business Central  
• Provide fields and options for acceptable input  
• Support template keys and values for creating rule templates (fundamental to the purpose of this asset)  
• Are optimal for creating many rules with the same rule structure but with different defined field values | Business Central| Designing a decision service using guided rule templates                      |
| DRL rules                   | • Are individual rules that you define directly in .drl text files  
• Provide the most flexibility for defining rules and other technicalities of rule behavior  
• Can be created in certain standalone environments and integrated with Red Hat Decision Manager  
• Are optimal for creating rules that require advanced DRL options  
• Have strict syntax requirements for rules to be compiled properly | Business Central or integrated development environment (IDE) | Designing a decision service using DRL rules                               |
<table>
<thead>
<tr>
<th>Asset</th>
<th>Highlights</th>
<th>Authoring tools</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Predictive Model Markup Language (PMML) models |  - Are predictive data-analytic models based on a notation standard defined by the Data Mining Group (DMG)  
  - Use an XML schema that allows the PMML models to be shared between PMML-compliant platforms  
  - Support Regression, Scorecard, Tree, Mining, and other model types  
  - Can be included with a standalone Red Hat Decision Manager project or imported into a project in Business Central  
  - Are optimal for incorporating predictive data into decision services in Red Hat Decision Manager | PMML or XML editor | Designing a decision service using PMML models |
CHAPTER 55. GUIDED RULE TEMPLATES

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

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

All data objects related to a guided rule template must be in the same project package as the guided rule template. Assets in the same package are imported by default. After you create the necessary data objects and the guided rule template, you can use the Data Objects tab of the guided rule templates designer to verify that all required data objects are listed or to import other existing data objects by adding a New item.
CHAPTER 56. DATA OBJECTS

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

56.1. CREATING DATA OBJECTS

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

Procedure

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

IMPORTING DATA OBJECTS FROM OTHER PACKAGES

You can import an existing data object from another package directly into the asset designers like guided rules or guided decision table designers. Select the relevant rule asset within the project and in the asset designer, go to Data Objects → New item to select the object to be imported.

4. To make your data object persistable, select the Persistable checkbox. Persistable data objects are able to be stored in a database according to the JPA specification. The default JPA is Hibernate.
5. Click Ok.
6. In the data object designer, click add field to add a field to the object with the attributes Id, Label, and Type. Required attributes are marked with an asterisk (*).

- Id: Enter the unique ID of the field.
- Label: (Optional) Enter a label for the field.
- Type: Enter the data type of the field.
- List: (Optional) Select this check box to enable the field to hold multiple items for the specified type.
7. Click **Create** to add the new field, or click **Create and continue** to add the new field and continue adding other fields.

**NOTE**

To edit a field, select the field row and use the **general properties** on the right side of the screen.
CHAPTER 57. CREATING GUIDED RULE TEMPLATES

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

Procedure

1. In Business Central, go to Menu → Design → Projects and click the project name.
2. Click Add Asset → Guided Rule Template.
3. Enter an informative Guided Rule Template name and select the appropriate Package. The package that you specify must be the same package where the required data objects have been assigned or will be assigned.
4. Click Ok to create the rule template.
   The new guided rule template is now listed in the Guided Rule Templates panel of the Project Explorer.
5. Click the Data Objects tab and confirm that all data objects required for your rules are listed. If not, click New item to import data objects from other packages, or create data objects within your package.
6. After all data objects are in place, return to the Model tab and use the buttons on the right side of the window to add and define the WHEN (condition) and THEN (action) sections of the rule template, based on the available data objects. For the field values that vary per rule, use template keys in the format $key in the rule designer or in the format @{key} in free form DRL (if used).

Figure 57.1. Sample guided rule template

```
WHEN

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. There is a Customer with:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

THEN

|   | Logically insert RecurringPayment: |
|   | amount $amount |

(show options..)
```
NOTE ON TEMPLATE KEYS

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

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

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

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

57.1. ADDING WHEN CONDITIONS IN GUIDED RULE TEMPLATES

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

Prerequisites

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

Procedure

1. In the guided rule templates designer, click the plus icon (➕) on the right side of the WHEN section. The Add a condition to the rule window with the available condition elements opens.
Figure 57.2. Add a condition to the rule

Add a condition to the rule...

Position: Bottom

Filter DSLs and Fact Types...

- When the credit rating is \{rating\}
- When the applicant date is after \{dos\}
- When the applicant approval is \{bool\}
- When the age is less than \{num\}

- Applicant ...
- Bankruptcy ...
- Customer ...
- IncomeSource ...
- LoanApplication ...
- RecurringPayment ...

- The following does not exist ...
- The following exists ...
- Any of the following are true ...

- From ...
- From Accumulate ...
- From Collect ...
- From Entry Point ...

- Free form DRL

- Only display DSL conditions

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

- **The following does not exist**: Use this to specify facts and constraints that must not exist.
- **The following exists**: Use this to specify facts and constraints that must exist. This option is triggered on only the first match, not subsequent matches.
- **Any of the following are true**: Use this to list any facts or constraints that must be true.
- **From**: Use this to define a **From** conditional element for the rule.
- From Accumulate: Use this to define an Accumulate conditional element for the rule.
- From Collect: Use this to define a Collect conditional element for the rule.
- From Entry Point: Use this to define an Entry Point for the pattern.
- Free form DRL: Use this to insert a free-form DRL field where you can define condition elements freely, without the guided rules designer. For template keys in free form DRL, use the format @{key}.

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

3. Click the condition element in the guided rule templates designer and use the Modify constraints for Customer window to add a restriction on a field, apply multiple field constraints, add a new formula style expression, apply an expression editor, or set a variable name.

Figure 57.3. Modify a condition

Modify constraints for Customer

Add a restriction on a field

Multiple field constraint

Advanced options:

Add a new formula style expression

Expression editor

Variable name

NOTE

A variable name enables you to identify a fact or field in other constructs within the guided rule. For example, you could set the variable of Customer to c and then reference c in a separate Applicant constraint that specifies that the Customer is the Applicant.

\[
c : \text{Customer()}
\]
\[
\text{Applicant( this == c )}
\]

After you select a constraint, the window closes automatically.

4. Choose an operator for the restriction (for example, equal to) from the drop-down menu next to the added restriction.
5. Click the edit icon (📝) to define the field value.

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

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

Figure 57.4. Add multiple field constraints

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

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

57.2. ADDING THEN ACTIONS IN GUIDED RULE TEMPLATES

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

Prerequisites

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

Procedure
1. In the guided rule templates designer, click the plus icon (✚) on the right side of the **THEN** section.

The **Add a new action** window with the available action elements opens.

**Figure 57.5. Add a new action to the rule**

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

- **Insert fact**: Use this to insert a fact and define resulting fields and values for the fact.
- **Logically Insert fact**: Use this to insert a fact logically into the decision engine and define
resulting fields and values for the fact. The decision engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts have to be retracted explicitly. After logical insertions, facts are automatically retracted when the conditions that originally asserted the facts are no longer true.

- **Add free form DRL**: Use this to insert a free-form DRL field where you can define condition elements freely, without the guided rules designer. For template keys in free form DRL, use the format `@{key}`.

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

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

**Figure 57.6. Add a field**

![Add a field window](image)

After you select a field, the window closes automatically.

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

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

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

**57.3. DEFINING ENUMERATIONS FOR DROP-DOWN LISTS IN RULE ASSETS**

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

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

**Procedure**

1. In Business Central, go to **Menu → Design → Projects** and click the project name.
2. Click **Add Asset → Enumeration**.

3. Enter an informative **Enumeration** name and select the appropriate **Package**. The package that you specify must be the same package where the required data objects and relevant rule assets have been assigned or will be assigned.

4. Click **Ok** to create the enumeration. The new enumeration is now listed in the **Enumeration Definitions** panel of the **Project Explorer**.

5. In the **Model** tab of the enumerations designer, click **Add enum** and define the following values for the enumeration:

   - **Fact**: Specify an existing data object within the same package of your project with which you want to associate this enumeration. Open the **Data Objects** panel in the **Project Explorer** to view the available data objects, or create the relevant data object as a new asset if needed.

   - **Field**: Specify an existing field identifier that you defined as part of the data object that you selected for the **Fact**. Open the **Data Objects** panel in the **Project Explorer** to select the relevant data object and view the list of available **Identifier** options. You can create the relevant identifier for the data object if needed.

   - **Context**: Specify a list of values in the format `[string1, string2, string3]` or `[integer1, integer2, integer3]` that you want to map to the **Fact** and **Field** definitions. These values will be displayed as a drop-down list for the relevant field of the rule asset.

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

**Figure 57.7. Example enumeration for applicant credit rating in Business Central**

<table>
<thead>
<tr>
<th>Fact</th>
<th>Field</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant</td>
<td>creditRating</td>
<td>['AA', 'OK', 'Sub prime']</td>
</tr>
</tbody>
</table>

**Example enumeration for applicant credit rating in the DRL source**

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

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

**Figure 57.8. Example enumeration drop-down options in a guided rule or guided rule template**

<table>
<thead>
<tr>
<th>WHEN</th>
<th>THEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. There is a LoanApplication [app]</td>
<td>1. Set value of LoanApplication [app]</td>
</tr>
<tr>
<td>Any of the following are true:</td>
<td><strong>set value LoanApplication [app]</strong></td>
</tr>
<tr>
<td>If creditRating equal to OK Then</td>
<td>2. Set value of LoanApplication [app]</td>
</tr>
<tr>
<td>There is an Applicant with:</td>
<td><strong>set value LoanApplication [app]</strong></td>
</tr>
<tr>
<td>creditRating equal to Sub prime Then</td>
<td>2. There is a LoanApplication [app]</td>
</tr>
<tr>
<td><strong>set value LoanApplication [app]</strong></td>
<td><strong>delete LoanApplication [app]</strong></td>
</tr>
</tbody>
</table>

(approved explanation)

| false | Only AA |
57.3.1. Advanced enumeration options for rule assets

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

Mapping between DRL values and values in Business Central
If you want the enumeration values to appear differently or more completely in the Business Central interface than they appear in the DRL source, use a mapping in the format `fact.field : ['sourceValue1=UIValue1','sourceValue2=UIValue2', ... ] for your enumeration definition values.

For example, in the following enumeration definition for loan status, the options A or D are used in the DRL file but the options Approved or Declined are displayed in Business Central:

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

Enumeration value dependencies
If you want the selected value in one drop-down list to determine the available options in a subsequent drop-down list, use the format `fact.fieldB[fieldA=value1] : ['value2', 'value3', ... ] for your enumeration definition.

For example, in the following enumeration definition for insurance policies, the `policyType` field accepts the values Home or Car. The type of policy that the user selects determines the policy `coverage` field options that are then available:

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

**NOTE**

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

External data sources in enumerations
If you want to retrieve a list of enumeration values from an external data source instead of defining the values directly in the enumeration definition, on the class path of your project, add a helper class that returns a `java.util.List` list of strings. In the enumeration definition, instead of specifying a list of values, identify the helper class that you configured to retrieve the values externally.

For example, in the following enumeration definition for loan applicant region, instead of defining applicant regions explicitly in the format `Applicant.region : ['country1', 'country2', ... ]`, the enumeration uses a helper class that returns the list of values defined externally:

```
'Applicant.region' : (new com.mycompany.DataHelper()).getListOfRegions()
```
In this example, a `DataHelper` class contains a `getListOfRegions()` method that returns a list of strings. The enumerations are loaded in the drop-down list for the relevant field in the rule asset.

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

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

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

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

```java
public class SampleDataSource {

    public Map<String, List<String>> loadData() {
        Map data = new HashMap();
        List d = new ArrayList();
        d.add("AU");
        d.add("DE");
        d.add("ES");
        d.add("UK");
        d.add("US");
        ...
        data.put("Applicant.region", d);

        return data;
    }
}
```

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

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

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

**NOTE**

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

### 57.4. ADDING OTHER RULE OPTIONS

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

1. In the rule designer, click *(show options...)* under the THEN section.

2. Click the plus icon ( ⚫️ ) on the right side of the window to add options.

3. Select an option to be added to the rule:
   - **Metadata:** Enter a metadata label and click the plus icon ( ⚫️ ). Then enter any needed data in the field provided in the rule designer.
   - **Attribute:** Select from the list of rule attributes. Then further define the value in the field or option displayed in the rule designer.
   - **Freeze areas for editing:** Select **Conditions** or **Actions** to restrict the area from being modified in the rule designer.

Figure 57.10. Rule options

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

57.4.1. Rule attributes

Rule attributes are additional specifications that you can add to business rules to modify rule behavior.

The following table lists the names and supported values of the attributes that you can assign to rules:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| salience        | An integer defining the priority of the rule. Rules with a higher salience value are given higher priority when ordered in the activation queue.  
Example: `salience 10` |
| enabled         | A Boolean value. When the option is selected, the rule is enabled. When the option is not selected, the rule is disabled.  
Example: `enabled true` |
| date-effective  | A string containing a date and time definition. The rule can be activated only if the current date and time is after a `date-effective` attribute.  
Example: `date-effective "4-Sep-2018"` |
| date-expires    | A string containing a date and time definition. The rule cannot be activated if the current date and time is after the `date-expires` attribute.  
Example: `date-expires "4-Oct-2018"` |
| no-loop         | A Boolean value. When the option is selected, the rule cannot be reactivated (looped) if a consequence of the rule re-triggers a previously met condition. When the condition is not selected, the rule can be looped in these circumstances.  
Example: `no-loop true` |
| agenda-group    | A string identifying an agenda group to which you want to assign the rule. Agenda groups allow you to partition the agenda to provide more execution control over groups of rules. Only rules in an agenda group that has acquired a focus are able to be activated.  
Example: `agenda-group "GroupName"` |
| activation-group| A string identifying an activation (or XOR) group to which you want to assign the rule. In activation groups, only one rule can be activated. The first rule to fire will cancel all pending activations of all rules in the activation group.  
Example: `activation-group "GroupName"` |
| duration        | A long integer value defining the duration of time in milliseconds after which the rule can be activated, if the rule conditions are still met.  
Example: `duration 10000` |
| timer           | A string identifying either int (interval) or cron timer definitions for scheduling the rule.  
Example: `timer ( cron::* 0/15 * * ? )` (every 15 minutes) |
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>calendar</td>
<td>A Quartz calendar definition for scheduling the rule.</td>
</tr>
<tr>
<td></td>
<td>Example: calendars &quot;** * 0-7,18-23 ? * *&quot; (exclude non-business hours)</td>
</tr>
<tr>
<td>auto-focus</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: auto-focus true</td>
</tr>
<tr>
<td>lock-on-active</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: lock-on-active true</td>
</tr>
<tr>
<td>ruleflow-group</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: ruleflow-group &quot;GroupName&quot;</td>
</tr>
<tr>
<td>dialect</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Example: dialect &quot;JAVA&quot;</td>
</tr>
</tbody>
</table>

**NOTE**

When you use Red Hat Decision Manager without the executable model, the dialect "JAVA" rule consequences support only Java 5 syntax. For more information about executable models, see *Packaging and deploying a Red Hat Decision Manager project*. 
CHAPTER 58. DEFINING DATA TABLES FOR GUIDED RULE TEMPLATES

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

Procedure

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

   NOTE

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

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

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

   Figure 58.1. Sample data table for a guided rule template

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

Example:

```drl
rule "PaymentRules_6"
when
    Customer( internetService == false,
```
phoneService == false, 
TVService == true ) 
then 
RecurringPayment fact0 = new RecurringPayment(); 
fact0.setAmount( 5 ); 
insertLogical( fact0 ); 
end

rule "PaymentRules_5" 
when 
Customer( internetService == false, 
phoneService == true, 
TVService == false ) 
then 
RecurringPayment fact0 = new RecurringPayment(); 
fact0.setAmount( 5 ); 
insertLogical( fact0 ); 
end 
…
//Other rules omitted for brevity.

4. As a visual aid, click the grid icon in the upper-left corner of the data table to toggle cell merging on and off, if needed. Cells in the same column with identical values are merged into a single cell.

Figure 58.2. Merge cells in a data table

![Data Table Image]

You can then use the expand/collapse icon [+/-] in the upper-left corner of each newly merged cell to collapse the rows corresponding to the merged cell, or to re-expand the collapsed rows.
5. After you define the template key data for all rules and adjust the table display as needed, click **Validate** in the upper-right toolbar of the guided rule templates designer to validate the guided rule template. If the rule template validation fails, address any problems described in the error message, review all components in the rule template and data defined in the data table, and try again to validate the rule template until the rule template passes.

6. Click **Save** in the guided rule templates designer to save your work.
CHAPTER 59. EXECUTING RULES

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

Prerequisites

- Business Central and KIE Server are installed and running. For installation options, see Planning a Red Hat Decision Manager installation.

Procedure

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

2. In the upper-right corner of the project Assets page, click Deploy to build the project and deploy it to KIE Server. If the build fails, address any problems described in the Alerts panel at the bottom of the screen. For more information about project deployment options, see Packaging and deploying a Red Hat Decision Manager project.

   **NOTE**

   If the rule assets in your project are not built from an executable rule model by default, verify that the following dependency is in the pom.xml file of your project and rebuild the project:

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

   This dependency is required for rule assets in Red Hat Decision Manager to be built from executable rule models by default. This dependency is included as part of the Red Hat Decision Manager core packaging, but depending on your Red Hat Decision Manager upgrade history, you may need to manually add this dependency to enable the executable rule model behavior.

   For more information about executable rule models, see Packaging and deploying a Red Hat Decision Manager project.

3. Create a Maven or Java project outside of Business Central, if not created already, that you can use for executing rules locally or that you can use as a client application for executing rules on KIE Server. The project must contain a pom.xml file and any other required components for executing the project resources. For example test projects, see “Other methods for creating and executing DRL rules”.

4. Open the pom.xml file of your test project or client application and add the following dependencies, if not added already:

   - **kie-ci**: Enables your client application to load Business Central project data locally using Releaseld
- **kie-server-client**: Enables your client application to interact remotely with assets on KIE Server

- **slf4j**: (Optional) Enables your client application to use Simple Logging Facade for Java (SLF4J) to return debug logging information after you interact with KIE Server

Example dependencies for Red Hat Decision Manager 7.10 in a client application **pom.xml** file:

```xml
<!-- For local execution -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.48.0.Final-redhat-00004</version>
</dependency>

<!-- For remote execution on KIE Server -->
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.48.0.Final-redhat-00004</version>
</dependency>

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

For available versions of these artifacts, search the group ID and artifact ID in the **Nexus Repository Manager** online.

**NOTE**

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

Example BOM dependency:

```xml
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.10.0.redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see What is the mapping between Red Hat Decision Manager and the Maven library version?.
5. Ensure that the dependencies for artifacts containing model classes are defined in the client application pom.xml file exactly as they appear in the pom.xml file of the deployed project. If dependencies for model classes differ between the client application and your projects, execution errors can occur.

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

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

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

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

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

7. In your client application, create a .java main class containing the necessary imports and a main() method to load the KIE base, insert facts, and execute the rules. For example, a Person object in a project contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person. The following Wage rule in a project calculates the wage and hourly rate values and displays a message based on the result:

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

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

**Executing rules locally**

```
import org.kie.api.KieServices;
import org.kie.api.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;
```
import org.kie.api.runtime.KieSession;
import org.drools.compiler.kproject.ReleaseIdImpl;

public class RulesTest {

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

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

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

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

            // Fire all rules:
            kSession.fireAllRules();
            kSession.dispose();
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}

package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;

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

**Executing rules on KIE Server**
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;
import com.sample.Person;

public class RulesTest {

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

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

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

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

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

            // Use rule services client to send request:
            RuleServicesClient ruleClient =
                    kieServicesClient.getServicesClient(RuleServicesClient.class);
            ServiceResponse<ExecutionResults> executeResponse =
                    ruleClient.executeCommandsWithResults(containerName, batch);
            System.out.println("number of fired rules:" +
                           executeResponse.getResult().getValue("numberOfFiredRules"));
        }
    }
}
8. Run the configured `.java` class from your project directory. You can run the file in your development platform (such as Red Hat CodeReady Studio) or in the command line. Example Maven execution (within project directory):

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

Example Java execution (within project directory)

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

9. Review the rule execution status in the command line and in the server log. If any rules do not execute as expected, review the configured rules in the project and the main class configuration to validate the data provided.
CHAPTER 60. NEXT STEPS

- Testing a decision service using test scenarios
- Packaging and deploying a Red Hat Decision Manager project
PART VIII. TESTING A DECISION SERVICE USING TEST SCENARIOS

As a business analyst or business rules developer, you can use test scenarios in Business Central to test a decision service before a project is deployed. You can test DMN-based and rules-based decision services to ensure these are functioning properly and as expected. Also, you can test a decision service at any time during project development.

Prerequisites

- The space and project for the decision service have been created in Business Central. For details, see Getting started with decision services.
- Business rules and their associated data objects have been defined for the rules-based decision service. For details, see Designing a decision service using guided decision tables.
- DMN decision logic and its associated custom data types have been defined for the DMN-based decision service. For details, see Designing a decision service using DMN models.

NOTE

Having defined business rules is not a technical prerequisite for test scenarios, because the scenarios can test the defined data that constitutes the business rules. However, creating the rules first is helpful so that you can also test entire rules in test scenarios and so that the scenarios more closely match the intended decision service. For DMN-based test scenarios ensure that the DMN decision logic and its associated custom data types are defined for the decision service.
CHAPTER 61. TEST SCENARIOS

Test scenarios in Red Hat Decision Manager 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.

Red Hat Decision Manager currently supports both the new Test Scenarios designer and the former Test Scenarios (Legacy) designer. The default designer is the new test scenarios designer, which supports testing of both rules and DMN models and provides an enhanced overall user experience with test scenarios. If required, you can continue to use the legacy test scenarios designer, which supports rule-based test scenarios only.

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 Business Central. You do not have to compile or deploy your decision service to run test scenarios.

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.

IMPORTANT

Throughout the test scenarios documentation, all references to test scenarios and the test scenarios designer are for the new version, unless explicitly noted as the legacy version.
CHAPTER 62. DATA OBJECTS

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

62.1. CREATING DATA OBJECTS

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

Procedure

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

   **IMPORTING DATA OBJECTS FROM OTHER PACKAGES**

   You can import an existing data object from another package directly into the asset designers like guided rules or guided decision table designers. Select the relevant rule asset within the project and in the asset designer, go to **Data Objects → New item** to select the object to be imported.

4. To make your data object persistable, select the **Persistable** checkbox. Persistable data objects are able to be stored in a database according to the JPA specification. The default JPA is Hibernate.
5. Click **Ok**.
6. In the data object designer, click **add field** to add a field to the object with the attributes **Id**, **Label**, and **Type**. Required attributes are marked with an asterisk (*).

   - **Id**: Enter the unique ID of the field.
   - **Label**: (Optional) Enter a label for the field.
   - **Type**: Enter the data type of the field.
   - **List**: (Optional) Select this check box to enable the field to hold multiple items for the specified type.
Figure 62.1. Add data fields to a data object

<table>
<thead>
<tr>
<th>New Field</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Id</strong></td>
<td>salary</td>
</tr>
<tr>
<td><strong>Label</strong></td>
<td>Salary</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>BigInteger</td>
</tr>
<tr>
<td><strong>List</strong></td>
<td></td>
</tr>
</tbody>
</table>

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

**NOTE**

To edit a field, select the field row and use the **general properties** on the right side of the screen.
CHAPTER 63. TEST SCENARIOS DESIGNER IN BUSINESS CENTRAL

The test scenarios designer provides a tabular layout that helps you in defining a scenario template and all the associated test cases. The designer layout consists of a table which has a header and the individual rows. The header consists of three parts, the GIVEN and EXPECT row, a row with instances, and a row with corresponding fields. The header is also known as test scenario template and the individual rows are called test scenarios definitions.

The test scenario template or header has the following two parts:

- **GIVEN** data objects and their fields - represents the input information
- **EXPECT** data objects and their fields - represents the objects and their fields whose exact values are checked based on the given information and which also constitutes the expected result.

The test scenarios definitions represent the separate test cases of a template.

You can access the Project Explorer from the left panel of the designer whereas from the right panel you can access the Settings, Test Tools, Scenario Cheatsheet, Test Report and the Coverage Report tabs. You can access the Settings tab to view and edit the global settings of rule-based and DMN-based test scenarios. You can use the Test Tools to configure the data object mappings. Scenario Cheatsheet tab contains notes and the cheat sheet which you can use as reference. The Test Report tab displays the overview of the tests and the scenario status. To view the test coverage statistics, you can use the Coverage Report tab from the right side of the test scenario designer.

### 63.1. IMPORTING DATA OBJECTS

The test scenarios designer loads all data objects that are located in the same package as the test scenario. You can view all the data objects from the Data Objects tab in the designer. The loaded data objects are also displayed in the Test Tools panel.

You need to close and reopen the designer in case the data objects change (for example, when a new data object is created or when an existing one is deleted). Select a data object from the list to display its fields and the field types.

In case you want to use a data object located in a different package than the test scenario, you need to import the data object first. Follow the procedure below to import a data object for rules-based test scenarios.

**NOTE**

You cannot import any data objects while creating DMN-based test scenarios. DMN-based test scenarios do not use any data objects from the project but uses the custom data types defined in the DMN file.

**Procedure**

1. Go to Project Explorer panel in the test scenarios designer.
2. From Test Scenario, select a test scenario.
3. Select Data Objects tab and click New Item.
4. In the **Add import** window, choose the data object from the drop-down list.

5. Click **Ok** and then **Save**.

6. Close and reopen the test scenarios designer to view the new data object from the data objects list.

### 63.2. IMPORTING A TEST SCENARIO

You can import an existing test scenario using the **Import Asset** button in the **Asset** tab from the project view.

**Procedure**

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

2. From the project’s **Asset** tab, click **Import Asset**.

3. In the **Create new Import Asset** window,
   - Enter the name of the import asset.
   - Select the package from the **Package** drop-down list.
   - From **Please select a file to upload** click **Choose File...** to browse to test scenario file.

4. Select the file and click **Open**.

5. Click **Ok** and the test scenario opens in the test scenario designer.

### 63.3. SAVING A TEST SCENARIO

You can save a test scenario at any time while creating a test scenario template or defining the test scenarios.

**Procedure**

1. From the test scenarios designer toolbar on the upper-right, click **Save**.

2. On the **Confirm Save** window,
   - If you wish to add a comment regarding the test scenario, click **add a comment**
   - Click **Save** again.

A message stating that the test scenario was saved successfully appears on the screen.

### 63.4. COPYING A TEST SCENARIO

You can copy an existing test scenario to the same package or to some other package by using the **Copy** button from the upper-right toolbar.

**Procedure**

1. From the test scenarios designer toolbar on the upper-right, click **Copy**.
2. In the Make a Copy window,
   a. Enter a name in the New Name field.
   b. Select the package you want to copy the test scenario to.
   c. Optional: To add a comment, click add a comment
   d. Click Make a Copy.

A message stating that the test scenario was copied successfully appears on the screen.

63.5. DOWNLOADING A TEST SCENARIO

You can download a copy of the test scenario to your local machine for future reference or as backup.

Procedure
In the test scenarios designer toolbar on the upper-right, click the Download icon.

The .scesim file is downloaded to your local machine.

63.6. SWITCHING BETWEEN VERSIONS OF A TEST SCENARIO

Business Central provides you the ability to switch between the various versions of a test scenario. Every time you save the scenario, a new version of the scenario is listed under Latest Versions. To use this feature, you must save the test scenario file at least once.

Procedure
1. From the test scenarios designer toolbar on the upper-right, click Latest Version. All the versions of the file are listed under Latest Version, if they exist.
2. Click the version you want to work on.
   The selected version of the test scenario opens in the test scenarios designer.
3. From the designer toolbar, click Restore.
4. In the Confirm Restore,
   a. To add a comment, click add a comment
   b. Click Restore to confirm.

A message stating that the selected version has been reloaded successfully in the designer appears on the screen.

63.7. VIEW OR HIDE THE ALERTS PANEL

The Alerts panel appears at the bottom of the test scenarios designer or the project view. It contains the build information and error messages in case the executed tests are failed.

Procedure
From the designer toolbar on the upper-right, click Hide Alerts/View Alerts to enable or disable the reporting panel.
63.8. CONTEXTUAL MENU OPTIONS

The test scenarios designer provides contextual menu options, which enables you to perform basic operations on the table such as adding, deleting, and, duplicating rows and columns. To use the contextual menus, you need to right-click a table element. Menu options differ based on the table element you select.

Table 63.1. Contextual menu options

<table>
<thead>
<tr>
<th>Table element</th>
<th>Cell label</th>
<th>Available context menu options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>#, Scenario description</td>
<td>Insert row below</td>
</tr>
<tr>
<td></td>
<td>GIVEN, EXPECT</td>
<td>Insert leftmost column, Insert rightmost column, Insert row below</td>
</tr>
<tr>
<td></td>
<td>INSTANCE 1, INSTANCE 2, PROPERTY 1, PROPERTY 2</td>
<td>Insert column left, Insert column right, Delete column, Duplicate Instance, Insert row below</td>
</tr>
<tr>
<td>Rows</td>
<td>All the cells with row numbers, test scenarios description or test scenarios definition</td>
<td>Insert row above, Insert row below, Duplicate row, Delete row, Run scenario</td>
</tr>
</tbody>
</table>

Table 63.2. Description of table interactions

<table>
<thead>
<tr>
<th>Table interaction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert leftmost column</td>
<td>Inserts a new leftmost column (in either the GIVEN or EXPECT section of the table based on user selection).</td>
</tr>
<tr>
<td>Insert rightmost column</td>
<td>Inserts a new rightmost column (in either the GIVEN or EXPECT section of the table based on user selection).</td>
</tr>
<tr>
<td>Insert column left</td>
<td>Inserts a new column to the left of the selected column. The new column is of the same type as the selected column (in either the GIVEN or EXPECT section of the table based on user selection).</td>
</tr>
<tr>
<td>Insert column right</td>
<td>Inserts a new column to the right of the selected column. The new column is of the same type as the selected column (in either the GIVEN or EXPECT section of the table based on user selection).</td>
</tr>
<tr>
<td>Delete column</td>
<td>Deletes the selected column.</td>
</tr>
<tr>
<td>Insert row above</td>
<td>Inserts a new row above the selected row.</td>
</tr>
<tr>
<td>Insert row below</td>
<td>Inserts a new row below the selected row. If invoked from a header cell, inserts a new row with index 1.</td>
</tr>
<tr>
<td>Duplicate row</td>
<td>Duplicates the selected row.</td>
</tr>
<tr>
<td>Table interaction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>Duplicate Instance</td>
<td>Duplicates the selected instance.</td>
</tr>
<tr>
<td>Delete row</td>
<td>Deletes the selected row.</td>
</tr>
<tr>
<td>Run scenario</td>
<td>Runs a single test scenario.</td>
</tr>
</tbody>
</table>

The **Insert column right** or **Insert column left** context menu options behave differently.

- If the selected column does not have a type defined, a new column without a type is added.
- If the selected column has a type defined, either a new empty column or a column with the parent instance type is created.
  - If the action is performed from an instance header, a new column without a type is created.
  - If the action is performed from a property header, a new column with the parent instance type is created.

### 63.9. GLOBAL SETTINGS FOR TEST SCENARIOS

You can use the global **Settings** tab on the right side of the test scenarios designer to set and modify the additional properties of assets.

#### 63.9.1. Configuring global settings for rule-based test scenarios

Follow the procedure below to view and edit the global settings of rule-based test scenarios.

**Procedure**

1. Click **Settings** tab on the right side of the test scenario designer to display the attributes.
2. Configure the following attributes in the **Settings** panel:
   - **Name**: You can change the name of the existing test scenarios by using the **Rename** option from the upper-right toolbar in the designer.
   - **Type**: This attribute specifies it is a rule-based test scenario and it is read-only.
   - **Stateless Session**: Select or clear this check box to specify if the KieSession is stateless or not.

   **NOTE**
   
   If the current KieSession is stateless and the check box is not selected, the tests will fail.

   - **KieSession**: (Optional) Enter the KieSession for the test scenario.
   - **RuleFlowGroup/AgendaGroup**: (Optional) Enter the RuleFlowGroup or AgendaGroup for the test scenario.
3. Optional: To skip the entire simulation from project level after test execution, select the check box.

4. Click Save.

63.9.2. Configuring global settings for DMN-based test scenarios

Follow the procedure below to view and edit the global settings of DMN-based test scenarios.

Procedure

1. Click **Settings** tab on the right side of the test scenario designer to display the attributes.

2. Configure the following attributes in the **Settings** panel:
   - **Name**: You can change the name of the existing test scenarios by using the **Rename** option from the upper-right toolbar in the designer.
   - **Type**: This attribute specifies it is a DMN-based test scenario and it is read-only.
   - **DMN model**: (Optional) Enter the DMN model for the test scenario.
   - **DMN name**: This is the name of the DMN model and it is read-only.
   - **DMN namespace**: This is the default namespace for DMN model and it is read-only.

3. Optional: To skip the entire simulation from project level after test execution, select the check box.

4. Click Save.
CHAPTER 64. TEST SCENARIO TEMPLATE

Before specifying test scenario definitions, you need to create a test scenario template. The header of the test scenario table defines the template for each scenario. You need to set the types of the instance and property headers for both the GIVEN and EXPECT sections. Instance headers map to a particular data object (a fact), whereas the property headers map to a particular field of the corresponding data object.

Using the test scenarios designer, you can create test scenario templates for both rule-based and DMN-based test scenarios.

64.1. CREATING A TEST SCENARIO TEMPLATE FOR RULE-BASED TEST SCENARIOS

Create a test scenario template for rule-based test scenarios by following the procedure below to validate your rules and data.

Procedure

1. In Business Central, go to Menu → Design → Projects and click the project for which you want to create the test scenario.

2. Click Add Asset → Test Scenario.

3. Enter a Test Scenario name and select the appropriate Package. The package you select must contain all the required data objects and rule assets have been assigned or will be assigned.

4. Select RULE as the Source type.

5. Click Ok to create and open the test scenario in the test scenarios designer.

6. To map the GIVEN column header to a data object:

   Figure 64.1. Test scenario GIVEN header cells

   a. Select an instance header cell in the GIVEN section.

   b. Select the data object from the Test Tools tab.

   c. Click Insert Data Object.

7. To map the EXPECT column header to a data object:
**Figure 64.2. Test scenario EXPECT header cells**

<table>
<thead>
<tr>
<th>#</th>
<th>Scenario Description</th>
<th>Event</th>
<th>Violation</th>
<th>Expect</th>
<th>Points</th>
<th>Age</th>
<th>Speed Limit</th>
<th>Violation</th>
<th>Actual Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Above speed limit: 100km/h and 30km/h</td>
<td>Violation</td>
<td></td>
<td></td>
<td>10</td>
<td>25</td>
<td>100</td>
<td></td>
<td>120</td>
</tr>
<tr>
<td>2</td>
<td>Above speed limit: more than 30 km/h</td>
<td>Violation</td>
<td></td>
<td></td>
<td>10</td>
<td>25</td>
<td>100</td>
<td></td>
<td>120</td>
</tr>
</tbody>
</table>

a. Select an instance header cell in the **EXPECT** section.

b. Select the data object from the **Test Tools** tab.

c. Click **Insert Data Object**.

8. To map a data object field to a property cell:

a. Select an instance header cell or property header cell.

b. Select the data object field from the **Test Tools** tab.

c. Click **Insert Data Object**.

d. Use the MVEL expression with the prefix `#` to define a java method for test scenario execution.

e. To insert more properties of the data object, right-click the property header cell and select **Insert column right** or **Insert column left** as required.

9. To insert more properties of the data object, right-click the property header and select **Insert column right** or **Insert column left** as required.

10. To define a java method to a property cell during test scenarios execution:

a. Select an instance header cell or property header cell.

b. Select the data object field from the **Test Tools** tab.

c. Click **Insert Data Object**

d. Use the MVEL expression with the prefix `#` to define a java method for test scenario execution.

e. To insert more properties of the data object, right-click the property header cell and select **Insert column right** or **Insert column left** as required.

11. Use the contextual menu to add or remove columns and rows as needed.

For more details about the expression syntax in rule-based scenarios, see Section 69.1, “Expression syntax in rule-based test scenarios”.

### 64.2. USING ALIASES IN RULE-BASED TEST SCENARIOS

In the test scenarios designer, once you map a header cell with a data object, the data object is removed from the **Test Tools** tab. You can re-map a data object to another header cell by using an alias. Aliases enable you to specify multiple instances of the same data object in a test scenario. You can also create property aliases to rename the used properties directly in the table.

**Procedure**

In the test scenarios designer in Business Central, double-click a header cell and manually change the name. Ensure that the aliases are uniquely named.

The instance now appears in the list of data objects in the **Test Tools** tab.
CHAPTER 65. TEST TEMPLATE FOR DMN-BASED TEST SCENARIOS

Business Central automatically generates the template for every DMN-based test scenario asset and it contains all the specified inputs and decisions of the related DMN model. For each input node in the DMN model, a GIVEN column is added, whereas each decision node is represented by an EXPECT column. You can modify the default template at any time as per your needs. Also, to test only a specific part of the whole DMN model, its possible to remove the generated columns as well as move decision nodes from the EXPECT to the GIVEN section.

65.1. CREATING A TEST SCENARIO TEMPLATE FOR DMN-BASED TEST SCENARIOS

Create a test scenario template for DMN-based scenarios by following the procedure below to validate your DMN models.

Procedure

1. In Business Central, go to Menu → Design → Projects and click the project that you want to create the test scenario for.

2. Click Add Asset → Test Scenario.

3. Enter a Test Scenario name and select the appropriate Package.

4. Select DMN as the Source type.

5. Select an existing DMN asset using the Choose DMN asset option.

6. Click Ok to create and open the test scenario in the test scenarios designer. The template is automatically generated and you can modify it as per your needs.

7. To define a java method to a property cell during test scenario execution:
   a. Click an instance header cell or property header cell.
   b. Select the data object field from the Test Tools tab.
   c. Click Insert Data Object
   d. Use an expression to define a java method for test scenario execution.
   e. To add more properties to the data object, right-click the property header cell and select Insert column right or Insert column left as required.

8. Use the contextual menu to add or remove columns and rows as needed.

For more details about the expression syntax in DMN-based scenarios, see Section 69.2, “Expression syntax in DMN-based scenarios”.

Red Hat Decision Manager 7.10 Developing decision services in Red Hat Decision Manager
CHAPTER 66. DEFINING A TEST SCENARIO

After creating a test scenario template you have to define the test scenario next. The rows of the test scenario table define the individual test scenarios. A test scenario has a unique index number, description, set of input values (the Given values), and a set of output values (the Expect values).

Prerequisites

- The test scenario template has been created for the selected test scenario.

Procedure

1. Open the test scenario in the test scenarios designer.
2. Enter a description of the test scenario and fill in required values in each cell of the row.
3. Use the contextual menu to add or remove rows as required.
   Double click a cell to start inline editing. To skip a particular cell from test evaluation, leave it empty.

After defining the test scenario, you can run the test next.
CHAPTER 67. BACKGROUND INSTANCE IN TEST SCENARIOS

In test scenario designer, you can use the **Background** tab to add and set background data for rule-based and DMN-based test scenarios. You can add and define the GIVEN data which is common for the entire test scenario simulation, based on the available data objects. **Background** tab has the ability to add and share the data among every test scenario. Data added using the **Background** tab can not be overridden by **Model** tab data.

For example, if the test scenario example requires the same value for the person **Age** in all test scenarios, you can define the **Age** value in the **Background** page and exclude that column from the test scenario table template. In this case, the **Age** is set to **25** for all test scenarios.

**Figure 67.1. Example test scenarios with repeated value for Age**

<table>
<thead>
<tr>
<th>#</th>
<th>Scenario description</th>
<th>GIVEN</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Driver</td>
<td>Violation</td>
</tr>
<tr>
<td>1</td>
<td>Insert value</td>
<td>25</td>
<td>Insert value</td>
</tr>
<tr>
<td>2</td>
<td>Insert value</td>
<td>25</td>
<td>Insert value</td>
</tr>
</tbody>
</table>

**Figure 67.2. Example background definition of repeated value for Age**

<table>
<thead>
<tr>
<th>Model</th>
<th>Background</th>
<th>Overview</th>
<th>Data Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Driver</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Age</td>
<td>25</td>
</tr>
</tbody>
</table>

**Figure 67.3. Modified test scenario template with excluded Age column**

<table>
<thead>
<tr>
<th>#</th>
<th>Scenario description</th>
<th>GIVEN</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Driver</td>
<td>Violation</td>
</tr>
<tr>
<td>1</td>
<td>Insert value</td>
<td>Insert value</td>
<td>Insert value</td>
</tr>
<tr>
<td>2</td>
<td>Insert value</td>
<td>Insert value</td>
<td>Insert value</td>
</tr>
</tbody>
</table>

**NOTE**

The **GIVEN** data which is defined in the **Background** tab can only be shared between the test scenarios of the same *.scesim* file and will not be shared among different test scenarios.

67.1. ADDING A BACKGROUND DATA IN RULE-BASED TEST SCENARIOS

Follow the procedure below to add and set a background data in rule-based test scenarios.

**Prerequisites**

- The rule-based test scenario template are created for the selected test scenario. For more information about creating rule-based test scenarios, see Section 64.1, “Creating a test scenario template for rule-based test scenarios”.
The individual test scenarios are defined. For more information about defining a test scenario, see Chapter 66, Defining a test scenario.

Procedure

1. Open the rule-based test scenarios in the test scenario designer.
2. Click the Background tab of the test scenarios designer.
3. Select an instance header cell in the GIVEN section to add a background data object field.
4. From the Test Tools panel, select the data object.
5. Click Insert Data Object.
6. Select a property header cell to add a background data object field.
7. From the Test Tools panel, select the data object.
8. Click Insert Data Object.
9. To add more properties to the data object, right-click the property header cell and select Insert column right or Insert column left as required.
10. Use the contextual menu to add or remove columns and rows as needed.
11. Run the defined test scenarios.

67.2. ADDING A BACKGROUND DATA IN DMN-BASED TEST SCENARIOS

Follow the procedure below to add and set a background data in DMN-based test scenarios.

Prerequisites

- The DMN-based test scenario template is created for the selected test scenario. For more information about creating DMN-based test scenarios, see Section 65.1, “Creating a test scenario template for DMN-based test scenarios”.
- The individual test scenarios are defined. For more information about defining a test scenario, see Chapter 66, Defining a test scenario.

Procedure

1. Open the DMN-based test scenarios in the test scenario designer.
2. Click the Background tab of the test scenarios designer.
3. Select an instance header cell in the GIVEN section to add a background data object field.
4. From the Test Tools panel, select the data object.
5. Click Insert Data Object.
6. Select a property header cell to add a background data object field.
7. From the Test Tools panel, select the data object.

8. Click Insert Data Object.

9. To add more properties to the data object, right-click the property header cell and select Insert column right or Insert column left as required.

10. Use the contextual menu to add or remove columns and rows as needed.

11. Run the defined test scenarios.
CHAPTER 68. USING LIST AND MAP COLLECTIONS IN TEST SCENARIOS

The test scenarios designer supports list and map collections for both DMN-based as well as rules-based test scenarios. You can create and define a collection like a list or a map as the value of a particular cell in both GIVEN and EXPECT columns.

NOTE

For map entries, an entry key must be a String data type.

To pass the parameter in the EXPECT column of Rule-based collection editor use the actualValue keyword whereas use the ? keyword in DMN-based test scenario.

Procedure

1. Set the column type first (use a field whose type is a list or a map).
2. Double click a cell in the column to input a value.
3. To create the list values for the data objects in the collection editor popup:
   a. Select Create List
   b. Click Add new item.
   c. Enter the required value and click the check icon to save each collection item that you add.
   d. Click Save.
   e. To edit an item from the collection, click the pencil icon in the collection popup editor.
   f. Click Save changes.
   g. To delete an item from the collection, click the bin icon in the collection popup editor.
4. To define the list values for the data objects in the collection editor popup:
   a. Select Define List
   b. Use the MVEL or FEEL expression to define a list value in the text field.
      Rule-based test scenario uses MVEL expression language and DMN-based test scenario uses FEEL expression language.
   c. Click Save.
5. To create the map values for the data objects in the collection editor popup:
   a. Select Create Map.
   b. Click Add new item.
c. Enter the required value and click the check icon to save each collection item that you add.

d. Click Save.

e. To edit an item from the collection, click the pencil icon in the collection popup editor.

f. Click Save changes.

g. To delete an item from the collection, click the bin icon in the collection popup editor.

6. To define the map values for the data objects in the collection editor popup:

a. Select Define Map.

b. Use the MVEL or FEEL expression to define a map value in the text field.
   Rule-based test scenario uses MVEL expression language and DMN-based test scenario uses FEEL expression language.

c. Click Save.

   **NOTE**
   
   To define the map values for DMN-based test scenario, you can add a fact and use the FEEL expression, instead of using the collection editor.

7. Click Remove to delete the entire collection.
CHAPTER 69. EXPRESSION SYNTAX IN TEST SCENARIOS

The test scenarios designer supports different expression languages for both rule-based and DMN-based test scenarios. While rule-based test scenarios support the MVFLEX Expression Language (MVEL) and DMN-based test scenarios support the Friendly Enough Expression Language (FEEL).

69.1. EXPRESSION SYNTAX IN RULE-BASED TEST SCENARIOS

Rule-based test scenario supports the following built-in data types:

- String
- Boolean
- Integer
- Long
- Double
- Float
- Character
- Byte
- Short
- LocalDate

**NOTE**

For any other data types, use the MVEL expression with the prefix `#`.

Follow the BigDecimal example in the test scenario designer to use the `#` prefix to set the java expression:

- Enter `# java.math.BigDecimal.valueOf(10)` for the GIVEN column value.
- Enter `# actualValue.intValue() == 10` for the EXPECT column value.

You can refer to the actual value of the EXPECT column in the java expression to execute a condition.

The following rule-based test scenario definition expressions are supported by the test scenarios designer:

**Table 69.1. Description of expressions syntax**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Specifies equal to a value. This is default for all columns and is the only operator supported by the GIVEN column.</td>
</tr>
</tbody>
</table>
### Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=, =</td>
<td>!, &lt;</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>Specifies a comparison: less than, greater than, less or equals than, and greater or equals than.</td>
</tr>
<tr>
<td>#</td>
<td>This operator is used to set the java expression value to a property header cell which can be executed as a java method.</td>
</tr>
<tr>
<td>[value1, value2, value3]</td>
<td>Specifies a list of values. If one or more values are valid, the scenario definition is evaluated as true.</td>
</tr>
<tr>
<td>expression1; expression2; expression3</td>
<td>Specifies a list of expressions. If all expressions are valid, the scenario definition is evaluated as true.</td>
</tr>
</tbody>
</table>

**NOTE**

An empty cell is skipped from evaluation. To define an empty string, use =[], or ;. To define a null value, use null.

### Table 69.2. Example expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>The actual value is equal to -1.</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>The actual value is less than 0.</td>
</tr>
<tr>
<td>! &gt; 0</td>
<td>The actual value is not greater than 0.</td>
</tr>
<tr>
<td>[-1, 0, 1]</td>
<td>The actual value is equal to either -1 or 0 or 1.</td>
</tr>
<tr>
<td>&lt;&gt; [1, -1]</td>
<td>The actual value is neither equal to 1 nor -1.</td>
</tr>
<tr>
<td>! 100; 0</td>
<td>The actual value is not equal to 100 but is equal to 0.</td>
</tr>
<tr>
<td>!= &lt; 0; &lt;&gt; &gt; 1</td>
<td>The actual value is neither less than 0 nor greater than 1.</td>
</tr>
<tr>
<td>&lt;&gt; &lt;= 0; &gt;= 1</td>
<td>The actual value is neither less than 0 nor equal to 0 but is greater than or equal to 1.</td>
</tr>
</tbody>
</table>

**NOTE**

You can refer to the supported commands and syntax in the Scenario Cheatsheet tab on the right of the rule-based test scenarios designer.
69.2. EXPRESSION SYNTAX IN DMN-BASED SCENARIOS

The following data types are supported by the DMN-based test scenarios in the test scenarios designer:

Table 69.3. Data types supported by DMN-based scenarios

<table>
<thead>
<tr>
<th>Supported data types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numbers &amp; strings</td>
<td>Strings must be delimited by quotation marks, for example, &quot;John Doe&quot;, &quot;Brno&quot; or &quot;&quot;.</td>
</tr>
<tr>
<td>boolean values</td>
<td>true, false, and null.</td>
</tr>
<tr>
<td>dates and time</td>
<td>For example, date(&quot;2019-05-13&quot;) or time(&quot;14:10:00+02:00&quot;).</td>
</tr>
<tr>
<td>functions</td>
<td>Supports built-in math functions, for example, avg, max.</td>
</tr>
<tr>
<td>contexts</td>
<td>For example, {x : 5, y : 3}.</td>
</tr>
<tr>
<td>ranges and lists</td>
<td>For example, ([1 .. 10]) or ([2, 3, 4, 5]).</td>
</tr>
</tbody>
</table>

NOTE

You can refer to the supported commands and syntax in the Scenario Cheatsheet tab on the right of the DMN-based test scenarios designer.
CHAPTER 70. RUNNING THE TEST SCENARIOS

After creating a test scenario template and defining the test scenarios, you can run the tests to validate your business rules and data.

Procedure

1. To run defined test scenarios, do any of the following tasks:
   - To execute all the available test scenarios in your project inside multiple assets, in the upper-right corner of your project page, click Test.
   - To execute all available test scenarios defined in a .scesim file, at the top of the Test Scenario designer, click the Run Test icon.
   - To run a single test scenario defined in a single .scesim file, right-click the row of the test scenario you want to run and select Run scenario.

2. The Test Report panel displays the overview of the tests and the scenario status. After the tests execute, if the values entered in the test scenario table do not match with the expected values, then the corresponding cells are highlighted.

3. If tests fail, you can do the following tasks to troubleshoot the failure:
   - To review the error message in the pop-up window, hover your mouse cursor over the highlighted cell.
   - To open the Alerts panel at the bottom of the designer or the project view for the error messages, click View Alerts.
   - Make the necessary changes and run the test again until the scenario passes.
CHAPTER 71. RUNNING A TEST SCENARIO LOCALLY

In Red Hat Decision Manager, you can either run the test scenarios directly in Business Central or locally using the command line.

Procedure

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

2. On the Project’s home page, select the Settings tab.

3. Select git URL and click the Clipboard to copy the git url.

4. Open a command terminal and navigate to the directory where you want to clone the git project.

5. Run the following command:

   ```
   git clone your_git_project_url
   ```

   Replace `your_git_project_url` with relevant data like `git://localhost:9418/MySpace/ProjectTestScenarios`.

6. Once the project is successfully cloned, navigate to the git project directory and execute the following command:

   ```
   mvn clean test
   ```

   Your project’s build information and the test results (such as, the number of tests run and whether the test run was a success or not) are displayed in the command terminal. In case of failures, make the necessary changes in Business Central, pull the changes and run the command again.
CHAPTER 72. EXPORTING AND IMPORTING TEST SCENARIO SPREADSHEETS

These sections show how to export and import test scenario spreadsheets in the test scenario designer. You can analyze and manage test scenario spreadsheets with software such as Microsoft Excel or LibreOffice Calc. Test scenario designer supports the .CSV file format. For more information about the RFC specification for the Comma-Separated Values (CSV) format, see Common Format and MIME Type for Comma-Separated Values (CSV) Files.

72.1. EXPORTING A TEST SCENARIO SPREADSHEET

Follow the procedure below to export a test scenario spreadsheet using the Test Scenario designer.

Procedure

1. In the Test Scenario designer toolbar on the upper-right, click Export button.
2. Select a destination in your local file directory and confirm to save the .CSV file.

The .CSV file is exported to your local machine.

72.2. IMPORTING A TEST SCENARIO SPREADSHEET

Follow the procedure below to import a test scenario spreadsheet using the Test Scenario designer.

Procedure

1. In the Test Scenario designer toolbar on the upper-right, click Import button.
2. In the Select file to Import prompt, click Choose File… and select the .CSV file you would like to import from your local file directory.
3. Click Import.

The .CSV file is imported to the Test Scenario designer.

WARNING

You must not modify the headers in the selected .CSV file. Otherwise, the spreadsheet may not be successfully imported.
CHAPTER 73. COVERAGE REPORTS FOR TEST SCENARIOS

The test scenario designer provides a clear and coherent way of displaying the test coverage statistics using the Coverage Report tab on the right side of the test scenario designer. You can also download the coverage report to view and analyze the test coverage statistics. Downloaded test scenario coverage report supports the .CSV file format. For more information about the RFC specification for the Comma-Separated Values (CSV) format, see Common Format and MIME Type for Comma-Separated Values (CSV) Files.

You can view the coverage report for rule-based and DMN-based test scenarios.

73.1. GENERATING COVERAGE REPORTS FOR RULE-BASED TEST SCENARIOS

In rule-based test scenarios, the Coverage Report tab contains the detailed information about the following:

- Number of available rules
- Number of fired rules
- Percentage of fired rules
- Percentage of executed rules represented as a pie chart
- Number of times each rule has executed
- The rules that are executed for each defined test scenario

Follow the procedure to generate a coverage report for rule-based test scenarios:

**Prerequisites**

- The rule-based test scenario template are created for the selected test scenario. For more information about creating rule-based test scenarios, see Section 64.1, “Creating a test scenario template for rule-based test scenarios”.
- The individual test scenarios are defined. For more information about defining a test scenario, see Chapter 66, Defining a test scenario.

**NOTE**

To generate the coverage report for rule-based test scenario, you must create at least one rule.

**Procedure**

1. Open the rule-based test scenarios in the test scenario designer.
2. Run the defined test scenarios.
3. Click Coverage Report on the right of the test scenario designer to display the test coverage statistics.
4. Optional: To download the test scenario coverage report, Click Download report.
73.2. Generating Coverage Reports for DMN-Based Test Scenarios

In DMN-based test scenarios, the **Coverage Report** tab contains the detailed information about the following:

- Number of available decisions
- Number of executed decisions
- Percentage of executed decisions
- Percentage of executed decisions represented as a pie chart
- Number of times each decision has executed
- Decisions that are executed for each defined test scenario

Follow the procedure to generate a coverage report for DMN-based test scenarios:

**Prerequisites**

- The DMN-based test scenario template is created for the selected test scenario. For more information about creating DMN-based test scenarios, see Section 65.1, “Creating a test scenario template for DMN-based test scenarios”.
- The individual test scenarios are defined. For more information about defining a test scenario, see Chapter 66, *Defining a test scenario*.

**Procedure**

1. Open the DMN-based test scenarios in the test scenario designer.
2. Run the defined test scenarios.
3. Click **Coverage Report** on the right of the test scenario designer to display the test coverage statistics.
4. Optional: To download the test scenario coverage report, Click **Download report**.
CHAPTER 74. EXECUTING A TEST SCENARIO USING THE KIE SERVER REST API

Directly interacting with the REST endpoints of KIE Server provides the most separation between the calling code and the decision logic definition. You can use the KIE Server REST API to execute the test scenarios externally. It executes the test scenarios against the deployed project.

NOTE

This functionality is disabled by default, use org.kie.scenariosimulation.server.ext.disabled system property to enable it.

For more information about the KIE Server REST API, see Interacting with Red Hat Decision Manager using KIE APIs.

Prerequisites

- KIE Server is installed and configured, including a known user name and credentials for a user with the kie-server role. For installation options, see Planning a Red Hat Decision Manager installation.
- You have built the project as a KJAR artifact and deployed it to KIE Server.
- You have the ID of the KIE container.

Procedure

1. Determine the base URL for accessing the KIE Server REST API endpoints. This requires knowing the following values (with the default local deployment values as an example):
   - Host (localhost)
   - Port (8080)
   - Root context (kie-server)
   - Base REST path (services/rest/)

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

http://localhost:8080/kie-server/services/rest/server/containers/traffic_1.0.0-SNAPSHOT

2. Determine user authentication requirements.
   When users are defined directly in the KIE Server configuration, HTTP Basic authentication is used and requires the user name and password. Successful requests require that the user have the kie-server role.

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

   curl -u username:password <request>

   If KIE Server is configured with Red Hat Single Sign-On, the request must include a bearer token:
3. Specify the format of the request and response. The REST API endpoints work with XML format and are set using request headers:

**XML**

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

4. Execute the test scenario:

```
[POST] server/containers/{containerId}/scesim
```

Example curl request:

```bash
curl -X POST "http://localhost:8080/kie-server/services/rest/server/containers/traffic_1.0.0-SNAPSHOT/scesim" -u 'wbadmin:wbadmin;' -H "accept: application/xml" -H "content-type: application/xml" -d @Violation.scesim
```

Example XML request:

```xml
<ScenarioSimulationModel version="1.8">
  <simulation>
    <scesimModelDescriptor>
      <factMappings>
        <FactMapping>
          <expressionElements/>
          <expressionIdentifier>
            <name>Index</name>
            <type>OTHER</type>
          </expressionIdentifier>
          <factIdentifier>
            <name>#{</name>
            <className>java.lang.Integer</className>
          </factIdentifier>
          <className>java.lang.Integer</className>
        </FactMapping>
        <FactMapping>
          <expressionElements/>
          <expressionIdentifier>
            <name>Description</name>
            <type>OTHER</type>
          </expressionIdentifier>
          <factIdentifier>
            <name>Scenario description</name>
            <className>java.lang.String</className>
          </factIdentifier>
          <className>java.lang.String</className>
        </FactMapping>
      </factMappings>
    </scesimModelDescriptor>
  </simulation>
</ScenarioSimulationModel>
```
<FactMapping>
  <expressionElements>
    <ExpressionElement>
      <step>Driver</step>
    </ExpressionElement>
    <ExpressionElement>
      <step>Points</step>
    </ExpressionElement>
  </expressionElements>
  <expressionIdentifier>
    <name>0|1</name>
    <type>GIVEN</type>
  </expressionIdentifier>
  <factIdentifier>
    <name>Driver</name>
    <className>Driver</className>
  </factIdentifier>
  <factAlias>Driver</factAlias>
  <expressionAlias>Points</expressionAlias>
  <factMappingValueType>NOT_EXPRESSION</factMappingValueType>
  <columnWidth>114.0</columnWidth>
</FactMapping>

<FactMapping>
  <expressionElements>
    <ExpressionElement>
      <step>Violation</step>
    </ExpressionElement>
    <ExpressionElement>
      <step>Type</step>
    </ExpressionElement>
  </expressionElements>
  <expressionIdentifier>
    <name>0|6</name>
    <type>GIVEN</type>
  </expressionIdentifier>
  <factIdentifier>
    <name>Violation</name>
    <className>Violation</className>
  </factIdentifier>
  <factAlias>Violation</factAlias>
  <expressionAlias>Type</expressionAlias>
  <factMappingValueType>NOT_EXPRESSION</factMappingValueType>
  <columnWidth>114.0</columnWidth>
</FactMapping>

<FactMapping>
  <expressionElements>
    <ExpressionElement>
      <step>Violation</step>
    </ExpressionElement>
    <ExpressionElement>
      <step>Speed Limit</step>
    </ExpressionElement>
  </expressionElements>
  <expressionIdentifier>
    <name>0|1</name>
    <type>GIVEN</type>
  </expressionIdentifier>
  <factIdentifier>
    <name>Driver</name>
    <className>Driver</className>
  </factIdentifier>
  <factAlias>Driver</factAlias>
  <expressionAlias>Points</expressionAlias>
  <factMappingValueType>NOT_EXPRESSION</factMappingValueType>
  <columnWidth>114.0</columnWidth>
</FactMapping>
<name>0|7</name>
<type>GIVEN</type>
</expressionIdentifier>
<factIdentifier>
    <name>Violation</name>
    <className>Violation</className>
</factIdentifier>
<className>number</className>
<factAlias>Violation</factAlias>
<expressionAlias>Speed Limit</expressionAlias>
<factMappingValueTyp>NOT_EXPRESSION</factMappingValueTyp>
<columnWidth>114.0</columnWidth>
</FactMapping>

<FactMapping>
<expressionElements>
    <ExpressionElement>
        <step>Violation</step>
    </ExpressionElement>
    <ExpressionElement>
        <step>Actual Speed</step>
    </ExpressionElement>
</expressionElements>
<expressionIdentifier>
    <name>0|8</name>
    <type>GIVEN</type>
</expressionIdentifier>
<className>number</className>
<factAlias>Violation</factAlias>
<expressionAlias>Actual Speed</expressionAlias>
<factMappingValueTyp>NOT_EXPRESSION</factMappingValueTyp>
<columnWidth>114.0</columnWidth>
</FactMapping>

<FactMapping>
<expressionElements>
    <ExpressionElement>
        <step>Fine</step>
    </ExpressionElement>
    <ExpressionElement>
        <step>Points</step>
    </ExpressionElement>
</expressionElements>
<expressionIdentifier>
    <name>0|11</name>
    <type>EXPECT</type>
</expressionIdentifier>
<className>number</className>
<factAlias>Fine</factAlias>
<expressionAlias>Points</expressionAlias>
<FactMapping>
  <expressionElements>
    <ExpressionElement>
      <step>Fine</step>
    </ExpressionElement>
    <ExpressionElement>
      <step>Amount</step>
    </ExpressionElement>
  </expressionElements>
  <expressionIdentifier>
    <name>0|12</name>
    <type>EXPECT</type>
  </expressionIdentifier>
</FactMapping>

<FactMapping>
  <expressionElements>
    <ExpressionElement>
      <step>Should the driver be suspended?</step>
    </ExpressionElement>
  </expressionElements>
  <expressionIdentifier>
    <name>0|13</name>
    <type>EXPECT</type>
  </expressionIdentifier>
</FactMapping>

<FactMappingValues>
  <FactMappingValue>
    <factIdentifier>
      <name>Scenario description</name>
      <className>java.lang.String</className>
    </factIdentifier>
  </FactMappingValue>
</FactMappingValues>
<expressionIdentifier>
  <name>Driver</name>
  <className>Driver</className>
</expressionIdentifier>
<expressionIdentifier>
  <name>0|1</name>
  <type>GIVEN</type>
</expressionIdentifier>
<rawValue class="string">10</rawValue>

<expressionIdentifier>
  <name>Violation</name>
  <className>Violation</className>
</expressionIdentifier>
<expressionIdentifier>
  <name>0|6</name>
  <type>GIVEN</type>
</expressionIdentifier>
<rawValue class="string">&quot;speed&quot;</rawValue>

<expressionIdentifier>
  <name>Violation</name>
  <className>Violation</className>
</expressionIdentifier>
<expressionIdentifier>
  <name>0|7</name>
  <type>GIVEN</type>
</expressionIdentifier>
<rawValue class="string">100</rawValue>

<expressionIdentifier>
  <name>Violation</name>
  <className>Violation</className>
</expressionIdentifier>
<expressionIdentifier>
  <name>0|8</name>
  <type>GIVEN</type>
</expressionIdentifier>
<rawValue class="string">120</rawValue>

<expressionIdentifier>
  <name>Fine</name>
  <className>Fine</className>
</expressionIdentifier>
<name>0|11</name> 
<type>EXPECT</type> 
</expressionIdentifier> 
<rawValue class="string">3</rawValue> </FactMappingValue> 
<FactMappingValue> 
<factIdentifier> 
<name>Fine</name> 
<className>Fine</className> </factIdentifier> 
<expressionIdentifier> 
<name>0|12</name> 
<type>EXPECT</type> 
</expressionIdentifier> 
<rawValue class="string">500</rawValue> </FactMappingValue> 
<FactMappingValue> 
<factIdentifier> 
<name>Should the driver be suspended?</name> 
<className>Should the driver be suspended?</className> </factIdentifier> 
<expressionIdentifier> 
<name>0|13</name> 
<type>EXPECT</type> 
</expressionIdentifier> 
<rawValue class="string">"No"</rawValue> </FactMappingValue> 
<FactMappingValue> 
<factIdentifier> 
<name>Index</name> 
<type>OTHER</type> 
</expressionIdentifier> 
<rawValue class="string">1</rawValue> </FactMappingValue> 
</factMappingValues> 
</Scenario> 
</scesimData> 
</simulation> 
</background> 
<scesimModelDescriptor> 
<factMappings> 
<FactMapping> 
<expressionElements/> 
<expressionIdentifier> 
<name>1|1</name> 
<type>GIVEN</type> 
</expressionIdentifier> 
<factIdentifier> 
<name>Empty</name> 
<className>java.lang.Void</className> </factIdentifier> 
</FactMapping> 
</factMappings> 
</scesimModelDescriptor>
Example XML response:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response type="SUCCESS" msg="Test Scenario successfully executed">
  <scenario-simulation-result>
    <run-count>5</run-count>
    <ignore-count>0</ignore-count>
    <run-time>31</run-time>
  </scenario-simulation-result>
</response>
```

Red Hat Decision Manager 7.10 Developing decision services in Red Hat Decision Manager
CHAPTER 75. CREATING TEST SCENARIO USING THE SAMPLE MORTGAGES PROJECT

This chapter illustrates creating and executing a test scenario from the sample Mortgages project shipped with Business Central using the test scenario designer. The test scenario example in this chapter is based on the Pricing loans guided decision table from the Mortgages project.

Procedure

1. In Business Central, go to Menu → Design → Projects and click Mortgages.
2. If the project is not listed under Projects, from MySpace, click Try Samples → Mortgages → OK. The Assets window appears.
3. Click Add Asset → Test Scenario.
4. Enter scenario_pricing_loans as the Test Scenario name and select the default mortgages.mortgages package from the Package drop-down list. The package you select must contain all the required rule assets.
5. Select RULE as the Source type.
6. Click Ok to create and open the test scenario in the test scenario designer.
7. Expand Project Explorer and verify the following:
   - Applicant, Bankruptcy, IncomeSource, and LoanApplication data objects exist.
   - Pricing loans guided decision table exists.
   - Verify that the new test scenario is listed under Test Scenario
8. After verifying that everything is in place, return to the Model tab of the test scenario designer and define the GIVEN and EXPECT data for the scenario, based on the available data objects.

Figure 75.1. A blank test scenario designer

9. Define the GIVEN column details:

```plaintext
a. Click the cell named INSTANCE 1 under the GIVEN column header.
```
a. Click the cell named INSTANCED 1 under the GIVEN column header.

b. From the Test Tools panel, select the LoanApplication data object.

c. Click Insert Data Object.

10. To create properties for the data object, right-click the property header cell and select Insert column right or Insert column left as required. For this example, you need to create two more property cells under the GIVEN column.

11. Select the first property header cell:

   a. From the Test Tools panel, select and expand the LoanApplication data object.

   b. Click amount.

   c. Click Insert Data Object to map the data object field to the property header cell.

12. Select the second property header cell:

   a. From the Test Tools panel, select and expand the LoanApplication data object.

   b. Click deposit.

   c. Click Insert Data Object

13. Select the third property header cell:

   a. From the Test Tools panel, select and expand the LoanApplication data object.

   b. Click lengthYears

   c. Click Insert Data Object

14. Right-click the LoanApplication header cell and select Insert column right. A new GIVEN column to the right is created.

15. Select the new header cell:

   a. From the Test Tools panel, select the IncomeSource data object.

   b. Click Insert Data Object to map the data object to the header cell.

16. Select the property header cell below IncomeSource:

   a. From the Test Tools panel, select and expand the IncomeSource data object.

   b. Click type.

   c. Click Insert Data Object to map the data object field to the property header cell.

   You have now defined all the GIVEN column cells.

17. Next, define the EXPECT column details:

   a. Click the cell named INSTANCE 2 under the EXPECT column header.

   b. From the Test Tools panel, select LoanApplication data object.

   c. Click Insert Data Object
18. To create properties for the data object, right-click the property header cell and select **Insert column right** or **Insert column left** as required. Create two more property cells under the **EXPECT** column.

19. Select the first property header cell:
   a. From the **Test Tools** panel, select and expand the **LoanApplication** data object.
   b. Click **approved**.
   c. Click **Insert Data Object** to map the data object field to the property header cell.

20. Select the second property header cell:
   a. From the **Test Tools** panel, select and expand the **LoanApplication** data object.
   b. Click **insuranceCost**.
   c. Click **Insert Data Object** to map the data object field to the property header cell.

21. Select the third property header cell:
   a. From the **Test Tools** panel, select and expand the **LoanApplication** data object.
   b. Click **approvedRate**.
   c. Click **Insert Data Object** to map the data object field to the property header cell.

22. To define the test scenario, enter the following data in the first row:
   - Enter **Row 1 test scenario** as the **Scenario Description**, **150000** as the **amount**, **19000** as the **deposit**, **30** as the **lengthYears**, and **Asset** as the **type** for the **GIVEN** column values.
   - Enter **true** as **approved**, **0** as the **insuranceCost** and **2** as the **approvedRate** for the **EXPECT** column values.

23. Next enter the following data in the second row:
   - Enter **Row 2 test scenario** as the **Scenario Description**, **100002** as the **amount**, **2999** as the **deposit**, **20** as the **lengthYears**, and **Job** as the **type** for the **GIVEN** column values.
   - Enter **true** as **approved**, **10** as the **insuranceCost** and **6** as the **approvedRate** for the **EXPECT** column values.

24. After you have defined all **GIVEN**, **EXPECT**, and other data for the scenario, click **Save** in the test scenario designer to save your work.

25. Click **Run Test** in the upper-right corner to run the **.scesim** file.
   The test result is displayed in the **Test Report** panel. Click **View Alerts** to display messages from the **Alerts** section. If a test fails, refer to the messages in the **Alerts** section at the bottom of the window, review and correct all components in the scenario, and try again to validate the scenario until the scenario passes.

26. Click **Save** in the test scenario designer to save your work after you have made all necessary changes.
CHAPTER 76. TEST SCENARIOS (LEGACY) DESIGNER IN BUSINESS CENTRAL

Red Hat Decision Manager currently supports both the new Test Scenarios designer and the former Test Scenarios (Legacy) designer. The default designer is the new test scenarios designer, which supports testing of both rules and DMN models and provides an enhanced overall user experience with test scenarios. If required, you can continue to use the legacy test scenarios designer, which supports rule-based test scenarios only.

76.1. CREATING AND RUNNING A TEST SCENARIO (LEGACY)

You can create test scenarios in Business Central to test the functionality of business rule data before deployment. A basic test scenario must have at least the following data:

- Related data objects
- GIVEN facts
- EXPECT results

**NOTE**

The legacy test scenarios designer supports the LocalDate java built-in data type. You can use the LocalDate java built-in data type in the dd-mmm-yyyy date format. For example, you can set this in the 17-Oct-2020 date format.

With this data, the test scenario can validate the expected and actual results for that rule instance based on the defined facts. You can also add a CALL METHOD and any available globals to a test scenario, but these scenario settings are optional.

**Procedure**

1. In Business Central, go to Menu → Design → Projects and click the project name.
2. Click Add Asset → Test Scenarios (Legacy).
3. Enter an informative Test Scenario name and select the appropriate Package. The package that you specify must be the same package where the required rule assets have been assigned or will be assigned. You can import data objects from any package into the asset’s designer.
4. Click Ok to create the test scenario.
   The new test scenario is now listed in the Test Scenarios panel of the Project Explorer,
5. Click the Data Objects tab to verify that all data objects required for the rules that you want to test are listed. If not, click New item to import the needed data objects from other packages, or create data objects within your package.
6. After all data objects are in place, return to the Model tab of the test scenarios designer and define the GIVEN and EXPECT data for the scenario, based on the available data objects.
The GIVEN section defines the input facts for the test. For example, if an Underage rule in the project declines loan applications for applicants under the age of 21, then the GIVEN facts in the test scenario could be Applicant with age set to some integer less than 21.

The EXPECT section defines the expected results based on the GIVEN input facts. That is, GIVEN the input facts, EXPECT these other facts to be valid or entire rules to be activated. For example, with the given facts of an applicant under the age of 21 in the scenario, the EXPECT results could be LoanApplication with approved set to false (as a result of the underage applicant), or could be the activation of the Underage rule as a whole.

7. Optional: Add a CALL METHOD and any globals to the test scenario:

- **CALL METHOD**: Use this to invoke a method from another fact when the rule execution is initiated. Click CALL METHOD, select a fact, and click to select the method to invoke. You can invoke any Java class methods (such as methods from an ArrayList) from the Java library or from a JAR that was imported for the project (if applicable).

- **globals**: Use this to add any global variables in the project that you want to validate in the test scenario. Click globals to select the variable to be validated, and then in the test scenarios designer, click the global name and define field values to be applied to the global variable. If no global variables are available, then they must be created as new assets in Business Central. Global variables are named objects that are visible to the decision engine but are different from the objects for facts. Changes in the object of a global do not trigger the re-evaluation of rules.
8. Click More at the bottom of the test scenarios designer to add other data blocks to the same scenario file as needed.

9. After you have defined all GIVEN, EXPECT, and other data for the scenario, click Save in the test scenarios designer to save your work.

10. Click Run scenario in the upper-right corner to run this .scenario file, or click Run all scenarios to run all saved .scenario files in the project package (if there are multiple). Although the Run scenario option does not require the individual .scenario file to be saved, the Run all scenarios option does require all .scenario files to be saved.
   If the test fails, address any problems described in the Alerts message at the bottom of the window, review all components in the scenario, and try again to validate the scenario until the scenario passes.

11. Click Save in the test scenarios designer to save your work after all changes are complete.

### 76.1.1. Adding GIVEN facts in test scenarios (legacy)

The GIVEN section defines input facts for the test. For example, if an Underage rule in the project declines loan applications for applicants under the age of 21, then the GIVEN facts in the test scenario could be Applicant with age set to some integer less than 21.

**Prerequisites**

- All data objects required for your test scenario have been created or imported and are listed in the Data Objects tab of the Test Scenarios (Legacy) designer.

**Procedure**

1. In the Test Scenarios (Legacy) designer, click GIVEN to open the New input window with the available facts.

**Figure 76.2. Add GIVEN input to the test scenario**

![New Input Window](image)

The list includes the following options, depending on the data objects available in the Data Objects tab of the test scenarios designer:

- **Insert a new fact:** Use this to add a fact and modify its field values. Enter a variable for the fact as the Fact name.
- **Modify an existing fact:** (Appears only after another fact has been added.) Use this to specify a previously inserted fact to be modified in the decision engine between executions of the scenario.

- **Delete an existing fact:** (Appears only after another fact has been added.) Use this to specify a previously inserted fact to be deleted from the decision engine between executions of the scenario.

- **Activate rule flow group:** Use this to specify a rule flow group to be activated so that all rules within that group can be tested.

2. Choose a fact for the desired input option and click **Add.** For example, set **Insert a new fact:** to **Applicant** and enter a or app or any other variable for the **Fact name.**

3. Click the fact in the test scenarios designer and select the field to be modified.

4. Click the edit icon (✎) and select from the following field values:
   - **Literal value:** Creates an open field in which you enter a specific literal value.
   - **Bound variable:** Sets the value of the field to the fact bound to a selected variable. The field type must match the bound variable type.
   - **Create new fact:** Enables you to create a new fact and assign it as a field value of the parent fact. Then you can click the child fact in the test scenarios designer and likewise assign field values or nest other facts similarly.

5. Continue adding any other **GIVEN** input data for the scenario and click **Save** in the test scenarios designer to save your work.

### 76.1.2. Adding EXPECT results in test scenarios (legacy)

The **EXPECT** section defines the expected results based on the **GIVEN** input facts. That is, **GIVEN** the input facts, **EXPECT** other specified facts to be valid or entire rules to be activated. For example, with the given facts of an applicant under the age of 21 in the scenario, the **EXPECT** results could be **LoanApplication** with **approved** set to false (as a result of the underage applicant), or could be the activation of the **Underage** rule as a whole.

**Prerequisites**

- All data objects required for your test scenario have been created or imported and are listed in the **Data Objects** tab of the **Test Scenarios (Legacy)** designer.
Procedure

1. In the Test Scenarios (Legacy) designer, click EXPECT to open the New expectation window with the available facts.

Figure 76.4. Add EXPECT results to the test scenario

![New expectation](image)

The list includes the following options, depending on the data in the GIVEN section and the data objects available in the Data Objects tab of the test scenarios designer:

- **Rule:** Use this to specify a particular rule in the project that is expected to be activated as a result of the GIVEN input. Type the name of a rule that is expected to be activated or select it from the list of rules, and then in the test scenarios designer, specify the number of times the rule should be activated.

- **Fact value:** Use this to select a fact and define values for it that are expected to be valid as a result of the facts defined in the GIVEN section. The facts are listed by the Fact name previously defined for the GIVEN input.

- **Any fact that matches:** Use this to validate that at least one fact with the specified values exists as a result of the GIVEN input.

2. Choose a fact for the desired expectation (such as Fact value: application) and click Add or OK.

3. Click the fact in the test scenarios designer and select the field to be added and modified.

Figure 76.5. Modify a fact field

![Choose a field to add](image)

4. Set the field values to what is expected to be valid as a result of the GIVEN input (such as approved | equals | false).
NOTE

In the legacy test scenarios designer, you can use ["value1", "value2"] string format in the EXPECT field to validate the list of strings.

5. Continue adding any other EXPECT input data for the scenario and click Save in the test scenarios designer to save your work.

6. After you have defined and saved all GIVEN, EXPECT, and other data for the scenario, click Run scenario in the upper-right corner to run this .scenario file, or click Run all scenarios to run all saved .scenario files in the project package (if there are multiple). Although the Run scenario option does not require the individual .scenario file to be saved, the Run all scenarios option does require all .scenario files to be saved.

   If the test fails, address any problems described in the Alerts message at the bottom of the window, review all components in the scenario, and try again to validate the scenario until the scenario passes.

7. Click Save in the test scenarios designer to save your work after all changes are complete.
CHAPTER 77. FEATURE COMPARISON OF LEGACY AND NEW TEST SCENARIO DESIGNER

Red Hat Decision Manager supports both the new test scenario designer and the former test scenario (Legacy) designer.

The default designer is the new test scenario designer, which supports testing of both rules and DMN models, and provides an enhanced overall user experience with test scenarios. You can continue to use the legacy test scenario designer, which only supports rule-based test scenarios.

**IMPORTANT**

The new test scenario designer has an improved layout and feature set and continues to be developed. However, the legacy test scenario designer is deprecated with Red Hat Decision Manager 7.3.0 and will be removed in a future Red Hat Decision Manager release.

The following table highlights the main features of legacy and new test scenario designer, which are supported in Red Hat Decision Manager to help you decide a suitable test scenario designer in your project.

- + indicates that the feature is present in the test scenario designer.
- - indicates that the feature is not present in the test scenario designer.

<table>
<thead>
<tr>
<th>Feature &amp; highlights</th>
<th>New designer</th>
<th>Legacy designer</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating and running a test scenario</td>
<td>+</td>
<td>+</td>
<td>- For more information about creating rule and DMN-based test scenarios, see Chapter 64, Test scenario template.</td>
</tr>
<tr>
<td>- You can create test scenarios in Business Central to test the functionality of business rule data before deployment.</td>
<td></td>
<td></td>
<td>- For more information about running the test scenarios, see Chapter 70, Running the test scenarios.</td>
</tr>
<tr>
<td>- A basic test scenario must have at least a related data objects, GIVEN facts, and EXPECT results.</td>
<td></td>
<td></td>
<td>- For more information about creating and running test scenarios (legacy), see Section 76.1, “Creating and running a test scenario (legacy)”</td>
</tr>
</tbody>
</table>
### Feature & highlights

<table>
<thead>
<tr>
<th>Feature &amp; highlights</th>
<th>New designer</th>
<th>Legacy designer</th>
<th>Documentation</th>
</tr>
</thead>
</table>
| Adding GIVEN facts in test scenarios | + | + | - For more information about adding GIVEN facts in new test scenario designer, see Chapter 64, Test scenario template.  
- For more information about adding GIVEN facts in test scenarios (legacy), see Section 76.1.1, “Adding GIVEN facts in test scenarios (legacy)”.
| Adding EXPECT results in test scenarios | + | + | - For more information about adding EXPECT results in new test scenario designer, see Chapter 64, Test scenario template.  
- For more information about adding EXPECT results in test scenarios (legacy), see Section 76.1.2, “Adding EXPECT results in test scenarios (legacy)”.
<p>| KIE session | + | + | NA |
| KIE base on test scenario level | - | + | NA |
| KIE base on project level | + | + | NA |</p>
<table>
<thead>
<tr>
<th>Feature &amp; highlights</th>
<th>New designer</th>
<th>Legacy designer</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulated date and time</td>
<td>-</td>
<td>+</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>● You can set a simulated date and time for the legacy test scenario designer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule flow group</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● You can specify a rule flow group to be activated to test all the rules within that group.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global variables</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● Global variables are named objects that are visible to the decision engine but are different from the objects for facts.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>● Setting global variables for new test scenario is deprecated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>● In case you want to reuse data sets for different scenarios, you can use the Background instance.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>● For more information about setting rule flow group in new test scenarios, see Section 63.9.1, “Configuring global settings for rule-based test scenarios”.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>● For more information about setting rule flow group in test scenarios (legacy), Section 76.1.1, “Adding GIVEN facts in test scenarios (legacy)”.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>● For more information about Background instance in new test scenarios, see Chapter 67, Background instance in test scenarios.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>● For more information about global variables in test scenarios (legacy), see Section 76.1, “Creating and running a test scenario (legacy)”.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Call method
- You can use this to invoke a method from another fact when the rule execution is initiated.
- You can invoke any Java class methods from the Java library or from a JAR that was imported for the project.

### Modify an existing fact
- You can modify a previously inserted fact in the decision engine between executions of the scenario.

### Bound variable
- You can set the value of a field to the fact bound to a selected variable.
- In the new test scenario designer, you can not define a variable inside a test scenario grid and reuse it inside GIVEN or EXPECTED cells.

### Documentation
- For more information about calling a method in new test scenarios, see Chapter 69, *Expression syntax in test scenarios*.
- For more information about calling a method in test scenarios (legacy), see Section 76.1, “Creating and running a test scenario (legacy)”.

- For more information about modifying an existing fact in test scenarios (legacy), see Section 76.1.1, “Adding GIVEN facts in test scenarios (legacy)”.

- For more information about how to set bound variables in test scenarios (legacy), see Section 76.1.1, “Adding GIVEN facts in test scenarios (legacy)”.

---

<table>
<thead>
<tr>
<th>Feature &amp; highlights</th>
<th>New designer</th>
<th>Legacy designer</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call method</strong></td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Modify an existing fact</strong></td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bound variable</strong></td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 78. NEXT STEPS

Packaging and deploying a Red Hat Decision Manager project
PART IX. DECISION ENGINE IN RED HAT DECISION MANAGER

As a business rules developer, your understanding of the decision engine in Red Hat Decision Manager can help you design more effective business assets and a more scalable decision management architecture. The decision engine is the Red Hat Decision Manager component that stores, processes, and evaluates data to execute business rules and to reach the decisions that you define. This document describes basic concepts and functions of the decision engine to consider as you create your business rule system and decision services in Red Hat Decision Manager.
CHAPTER 79. DECISION ENGINE IN RED HAT DECISION MANAGER

The decision engine is the rules engine in Red Hat Decision Manager. The decision engine stores, processes, and evaluates data to execute the business rules or decision models that you define. The basic function of the decision engine is to match incoming data, or facts, to the conditions of rules and determine whether and how to execute the rules.

The decision 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 decision engine that the decision engine matches to rule conditions to execute applicable rules.

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

- **Working memory**: Location where facts are stored in the decision 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 Red Hat Decision Manager, that information is inserted into the working memory of the decision engine in the form of one or more facts. The decision 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 decision engine activates and registers rules in the agenda, where the decision engine then sorts prioritized or conflicting rules in preparation for execution.

The following diagram illustrates these basic components of the decision engine:

**Figure 79.1. Overview of basic decision engine components**

For more details and examples of rule and fact behavior in the decision engine, see Chapter 81, *Inference and truth maintenance in the decision engine*.

These core concepts can help you to better understand other more advanced components, processes, and sub-processes of the decision engine, and as a result, to design more effective business assets in Red Hat Decision Manager.
CHAPTER 80. KIE SESSIONS

In Red Hat Decision Manager, 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

```xml
<kmodule>
  ...
  <kbase>
    ...
    <ksession name="KSession2_1" type="stateless" default="true" clockType="realtime">
      ...
    </ksession>
    ...
  </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 rules and other business assets in Red Hat Decision Manager, but does not contain any runtime data.

Example KIE base configuration in a kmodule.xml file

```xml
<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.

80.1. 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

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

Sample DRL rule for driver’s license application

```java
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 decision engine, the decision 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.

NOTE

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 Red Hat Decision Manager project are built with the following code:

Create the KIE container

```java
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**

```java
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**

```drl
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**

```java
import com.company.license.Applicant;
import com.company.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
}
```
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.

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

`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`.

### 80.1.1. 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 sessionglobals. 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**

```java
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");
```

### 80.2. 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.

**WARNING**

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**

```java
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

guard

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

guard

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

guard

rule "Status output when things are ok"
when
```
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 decision engine of changes. The decision engine then reasons over the changes and assesses impact on subsequent rule executions. This process is part of the decision 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 Red Hat Decision Manager project are built with the following code:

**Create the KIE container**

```java
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**

```java
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**

```bash
> Everything is ok
```

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

Enter new data to trigger rules

```java
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

```java
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).

**80.3. 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**

```java
// Obtain a KIE session pool from the KIE container
KieContainerSessionsPool pool = kContainer.newKieSessionsPool(10);
```
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.
CHAPTER 81. INFERENCE AND TRUTH MAINTENANCE IN THE DECISION ENGINE

The basic function of the decision 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 decision 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 decision 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 decision engine asserts data as a result of executing a rule, the decision 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 decision engine chooses the action based on assumptions from previously calculated conclusions.

The decision 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 decision 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
```

Red Hat Decision Manager 7.10 Developing decision services in Red Hat Decision Manager
These rules are not easily maintained in the decision 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**
```
rule "Infer Child"
when
  $p : Person(age < 18)
then
  insertLogical(new IsChild($p))
end
```

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

### IMPORTANT

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 decision engine supports chaining of logical insertions for a cascading set of retracts.

### Rules to issue bus pass, logical insertion

**Rule: Issue Child Bus Pass**
```
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 decision 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 81.1. Stated insertion

```
Inserted fact

Does the object already exist?

No

Is there an existing equal object?

No

Is the object stated or justified?

Stated

Return new FactHandle.

Yes

Justified

Discard logical assertion?

Yes

Override justification and set to stated, set existing handle to new object.

No

Override justification and set to stated, remove justifications and return existing FactHandle.

Yes

Justified

Is the object stated or justified?

Stated

Return existing FactHandle.

No

Yes

Justified
```
When the decision engine logically inserts an object during a rule execution, the decision 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.

### 81.1. FACT EQUALITY MODES IN THE DECISION ENGINE

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

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

- **equality**: The decision engine uses a `HashMap` to store all inserted facts. The decision 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 decision 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**

```java
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 decision 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**

```java
ksession.insert(p1);
ksession.getFactHandle(p1);
```

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

```java
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:

```java
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 Red Hat Decision Manager project:

```xml
<kmodule>
    ...
    <kbase name="KBase2" default="false" equalsBehavior="equality" packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
        ...
    </kbase>
    ...
</kmodule>
```
CHAPTER 82. EXECUTION CONTROL IN THE DECISION ENGINE

When new rule data enters the working memory of the decision 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 decision engine creates an activation instance, referencing the rule and the matched facts, and adds the activation onto the decision 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 decision engine cycles repeatedly through two phases:

- **Agenda evaluation.** In this phase, the decision engine selects all rules that can be executed. If no executable rules exist, the execution cycle ends. If an executable rule is found, the decision 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 decision 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 decision engine returns to the agenda evaluation phase to reassess rules.

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 decision engine. Some common methods for defining rule execution order are by using rule salience, agenda groups, activation groups, or rule units for DRL rule sets.

### 82.1. 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 decision engine stack in the order shown:

```drl
rule "RuleA"
salience 95
when
```
The `RuleB` rule is listed second, but it has a higher salience value than the `RuleA` rule and is therefore executed first.

### 82.2. 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 decision 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 decision 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 decision engine stack in the order shown:

#### Sample DRL rules for banking application

```drl
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()
```

---

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

82.3. 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 decision 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.

82.4. RULE EXECUTION MODES AND THREAD SAFETY IN THE DECISION ENGINE

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

- **Passive mode**: (Default) The decision engine evaluates rules when a user or an application explicitly calls `fireAllRules()`. Passive mode in the decision 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 decision engine.

  Example CEP application code with the decision engine in passive mode

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

  Example CEP application code with the decision 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() {
      ...
    }
  });
  ```
This example calls `fireUntilHalt()` from a dedicated execution thread to prevent the current thread from being blocked indefinitely while the decision 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 decision 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 decision 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 decision engine is running in active mode following a `fireUntilHalt()` call and you call `fireAllRules()`, the `fireAllRules()` call is ignored and the decision engine continues to run in active mode until you call `halt()`.

For added thread safety in active mode, the decision 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**

```java
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 );
    }
} );
```

Red Hat Decision Manager 7.10 Developing decision services in Red Hat Decision Manager

568
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 decision engine to consider the insertions as an atomic operation and to wait until all the insertions are complete before evaluating the rules again.

### 82.5. FACT PROPAGATION MODES IN THE DECISION ENGINE

The decision engine supports the following fact propagation modes that determine how the decision 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 decision 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 decision 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 decision 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

```plaintext
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

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

However, due to the default lazy propagation behavior in Phreak, the decision 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 decision 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

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

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

82.6. AGENDA EVALUATION FILTERS

The decision 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 decision engine agenda.

Example agenda filter definition

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

82.7. RULE UNITS IN DRL RULE SETS

Rule units are groups of data sources, global variables, and DRL rules that function together for a specific purpose. You can use rule units to partition a rule set into smaller units, bind different data sources to those units, and then execute the individual unit. Rule units are an enhanced alternative to rule-grouping DRL attributes such as rule agenda groups or activation groups for execution control.

Rule units are helpful when you want to coordinate rule execution so that the complete execution of one...
rule unit triggers the start of another rule unit and so on. For example, assume that you have a set of rules for data enrichment, another set of rules that processes that data, and another set of rules that extract the output from the processed data. If you add these rule sets into three distinct rule units, you can coordinate those rule units so that complete execution of the first unit triggers the start of the second unit and the complete execution of the second unit triggers the start of third unit.

To define a rule unit, implement the `RuleUnit` interface as shown in the following example:

**Example rule unit class**

```java
package org.mypackage.myunit;

public static class AdultUnit implements RuleUnit {
    private int adultAge;
    private DataSource<Person> persons;

    public AdultUnit() {
    }

    public AdultUnit(DataSource<Person> persons, int age) {
        this.persons = persons;
        this.age = age;
    }

    // A data source of `Persons` in this rule unit:
    public DataSource<Person> getPersons() {
        return persons;
    }

    // A global variable in this rule unit:
    public int getAdultAge() {
        return adultAge;
    }

    // Life-cycle methods:
    @Override
    public void onStart() {
        System.out.println("AdultUnit started.");
    }

    @Override
    public void onEnd() {
        System.out.println("AdultUnit ended.");
    }
}
```

In this example, `persons` is a source of facts of type `Person`. A rule unit data source is a source of the data processed by a given rule unit and represents the entry point that the decision engine uses to evaluate the rule unit. The `adultAge` global variable is accessible from all the rules belonging to this rule unit. The last two methods are part of the rule unit life cycle and are invoked by the decision engine.

The decision engine supports the following optional life-cycle methods for rule units:

**Table 82.1. Rule unit life-cycle methods**
### Method Invoked when

<table>
<thead>
<tr>
<th>Method</th>
<th>Invoked when</th>
</tr>
</thead>
<tbody>
<tr>
<td>onStart()</td>
<td>Rule unit execution starts</td>
</tr>
<tr>
<td>onEnd()</td>
<td>Rule unit execution ends</td>
</tr>
<tr>
<td>onSuspend()</td>
<td>Rule unit execution is suspended (used only with runUntilHalt())</td>
</tr>
<tr>
<td>onResume()</td>
<td>Rule unit execution is resumed (used only with runUntilHalt())</td>
</tr>
<tr>
<td>onYield(RuleUnit other)</td>
<td>The consequence of a rule in the rule unit triggers the execution of a different rule unit</td>
</tr>
</tbody>
</table>

You can add one or more rules to a rule unit. By default, all the rules in a DRL file are automatically associated with a rule unit that follows the naming convention of the DRL file name. If the DRL file is in the same package and has the same name as a class that implements the `RuleUnit` interface, then all of the rules in that DRL file implicitly belong to that rule unit. For example, all the rules in the `AdultUnit.drl` file in the `org.mypackage.myunit` package are automatically part of the rule unit `org.mypackage.myunit.AdultUnit`.

To override this naming convention and explicitly declare the rule unit that the rules in a DRL file belong to, use the `unit` keyword in the DRL file. The `unit` declaration must immediately follow the package declaration and contain the name of the class in that package that the rules in the DRL file are part of.

#### Example rule unit declaration in a DRL file

```drl
package org.mypackage.myunit
unit AdultUnit

rule Adult
    when
        $p : Person(age >= adultAge) from persons
    then
        System.out.println($p.getName() + " is adult and greater than "+ adultAge);
end
```

**WARNING**

Do not mix rules with and without a rule unit in the same KIE base. Mixing two rule paradigms in a KIE base results in a compilation error.

You can also rewrite the same pattern in a more convenient way using OOPPath notation, as shown in the following example:

#### Example rule unit declaration in a DRL file that uses OOPPath notation
package org.mypackage.myunit
unit AdultUnit

rule Adult
when
  $p : /persons[age >= adultAge]
then
  System.out.println($p.getName() + " is adult and greater than " + adultAge);
end

NOTE

OOPath is an object-oriented syntax extension of XPath that is designed for browsing graphs of objects in DRL rule condition constraints. OOPath uses the compact notation from XPath for navigating through related elements while handling collections and filtering constraints, and is specifically useful for graphs of objects.

In this example, any matching facts in the rule conditions are retrieved from the persons data source defined in the DataSource definition in the rule unit class. The rule condition and action use the adultAge variable in the same way that a global variable is defined at the DRL file level.

To execute one or more rule units defined in a KIE base, create a new RuleUnitExecutor class bound to the KIE base, create the rule unit from the relevant data source, and run the rule unit executor:

Example rule unit execution

```java
// Create a `RuleUnitExecutor` class and bind it to the KIE base:
KieBase kbase = kieContainer.getKieBase();
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );

// Create the `AdultUnit` rule unit using the `persons` data source and run the executor:
RuleUnit adultUnit = new AdultUnit(persons, 18);
executor.run( adultUnit );
```

Rules are executed by the RuleUnitExecutor class. The RuleUnitExecutor class creates KIE sessions and adds the required DataSource objects to those sessions, and then executes the rules based on the RuleUnit that is passed as a parameter to the run() method.

The example execution code produces the following output when the relevant Person facts are inserted in the persons data source:

Example rule unit execution output

```
org.mypackage.myunit.AdultUnit started.
Jane is adult and greater than 18
John is adult and greater than 18
org.mypackage.myunit.AdultUnit ended.
```

Instead of explicitly creating the rule unit instance, you can register the rule unit variables in the executor and pass to the executor the rule unit class that you want to run, and then the executor creates an instance of the rule unit. You can then set the DataSource definition and other variables as needed before running the rule unit.

Alternate rule unit execution option with registered variables

```java
```
The name that you pass to the `RuleUnitExecutor.bindVariable()` method is used at run time to bind the variable to the field of the rule unit class with the same name. In the previous example, the `RuleUnitExecutor` inserts into the new rule unit the data source bound to the "persons" name and inserts the value 18 bound to the String "adultAge" into the fields with the corresponding names inside the `AdultUnit` class.

To override this default variable-binding behavior, use the `@UnitVar` annotation to explicitly define a logical binding name for each field of the rule unit class. For example, the field bindings in the following class are redefined with alternative names:

```java
class AdultUnit implements RuleUnit {
    @UnitVar("minAge")
    private int adultAge = 18;

    @UnitVar("data")
    private DataSource<Person> persons;
}
```

You can then bind the variables to the executor using those alternative names and run the rule unit:

```java
executor.bindVariable( "data", persons );
.executor.bindVariable( "minAge", 18 );
executor.run( AdultUnit.class );
```

You can execute a rule unit in passive mode by using the `run()` method (equivalent to invoking `fireAllRules()` on a KIE session) or in active mode using the `runUntilHalt()` method (equivalent to invoking `fireUntilHalt()` on a KIE session). By default, the decision engine runs in passive mode and evaluates rule units only when a user or an application explicitly calls `run()` (or `fireAllRules()` for standard rules). If a user or application calls `runUntilHalt()` for rule units (or `fireUntilHalt()` for standard rules), the decision engine starts in active mode and evaluates rule units continually until the user or application explicitly calls `halt()`.

If you use the `runUntilHalt()` method, invoke the method on a separate execution thread to avoid blocking the main thread:

```java
new Thread( () -> executor.runUntilHalt( adultUnit ) ).start();
```

82.7.1. Data sources for rule units

A rule unit data source is a source of the data processed by a given rule unit and represents the entry point that the decision engine uses to evaluate the rule unit. A rule unit can have zero or more data
sources and each **DataSource** definition declared inside a rule unit can correspond to a different entry point into the rule unit executor. Multiple rule units can share a single data source, but each rule unit must use different entry points through which the same objects are inserted.

You can create a **DataSource** definition with a fixed set of data in a rule unit class, as shown in the following example:

**Example data source definition**

```java
DataSource<Person> persons = DataSource.create( new Person( "John", 42 ),
        new Person( "Jane", 44 ),
        new Person( "Sally", 4 ) );
```

Because a data source represents the entry point of the rule unit, you can insert, update, or delete facts in a rule unit:

**Example code to insert, modify, and delete a fact in a rule unit**

```java
// Insert a fact:
Person john = new Person( "John", 42 );
FactHandle johnFh = persons.insert( john );

// Modify the fact and optionally specify modified properties (for property reactivity):
john.setAge( 43 );
persons.update( johnFh, john, "age" );

// Delete the fact:
persons.delete( johnFh );
```

**82.7.2. Rule unit execution control**

Rule units are helpful when you want to coordinate rule execution so that the execution of one rule unit triggers the start of another rule unit and so on.

To facilitate rule unit execution control, the decision engine supports the following rule unit methods that you can use in DRL rule actions to coordinate the execution of rule units:

- **drools.run()**: Triggers the execution of a specified rule unit class. This method imperatively interrupts the execution of the rule unit and activates the other specified rule unit.

- **drools.guard()**: Prevents (guards) a specified rule unit class from being executed until the associated rule condition is met. This method declaratively schedules the execution of the other specified rule unit. When the decision engine produces at least one match for the condition in the guarding rule, the guarded rule unit is considered active. A rule unit can contain multiple guarding rules.

As an example of the **drools.run()** method, consider the following DRL rules that each belong to a specified rule unit. The **NotAdult** rule uses the **drools.run( AdultUnit.class )** method to trigger the execution of the **AdultUnit** rule unit:

**Example DRL rules with controlled execution using drools.run()**

```java
package org.mypackage.myunit
unit AdultUnit
```
rule Adult
  when
  Person(age >= 18, $name : name) from persons
  then
  System.out.println($name + " is adult");
end

package org.mypackage.myunit
unit NotAdultUnit

rule NotAdult
  when
  $p : Person(age < 18, $name : name) from persons
  then
  System.out.println($name + " is NOT adult");
  modify($p) { setAge(18); }
  drools.run( AdultUnit.class );
end

The example also uses a RuleUnitExecutor class created from the KIE base that was built from these rules and a DataSource definition of persons bound to it:

Example rule executor and data source definitions

```java
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );
DataSource<Person> persons = executor.newDataSource( "persons",
  new Person( "John", 42 ),
  new Person( "Jane", 44 ),
  new Person( "Sally", 4 ) );
```

In this case, the example creates the DataSource definition directly from the RuleUnitExecutor class and binds it to the "persons" variable in a single statement.

The example execution code produces the following output when the relevant Person facts are inserted in the persons data source:

Example rule unit execution output

```
Sally is NOT adult
John is adult
Jane is adult
Sally is adult
```

The NotAdult rule detects a match when evaluating the person "Sally", who is under 18 years old. The rule then modifies her age to 18 and uses the drools.run( AdultUnit.class ) method to trigger the execution of the AdultUnit rule unit. The AdultUnit rule unit contains a rule that can now be executed for all of the 3 persons in the DataSource definition.

As an example of the drools.guard() method, consider the following BoxOffice class and BoxOfficeUnit rule unit class:

Example BoxOffice class

```java
public class BoxOffice {
```
The example also uses the following `BoxOfficeUnit` rule unit class to keep selling box office tickets for the event as long as at least one box office is open. This rule unit uses `DataSource` definitions of `persons` and `tickets`:

**Example BoxOfficeUnit rule unit class**

```java
private boolean open;

public BoxOffice( boolean open ) {
    this.open = open;
}

public boolean isOpen() {
    return open;
}

public void setOpen( boolean open ) {
    this.open = open;
}
```

```java
public BoxOfficeUnit implements RuleUnit {
    private DataSource<BoxOffice> boxOffices;

    public DataSource<BoxOffice> getBoxOffices() {
        return boxOffices;
    }
}
```

The example also uses the following `TicketIssuerUnit` rule unit class to keep selling box office tickets for the event as long as at least one box office is open. This rule unit uses `DataSource` definitions of `persons` and `tickets`:

**Example TicketIssuerUnit rule unit class**

```java
public class TicketIssuerUnit implements RuleUnit {
    private DataSource<Person> persons;
    private DataSource<AdultTicket> tickets;

    private List<String> results;

    public TicketIssuerUnit() {
    }

    public TicketIssuerUnit( DataSource<Person> persons, DataSource<AdultTicket> tickets ) {
        this.persons = persons;
        this.tickets = tickets;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public DataSource<AdultTicket> getTickets() {
        return tickets;
    }

    public List<String> getResults() {
```
return results;
}
}

The **BoxOfficeUnit** rule unit contains a **BoxOfficelsOpen** DRL rule that uses the `drools.guard(TicketIssuerUnit.class)` method to guard the execution of the **TicketIssuerUnit** rule unit that distributes the event tickets, as shown in the following DRL rule examples:

### Example DRL rules with controlled execution using `drools.guard()`

```drl
package org.mypackage.myunit;
unit TicketIssuerUnit;

rule IssueAdultTicket when
  $p: /persons[ age >= 18 ]
then
  tickets.insert(new AdultTicket($p));
end

rule RegisterAdultTicket when
  $t: /tickets
then
  results.add( $t.getPerson().getName() );
end

package org.mypackage.myunit;
unit BoxOfficeUnit;

rule BoxOfficelsOpen when
  $box: /boxOffices[ open ]
then
  drools.guard( TicketIssuerUnit.class );
end
```

In this example, so long as at least one box office is **open**, the guarded **TicketIssuerUnit** rule unit is active and distributes event tickets. When no more box offices are in **open** state, the guarded **TicketIssuerUnit** rule unit is prevented from being executed.

The following example class illustrates a more complete box office scenario:

### Example class for the box office scenario

```java
DataSource<Person> persons = executor.newDataSource( "persons" );
DataSource<BoxOffice> boxOffices = executor.newDataSource( "boxOffices" );
DataSource<AdultTicket> tickets = executor.newDataSource( "tickets" );

List<String> list = new ArrayList<>();
executor.bindVariable( "results", list );

// Two box offices are open:
BoxOffice office1 = new BoxOffice(true);
FactHandle officeFH1 = boxOffices.insert( office1 );
BoxOffice office2 = new BoxOffice(true);
FactHandle officeFH2 = boxOffices.insert( office2 );
```
82.7.3. Rule unit identity conflicts

In rule unit execution scenarios with guarded rule units, a rule can guard multiple rule units and at the same time a rule unit can be guarded and then activated by multiple rules. For these two-way guarding scenarios, rule units must have a clearly defined identity to avoid identity conflicts.

By default, the identity of a rule unit is the rule unit class name and is treated as a singleton class by the RuleUnitExecutor. This identification behavior is encoded in the getUnitIdentity() default method of the RuleUnit interface:

```java
default Identity getUnitIdentity() {
    return new Identity( getClass() );
}
```

In some cases, you may need to override this default identification behavior to avoid conflicting identities between rule units.
For example, the following `RuleUnit` class contains a `DataSource` definition that accepts any kind of object:

**Example Unit0 rule unit class**

```java
public class Unit0 implements RuleUnit {
    private DataSource<Object> input;

    public DataSource<Object> getInput() {
        return input;
    }
}
```

This rule unit contains the following DRL rule that guards another rule unit based on two conditions (in OOPath notation):

**Example GuardAgeCheck DRL rule in the rule unit**

```drl
package org.mypackage.myunit
unit Unit0

rule GuardAgeCheck
    when
        $i: /input#Integer
        $s: /input#String
    then
        drools.guard( new AgeCheckUnit($i) );
        drools.guard( new AgeCheckUnit($s.length()) );
end
```

The guarded `AgeCheckUnit` rule unit verifies the age of a set of `persons`. The `AgeCheckUnit` contains a `DataSource` definition of the `persons` to check, a `minAge` variable that it verifies against, and a `List` for gathering the results:

**Example AgeCheckUnit rule unit**

```java
public class AgeCheckUnit implements RuleUnit {
    private final int minAge;
    private DataSource<Person> persons;
    private List<String> results;

    public AgeCheckUnit( int minAge ) {
        this.minAge = minAge;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public int getMinAge() {
        return minAge;
    }

    public List<String> getResults() {
```
The **AgeCheckUnit** rule unit contains the following DRL rule that performs the verification of the **persons** in the data source:

**Example CheckAge DRL rule in the rule unit**

```drl
package org.mypackage.myunit
unit AgeCheckUnit

rule CheckAge
  when
    $p : /persons{ age > minAge }
  then
    results.add($p.getName() + "">" + minAge);
end
```

This example creates a **RuleUnitExecutor** class, binds the class to the KIE base that contains these two rule units, and creates the two **DataSource** definitions for the same rule units:

**Example executor and data source definitions**

```java
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );
DataSource<Object> input = executor.newDataSource( "input" );
DataSource<Person> persons = executor.newDataSource( "persons",
    new Person( "John", 42 ),
    new Person( "Sally", 4 ) );

List<String> results = new ArrayList<>();
executor.bindVariable( "results", results );
```

You can now insert some objects into the input data source and execute the **Unit0** rule unit:

**Example rule unit execution with inserted objects**

```java
ds.insert("test");
ds.insert(3);
ds.insert(4);
executor.run(Unit0.class);
```

**Example results list from the execution**

```
[Sally>3, John>3]
```

In this example, the rule unit named **AgeCheckUnit** is considered a singleton class and then executed only once, with the **minAge** variable set to **3**. Both the String "**test**" and the Integer **4** inserted into the input data source can also trigger a second execution with the **minAge** variable set to **4**. However, the second execution does not occur because another rule unit with the same identity has already been evaluated.
To resolve this rule unit identity conflict, override the `getUnitIdentity()` method in the `AgeCheckUnit` class to include also the `minAge` variable in the rule unit identity:

**Modified AgeCheckUnit rule unit to override the `getUnitIdentity()` method**

```java
public class AgeCheckUnit implements RuleUnit {

    ...

    @Override
    public Identity getUnitIdentity() {
        return new Identity(getClass(), minAge);
    }
}
```

With this override in place, the previous example rule unit execution produces the following output:

**Example results list from executing the modified rule unit**

```
[John>4, Sally>3, John>3]
```

The rule units with `minAge` set to 3 and 4 are now considered two different rule units and both are executed.
CHAPTER 83. PHREAK RULE ALGORITHM IN THE DECISION ENGINE

The decision engine in Red Hat Decision Manager 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 Red Hat Decision Manager 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

83.1. RULE EVALUATION IN PHREAK

When the decision 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 decision 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 decision 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 83.1. 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.
Example 2: Rule R2 is added and shares pattern A.

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 83.4. 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.

Figure 83.5. 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.
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.

### 83.1.1. Rule evaluation with forward and backward chaining

The decision engine in Red Hat Decision Manager 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 decision 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 decision 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 decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:
83.2. RULE BASE CONFIGURATION

Red Hat Decision Manager contains a RuleBaseConfiguration.java object that you can use to configure exception handler settings, multithreaded execution, and sequential mode in the decision engine.

For the rule base configuration options, download the Red Hat Decision Manager 7.10.0 Source Distribution ZIP file from the Red Hat Customer Portal and navigate to ~/rhdm-7.10.0-sources/src/drools-$VERSION/drools-core/src/main/java/org/drools/core/RuleBaseConfiguration.java.
The following rule base configuration options are available for the decision 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 decision 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.MyCustomConsequenceExceptionHandler
  ```

- Specify the exception handler while creating the KIE base programmatically:
  ```java
  KieServices ks = KieServices.Factory.get();
  KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
  kieBaseConf.setOption(ConsequenceExceptionHandlerOption.get(MyCustomConsequenceExceptionHandler.class));
  KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
  ```

**drools.multithreadEvaluation**

When enabled, this system property enables the decision 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:
  ```java
  KieServices ks = KieServices.Factory.get();
  KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
  kieBaseConf.setOption(MultithreadEvaluationOption.YES);
  KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
  ```
**WARNING**

Rules that use queries, salience, or agenda groups are currently not supported by the parallel decision 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 decision engine might not detect the unsupported rule elements and rules might be evaluated incorrectly. For example, the decision 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 decision engine. In sequential mode, the decision engine evaluates rules one time in the order that they are listed in the decision engine agenda without regard to changes in the working memory. This means that the decision 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:

  ```java
  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 Red Hat Decision Manager project:

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

---

**83.3. SEQUENTIAL MODE IN PHREAK**

---

590
Sequential mode is an advanced rule base configuration in the decision engine, supported by Phreak, that enables the decision engine to evaluate rules one time in the order that they are listed in the decision engine agenda without regard to changes in the working memory. In sequential mode, the decision 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 decision 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:
  ```java
  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 Red Hat Decision Manager project:
  ```xml
  <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:
  ```java
  KieServices ks = KieServices.Factory.get();
  KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
  kieBaseConf.setOption(SequentialAgendaOption.DYNAMIC);
  KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
  ```

When you enable sequential mode, the decision 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 decision 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 decision 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 decision engine iterates over the list of **LeftInputAdapterNode 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.
In Red Hat Decision Manager, 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 decision engine in Red Hat Decision Manager 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 Red Hat Decision Manager 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 decision engine (pipeline)

84.1. EVENTS IN COMPLEX EVENT PROCESSING

In Red Hat Decision Manager, 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.

  **NOTE**

  The decision 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 decision 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 decision 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.

84.2. DECLARING FACTS AS EVENTS

You can declare facts as events in your Java class or DRL rule file so that the decision 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 decision 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**

```java
import some.package.StockPoint
declarate 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**

```java
declare StockPoint
   @role( event )
   datetime : java.util.Date
   symbol : String
   price : double
end
```

**84.3. METADATA TAGS FOR EVENTS**

The decision engine uses the following metadata tags for events that are inserted into the working memory of the decision engine. You can change the default metadata tag values in your Java class or DRL rule file as needed.
NOTE

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

```java
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 decision engine during complex event processing.
Default parameter: `fact`

Supported parameters: `fact`, `event`

Example: Declare VoiceCall as event type

```java
declare VoiceCall
    @role( event )
end
```

@timestamp

This tag is automatically assigned to every event in the decision 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 decision 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 decision engine session clock

Supported parameters: Session clock time or custom time stamp attribute

Example: Declare VoiceCall timestamp attribute

```java
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
end
```

@duration

This tag determines the duration time for events in the decision 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 decision 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 decision 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( <attributeName> )
```

**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 decision 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 decision 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( <timeOffset> )
```

**Example: Declare expiration offset for VoiceCall events**

```
declare VoiceCall
  @role( event )
  @timestamp( callDateTime )
  @duration( callDuration )
  @expires( 1h35m )
end
```

### 84.4. EVENT PROCESSING MODES IN THE DECISION ENGINE

The decision engine runs in either cloud mode or stream mode. In cloud mode, the decision engine processes facts as facts with no temporal constraints, independent of time, and in no particular order. In stream mode, the decision 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 Red Hat Decision Manager.

**Cloud mode**

Cloud mode is the default operating mode of the decision engine. In cloud mode, the decision engine
treats events as an unordered cloud. Events still have time stamps, but the decision 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 decision 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 decision engine has no notion of time
- No ordering of events because the decision engine processes events as an unordered cloud, against which the decision 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**

```groovy
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 Red Hat Decision Manager 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 decision engine to process events chronologically and in real time as they are inserted into the decision engine. In stream mode, the decision 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.

NOTE

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 Red Hat Decision Manager project:

Set stream mode using project kmodule.xml file

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

84.4.1. Negative patterns in decision 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

```drl
rule "Sound the alarm"
when
  $f : FireDetected() not(SprinklerActivated())
then
  // Sound the alarm.
end
```
In cloud mode, the decision engine assumes all facts (regular facts and events) are known in advance and evaluates negative patterns immediately. In stream mode, the decision 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)**

```plaintext
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 \([0s, \ldots]\) does not inherently exclude that event from being matched again.

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

```plaintext
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
```

84.5. PROPERTY-CHANGE SETTINGS AND LISTENERS FOR FACT TYPES

By default, the decision 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 decision 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**

```java
KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration().
config.setProperty(PropertySpecificOption.ALLOWED);
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder(config);
```

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

**Example property reactivity setting in system properties**

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

The decision 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 decision 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 decision 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**

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

**Example: Disable default property reactivity in a Java class**

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

- **@propertyReactive**
  
  If property reactivity is set to **ALLOWED** in the decision 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 decision 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 decision 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 decision 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 decision 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 decision engine to monitor changes in the fact properties.

Example: Declare property change support in JavaBeans object

```
declare Person
    @propertyChangeSupport
end
```

84.6. TEMPORAL OPERATORS FOR EVENTS

In stream mode, the decision engine supports the following temporal operators for events that are inserted into the working memory of the decision 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 decision 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:

\[
3m30s \leq \text{$eventA.startTimestamp - $eventB.endTimestamp} \leq 4m
\]

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:

\[
\text{$eventA : EventA(this after[-3m30s, -2m] $eventB)}
\]

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

\[
\text{$eventA : EventA(this after[-3m30s, -2m] $eventB)}
\]

\[
\text{$eventA : EventA(this after[-2m, -3m30s] $eventB)}
\]

**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.

\[
\text{$eventA : EventA(this before[3m30s, 4m] $eventB)}
\]

You can also express this operator in the following way:

\[
3m30s \leq $eventB.startTimestamp - $eventA.endTimestamp \leq 4m
\]

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

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

WARNING

The decision engine does not support negative intervals for the coincides operator. If you use negative intervals, the decision 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:

\[
\text{\$eventA : EventA(this finishes \$eventB)}
\]

You can also express this operator in the following way:

\[
\text{\$eventB.startTimestamp < \$eventA.startTimestamp}
\text{\&\&}
\text{\$eventA.endTimestamp == \$eventB.endTimestamp}
\]

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

\[
\text{\$eventA : EventA(this finishes[5s] \$eventB)}
\]

This pattern matches if these conditions are met:

\[
\text{\$eventB.startTimestamp < \$eventA.startTimestamp}
\text{\&\&}
\text{abs(\$eventA.endTimestamp - \$eventB.endTimestamp) <= 5s}
\]

**WARNING**

The decision engine does not support negative intervals for the **finishes** operator. If you use negative intervals, the decision 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:

\[
\text{\$eventA : EventA(this finishedby \$eventB)}
\]

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

**WARNING**

The decision engine does not support negative intervals for the finished by operator. If you use negative intervals, the decision 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
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

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
```
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

84.7. SESSION CLOCK IMPLEMENTATIONS IN THE DECISION ENGINE

During complex event processing, events in the decision 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 decision engine must be able to compare the stock price event time stamp with the current time in the session clock.

The decision 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 decision 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 decision engine.

**Real-time clock**

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

**Configure real-time clock in KIE session**

```java
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 decision engine is helpful for testing temporal rules and it can be controlled by the application. To configure the decision engine to use the pseudo clock, set the KIE session configuration parameter to `pseudo`:

**Configure pseudo clock in KIE session**

```java
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

```java
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);
```

84.8. EVENT STREAMS AND ENTRY POINTS

The decision 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 decision 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 decision engine. Facts always remain associated with the entry point through which they entered the decision engine. Facts of the same type can enter the decision engine through several entry points, but facts that enter the decision 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.

84.8.1. Declaring entry points for rule data
You can declare an entry point (event stream) for events so that the decision 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 decision 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.

84.9. SLIDING WINDOWS OF TIME OR LENGTH

In stream mode, the decision 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 decision 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 decision 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) |

84.9.1. 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 decision 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) |

CHAPTER 84. COMPLEX EVENT PROCESSING (CEP)
Number(doubleValue > $max) from accumulate(
    SensorReading($temp : temperature) over window:length(100),
    average($temp))
then
  // Sound the alarm.
end

The decision 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 recalculation the average as the minutes or readings "slide" forward in real time.

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

### 84.10. MEMORY MANAGEMENT FOR EVENTS

In stream mode, the decision engine uses automatic memory management to maintain events that are stored in KIE sessions. The decision 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 decision engine uses either explicit or inferred expiration to retract outdated events:

- **Explicit expiration**: The decision 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 decision 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 decision engine automatically calculates that whenever a BuyOrder event occurs, the decision engine needs to store the event for up to 10 seconds and wait for the matching AckOrder event. After 10 seconds, the decision engine infers the expiration and
removes the event from the KIE session. An AckOrder event can only match an existing BuyOrder event, so the decision engine infers the expiration if no match occurs and removes the event immediately.

The decision 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 decision engine uses the greater time frame of the two to store the event longer.
CHAPTER 85. DECISION ENGINE QUERIES AND LIVE QUERIES

You can use queries with the decision 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 decision 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.

Example query definition in a DRL file

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

Example application code to obtain and iterate over query results

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

```drl
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" ) );
    }
}
```

Red Hat Decision Manager 7.10 Developing decision services in Red Hat Decision Manager
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()
CHAPTER 86. DECISION ENGINE EVENT LISTENERS AND DEBUG LOGGING

In Red Hat Decision Manager, you can add or remove listeners for decision engine events, such as fact insertions and rule executions. With decision engine event listeners, you can be notified of decision engine activity and separate your logging and auditing work from the core of your application.

The decision engine supports the following default event listeners for the agenda and working memory:

- AgendaEventListener
- WorkingMemoryEventListener

For each event listener, the decision engine also supports the following specific events that you can specify to be monitored:

- MatchCreatedEvent
- MatchCancelledEvent
- BeforeMatchFiredEvent
- AfterMatchFiredEvent
- AgendaGroupPushedEvent
- AgendaGroupPoppedEvent
- ObjectInsertEvent
- ObjectDeletedEvent
- ObjectUpdatedEvent
- ProcessCompletedEvent
- ProcessNodeLeftEvent
- ProcessNodeTriggeredEvent
- ProcessStartEvent

For example, the following code uses a DefaultAgendaEventListener listener attached to a KIE session and specifies the AfterMatchFiredEvent event to be monitored. The code prints pattern matches after the rules are executed (fired):

Example code to monitor and print AfterMatchFiredEvent events in the agenda

```java
ksession.addEventListener( new DefaultAgendaEventListener() {
    public void afterMatchFired(AfterMatchFiredEvent event) {
        super.afterMatchFired( event );
        System.out.println( event );
    }
});
```
The decision engine also supports the following agenda and working memory 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 a specific supported event to be monitored and documented, or monitor all agenda or working memory activity.

For example, the following code uses the **DebugRuleRuntimeEventListener** event listener to monitor and print all working memory events:

Example code to monitor and print all working memory events

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

### 86.1. CONFIGURING A LOGGING UTILITY IN THE DECISION ENGINE

The decision engine uses the Java logging API SLF4J for system logging. You can use one of the following logging utilities with the decision engine to investigate decision 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 Red Hat Decision Manager distribution:

**Example Maven dependency for Logback**

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

**Example logback.xml configuration file in org.drools package**

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

**Example log4j.xml configuration file in org.drools package**

```xml
```
NOTE

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

```xml
  <category name="org.drools">
    <priority value="debug"/>
  </category>
  ...
</log4j:configuration>
```
CHAPTER 87. EXAMPLE DECISIONS IN RED HAT DECISION MANAGER FOR AN IDE

Red Hat Decision Manager provides example decisions distributed as Java classes that you can import into your integrated development environment (IDE). You can use these examples to better understand decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

The following example decision sets are some of the examples available in Red Hat Decision Manager:

- **Hello World example**: Demonstrates basic rule execution and use of debug output
- **State example**: Demonstrates forward chaining and conflict resolution through rule salience and agenda groups
- **Fibonacci example**: Demonstrates recursion and conflict resolution through rule salience
- **Banking example**: Demonstrates pattern matching, basic sorting, and calculation
- **Pet Store example**: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration
- **Sudoku example**: Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration
- **House of Doom example**: Demonstrates backward chaining and recursion

**NOTE**

For optimization examples provided with Red Hat Business Optimizer, see [Getting started with Red Hat Business Optimizer](#).

87.1. IMPORTING AND EXECUTING RED HAT DECISION MANAGER EXAMPLE DECISIONS IN AN IDE

You can import Red Hat Decision Manager example decisions into your integrated development environment (IDE) and execute them to explore how the rules and code function. You can use these examples to better understand decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

**Prerequisites**

- Java 8 or later is installed.
- Maven 3.5.x or later is installed.
- An IDE is installed, such as Red Hat CodeReady Studio.

**Procedure**

1. Download and unzip the Red Hat Decision Manager 7.10.0 Source Distribution from the Red Hat Customer Portal to a temporary directory, such as `/rhdm-7.10.0-sources`. 
2. Open your IDE and select **File → Import → Maven → Existing Maven Projects** or the equivalent option for importing a Maven project.

3. Click **Browse**, navigate to `~/rhdm-7.10.0-sources/src/drools-$VERSION/drools-examples` (or, for the Conway's Game of Life example, `~/rhdm-7.10.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples`), and import the project.

4. Navigate to the example package that you want to run and find the Java class with the **main** method.

5. Right-click the Java class and select **Run As → Java Application** to run the example.

   To run all examples through a basic user interface, run the **DroolsExamplesApp.java** class (or, for Conway's Game of Life, the **DroolsJbpmIntegrationExamplesApp.java** class) in the **org.drools.examples** main class.
Figure 87.1. Interface for all examples in drools-examples (DroolsExamplesApp.java)
87.2. HELLO WORLD EXAMPLE DECISIONS (BASIC RULES AND DEBUGGING)

The Hello World example decision set demonstrates how to insert objects into the decision engine working memory, how to match the objects using rules, and how to configure logging to trace the internal activity of the decision engine.

The following is an overview of the Hello World example:

- **Name:** helloworld
- **Main class:** org.drools.examples.helloworld.HelloWorldExample (in src/main/java)
- **Module:** drools-examples
- **Type:** Java application
- **Rule file:** org.drools.examples.helloworld.HelloWorld.drl (in src/main/resources)
- **Objective:** Demonstrates basic rule execution and use of debug output

In the Hello World example, a KIE session is generated to enable rule execution. All rules require a KIE session for execution.

**KIE session for rule execution**

```java
KieServices ks = KieServices.Factory.get();  // 1
KieContainer kc = ks.getKieClasspathContainer();  // 2
KieSession ksession = kc.newKieSession("HelloWorldKS");  // 3
```

1. Obtains the `KieServices` factory. This is the main interface that applications use to interact with the decision engine.

2. Creates a `KieContainer` from the project class path. This detects a `/META-INF/kmodule.xml` file from which it configures and instantiates a `KieContainer` with a `KieModule`.

3. Creates a `KieSession` based on the "HelloWorldKS" KIE session configuration defined in the `/META-INF/kmodule.xml` file.
NOTE
For more information about Red Hat Decision Manager project packaging, see Packaging and deploying a Red Hat Decision Manager project.

Red Hat Decision Manager has an event model that exposes internal engine activity. Two default debug listeners, DebugAgendaEventListener and DebugRuleRuntimeEventListener, print debug event information to the System.err output. The KieRuntimeLogger provides execution auditing, the result of which you can view in a graphical viewer.

Debug listeners and audit loggers

```java
// Set up listeners.
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugRuleRuntimeEventListener() );

// Set up a file-based audit logger.
KieRuntimeLogger logger = KieServices.get().getLoggers().newFileLogger( ksession, "/target/helloworld");

// Set up a ThreadedFileLogger so that the audit view reflects events while debugging.
KieRuntimeLogger logger = ks.getLoggers().newThreadedFileLogger( ksession, "/target/helloworld", 1000);
```

The logger is a specialized implementation built on the Agenda and RuleRuntime listeners. When the decision engine has finished executing, logger.close() is called.

The example creates a single Message object with the message "Hello World", inserts the status HELLO into the KieSession, executes rules with fireAllRules().

Data insertion and execution

```java
// Insert facts into the KIE session.
final Message message = new Message();
message.setMessage( "Hello World" );
message.setStatus( Message.HELLO );
ksession.insert( message );

// Fire the rules.
ksession.fireAllRules();
```

Rule execution uses a data model to pass data as inputs and outputs to the KieSession. The data model in this example has two fields: the message, which is a String, and the status, which can be HELLO or GOODBYE.

Data model class

```java
public static class Message {
    public static final int HELLO = 0;
    public static final int GOODBYE = 1;

    private String message;
```
The two rules are located in the file
src/main/resources/org/drools/examples/helloworld/HelloWorld.drl.

The `when` condition of the "Hello World" rule states that the rule is activated for each `Message` object inserted into the KIE session that has the status `Message.HELLO`. Additionally, two variable bindings are created: the variable `message` is bound to the `message` attribute and the variable `m` is bound to the matched `Message` object itself.

The `then` action of the rule specifies to print the content of the bound variable `message` to `System.out`, and then changes the values of the `message` and `status` attributes of the `Message` object bound to `m`. The rule uses the `modify` statement to apply a block of assignments in one statement and to notify the decision engine of the changes at the end of the block.

"Hello World" rule

```java
rule "Hello World"
when
  m : Message( status == Message.HELLO, message : message )
then
  System.out.println( message );
  modify ( m ) { message = "Goodbye cruel world",
               status = Message.GOODBYE };" end
```

The "Good Bye" rule is similar to the "Hello World" rule except that it matches `Message` objects that have the status `Message.GOODBYE`.

"Good Bye" rule

```java
rule "Good Bye"
when
  Message( status == Message.GOODBYE, message : message )
then
  System.out.println( message );
end
```

To execute the example, run the `org.drools.examples.helloworld.HelloWorldExample` class as a Java application in your IDE. The rule writes to `System.out`, the debug listener writes to `System.err`, and the audit logger creates a log file in `target/helloworld.log`.

System.out output in the IDE console

```
Hello World
Goodbye cruel world
```

System.err output in the IDE console

```text
==>[ActivationCreated(0): rule=Hello World;
  tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[ObjectInserted: handle=
```

Red Hat Decision Manager 7.10 Developing decision services in Red Hat Decision Manager
To better understand the execution flow of this example, you can load the audit log file from target/helloworld.log into your IDE debug view or Audit View, if available (for example, in Window → Show View in some IDEs).

In this example, the Audit view shows that the object is inserted, which creates an activation for the "Hello World" rule. The activation is then executed, which updates the Message object and causes the "Good Bye" rule to activate. Finally, the "Good Bye" rule is executed. When you select an event in the Audit View, the origin event, which is the "Activation created" event in this example, is highlighted in green.

Figure 87.3. Hello World example Audit View

87.3. STATE EXAMPLE DECISIONS (FORWARD CHAINING AND CONFLICT RESOLUTION)

The State example decision set demonstrates how the decision engine uses forward chaining and any changes to facts in the working memory to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values or through agenda groups that you can define in rules.

The following is an overview of the State example:

- **Name:** state
- **Main classes:** org.drools.examples.state.StateExampleUsingSalience, org.drools.examples.state.StateExampleUsingAgendaGroup (in src/main/java)
- **Module:** drools-examples
- **Type:** Java application
- Rule files: `org.drools.examples.state.*.drl` (in `src/main/resources`)

- **Objective**: Demonstrates forward chaining and conflict resolution through rule salience and agenda groups

A forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision 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 decision 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 decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:
In the State example, each State class has fields for its name and its current state (see the class org.drools.examples.state.State). The following states are the two possible states for each object:

- NOTRUN
- FINISHED

State class

```java
public class State {
    public static final int NOTRUN = 0;
```
The State example contains two versions of the same example to resolve rule execution conflicts:

- A **StateExampleUsingSalience** version that resolves conflicts by using rule salience
- A **StateExampleUsingAgendaGroups** version that resolves conflicts by using rule agenda groups

Both versions of the state example involve four State objects: A, B, C, and D. Initially, their states are set to NOTRUN, which is the default value for the constructor that the example uses.

**State example using salience**

The **StateExampleUsingSalience** version of the State example uses salience values in rules to resolve rule execution conflicts. Rules with a higher salience value are given higher priority when ordered in the activation queue.

The example inserts each State instance into the KIE session and then calls `fireAllRules()`.

**Salience State example execution**

```java
public static final int FINISHED = 1;

private final PropertyChangeSupport changes =
    new PropertyChangeSupport( this );

private String name;
private int state;

... setters and getters go here...
}
```

```java
The State example uses Salience values in rules to resolve rule execution conflicts. Rules with a higher salience value are given higher priority when ordered in the activation queue.

```java
definal State a = new State( "A" );
definal State b = new State( "B" );
definal State c = new State( "C" );
definal State d = new State( "D" );
ksession.insert( a );
ksession.insert( b );
ksession.insert( c );
ksession.insert( d );

ksession.fireAllRules();

// Dispose KIE session if stateful (not required if stateless).
ksession.dispose();
```

To execute the example, run the **org.drools.examples.state.StateExampleUsingSalience** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

**Salience State example output in the IDE console**

```
A finished
B finished
C finished
```
D finished

Four rules are present.

First, the "Bootstrap" rule fires, setting \( A \) to state \textbf{FINISHED}, which then causes \( B \) to change its state to \textbf{FINISHED}. Objects \( C \) and \( D \) are both dependent on \( B \), causing a conflict that is resolved by the salience values.

To better understand the execution flow of this example, you can load the audit log file from \texttt{target/state.log} into your IDE debug view or Audit View, if available (for example, in \texttt{Window \rightarrow Show View} in some IDEs).

In this example, the Audit View shows that the assertion of the object \( A \) in the state \textbf{NOTRUN} activates the "Bootstrap" rule, while the assertions of the other objects have no immediate effect.

Figure 87.5. Salience State example Audit View

Rule "Bootstrap" in salience State example

```java
rule "Bootstrap"
    when
        a : State(name == "A", state == State.NOTRUN )
    then
        System.out.println(a.getName() + " finished");
        a.setState( State.FINISHED );
end
```

The execution of the "Bootstrap" rule changes the state of \( A \) to \textbf{FINISHED}, which activates rule "\( A \) to \( B \)".

Rule "\( A \) to \( B \)" in salience State example
rule "A to B"
    when
        State(name == "A", state == State.FINISHED )
    b : State(name == "B", state == State.NOTRUN )
    then
        System.out.println(b.getName() + " finished");
        b.setState( State.FINISHED );
    end

The execution of rule "A to B" changes the state of B to FINISHED, which activates both rules "B to C" and "B to D", placing their activations onto the decision engine agenda.

Rules "B to C" and "B to D" in salience State example

rule "B to C"
    salience 10
    when
        State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
    then
        System.out.println(c.getName() + " finished");
        c.setState( State.FINISHED );
    end

rule "B to D"
    when
        State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
    then
        System.out.println(d.getName() + " finished");
        d.setState( State.FINISHED );
    end

From this point on, both rules may fire and, therefore, the rules are in conflict. The conflict resolution strategy enables the decision engine agenda to decide which rule to fire. Rule "B to C" has the higher salience value (10 versus the default salience value of 0), so it fires first, modifying object C to state FINISHED.

The Audit View in your IDE shows the modification of the State object in the rule "A to B", which results in two activations being in conflict.

You can also use the Agenda View in your IDE to investigate the state of the decision engine agenda. In this example, the Agenda View shows the breakpoint in the rule "A to B" and the state of the agenda with the two conflicting rules. Rule "B to D" fires last, modifying object D to state FINISHED.
State example using agenda groups

The **StateExampleUsingAgendaGroups** version of the State example uses agenda groups in rules to resolve rule execution conflicts. Agenda groups enable you to partition the decision engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.
Initially, a working memory has its focus on the agenda group **MAIN**. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method `setFocus()` or the rule attribute `auto-focus`. The `auto-focus` attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

In this example, the `auto-focus` attribute enables rule "B to C" to fire before "B to D".

**Rule "B to C" in agenda group State example**

```java
rule "B to C"
    agenda-group "B to C"
    auto-focus true
    when
        State(name == "B", state == State.FINISHED )
        c : State(name == "C", state == State.NOTRUN )
    then
        System.out.println(c.getName() + " finished");
        c.setState( State.FINISHED );
        kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup("B to D").setFocus();
end
```

The rule "B to C" calls `setFocus()` on the agenda group "B to D", enabling its active rules to fire, which then enables the rule "B to D" to fire.

**Rule "B to D" in agenda group State example**

```java
rule "B to D"
    agenda-group "B to D"
    when
        State(name == "B", state == State.FINISHED )
        d : State(name == "D", state == State.NOTRUN )
    then
        System.out.println(d.getName() + " finished");
        d.setState( State.FINISHED );
end
```

To execute the example, run the `org.drools.examples.state.StateExampleUsingAgendaGroups` class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window (same as the salience version of the State example):

**Agenda group State example output in the IDE console**

```
A finished
B finished
C finished
D finished
```

**Dynamic facts in the State example**

Another notable concept in this State example is the use of **dynamic facts**, based on objects that implement a `PropertyChangeListener` object. In order for the decision engine to see and react to changes of fact properties, the application must notify the decision engine that changes occurred. You
can configure this communication explicitly in the rules by using the modify statement, or implicitly by specifying that the facts implement the PropertyChangeSupport interface as defined by the JavaBeans specification.

This example demonstrates how to use the PropertyChangeSupport interface to avoid the need for explicit modify statements in the rules. To make use of this interface, ensure that your facts implement PropertyChangeSupport in the same way that the class org.drools.example.State implements it, and then use the following code in the DRL rule file to configure the decision engine to listen for property changes on those facts:

**Declaring a dynamic fact**

```java
declare type State
    @propertyChangeSupport
end
```

When you use PropertyChangeListener objects, each setter must implement additional code for the notification. For example, the following setter for state is in the class org.drools.examples:

**Setter example with PropertyChangeSupport**

```java
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",
        oldState,
        newState );
}
```

### 87.4. FIBONACCI EXAMPLE DECISIONS (RECURSION AND CONFLICT RESOLUTION)

The Fibonacci example decision set demonstrates how the decision engine uses recursion to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values that you can define in rules.

The following is an overview of the Fibonacci example:

- **Name:** fibonacci
- **Main class:** org.drools.examples.fibonacci.FibonacciExample (in src/main/java)
- **Module:** drools-examples
- **Type:** Java application
- **Rule file:** org.drools.examples.fibonacci.Fibonacci.drl (in src/main/resources)
- **Objective:** Demonstrates recursion and conflict resolution through rule salience

The Fibonacci Numbers form a sequence starting with 0 and 1. The next Fibonacci number is obtained by adding the two preceding Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, and so on.

The Fibonacci example uses the single fact class Fibonacci with the following two fields:
The sequence field indicates the position of the object in the Fibonacci number sequence. The value field shows the value of that Fibonacci object for that sequence position, where -1 indicates a value that still needs to be computed.

**Fibonacci class**

```java
public static class Fibonacci {
    private int sequence;
    private long value;

    public Fibonacci(final int sequence) {
        this.sequence = sequence;
        this.value = -1;
    }

    // ... setters and getters go here...
}
```

To execute the example, run the `org.drools.examples.fibonacci.FibonacciExample` class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

**Fibonacci example output in the IDE console**

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
6 == 8
...
47 == 2971215073
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

To achieve this behavior in Java, the example inserts a single Fibonacci object with a sequence field of 50. The example then uses a recursive rule to insert the other 49 Fibonacci objects.
Instead of implementing the `PropertyChangeSupport` interface to use dynamic facts, this example uses the MVEL dialect `modify` keyword to enable a block setter action and notify the decision engine of changes.

**Fibonacci example execution**

```java
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

This example uses the following three rules:

- "Recurse"
- "Bootstrap"
- "Calculate"

The rule "Recurse" matches each asserted `Fibonacci` object with a value of -1, creating and asserting a new `Fibonacci` object with a sequence of one less than the currently matched object. Each time a Fibonacci object is added while the one with a sequence field equal to 1 does not exist, the rule re-matches and fires again. The `not` conditional element is used to stop the rule matching once you have all 50 Fibonacci objects in memory. The rule also has a `salience` value because you need to have all 50 `Fibonacci` objects asserted before you execute the "Bootstrap" rule.

**Rule "Recurse"**

```java
rule "Recurse"
  salience 10
  when
    f : Fibonacci ( value == -1 )
    not ( Fibonacci ( sequence == 1 ) )
  then
    insert( new Fibonacci( f.sequence - 1 ) );
    System.out.println( "recurse for " + f.sequence );
end
```

To better understand the execution flow of this example, you can load the audit log file from `target/fibonacci.log` into your IDE debug view or Audit View, if available (for example, in Window → Show View in some IDEs).

In this example, the Audit View shows the original assertion of the `Fibonacci` object with a `sequence` field of 50, done from Java code. From there on, the Audit View shows the continual recursion of the rule, where each asserted `Fibonacci` object causes the "Recurse" rule to become activated and to fire again.
When a `Fibonacci` object with a sequence field of 2 is asserted, the "Bootstrap" rule is matched and activated along with the "Recurse" rule. Notice the multiple restrictions on field sequence that test for equality with 1 or 2:

**Rule "Bootstrap"**

```java
rule "Bootstrap"
when
  f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-restriction
then
  modify ( f ){ value = 1 };
  System.out.println( f.sequence + " == " + f.value );
end
```

You can also use the Agenda View in your IDE to investigate the state of the decision engine agenda. The "Bootstrap" rule does not fire yet because the "Recurse" rule has a higher salience value.
When a Fibonacci object with a sequence of 1 is asserted, the "Bootstrap" rule is matched again, causing two activations for this rule. The "Recurse" rule does not match and activate because the not conditional element stops the rule matching as soon as a Fibonacci object with a sequence of 1 exists.
The "Bootstrap" rule sets the objects with a sequence of 1 and 2 to a value of 1. Now that you have two Fibonacci objects with values not equal to -1, the "Calculate" rule is able to match.

At this point in the example, nearly 50 Fibonacci objects exist in the working memory. You need to select a suitable triple to calculate each of their values in turn. If you use three Fibonacci patterns in a rule without field constraints to confine the possible cross products, the result would be 50x49x48 possible combinations, leading to about 125,000 possible rule firings, most of them incorrect.

The "Calculate" rule uses field constraints to evaluate the three Fibonacci patterns in the correct order. This technique is called cross-product matching.

The first pattern finds any Fibonacci object with a value != -1 and binds both the pattern and the field. The second Fibonacci object does the same thing, but adds an additional field constraint to ensure that its sequence is greater by one than the Fibonacci object bound to f1. When this rule fires for the first time, you know that only sequences 1 and 2 have values of 1, and the two constraints ensure that f1 references sequence 1 and that f2 references sequence 2.

The final pattern finds the Fibonacci object with a value equal to -1 and with a sequence one greater than f2.

At this point in the example, three Fibonacci objects are correctly selected from the available cross products, and you can calculate the value for the third Fibonacci object that is bound to f3.

**Rule "Calculate"**

```java
rule "Calculate"
when
    // Bind f1 and s1.
    f1 : Fibonacci( s1 : sequence, value != -1 )
    // Bind f2 and v2, refer to bound variable s1.
    f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )
    // Bind f3 and s3, alternative reference of f2.sequence.
    f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )
then
    // Note the various referencing techniques.
    modify ( f3 ) { value = f1.value + v2 };
    System.out.println( s3 + " == " + f3.value );
end
```

The modify statement updates the value of the Fibonacci object bound to f3. This means that you now have another new Fibonacci object with a value not equal to -1, which allows the "Calculate" rule to re-match and calculate the next Fibonacci number.

The debug view or Audit View of your IDE shows how the firing of the last "Bootstrap" rule modifies the Fibonacci object, enabling the "Calculate" rule to match, which then modifies another Fibonacci object that enables the "Calculate" rule to match again. This process continues until the value is set for all Fibonacci objects.
87.5. PRICING EXAMPLE DECISIONS (DECISION TABLES)

The Pricing example decision set demonstrates how to use a spreadsheet decision table for calculating the retail cost of an insurance policy in tabular format instead of directly in a DRL file.

The following is an overview of the Pricing example:

- **Name**: decisiontable
- **Main class**: org.drools.examples.decisiontable.PricingRuleDTExample (in src/main/java)
- **Module**: drools-examples
- **Type**: Java application
- **Rule file**: org.drools.examples.decisiontable.ExamplePolicyPricing.xls (in src/main/resources)
- **Objective**: Demonstrates use of spreadsheet decision tables to define rules

Spreadsheet decision tables are XLS or XLSX spreadsheets that contain business rules defined in a tabular format. You can include spreadsheet decision tables with standalone Red Hat Decision Manager projects or upload them to projects in Business Central. Each row in a decision table is a rule, and each column is a condition, an action, or another rule attribute. After you create and upload your decision tables into your Red Hat Decision Manager project, the rules you defined are compiled into Drools Rule Language (DRL) rules as with all other rule assets.
The purpose of the Pricing example is to provide a set of business rules to calculate the base price and a
discount for a car driver applying for a specific type of insurance policy. The driver’s age and history and
the policy type all contribute to calculate the basic premium, and additional rules calculate potential
discounts for which the driver might be eligible.

To execute the example, run the `org.drools.examples.decisiontable.PricingRuleDTExample` class as
a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

```
Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
```

The code to execute the example follows the typical execution pattern: the rules are loaded, the facts
are inserted, and a stateless KIE session is created. The difference in this example is that the rules are
defined in an `ExamplePolicyPricing.xls` file instead of a DRL file or other source. The spreadsheet file
is loaded into the decision engine using templates and DRL rules.

**Spreadsheet decision table setup**
The `ExamplePolicyPricing.xls` spreadsheet contains two decision tables in the first tab:

- **Base pricing rules**
- **Promotional discount rules**

As the example spreadsheet demonstrates, you can use only the first tab of a spreadsheet to create
decision tables, but multiple tables can be within a single tab. Decision tables do not necessarily follow
top-down logic, but are more of a means to capture data resulting in rules. The evaluation of the rules is
not necessarily in the given order, because all of the normal mechanics of the decision engine still apply.
This is why you can have multiple decision tables in the same tab of a spreadsheet.

The decision tables are executed through the corresponding rule template files `BasePricing.drt` and
`PromotionalPricing.drt`. These template files reference the decision tables through their template
parameter and directly reference the various headers for the conditions and actions in the decision
tables.

**BasePricing.drt rule template file**

```java
package org.drools.examples.decisiontable;

template "Pricing bracket"
    age
    policyType
    base
    reason

rule "Pricing bracket_@{row.rowNumber}"
    when
```

Red Hat Decision Manager 7.10 Developing decision services in Red Hat Decision Manager
PromotionalPricing.drt rule template file

```java
template header
age[]
priorClaims
policyType
discount

discount

package org.drools.examples.decisiontable;

template "discounts"
age
priorClaims
policyType
discount

rule "Discounts_@{row.rowNumber}"
when
    Driver(age >= @{age0}, age <= @{age1}, priorClaims == "@{priorClaims}")
policy: Policy(type == "@{policyType}")
then
    policy.applyDiscount(@{discount});
end
end template
```

The rules are executed through the `kmodule.xml` reference of the KIE Session `DTableWithTemplateKB`, which specifically mentions the `ExamplePolicyPricing.xls` spreadsheet and is required for successful execution of the rules. This execution method enables you to execute the rules as a standalone unit (as in this example) or to include the rules in a packaged knowledge JAR (KJAR) file, so that the spreadsheet is packaged along with the rules for execution.

The following section of the `kmodule.xml` file is required for the execution of the rules and spreadsheet to work successfully:

```xml
<base name="DecisionTableKB" packages="org.drools.examples.decisiontable">
    <ksession name="DecisionTableKS" type="stateless"/>
</base>

<base name="DTableWithTemplateKB" packages="org.drools.examples.decisiontable-template">
    <ruleTemplate dtable="org/drools/examples/decisiontable-template/ExamplePolicyPricingTemplateData.xls" template="org/drools/examples/decisiontable-template/BasePricing.drt" row="3" col="3"/>
    <ruleTemplate dtable="org/drools/examples/decisiontable-template/
```
As an alternative to executing the decision tables using rule template files, you can use the DecisionTableConfiguration object and specify an input spreadsheet as the input type, such as DecisionTableInputType.xls:

```java
template/ExamplePolicyPricingTemplateData.xls
template="org/drools/examples/decisiontable-template/PromotionalPricing.drt"
row="18" col="3"/>
<ksession name="DTableWithTemplateKS"/>
</kbase>
```

As an alternative to executing the decision tables using rule template files, you can use the DecisionTableConfiguration object and specify an input spreadsheet as the input type, such as DecisionTableInputType.xls:

```
DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
    dtableconfiguration.setInputType( DecisionTableInputType.XLS );

    KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

        getClass() );

    kbuilder.add( xlsRes,
        ResourceType.DTABLE,
        dtableconfiguration );
```

The Pricing example uses two fact types:

- **Driver**
- **Policy**

The example sets the default values for both facts in their respective Java classes Driver.java and Policy.java. The Driver is 30 years old, has had no prior claims, and currently has a risk profile of LOW. The Policy that the driver is applying for is COMPREHENSIVE.

In any decision table, each row is considered a different rule and each column is a condition or an action. Each row is evaluated in a decision table unless the agenda is cleared upon execution.

Decision table spreadsheets (XLS or XLSX) require two key areas that define rule data:

- A RuleSet area
- A RuleTable area

The RuleSet area of the spreadsheet defines elements that you want to apply globally to all rules in the same package (not only the spreadsheet), such as a rule set name or universal rule attributes. The RuleTable area defines the actual rules (rows) and the conditions, actions, and other rule attributes (columns) that constitute that rule table within the specified rule set. A decision table spreadsheet can contain multiple RuleTable areas, but only one RuleSet area.
The RuleTable area also defines the objects to which the rule attributes apply, in this case Driver and Policy, followed by constraints on the objects. For example, the Driver object constraint that defines the Age Bracket column is \texttt{age >= $1, age <= $2}, where the comma-separated range is defined in the table column values, such as 18,24.

**Base pricing rules**

The Base pricing rules decision table in the Pricing example evaluates the age, risk profile, number of claims, and policy type of the driver and produces the base price of the policy based on these conditions.

The Driver attributes are defined in the following table columns:

- **Age Bracket**: The age bracket has a definition for the condition \texttt{age >=$1, age <=$2}, which defines the condition boundaries for the driver’s age. This condition column highlights the use of $1 and $2, which is comma delimited in the spreadsheet. You can write these values as 18,24 or 18, 24 and both formats work in the execution of the business rules.
- **Location risk profile**: The risk profile is a string that the example program passes always as **LOW** but can be changed to reflect **MED** or **HIGH**.

- **Number of prior claims**: The number of claims is defined as an integer that the condition column must exactly equal to trigger the action. The value is not a range, only exact matches.

The **Policy** of the decision table is used in both the conditions and the actions of the rule and has attributes defined in the following table columns:

- **Policy type applying for**: The policy type is a condition that is passed as a string that defines the type of coverage: **COMPREHENSIVE**, **FIRE_THEFT**, or **THIRD_PARTY**.

- **Base $ AUD**: The **basePrice** is defined as an **ACTION** that sets the price through the constraint `policy.setBasePrice($param);` based on the spreadsheet cells corresponding to this value. When you execute the corresponding DRL rule for this decision table, the **then** portion of the rule executes this action statement on the true conditions matching the facts and sets the base price to the corresponding value.

- **Record Reason**: When the rule successfully executes, this action generates an output message to the **System.out** console reflecting which rule fired. This is later captured in the application and printed.

The example also uses the first column on the left to categorize rules. This column is for annotation only and has no affect on rule execution.

### Promotional discount rules

The **Promotional discount rules** decision table in the Pricing example evaluates the age, number of prior claims, and policy type of the driver to generate a potential discount on the price of the insurance policy.

#### Figure 87.13. Discount calculation

This decision table contains the conditions for the discount for which the driver might be eligible. Similar to the base price calculation, this table evaluates the **Age**, **Number of prior claims** of the driver, and the **Policy type applying for** to determine a **Discount %** rate to be applied. For example, if the driver is 30 years old, has no prior claims, and is applying for a **COMPREHENSIVE** policy, the driver is given a discount of **20** percent.

### 87.6. PET STORE EXAMPLE DECISIONS (AGENDA GROUPS, GLOBAL VARIABLES, CALLBACKS, AND GUI INTEGRATION)

The Pet Store example decision set demonstrates how to use agenda groups and global variables in rules and how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based desktop application. The example also demonstrates how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.
The following is an overview of the Pet Store example:

- **Name**: petstore  
- **Main class**: org.drools.examples.petstore.PetStoreExample (in src/main/java)  
- **Module**: drools-examples  
- **Type**: Java application  
- **Rule file**: org.drools.examples.petstore.PetStore.drl (in src/main/resources)  
- **Objective**: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration

In the Pet Store example, the sample **PetStoreExample.java** class defines the following principal classes (in addition to several classes to handle Swing events):

- **Petstore** contains the **main()** method.  
- **PetStoreUI** is responsible for creating and displaying the Swing-based GUI. This class contains several smaller classes, mainly for responding to various GUI events, such as user mouse clicks.  
- **TableModel** holds the table data. This class is essentially a JavaBean that extends the Swing class **AbstractTableModel**.  
- **CheckoutCallback** enables the GUI to interact with the rules.  
- **Ordershow** keeps the items that you want to buy.  
- **Purchase** stores details of the order and the products that you are buying.  
- **Product** is a JavaBean containing details of the product available for purchase and its price.

Much of the Java code in this example is either plain JavaBean or Swing based. For more information about Swing components, see the Java tutorial on **Creating a GUI with JFC/Swing**.

**Rule execution behavior in the Pet Store example**

Unlike other example decision sets where the facts are asserted and fired immediately, the Pet Store example does not execute the rules until more facts are gathered based on user interaction. The example executes rules through a **PetStoreUI** object, created by a constructor, that accepts the **Vector** object **stock** for collecting the products. The example then uses an instance of the **CheckoutCallback** class containing the rule base that was previously loaded.

**Pet Store KIE container and fact execution setup**

```java
// KieServices is the factory for all KIE services.
KieServices ks = KieServices.Factory.get();

// Create a KIE container on the class path.
KieContainer kc = ks.getKieClasspathContainer();

// Create the stock.
Vector<Product> stock = new Vector<Product>();
stock.add(new Product( "Gold Fish", 5 ) );
stock.add(new Product( "Fish Tank", 25 ) );
stock.add(new Product( "Fish Food", 2 ) );
```
The Java code that fires the rules is in the `CheckoutCallBack.checkout()` method. This method is triggered when the user clicks `Checkout` in the UI.

**Rule execution from CheckoutCallBack.checkout()**

```java
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();

    // Iterate through list and add to cart.
    for (Product p: items) {
        order.addItem(new Purchase(order, p));
    }

    // Add the JFrame to the ApplicationData to allow for user interaction.
    // From the KIE container, a KIE session is created based on
    // its definition and configuration in the META-INF/kmodule.xml file.
    KieSession ksession = kcontainer.newKieSession("PetStoreKS");

    ksession.setGlobal("frame", frame);
    ksession.setGlobal("textArea", this.output);

    ksession.insert(new Product("Gold Fish", 5));
    ksession.insert(new Product("Fish Tank", 25));
    ksession.insert(new Product("Fish Food", 2));

    ksession.insert(new Product("Fish Food Sample", 0));

    ksession.setGlobal( "frame", frame );
    ksession.setGlobal( "textArea", this.output );

    ksession.insert( new Product( "Gold Fish", 5 ) );
    ksession.insert( new Product( "Fish Tank", 25 ) );
    ksession.insert( new Product( "Fish Food", 2 ) );

    ksession.insert( new Product( "Fish Food Sample", 0 ) );

    ksession.insert( order );

    // Execute rules.
    ksession.fireAllRules();

    // Return the state of the cart
    return order.toString();
}
```

The example code passes two elements into the `CheckoutCallBack.checkout()` method. One element is the handle for the `JFrame` Swing component surrounding the output text frame, found at the bottom of the GUI. The second element is a list of order items, which comes from the `TableModel` that stores the information from the `Table` area at the upper-right section of the GUI.

The `for` loop transforms the list of order items coming from the GUI into the `Order` JavaBean, also contained in the file `PetStoreExample.java`.

In this case, the rule is firing in a stateless KIE session because all of the data is stored in Swing components and is not executed until the user clicks `Checkout` in the UI. Each time the user clicks `Checkout`, the content of the list is moved from the Swing `TableModel` into the KIE session working memory and is then executed with the `ksession.fireAllRules()` method.
Within this code, there are nine calls to `KieSession`. The first of these creates a new `KieSession` from the `KieContainer` (the example passed in this `KieContainer` from the `CheckoutCallBack` class in the `main()` method). The next two calls pass in the two objects that hold the global variables in the rules: the Swing text area and the Swing frame used for writing messages. More inserts put information on products into the `KieSession`, as well as the order list. The final call is the standard `fireAllRules()`.

**Pet Store rule file imports, global variables, and Java functions**

The `PetStore.drl` file contains the standard package and import statements to make various Java classes available to the rules. The rule file also includes global variables to be used within the rules, defined as `frame` and `textArea`. The global variables hold references to the Swing components `JFrame` and `JTextArea` components that were previously passed on by the Java code that called the `setGlobal()` method. Unlike standard variables in rules, which expire as soon as the rule has fired, global variables retain their value for the lifetime of the KIE session. This means the contents of these global variables are available for evaluation on all subsequent rules.

**PetStore.drl package, imports, and global variables**

```java
package org.drools.examples;

import org.kie.api.runtime.KieRuntime;
import org.drools.examples.petstore.PetStoreExample.Order;
import org.drools.examples.petstore.PetStoreExample.Purchase;
import org.drools.examples.petstore.PetStoreExample.Product;
import java.util.ArrayList;
import javax.swing.JOptionPane;
import javax.swing.JFrame;

global JFrame frame
global javax.swing.JTextArea textArea
```

The `PetStore.drl` file also contains two functions that the rules in the file use:

**PetStore.drl Java functions**

```java
function void doCheckout(JFrame frame, KieRuntime krt) {
    Object[] options = {"Yes", "No"};

    int n = JOptionPane.showOptionDialog(frame,
        "Would you like to checkout?",
        "",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null,
        options,
        options[0]);

    if (n == 0) {
        krt.getAgenda().getAgendaGroup( "checkout" ).setFocus();
    }
}

function boolean requireTank(JFrame frame, KieRuntime krt, Order order, Product fishTank, int total) {
```
The two functions perform the following actions:

- **doCheckout()** displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the checkout agenda group, enabling rules in that group to (potentially) fire.

- **requireTank()** displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank **Product** is added to the order list in the working memory.

```java
Object[] options = {"Yes", "No"};

int n = JOptionPane.showOptionDialog(frame,
   "Would you like to buy a tank for your " + total + " fish?",
   "Purchase Suggestion",
   JOptionPane.YES_NO_OPTION,
   JOptionPane.QUESTION_MESSAGE,
   null,
   options,
   options[0]);

System.out.print( "SUGGESTION: Would you like to buy a tank for your " + total + " fish? - ");

if (n == 0) {
   Purchase purchase = new Purchase( order, fishTank );
   krt.insert( purchase );
   order.addItem( purchase );
   System.out.println( "Yes" );
} else {
   System.out.println( "No" );
}
return true;
```

The Pet Store example uses the following agenda groups for rules:

**NOTE**

For this example, all rules and functions are within the same rule file for efficiency. In a production environment, you typically separate the rules and functions in different files or build a static Java method and import the files using the import function, such as `import function my.package.name.hello`.

**Pet Store rules with agenda groups**

Most of the rules in the Pet Store example use agenda groups to control rule execution. Agenda groups allow you to partition the decision engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the `agenda-group` attribute to specify a different agenda group for the rule.

Initially, a working memory has its focus on the agenda group **MAIN**. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method `setFocus()` or the rule attribute `auto-focus`. The `auto-focus` attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

The Pet Store example uses the following agenda groups for rules:
For example, the sample rule "Explode Cart" uses the "init" agenda group to ensure that it has the option to fire and insert shopping cart items into the KIE session working memory:

Rule "Explode Cart"

```
// Insert each item in the shopping cart into the working memory.
rule "Explode Cart"
    agenda-group "init"
    auto-focus true
    salience 10
    when
        $order : Order( grossTotal == -1 )
        $item : Purchase() from $order.items
    then
        insert( $item );
        kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show items" ).setFocus();
        kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate" ).setFocus();
end
```

This rule matches against all orders that do not yet have their grossTotal calculated. The execution loops for each purchase item in that order.

The rule uses the following features related to its agenda group:

- **agenda-group "init"** defines the name of the agenda group. In this case, only one rule is in the group. However, neither the Java code nor a rule consequence sets the focus to this group, and therefore it relies on the **auto-focus** attribute for its chance to fire.

- **auto-focus true** ensures that this rule, while being the only rule in the agenda group, gets a chance to fire when fireAllRules() is called from the Java code.

- **kcontext....setFocus()** sets the focus to the "show items" and "evaluate" agenda groups, enabling their rules to fire. In practice, you loop through all items in the order, insert them into memory, and then fire the other rules after each insertion.

The "show items" agenda group contains only one rule, "Show Items". For each purchase in the order currently in the KIE session working memory, the rule logs details to the text area at the bottom of the GUI, based on the **textArea** variable defined in the rule file.

Rule "Show Items"

```
rule "Show Items"
    agenda-group "show items"
    when
        $order : Order()
        $p : Purchase( order == $order )
```

CHAPTER 87. EXAMPLE DECISIONS IN RED HAT DECISION MANAGER FOR AN IDE
then
textArea.append( $p.product + "\n" );
end

The "evaluate" agenda group also gains focus from the "Explode Cart" rule. This agenda group contains two rules, "Free Fish Food Sample" and "Suggest Tank", which are executed in that order.

Rule "Free Fish Food Sample"

// Free fish food sample when users buy a goldfish if they did not already buy
// fish food and do not already have a fish food sample.
rule "Free Fish Food Sample"
    agenda-group "evaluate"  
    when
        $order : Order()
        not ( $p : Product( name == "Fish Food") && Purchase( product == $p ) )  
        not ( $p : Product( name == "Fish Food Sample") && Purchase( product == $p ) )  
        exists ( $p : Product( name == "Gold Fish") && Purchase( product == $p ) )  
        $fishFoodSample : Product( name == "Fish Food Sample" );
    then
        System.out.println( "Adding free Fish Food Sample to cart" );
        purchase = new Purchase($order, $fishFoodSample);
        insert( purchase );
        $order.addItem( purchase );
    end

The rule "Free Fish Food Sample" fires only if all of the following conditions are true:

1. The agenda group "evaluate" is being evaluated in the rules execution.
2. User does not already have fish food.
3. User does not already have a free fish food sample.
4. User has a goldfish in the order.

If the order facts meet all of these requirements, then a new product is created (Fish Food Sample) and is added to the order in working memory.

Rule "Suggest Tank"

// Suggest a fish tank if users buy more than five goldfish and
// do not already have a tank.
rule "Suggest Tank"
    agenda-group "evaluate"
    when
        $order : Order()
        not ( $p : Product( name == "Fish Tank") && Purchase( product == $p ) )  
        ArrayList( $total : size > 5 ) from collect( Purchase( product.name == "Gold Fish" ) )  
        $fishTank : Product( name == "Fish Tank" )
    then
        requireTank(frame, kcontext.getKieRuntime(), $order, $fishTank, $total);
    end
The rule "Suggest Tank" fires only if the following conditions are true:

1. User does not have a fish tank in the order.
2. User has more than five fish in the order.

When the rule fires, it calls the requireTank() function defined in the rule file. This function displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank Product is added to the order list in the working memory. When the rule calls the requireTank() function, the rule passes the frame global variable so that the function has a handle for the Swing GUI.

The "do checkout" rule in the Pet Store example has no agenda group and no when conditions, so the rule is always executed and considered part of the default MAIN agenda group.

Rule "do checkout"

```java
rule "do checkout"
    when
    then
        doCheckout(frame, kcontext.getKieRuntime());
    end
```

When the rule fires, it calls the doCheckout() function defined in the rule file. This function displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the checkout agenda group, enabling rules in that group to (potentially) fire. When the rule calls the doCheckout() function, the rule passes the frame global variable so that the function has a handle for the Swing GUI.

**NOTE**

This example also demonstrates a troubleshooting technique if results are not executing as you expect: You can remove the conditions from the when statement of a rule and test the action in the then statement to verify that the action is performed correctly.

The "checkout" agenda group contains three rules for processing the order checkout and applying any discounts: "Gross Total", "Apply 5% Discount", and "Apply 10% Discount".

Rules "Gross Total", "Apply 5% Discount", and "Apply 10% Discount"

```java
rule "Gross Total"
    agenda-group "checkout"
    when
        $order : Order( grossTotal == -1)
        Number( total : doubleValue ) from accumulate( Purchase( $price : product.price ),
                                            sum( $price ) )
    then
        modify( $order ) { grossTotal = total }
        textArea.append( "gross total=\n" + total + "\n" );
    end

rule "Apply 5% Discount"
    agenda-group "checkout"
    when
        $order : Order( grossTotal >= 10 && < 20 )
then
   $order.discountedTotal = $order.grossTotal * 0.95;
   textArea.append( "discountedTotal total=\n" + $order.discountedTotal + \"n\" );
end

rule "Apply 10% Discount"
   agenda-group "checkout"
   when
     $order : Order( grossTotal >= 20 )
   then
     $order.discountedTotal = $order.grossTotal * 0.90;
     textArea.append( "discountedTotal total=\n" + $order.discountedTotal + \"n\" );
   end

If the user has not already calculated the gross total, the **Gross Total** accumulates the product prices into a total, puts this total into the KIE session, and displays it through the Swing **JTextArea** using the **textArea** global variable.

If the gross total is between 10 and 20 (currency units), the "**Apply 5% Discount**" rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

If the gross total is not less than 20, the "**Apply 10% Discount**" rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

**Pet Store example execution**

Similar to other Red Hat Decision Manager decision examples, you execute the Pet Store example by running the **org.drools.examples.petstore.PetStoreExample** class as a Java application in your IDE.

When you execute the Pet Store example, the **Pet Store Demo** GUI window appears. This window displays a list of available products (upper left), an empty list of selected products (upper right), **Checkout** and **Reset** buttons (middle), and an empty system messages area (bottom).
The following events occurred in this example to establish this execution behavior:

1. The `main()` method has run and loaded the rule base but has not yet fired the rules. So far, this is the only code in connection with rules that has been run.

2. A new `PetStoreUI` object has been created and given a handle for the rule base, for later use.

3. Various Swing components have performed their functions, and the initial UI screen is displayed and waits for user input.

You can click various products from the list to explore the UI setup:
No rules code has been fired yet. The UI uses Swing code to detect user mouse clicks and add selected products to the `TableModel` object for display in the upper-right corner of the UI. This example illustrates the Model-View-Controller design pattern.

When you click **Checkout**, the rules are then fired in the following way:

1. Method `CheckOutCallBack.checkout()` is called (eventually) by the Swing class waiting for a user to click **Checkout**. This inserts the data from the `TableModel` object (upper-right corner of the UI) into the KIE session working memory. The method then fires the rules.
2. The "Explode Cart" rule is the first to fire, with the auto-focus attribute set to true. The rule loops through all of the products in the cart, ensures that the products are in the working memory, and then gives the "show items" and "evaluate" agenda groups the option to fire. The rules in these groups add the contents of the cart to the text area (bottom of the UI), evaluate if you are eligible for free fish food, and determine whether to ask if you want to buy a fish tank.

Figure 87.16. Fish tank qualification

3. The "do checkout" rule is the next to fire because no other agenda group currently has focus and because it is part of the default MAIN agenda group. This rule always calls the doCheckout() function, which asks you if you want to check out.

4. The doCheckout() function sets the focus to the "checkout" agenda group, giving the rules in that group the option to fire.

5. The rules in the "checkout" agenda group display the contents of the cart and apply the appropriate discount.

6. Swing then waits for user input to either select more products (and cause the rules to fire again) or to close the UI.
You can add more `System.out` calls to demonstrate this flow of events in your IDE console:

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

87.7. HONEST POLITICIAN EXAMPLE DECISIONS (TRUTH MAINTENANCE AND SALIENCE)
The Honest Politician example decision set demonstrates the concept of truth maintenance with logical insertions and the use of salience in rules.

The following is an overview of the Honest Politician example:

- **Name**: honestpolitician
- **Main class**: org.drools.examples.honestpolitician.HonestPoliticianExample (in src/main/java)
- **Module**: drools-examples
- **Type**: Java application
- **Rule file**: org.drools.examples.honestpolitician.HonestPolitician.drl (in src/main/resources)
- **Objective**: Demonstrates the concept of truth maintenance based on the logical insertion of facts and the use of salience in rules

The basic premise of the Honest Politician example is that an object can only exist while a statement is true. A rule consequence can logically insert an object with the `insertLogical()` method. This means the object remains in the KIE session working memory as long as the rule that logically inserted it remains true. When the rule is no longer true, the object is automatically retracted.

In this example, rule execution causes a group of politicians to change from being honest to being dishonest as a result of a corrupt corporation. As each politician is evaluated, they start out with their honesty attribute being set to `true`, but a rule fires that makes the politicians no longer honest. As they switch their state from being honest to dishonest, they are then removed from the working memory. The rule salience notifies the decision engine how to prioritize any rules that have a salience defined for them, otherwise utilizing the default salience value of 0. Rules with a higher salience value are given higher priority when ordered in the activation queue.

**Politician and Hope classes**
The sample class `Politician` in the example is configured for an honest politician. The `Politician` class is made up of a String item `name` and a Boolean item `honest`:

**Politician class**

```java
public class Politician {
    private String name;
    private boolean honest;
    ... 
}
```

The `Hope` class determines if a `Hope` object exists. This class has no meaningful members, but is present in the working memory as long as society has hope.

**Hope class**

```java
public class Hope {
    public Hope() {
    }
}
```
Rule definitions for politician honesty
In the Honest Politician example, when at least one honest politician exists in the working memory, the "We have an honest Politician" rule logically inserts a new Hope object. As soon as all politicians become dishonest, the Hope object is automatically retracted. This rule has a salience attribute with a value of 10 to ensure that it fires before any other rule, because at that stage the "Hope is Dead" rule is true.

Rule "We have an honest politician"

```java
rule "We have an honest Politician"
   salience 10
   when
   exists( Politician( honest == true ) )
   then
   insertLogical( new Hope() );
end
```

As soon as a Hope object exists, the "Hope Lives" rule matches and fires. This rule also has a salience value of 10 so that it takes priority over the "Corrupt the Honest" rule.

Rule "Hope Lives"

```java
rule "Hope Lives"
   salience 10
   when
   exists( Hope() )
   then
   System.out.println("Hurrah!!! Democracy Lives");
end
```

Initially, four honest politicians exist so this rule has four activations, all in conflict. Each rule fires in turn, corrupting each politician so that they are no longer honest. When all four politicians have been corrupted, no politicians have the property honest == true. The rule "We have an honest Politician" is no longer true and the object it logically inserted (due to the last execution of new Hope()) is automatically retracted.

Rule "Corrupt the Honest"

```java
rule "Corrupt the Honest"
   when
   politician : Politician( honest == true )
   exists( Hope() )
   then
   System.out.println("I'm an evil corporation and I have corrupted " + politician.getName() );
   modify ( politician ){ honest = false };
end
```

With the Hope object automatically retracted through the truth maintenance system, the conditional element not applied to Hope is no longer true so that the "Hope is Dead" rule matches and fires.

Rule "Hope is Dead"

```java
rule "Hope is Dead"
   when
```
not( Hope() )
then
  System.out.println( "We are all Doomed!!! Democracy is Dead" );
end

Example execution and audit trail

In the HonestPoliticianExample.java class, the four politicians with the honest state set to true are inserted for evaluation against the defined business rules:

HonestPoliticianExample.java class execution

```java
public static void execute( KieContainer kc ) {
    KieSession ksession = kc.newKieSession("HonestPoliticianKS");

    final Politician p1 = new Politician( "President of Umpa Lumpa", true );
    final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
    final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
    final Politician p4 = new Politician( "Omnipotence Om", true );

    ksession.insert( p1 );
    ksession.insert( p2 );
    ksession.insert( p3 );
    ksession.insert( p4 );

    ksession.fireAllRules();
    ksession.dispose();
}
```

To execute the example, run the org.drools.examples.honestpolitician.HonestPoliticianExample class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Execution output in the IDE console

Hurrah!!! Democracy Lives
I'm an evil corporation and I have corrupted President of Umpa Lumpa
I'm an evil corporation and I have corrupted Prime Minster of Cheeseland
I'm an evil corporation and I have corrupted Tsar of Pringapopaloo
I'm an evil corporation and I have corrupted Omnipotence Om
We are all Doomed!!! Democracy is Dead

The output shows that, while there is at least one honest politician, democracy lives. However, as each politician is corrupted by some corporation, all politicians become dishonest, and democracy is dead.

To better understand the execution flow of this example, you can modify the HonestPoliticianExample.java class to include a DebugRuleRuntimeEventListener listener and an audit logger to view execution details:

HonestPoliticianExample.java class with an audit logger

```java
package org.drools.examples.honestpolitician;

import org.kie.api.KieServices;
```
import org.kie.api.event.rule.DebugAgendaEventListener;
import org.kie.api.event.rule.DebugRuleRuntimeEventListener;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class HonestPoliticianExample {

    /**
     * @param args
     */
    public static void main(final String[] args) {
        KieServices ks = KieServices.Factory.get();
        KieContainer kc = KieServices.Factory.get().getKieClasspathContainer();
        System.out.println(kc.verify().getMessages().toString());
        execute(kc);
    }

    public static void execute(KieServices ks, KieContainer kc) {
        KieSession ksession = kc.newKieSession("HonestPoliticianKS");
        final Politician p1 = new Politician("President of Umpa Lumpa", true);
        final Politician p2 = new Politician("Prime Minister of Cheeseland", true);
        final Politician p3 = new Politician("Tsar of Pringapopaloo", true);
        final Politician p4 = new Politician("Omnipotence Om", true);
        ksession.insert(p1);
        ksession.insert(p2);
        ksession.insert(p3);
        ksession.insert(p4);
        ksession.addEventListener(new DebugAgendaEventListener());
        ksession.addEventListener(new DebugRuleRuntimeEventListener());
        // Set up a file-based audit logger.
        ks.getLoggers().newFileLogger(ksession, ".target/honestpolitician");
        ksession.fireAllRules();
        ksession.dispose();
    }

    // The application can also setup listeners.
    }
Creates the listeners

Builds the log that can be passed into the debug view or Audit View or your IDE after executing of the rules

When you run the Honest Politician with this modified logging capability, you can load the audit log file from `target/honestpolitician.log` into your IDE debug view or Audit View, if available (for example, in Window → Show View in some IDEs).

In this example, the Audit View shows the flow of executions, insertions, and retractions as defined in the example classes and rules:

Figure 87.18. Honest Politician example Audit View

When the first politician is inserted, two activations occur. The rule "We have an honest Politician" is activated only one time for the first inserted politician because it uses an `exists` conditional element, which matches when at least one politician is inserted. The rule "Hope is Dead" is also activated at this stage because the Hope object is not yet inserted. The rule "We have an honest Politician" fires first because it has a higher salience value than the rule "Hope is Dead", and inserts the Hope object (highlighted in green). The insertion of the Hope object activates the rule "Hope Lives" and deactivates the rule "Hope is Dead". The insertion also activates the rule "Corrupt the Honest" for each inserted honest politician. The rule "Hope Lives" is executed and prints "Hurrah!!! Democracy Lives".

Next, for each politician, the rule "Corrupt the Honest" fires, printing "I'm an evil corporation and I have corrupted X", where X is the name of the politician, and modifies the politician honesty value to `false`. When the last honest politician is corrupted, Hope is automatically retracted by the truth maintenance system (highlighted in blue). The green highlighted area shows the origin of the currently selected blue highlighted area. After the Hope fact is retracted, the rule "Hope is dead" fires, printing "We are all Doomed!!! Democracy is Dead".

When the first politician is inserted, two activations occur. The rule "We have an honest Politician" is activated only one time for the first inserted politician because it uses an `exists` conditional element, which matches when at least one politician is inserted. The rule "Hope is Dead" is also activated at this stage because the Hope object is not yet inserted. The rule "We have an honest Politician" fires first because it has a higher salience value than the rule "Hope is Dead", and inserts the Hope object (highlighted in green). The insertion of the Hope object activates the rule "Hope Lives" and deactivates the rule "Hope is Dead". The insertion also activates the rule "Corrupt the Honest" for each inserted honest politician. The rule "Hope Lives" is executed and prints "Hurrah!!! Democracy Lives".
87.8. SUDOKU EXAMPLE DECISIONS (COMPLEX PATTERN MATCHING, CALLBACKS, AND GUI INTEGRATION)

The Sudoku example decision set, based on the popular number puzzle Sudoku, demonstrates how to use rules in Red Hat Decision Manager to find a solution in a large potential solution space based on various constraints. This example also shows how to integrate Red Hat Decision Manager rules into a graphical user interface (GUI), in this case a Swing-based desktop application, and how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.

The following is an overview of the Sudoku example:

- **Name:** sudoku
- **Main class:** `org.drools.examples.sudoku.SudokuExample` (in `src/main/java`)
- **Module:** drools-examples
- **Type:** Java application
- **Rule files:** `org.drools.examples.sudoku.*.drl` (in `src/main/resources`)
- **Objective:** Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration

Sudoku is a logic-based number placement puzzle. The objective is to fill a 9x9 grid so that each column, each row, and each of the nine 3x3 zones contains the digits from 1 to 9 only one time. The puzzle setter provides a partially completed grid and the puzzle solver’s task is to complete the grid with these constraints.

The general strategy to solve the problem is to ensure that when you insert a new number, it must be unique in its particular 3x3 zone, row, and column. This Sudoku example decision set uses Red Hat Decision Manager rules to solve Sudoku puzzles from a range of difficulty levels, and to attempt to resolve flawed puzzles that contain invalid entries.

**Sudoku example execution and interaction**

Similar to other Red Hat Decision Manager decision examples, you execute the Sudoku example by running the `org.drools.examples.sudoku.SudokuExample` class as a Java application in your IDE.

When you execute the Sudoku example, the Drools Sudoku Example GUI window appears. This window contains an empty grid, but the program comes with various grids stored internally that you can load and solve.

Click File → Samples → Simple to load one of the examples. Notice that all buttons are disabled until a grid is loaded.
Figure 87.19. Sudoku example GUI after launch

When you load the Simple example, the grid is filled according to the puzzle’s initial state.
Choose from the following options:

- Click **Solve** to fire the rules defined in the Sudoku example that fill out the remaining values and that make the buttons inactive again.
Click **Step** to see the next digit found by the rule set. The console window in your IDE displays detailed information about the rules that are executing to solve the step.

**Step execution output in the IDE console**

- single 8 at [0,1]
- column elimination due to [1,2]: remove 9 from [4,2]
- hidden single 9 at [1,2]
- row elimination due to [2,8]: remove 7 from [2,4]
- remove 6 from [3,8] due to naked pair at [3,2] and [3,7]
- hidden pair in row at [4,6] and [4,4]

Click **Dump** to see the state of the grid, with cells showing either the established value or the remaining possibilities.

**Dump execution output in the IDE console**

```
Col: 0  Col: 1  Col: 2  Col: 3  Col: 4  Col: 5  Col: 6  Col: 7  Col: 8
Row 0: 123456789  --- 5  --- 6  --- 8  --- 123456789  --- 1  --- 9  --- 4  --- 123456789
Row 1:  --- 9  --- 123456789 123456789  --- 6  --- 123456789  --- 5  --- 123456789 123456789  --- 3  ---
Row 2:  --- 7  --- 123456789 123456789  --- 4  --- 9  --- 3  --- 123456789 123456789  --- 8  ---
```
The Sudoku example includes a deliberately broken sample file that the rules defined in the example can resolve.

Click File → Samples → !DELIBERATELY BROKEN! to load the broken sample. The grid starts with some issues, for example, the value 5 appears two times in the first row, which is not allowed.

Figure 87.22. Broken Sudoku example initial state

Click Solve to apply the solving rules to this invalid grid. The associated solving rules in the Sudoku example detect the issues in the sample and attempts to solve the puzzle as far as possible. This process does not complete and leaves some cells empty.

The solving rule activity is displayed in the IDE console window:

Detected issues in the broken sample
The sample Sudoku files labeled **Hard** are more complex and the solving rules might not be able to solve them. The unsuccessful solution attempt is displayed in the IDE console window:

**Hard sample unresolved**

Validation complete.

... 

Sorry - can't solve this grid.

The rules that work to solve the broken sample implement standard solving techniques based on the sets of values that are still candidates for a cell. For example, if a set contains a single value, then this is the value for the cell. For a single occurrence of a value in one of the groups of nine cells, the rules insert a fact of type **Setting** with the solution value for some specific cell. This fact causes the elimination of this value from all other cells in any of the groups the cell belongs to and the value is retracted.

Other rules in the example reduce the permissible values for some cells. The rules "**naked pair**", "**hidden pair in row**", "**hidden pair in column**", and "**hidden pair in square**" eliminate possibilities but do not establish solutions. The rules "**X-wings in rows**", "**X-wings in columns**", "**intersection removal**"
row”, and "intersection removal column" perform more sophisticated eliminations.

Sudoku example classes
The package org.drools.examples.sudoku.swing contains the following core set of classes that implement a framework for Sudoku puzzles:

- The SudokuGridModel class defines an interface that is implemented to store a Sudoku puzzle as a 9x9 grid of Cell objects.
- The SudokuGridView class is a Swing component that can visualize any implementation of the SudokuGridModel class.
- The SudokuGridEvent and SudokuGridListener classes communicate state changes between the model and the view. Events are fired when a cell value is resolved or changed.
- The SudokuGridSamples class provides partially filled Sudoku puzzles for demonstration purposes.

NOTE
This package does not have any dependencies on Red Hat Decision Manager libraries.

The package org.drools.examples.sudoku contains the following core set of classes that implement the elementary Cell object and its various aggregations:

- The CellFile class, with subtypes CellRow, CellCol, and CellSqr, all of which are subtypes of the CellGroup class.
- The Cell and CellGroup subclasses of SetOfNine, which provides a property free with the type Set<Integer>. For a Cell class, the set represents the individual candidate set. For a CellGroup class, the set is the union of all candidate sets of its cells (the set of digits that still need to be allocated).

In the Sudoku example are 81 Cell and 27 CellGroup objects and a linkage provided by the Cell properties cellRow, cellCol, and cellSqr, and by the CellGroup property cells (a list of Cell objects). With these components, you can write rules that detect the specific situations that permit the allocation of a value to a cell or the elimination of a value from some candidate set.

- The Setting class is used to trigger the operations that accompany the allocation of a value. The presence of a Setting fact is used in all rules that detect a new situation in order to avoid reactions to inconsistent intermediary states.
- The Stepping class is used in a low priority rule to execute an emergency halt when a “Step” does not terminate regularly. This behavior indicates that the program cannot solve the puzzle.
- The main class org.drools.examples.sudoku.SudokuExample implements a Java application combining all of these components.

Sudoku validation rules (validate.drl)
The validate.drl file in the Sudoku example contains validation rules that detect duplicate numbers in cell groups. They are combined in a “validate” agenda group that enables the rules to be explicitly activated after a user loads the puzzle.

The when conditions of the three rules "duplicate in cell ..." all function in the following ways:

- The first condition in the rule locates a cell with an allocated value.
The second condition in the rule pulls in any of the three cell groups to which the cell belongs.

The final condition finds a cell (other than the first one) with the same value as the first cell and in the same row, column, or square, depending on the rule.

Rules "duplicate in cell ..."

```java
rule "duplicate in cell row"
  when
  $c: Cell( $v: value != null )
  $cr: CellRow( cells contains $c )
  exists Cell( this != $c, value == $v, cellRow == $cr )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in row " + $cr.getNumber() );
end

rule "duplicate in cell col"
  when
  $c: Cell( $v: value != null )
  $cc: CellCol( cells contains $c )
  exists Cell( this != $c, value == $v, cellCol == $cc )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in col " + $cc.getNumber() );
end

rule "duplicate in cell sqr"
  when
  $c: Cell( $v: value != null )
  $cs: CellSqr( cells contains $c )
  exists Cell( this != $c, value == $v, cellSqr == $cs )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in its square of nine." );
end
```

The rule "terminate group" is the last to fire. This rule prints a message and stops the sequence.

Rule "terminate group"

```java
rule "terminate group"
  salience -100
  when
  then
    System.out.println( "Validation complete." );
drools.halt();
end
```

Sudoku solving rules (sudoku.drl)
The `sudoku.drl` file in the Sudoku example contains three types of rules: one group handles the allocation of a number to a cell, another group detects feasible allocations, and the third group eliminates values from candidate sets.

The rules "set a value", "eliminate a value from Cell", and "retract setting" depend on the presence of a `Setting` object. The first rule handles the assignment to the cell and the operations for removing the value from the `free` sets of the three groups of the cell. This group also reduces a counter that, when zero, returns control to the Java application that has called `fireUntilHalt()`.
The purpose of the rule "eliminate a value from Cell" is to reduce the candidate lists of all cells that are related to the newly assigned cell. Finally, when all eliminations have been made, the rule "retract setting" retracts the triggering Setting fact.

Rules "set a value", "eliminate a value from a Cell", and "retract setting"

// A Setting object is inserted to define the value of a Cell.  
// Rule for updating the cell and all cell groups that contain it  
rule "set a value"  
  when  
    // A Setting with row and column number, and a value  
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )  
    // A matching Cell, with no value set  
    $c: Cell( rowNo == $rn, colNo == $cn, value == null,  
              $cr: cellRow, $cc: cellCol, $cs: cellSqr )  
    // Count down  
    $ctr: Counter( $count: count )  
  then  
    // Modify the Cell by setting its value.  
    modify( $c ){ setValue( $v ) }  
    // System.out.println( "set cell " + $c.toString() );  
    modify( $cr ){ blockValue( $v ) }  
    modify( $cc ){ blockValue( $v ) }  
    modify( $cs ){ blockValue( $v ) }  
    modify( $ctr ){ setCount( $count - 1 ) }  
  end

// Rule for removing a value from all cells that are siblings  
// in one of the three cell groups  
rule "eliminate a value from Cell"  
  when  
    // A Setting with row and column number, and a value  
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )  
    // The matching Cell, with the value already set  
    Cell( rowNo == $rn, colNo == $cn, value == $v, $exCells: exCells )  
    // For all Cells that are associated with the updated cell  
    $c: Cell( free contains $v ) from $exCells  
  then  
    // System.out.println( "clear " + $v + " from cell " + $c.posAsString() );  
    modify( $c ){ blockValue( $v ) }  
  end

// Rule for eliminating the Setting fact  
rule "retract setting"  
  when  
    // A Setting with row and column number, and a value  
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )  
    // The matching Cell, with the value already set  
    $c: Cell( rowNo == $rn, colNo == $cn, value == $v )
Two solving rules detect a situation where an allocation of a number to a cell is possible. The rule "single" fires for a Cell with a candidate set containing a single number. The rule "hidden single" fires when no cell exists with a single candidate, but when a cell exists containing a candidate, this candidate is absent from all other cells in one of the three groups to which the cell belongs. Both rules create and insert a Setting fact.

Rules "single" and "hidden single"

// Detect a set of candidate values with cardinality 1 for some Cell.  
// This is the value to be set.  
rule "single"  
when  
  // Currently no setting underway  
  not Setting()  
  // One element in the "free" set  
  $c: Cell( $rn: rowNo, $cn: colNo, freeCount == 1 )  
then  
  Integer i = $c.getFreeValue();  
  if (explain) System.out.println( "single " + i + " at " + $c.posAsString() );  
  // Insert another Setter fact.  
  insert( new Setting( $rn, $cn, i ) );  
end  

// Detect a set of candidate values with a value that is the only one  
in one of its groups. This is the value to be set.  
rule "hidden single"  
when  
  // Currently no setting underway  
  not Setting()  
  not Cell( freeCount == 1 )  
  // Some integer  
  $i: Integer()  
  // The "free" set contains this number  
  $c: Cell( $rn: rowNo, $cn: colNo, freeCount > 1, free contains $i )  
  // A cell group contains this cell $c.  
  $cg: CellGroup( cells contains $c )  
  // No other cell from that group contains $i.  
  not ( Cell( this != $c, free contains $i ) from $cg.getCells() )  
then  
  if (explain) System.out.println( "hidden single " + $i + " at " + $c.posAsString() );
Rules from the largest group, either individually or in groups of two or three, implement various solving techniques used for solving Sudoku puzzles manually.

The rule "naked pair" detects identical candidate sets of size 2 in two cells of a group. These two values may be removed from all other candidate sets of that group.

**Rule "naked pair"**

// A "naked pair" is two cells in some cell group with their sets of permissible values being equal with cardinality 2. These two values can be removed from all other candidate lists in the group.
rule "naked pair"
  when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )
  
    // One cell with two candidates
    $c1: Cell( freeCount == 2, $f1: free, $r1: cellRow, $rn1: rowNo, $cn1: colNo, $b1: cellSqr )
  
    // The containing cell group
    $cg: CellGroup( freeCount > 2, cells contains $c1 )
  
    // Another cell with two candidates, not the one we already have
    $c2: Cell( this != $c1, free == $f1 , rowNo >= $rn1, colNo >= $cn1 ) from $cg.cells
  
    // Get one of the "naked pair".
    Integer( $v: intValue ) from $c1.getFree()
  
    // Get some other cell with a candidate equal to one from the pair.
    $c3: Cell( this != $c1 && != $c2, freeCount > 1, free contains $v ) from $cg.cells
  
  then
    if (explain) System.out.println( "remove " + $v + " from " + $c3.posAsString() + " due to naked pair at " + $c1.posAsString() + " and " + $c2.posAsString() );
    // Remove the value.
    modify( $c3 ){ blockValue( $v )}
  end

The three rules "hidden pair in ..." functions similarly to the rule "naked pair". These rules detect a subset of two numbers in exactly two cells of a group, with neither value occurring in any of the other cells of the group. This means that all other candidates can be eliminated from the two cells harboring the hidden pair.

**Rules "hidden pair in ..."**

// If two cells within the same cell group contain candidate sets with more than two values, with two values being in both of them but in none of the other cells, then we have a "hidden pair". We can remove all other candidates from these two cells.
rule "hidden pair in row"
  when
    // Insert another Setter fact.
    insert( new Setting( $rn, $cn, $i ) );
end
// Currently no setting underway
not Setting()
not Cell( freeCount == 1 )

// Establish a pair of Integer facts.
$i1: Integer()
$i2: Integer( this > $i1 )

// Look for a Cell with these two among its candidates. (The upper bound on
// the number of candidates avoids a lot of useless work during startup.)
$c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 & & < 9, free contains $i1 & & contains $i2,
$cellRow: cellRow )

// Get another one from the same row, with the same pair among its candidates.
$c2: Cell( this != $c1, cellRow == $cellRow, freeCount > 2, free contains $i1 & & contains $i2 )

// Ascertain that no other cell in the group has one of these two values.
not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellRow.getCells() )
then
if ( explain ) System.out.println( "hidden pair in row at " + $c1.posAsString() + " and " +
$c2.posAsString() ) ;
// Set the candidate lists of these two Cells to the "hidden pair".
modify( $c1 ){ blockExcept( $i1, $i2 ) }
modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in column"
when
not Setting()
not Cell( freeCount == 1 )

$i1: Integer()
$i2: Integer( this > $i1 )
$c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 & & < 9, free contains $i1 & & contains $i2,
$cellCol: cellCol )
$c2: Cell( this != $c1, cellCol == $cellCol, freeCount > 2, free contains $i1 & & contains $i2 )
not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellCol.getCells() )
then
if ( explain ) System.out.println( "hidden pair in column at " + $c1.posAsString() + " and " +
$c2.posAsString() ) ;
modify( $c1 ){ blockExcept( $i1, $i2 ) }
modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in square"
when
not Setting()
not Cell( freeCount == 1 )

$i1: Integer()
$i2: Integer( this > $i1 )
$c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 & & < 9, free contains $i1 & & contains $i2,
$cellSqr: cellSqr )
$c2: Cell( this != $c1, cellSqr == $cellSqr, freeCount > 2, free contains $i1 & & contains $i2 )
not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellSqr.getCells() )
then
if (explain) System.out.println("hidden pair in square " + $c1.posAsString() + " and " + $c2.posAsString() );
modify( $c1 ){ blockExcept( $i1, $i2 ) }
modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

Two rules deal with "X-wings" in rows and columns. When only two possible cells for a value exist in each of two different rows (or columns) and these candidates lie also in the same columns (or rows), then all other candidates for this value in the columns (or rows) can be eliminated. When you follow the pattern sequence in one of these rules, notice how the conditions that are conveniently expressed by words such as same or only result in patterns with suitable constraints or that are prefixed with not.

Rules "X-wings in ..."

rule "X-wings in rows"
when
not Setting()
not Cell( freeCount == 1 )

$i: Integer()$
$ca1: Cell( freeCount > 1, free contains $i,$ra: cellRow, $rano: rowNo, $c1: cellCol, $c1no: colNo )$
$cb1: Cell( freeCount > 1, free contains $i,$rb: cellRow, $rbno: rowNo > $rano, cellCol == $c1 )
not( Cell( this != $ca1 && != $cb1, free contains $i ) from $c1.getCells() )$

$ca2: Cell( freeCount > 1, free contains $i,$c2: cellCol, $c2no: colNo > $c1no )$
$cb2: Cell( freeCount > 1, free contains $i,$ra: cellRow, $c2no: colNo > $c1no, cellCol == $c2 )
not( Cell( this != $ca2 && != $cb2, free contains $i ) from $c2.getCells() )$

$cx: Cell( rowNo == $rano || == $rbno, colNo != $c1no && != $c2no,
freeCount > 1, free contains $i )$
then
if (explain) {
    System.out.println("X-wing with " + $i + " in rows " +
    $ca1.posAsString() + " - " + $cb1.posAsString() +
    $ca2.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString() );
} modify( $cx ){ blockValue( $i ) }
end

rule "X-wings in columns"
when
not Setting()
not Cell( freeCount == 1 )

$i: Integer()$
$ca1: Cell( freeCount > 1, free contains $i,$c1: cellCol, $c1no: colNo, $ra: cellRow, $rano: rowNo )$
$ca2: Cell( freeCount > 1, free contains $i,$c2: cellCol, $c2no: colNo > $c1no, cellRow == $ra )
not( Cell( this != $ca1 && != $ca2, free contains $i ) from $ra.getCells() )

$cb1: Cell( freeCount > 1, free contains $i,
The two rules "intersection removal ..." are based on the restricted occurrence of some number within one square, either in a single row or in a single column. This means that this number must be in one of those two or three cells of the row or column and can be removed from the candidate sets of all other cells of the group. The pattern establishes the restricted occurrence and then fires for each cell outside of the square and within the same cell file.

Rules "intersection removal ...

rule "intersection removal column"
when
not Setting() not Cell( freeCount == 1 )

$i$: Integer() // Occurs in a Cell
$c$: Cell( free contains $i$, $cs$: cellSqr, $cc$: cellCol ) // Does not occur in another cell of the same square and a different column not Cell( this !=$c$, free contains $i$, cellSqr == $cs$, cellCol != $cc$ ) // A cell exists in the same column and another square containing this value.
$cx$: Cell( freeCount > 1, free contains $i$, cellCol == $cc$, cellSqr != $cs$ ) then // Remove the value from that other cell.
if (explain) {
    System.out.println( "column elimination due to " + $c.posAsString() + "\: remove "$ + $i + " from " + $cx.posAsString() );
}
modify( $cx ){ blockValue( $i ) } end

rule "intersection removal row"
when
not Setting() not Cell( freeCount == 1 )

$i$: Integer() // Occurs in a Cell
$c$: Cell( free contains $i$, $cs$: cellSqr, $cr$: cellRow ) // Does not occur in another cell of the same square and a different row. not Cell( this !=$c$, free contains $i$, cellSqr == $cs$, cellRow != $cr$ )
// A cell exists in the same row and another square containing this value.
$cx: Cell( freeCount > 1, free contains $i, cellRow == $cr, cellSqr != $cs )
then
// Remove the value from that other cell.
if (explain) {
    System.out.println("row elimination due to "+$c.posAsString() +
                      ": remove "+$i+" from "+$cx.posAsString());
}
modify($cx){blockValue($i)}
end

These rules are sufficient for many but not all Sudoku puzzles. To solve very difficult grids, the rule set requires more complex rules. (Ultimately, some puzzles can be solved only by trial and error.)

### 87.9. CONWAY’S GAME OF LIFE EXAMPLE DECISIONS (RULEFLOW GROUPS AND GUI INTEGRATION)

The Conway’s Game of Life example decision set, based on the famous cellular automaton by John Conway, demonstrates how to use ruleflow groups in rules to control rule execution. The example also demonstrates how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based implementation of Conway’s Game of Life.

The following is an overview of the Conway’s Game of Life (Conway) example:

- **Name**: conway
- **Main classes**: org.drools.examples.conway.ConwayRuleFlowGroupRun, org.drools.examples.conway.ConwayAgendaGroupRun (in src/main/java)
- **Module**: droolsjbpm-integration-examples
- **Type**: Java application
- **Rule files**: org.drools.examples.conway.*.drl (in src/main/resources)
- **Objective**: Demonstrates ruleflow groups and GUI integration

**NOTE**

The Conway’s Game of Life example is separate from most of the other example decision sets in Red Hat Decision Manager and is located in ~/rhdm-7.10.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples of the Red Hat Decision Manager 7.10.0 Source Distribution from the Red Hat Customer Portal.

In Conway’s Game of Life, a user interacts with the game by creating an initial configuration or an advanced pattern with defined properties and then observing how the initial state evolves. The objective of the game is to show the development of a population, generation by generation. Each generation results from the preceding one, based on the simultaneous evaluation of all cells.

The following basic rules govern what the next generation looks like:

- If a live cell has fewer than two live neighbors, it dies of loneliness.
• If a live cell has more than three live neighbors, it dies from overcrowding.
• If a dead cell has exactly three live neighbors, it comes to life.

Any cell that does not meet any of those criteria is left as is for the next generation.

The Conway’s Game of Life example uses Red Hat Decision Manager rules with ruleflow-group attributes to define the pattern implemented in the game. The example also contains a version of the decision set that achieves the same behavior using agenda groups. Agenda groups enable you to partition the decision engine agenda to provide execution control over groups of rules. By default, all rules are in the agenda group MAIN. You can use the agenda-group attribute to specify a different agenda group for the rule.

This overview does not explore the version of the Conway example using agenda groups. For more information about agenda groups, see the Red Hat Decision Manager example decision sets that specifically address agenda groups.

Conway example execution and interaction
Similar to other Red Hat Decision Manager decision examples, you execute the Conway ruleflow example by running the org.drools.examples.conway.ConwayRuleFlowGroupRun class as a Java application in your IDE.

When you execute the Conway example, the Conway’s Game of Life GUI window appears. This window contains an empty grid, or “arena” where the life simulation takes place. Initially the grid is empty because no live cells are in the system yet.

Figure 87.24. Conway example GUI after launch

Select a predefined pattern from the Pattern drop-down menu and click Next Generation to click through each population generation. Each cell is either alive or dead, where live cells contain a green ball. As the population evolves from the initial pattern, cells live or die relative to neighboring cells, according to the rules of the game.
Neighbors include not only cells to the left, right, top, and bottom but also cells that are connected diagonally, so that each cell has a total of eight neighbors. Exceptions are the corner cells, which have only three neighbors, and the cells along the four borders, with five neighbors each.

You can manually intervene to create or kill cells by clicking the cell.

To run through an evolution automatically from the initial pattern, click Start.

**Conway example rules with ruleflow groups**

The rules in the ConwayRuleFlowGroupRun example use ruleflow groups to control rule execution. A ruleflow group is a group of rules associated by the ruleflow-group rule attribute. These rules can only fire when the group is activated. The group itself can only become active when the elaboration of the ruleflow diagram reaches the node representing the group.

The Conway example uses the following ruleflow groups for rules:

- "register neighbor"
- "evaluate"
- "calculate"
- "reset calculate"
- "birth"
- "kill"
- "kill all"

All of the Cell objects are inserted into the KIE session and the "register ..." rules in the ruleflow group "register neighbor" are allowed to execute by the ruleflow process. This group of four rules creates Neighbor relations between some cell and its northeastern, northern, northwestern, and western neighbors.
This relation is bidirectional and handles the other four directions. Border cells do not require any special treatment. These cells are not paired with neighboring cells where there is not any.

By the time all activations have fired for these rules, all cells are related to all their neighboring cells.

Rules "register ..."

```plaintext
rule "register north east"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northEast : Cell( row == ($row - 1), col == ( $col + 1 ) )
  then
    insert( new Neighbor( $cell, $northEast ) );
    insert( new Neighbor( $northEast, $cell ) );
end

rule "register north"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $north : Cell( row == ($row - 1), col == $col )
  then
    insert( new Neighbor( $cell, $north ) );
    insert( new Neighbor( $north, $cell ) );
end

rule "register north west"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northWest : Cell( row == ($row - 1), col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $northWest ) );
    insert( new Neighbor( $northWest, $cell ) );
end

rule "register west"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $west : Cell( row == $row, col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $west ) );
    insert( new Neighbor( $west, $cell ) );
end
```

After all the cells are inserted, some Java code applies the pattern to the grid, setting certain cells to Live. Then, when the user clicks Start or Next Generation, the example executes the Generation ruleflow. This ruleflow manages all changes of cells in each generation cycle.
The ruleflow process enters the "evaluate" ruleflow group and any active rules in the group can fire. The rules "Kill the …" and "Give Birth" in this group apply the game rules to birth or kill cells. The example uses the phase attribute to drive the reasoning of the Cell object by specific groups of rules. Typically, the phase is tied to a ruleflow group in the ruleflow process definition.

Notice that the example does not change the state of any Cell objects at this point because it must complete the full evaluation before those changes can be applied. The example sets the cell to a phase that is either Phase.KILL or Phase.BIRTH, which is used later to control actions applied to the Cell object.

Rules "Kill the ..." and "Give Birth"

```java
rule "Kill The Lonely"
  ruleflow-group "evaluate"
  no-loop
  when
    // A live cell has fewer than 2 live neighbors.
    theCell: Cell( liveNeighbors < 2, cellState == CellState.LIVE,
        phase == Phase.EVALUATE )
  then
    modify( theCell ){ setPhase( Phase.KILL ); }
end

rule "Kill The Overcrowded"
  ruleflow-group "evaluate"
  no-loop
  when
    // A live cell has more than 3 live neighbors.
    theCell: Cell( liveNeighbors > 3, cellState == CellState.LIVE,
        phase == Phase.EVALUATE )
  then
```
```java
modify( theCell ){
    setPhase( Phase.KILL );
}
end

rule "Give Birth"
    ruleflow-group "evaluate"
    no-loop
    when
        // A dead cell has 3 live neighbors.
        theCell: Cell( liveNeighbors == 3, cellState == CellState.DEAD,
                       phase == Phase.EVALUATE )
    then
        modify( theCell ){
            theCell.setPhase( Phase.BIRTH );
        }
end

After all Cell objects in the grid have been evaluated, the example uses the "reset calculate" rule to clear any activations in the "calculate" ruleflow group. The example then enters a split in the ruleflow that enables the rules "kill" and "birth" to fire, if the ruleflow group is activated. These rules apply the state change.

Rules "reset calculate", "kill", and "birth"

```
At this stage, several **Cell** objects have been modified with the state changed to either **LIVE** or **DEAD**. When a cell becomes live or dead, the example uses the **Neighbor** relation in the rules "Calculate ..." to iterate over all surrounding cells, increasing or decreasing the **liveNeighbor** count. Any cell that has its count changed is also set to the **EVALUATE** phase to make sure it is included in the reasoning during the evaluation stage of the ruleflow process.

After the live count has been determined and set for all cells, the ruleflow process ends. If the user initially clicked **Start**, the decision engine restarts the ruleflow at that point. If the user initially clicked **Next Generation**, the user can request another generation.

Rules "Calculate ...

```java
rule "Calculate Live"
  ruleflow-group "calculate"
  lock-on-active
  when
    theCell: Cell( cellState == CellState.LIVE )
    Neighbor( cell == theCell, $neighbor : neighbor )
  then
    modify( $neighbor ){
      setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 ),
      setPhase( Phase.EVALUATE );
    }
  end

rule "Calculate Dead"
  ruleflow-group "calculate"
  lock-on-active
  when
    theCell: Cell( cellState == CellState.DEAD )
    Neighbor( cell == theCell, $neighbor : neighbor )
  then
    modify( $neighbor ){
      setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 ),
      setPhase( Phase.EVALUATE );
    }
  end
```

87.10. HOUSE OF DOOM EXAMPLE DECISIONS (BACKWARD CHAINING AND RECURSION)

The House of Doom example decision set demonstrates how the decision engine uses backward chaining and recursion to reach defined goals or subgoals in a hierarchical system.

The following is an overview of the House of Doom example:

- **Name**: backwardchaining
- **Main class**: `org.drools.examples.backwardchaining.HouseOfDoomMain` (in `src/main/java`)
- **Module**: drools-examples
- **Type**: Java application
- **Rule file:** `org.drools.examples.backwardchaining.BC-Example.drl` (in `src/main/resources`)
- **Objective:** Demonstrates backward chaining and recursion

A backward-chaining rule system is a goal-driven system that starts with a conclusion that the decision 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.

In contrast, a forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision 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.

The decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:
The House of Doom example uses rules with various types of queries to find the location of rooms and items within the house. The sample class `Location.java` contains the item and location elements used in the example. The sample class `HouseOfDoomMain.java` inserts the items or rooms in their respective locations in the house and executes the rules.

**Items and locations in HouseOfDoomMain.java class**

```java
ksession.insert(new Location("Office", "House"));
ksession.insert(new Location("Kitchen", "House"));
ksession.insert(new Location("Knife", "Kitchen"));
ksession.insert(new Location("Cheese", "Kitchen"));
```
The example rules rely on backward chaining and recursion to determine the location of all items and rooms in the house structure.

The following diagram illustrates the structure of the House of Doom and the items and rooms within it:

**Figure 87.28. House of Doom structure**

```java
ksession.insert( new Location("Desk", "Office"));
ksession.insert( new Location("Chair", "Office"));
ksession.insert( new Location("Computer", "Desk"));
ksession.insert( new Location("Drawer", "Desk"));
```

To execute the example, run the `org.drools.examples.backwardchaining.HouseOfDoomMain` class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

**Execution output in the IDE console**

```
go1
Office is in the House
---
go2
Drawer is in the House
---
go3
---
Key is in the Office
---
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
---
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
```
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk

All rules in the example have fired to detect the location of all items in the house and to print the location of each in the output.

**Recursive query and related rules**

A recursive query repeatedly searches through the hierarchy of a data structure for relationships between elements.

In the House of Doom example, the `BC-Example.drl` file contains an `isContainedIn` query that most of the rules in the example use to recursively evaluate the house data structure for data inserted into the decision engine:

**Recursive query in BC-Example.drl**

```drl
query isContainedIn( String x, String y )
    Location( x, y; )
    or
    ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

The rule "go" prints every string inserted into the system to determine how items are implemented, and the rule "go1" calls the query `isContainedIn`:

**Rules "go" and "go1"**

```drl
rule "go" salience 10
    when
        $s : String()
    then
        System.out.println( $s );
end

rule "go1"
    when
        String( this == "go1" )
        isContainedIn("Office", "House");
    then
        System.out.println( "Office is in the House" );
end
```

690
The example inserts the "go1" string into the decision engine and activates the "go1" rule to detect that item **Office** is in the location **House**:

**Insert string and fire rules**

```java
ksession.insert( "go1" );
ksession.fireAllRules();
```

**Rule "go1" output in the IDE console**

```
go1
Office is in the House
```

**Transitive closure rule**

Transitive closure is a relationship between an element contained in a parent element that is multiple levels higher in a hierarchical structure.

The rule "go2" identifies the transitive closure relationship of the **Drawer** and the **House**: The **Drawer** is in the **Desk** in the **Office** in the **House**.

```java
rule "go2"
when
  String( this == "go2" )
  isContainedIn("Drawer", "House"; )
then
  System.out.println( "Drawer is in the House" );
end
```

The example inserts the "go2" string into the decision engine and activates the "go2" rule to detect that item **Drawer** is ultimately within the location **House**:

**Insert string and fire rules**

```java
ksession.insert( "go2" );
ksession.fireAllRules();
```

**Rule "go2" output in the IDE console**

```
go2
Drawer is in the House
```

The decision engine determines this outcome based on the following logic:

1. The query recursively searches through several levels in the house to detect the transitive closure between **Drawer** and **House**.

2. Instead of using `Location( x, y; )`, the query uses the value of `(z, y; )` because **Drawer** is not directly in **House**.

3. The `z` argument is currently unbound, which means it has no value and returns everything that is in the argument.

4. The `y` argument is currently bound to **House**, so `z` returns **Office** and **Kitchen**.

---

**CHAPTER 87. EXAMPLE DECISIONS IN RED HAT DECISION MANAGER FOR AN IDE**

691
5. The query gathers information from the Office and checks recursively if the Drawer is in the Office. The query line `isContainedIn(x, z; )` is called for these parameters.

6. No instance of Drawer exists directly in Office, so no match is found.

7. With z unbound, the query returns data within the Office and determines that z == Desk.

```
isContainedIn(x==drawer, z==desk)
```

8. The `isContainedIn` query recursively searches three times, and on the third time, the query detects an instance of Drawer in Desk.

```
Location(x==drawer, y==desk)
```

9. After this match on the first location, the query recursively searches back up the structure to determine that the Drawer is in the Desk, the Desk is in the Office, and the Office is in the House. Therefore, the Drawer is in the House and the rule is satisfied.

Reactive query rule
A reactive query searches through the hierarchy of a data structure for relationships between elements and is dynamically updated when elements in the structure are modified.

The rule "go3" functions as a reactive query that detects if a new item Key ever becomes present in the Office by transitive closure: A Key in the Drawer in the Office.

**Rule "go3"**

```
rule "go3"
when
  String( this == "go3" )
  isContainedIn("Key", "Office"; )
then
  System.out.println( "Key is in the Office" );
end
```

The example inserts the "go3" string into the decision engine and activates the "go3" rule. Initially, this rule is not satisfied because no item Key exists in the house structure, so the rule produces no output.

**Insert string and fire rules**

```java
ksession.insert( "go3" );
ksession.fireAllRules();
```

**Rule "go3" output in the IDE console (unsatisfied)**

```
go3
```

The example then inserts a new item Key in the location Drawer, which is in Office. This change satisfies the transitive closure in the "go3" rule and the output is populated accordingly.

**Insert new item location and fire rules**
ksession.insert( new Location("Key", "Drawer") );
ksession.fireAllRules();

**Rule "go3" output in the IDE console (satisfied)**

Key is in the Office

This change also adds another level in the structure that the query includes in subsequent recursive searches.

**Queries with unbound arguments in rules**

A query with one or more unbound arguments returns all undefined (unbound) items within a defined (bound) argument of the query. If all arguments in a query are unbound, then the query returns all items within the scope of the query.

The rule **"go4"** uses an unbound argument **thing** to search for all items within the bound argument **Office**, instead of using a bound argument to search for a specific item in the **Office**:

**Rule "go4"**

```
rule "go4"
  when
    String( this == "go4" )
    isContainedIn(thing, "Office"; )
  then
    System.out.println( thing + "is in the Office" );
end
```

The example inserts the **"go4"** string into the decision engine and activates the **"go4"** rule to return all items in the **Office**:

**Insert string and fire rules**

```
ksession.insert( "go4" );
ksession.fireAllRules();
```

**Rule "go4" output in the IDE console**

```
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
```

The rule **"go5"** uses both unbound arguments **thing** and **location** to search for all items and their locations in the entire **House** data structure:

**Rule "go5"**

```
rule "go5"
  when
    String( this == "go5" )
```
isContainedIn(thing, location; )
then
    System.out.println(thing + " is in " + location);
end

The example inserts the "go5" string into the decision engine and activates the "go5" rule to return all items and their locations in the House data structure:

Insert string and fire rules

ksession.insert( "go5" );
ksession.fireAllRules();

Rule "go5" output in the IDE console

go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk
CHAPTER 88. PERFORMANCE TUNING CONSIDERATIONS WITH THE DECISION ENGINE

The following key concepts or suggested practices can help you optimize decision 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 Red Hat Decision Manager.

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

Sequential mode is an advanced rule base configuration in the decision engine that enables the decision engine to evaluate rules one time in the order that they are listed in the decision 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 Section 83.3, “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

```java
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 decision engine, see Chapter 86, Decision 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.
CHAPTER 89. ADDITIONAL RESOURCES

- Designing your decision management architecture for Red Hat Decision Manager
- Getting started with decision services
- Designing a decision service using DRL rules
- Packaging and deploying a Red Hat Decision Manager project
PART X. INTEGRATING MACHINE LEARNING WITH RED HAT DECISION MANAGER

As a business analyst or business rules developer, you can integrate machine learning with Red Hat Decision Manager by using PMML files with Decision Model and Notation (DMN) models.
When you think about artificial intelligence (AI), machine learning and big data might come to mind. But machine learning is only part of the picture. Artificial intelligence includes the following technologies:

- **Robotics:** The integration of technology, science, and engineering that produces machines that can perform physical tasks that are performed by humans.
- **Machine learning:** The ability of a collection of algorithms to learn or improve when exposed to data without being explicitly programmed to do so.
- **Natural language processing:** A subset of machine learning that processes human speech.
- **Mathematical optimization:** The use of conditions and constraints to find the optimal solution to a problem.
- **Digital decisioning:** The use of defined criteria, conditions, and a series of machine and human tasks to make decisions.

While science fiction is filled with what is referred to as artificial general intelligence (AGI), machines that perform better than people and cannot be distinguished from them and learn and evolve without human intervention or control, AGI is decades away. Meanwhile, we have pragmatic AI which is much less frightening and much more useful to us today. Pragmatic AI is a collection of AI technologies that, when combined, provide solutions to problems such as predicting customer behavior, providing automated customer service, and helping customers make purchasing decisions.

Leading industry analysts report that previously organizations have struggled with AI technologies because they invested in the potential of AI rather than the reality of what AI can deliver today. AI projects were not productive and as a result investment in AI slowed and budgets for AI projects were reduced. This disillusionment with AI is often referred to as an AI winter. AI has experienced several cycles of AI winters followed by AI springs and we are now decidedly in an AI spring. Organizations are seeing the practical reality of what AI can deliver. Being pragmatic means being practical and realistic. A pragmatic approach to AI considers AI technologies that are available today, combines them where useful, and adds human intervention when needed to create solutions to real-world problems.

**Pragmatic AI solution example**

One application of pragmatic AI is in customer support. A customer files a support ticket that reports a problem, for example, a login error. A machine learning algorithm is applied to the ticket to match the ticket content with existing solutions, based on keywords or natural language processing (NLP). The keywords might appear in many solutions, some relevant and some not as relevant. You can use digital decisioning to determine which solutions to present to the customer. However, sometimes none of the solutions proposed by the algorithm are appropriate to propose to the customer. This can be because all solutions have a low confidence score or multiple solutions have a high confidence score. In cases where
an appropriate solution cannot be found, the digital decisioning can involve the human support team. To find the best support person based on availability and expertise, mathematical optimization selects the best assignee for the support ticket by considering employee rostering constraints.

As this example shows, you can combine machine learning to extract information from data analysis and digital decisioning to model human knowledge and experience. You can then apply mathematical optimization to schedule human assistance. This is a pattern that you can apply to other situations, for example, a credit card dispute and credit card fraud detection.

These technologies use four industry standards:

- **Case Management Model and Notation (CMMN)**
  CMMN is used to model work methods that include various activities that might be performed in an unpredictable order depending on circumstances. CMMN models are event centered. CMMN overcomes limitations of what can be modeled with BPMN2 by supporting less structured work tasks and tasks driven by humans. By combining BPMN and CMMN you can create much more powerful models.

- **Business Process Model and Notation (BPMN2)**
  The BPMN2 specification is an Object Management Group (OMG) specification that defines standards for graphically representing a business process, defines execution semantics for the elements, and provides process definitions in XML format. BPMN2 can model computer and human tasks.

- **Decision Model and Notation (DMN)**
  Decision Model and Notation (DMN) is a standard established by the 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.

- **Predictive Model Markup Language (PMML)**
  PMML is the language used to represent predictive models, mathematical models that use statistical techniques to uncover, or learn, patterns in large volumes of data. Predictive models use the patterns that they learn to predict the existence of patterns in new data. With PMML, you can share predictive models between applications. This data is exported as a PMML file that can be consumed by a DMN model. As a machine learning framework continues to train the model, the updated data can be saved to the existing PMML file. This means that you can use predictive models created by any application that can save the model as a PMML file. Therefore, DMN and PMML integrate well.

**Putting it all together**
This illustration shows how predictive decision automation works.

1. Business data enters the system, for example, data from a loan application.

2. A decision model that is integrated with a predictive model decides whether or not to approve the loan or whether additional tasks are required.

3. A business action results, for example, a rejection letter or loan offer is sent to the customer.

The next section demonstrates how predictive decision automation works with Red Hat Decision Manager.
CHAPTER 91. CREDIT CARD FRAUD DISPUTE USE CASE

The financial industry uses pragmatic AI for decisioning in several areas. One area is credit card charge disputes. When a customer identifies an incorrect or unrecognized charge on a credit card bill, the customer can dispute the charge. Human intervention in credit card fraud detection is required in some cases but the majority of reported credit card fraud can be completely or partially resolved with pragmatic AI.

Machine learning models such as Tensorflow™ and R™ produce predictive models. You can save these predictive models in an open standard such as PMML so that you can use the model with Red Hat Decision Manager or other products that support the PMML standard.

91.1. USING A PMML MODEL WITH A DMN MODEL TO RESOLVE CREDIT CARD TRANSACTION DISPUTES

This example shows you how to use Red Hat Decision Manager to create a DMN model that uses a PMML model to resolve credit card transaction disputes. When a customer disputes a credit card transaction, the system decides whether or not to process the transaction automatically.

Prerequisites

- Red Hat Decision Manager is available and the following JAR file has been added to the ~/kie-server.war/WEB-INF/lib and ~/business-central.war/WEB-INF/lib directories in your Red Hat Decision Manager installation:
  - kie-dmn-jpmml-7.48.0.Final-redhat-00004.jar
    This file is available in the Red Hat Decision Manager 7.10 Maven Repository distribution available from the Software Downloads page in the Red Hat Customer Portal (login required). The group ID, artifact ID, and version (GAV) identifier of this file is org.kie:kie-dmn-jpmml:7.48.0.Final-redhat-0004. For more information, see the "Including PMML models within a DMN file in Business Central" section of Designing a decision service using DMN models.
  - JPMML Evaluator 1.5.1 JAR file
  - JPMML Evaluator Extensions 1.5.1 JAR file
    These files are required to enable JPMML evaluation in KIE Server and Business Central.

    IMPORTANT
    Red Hat supports integration with the Java Evaluator API for PMML (JPMML) for PMML execution in Red Hat Decision Manager. However, Red Hat does not support the JPMML libraries directly. If you include JPMML libraries in your Red Hat Decision Manager distribution, see the Openscoring.io licensing terms for JPMML.

Procedure

1. Create the dtree_risk_predictor.pmml file with the contents of the XML example in Section 91.2, “Credit card transaction dispute exercise PMML file”.

2. In Business Central, create the Credit Card Dispute project:
   a. Navigate to Menu → Design → Projects
b. Click Add Project.

c. In the Name box, enter Credit Card Dispute and click Add.

3. In the Assets window of the Credit Card Dispute project, import the dtree_risk_predictor.pmml file into the com package:

   a. Click Import Asset.

   b. In the Create new Import Asset dialog, enter dtree_risk_predictor in the Name box, select com from the Package menu, select the dtree_risk_predictor.pmml file, and click OK.

   The content of the dtree_risk_predictor.pmml file appears in the Overview window.

4. Create the Dispute Transaction Check DMN model in com package:

   a. To return to the project window, click Credit Card Dispute in the breadcrumb trail.

   b. Click Add Asset.

   c. Click DMN in the asset library.

   d. In the Create new DMN dialog, enter Dispute Transaction Check in the Name box, select com from the Package menu, and click OK.

   The DMN editor opens with the Dispute Transaction Check DMN model.

5. Create the tTransaction custom data type:
a. Click the **Data Types** tab.

b. Click **Add a custom Data Type**

c. In the **Name** box, enter **tTransaction**.

d. Select **Structure** from the **Type** menu.

e. To create the data type, click the check mark. The **tTransaction** custom data type appears with one variable row.

f. In the **Name** field of the variable row, enter **transaction_amount**, select **Number** from the **Type** menu, and then click the check mark.

g. To add a new variable row, click the plus symbol on the **transaction_amount** row. A new row appears.

h. In the **Name** field, enter **cardholder_identifier**, select **Number** from the **Type** menu, and then click the check mark.

6. Add the **Risk Predictor dtree_risk_predictor.pmml** model:

   a. In the **Included Models** window of the DMN editor, click **Include Model**.
b. In the Include Model dialog, select `dtree_risk_predictor.pmml` from the Models menu.

c. Enter **Risk Predictor** in the Provide a unique name box and click **OK**.

7. Create the **Risk Predictor** Business Knowledge Model (BKM) node with the **Risk Predictor** and **DecisionTreeClassifier** model:

   ![Risk Predictor Business Knowledge Model](image)

   a. In the **Model** window of the DMN editor, drag a BKM node to the DMN editor palette.

   ![Model Window](image)

   b. Rename the node **Risk Predictor**.

   c. Click the edit icon located below the trash can icon on the left side of the node.
d. Click F in the Risk Predictor box and select PMML from the Select Function Kind menu. The F changes to P.

e. Double-click the First select PMML document box and select Risk Predictor.

f. Double-click the Second select PMML model box and select DecisionTreeClassifier.

g. To return to the DMN editor palette, click Back to Dispute Transaction Check

8. Create the Transaction input data node with the data type tTransaction:

   a. In the Model window of the DMN editor, drag an input data node to the DMN editor palette.
b. Rename the node Transaction.

c. Select the node then click the properties pencil icon in the upper-right corner of the window.

d. In the Properties panel, select Information Item → Data type → tTransaction then close the panel.

9. Create the Transaction Dispute Risk decision node and add the Transaction node for data input and the Risk Predictor node for the function:

   a. In the Model window of the DMN editor, drag a decision data node to the DMN editor palette.
b. Rename the node Transaction Dispute Risk.

c. Select the Risk Predictor node and drag the arrow from the top right of the node to the Transaction Dispute Risk node.

d. Select the Transaction node and drag the arrow from the bottom right of the node to the Transaction Dispute Risk node.

10. In the Transaction Dispute Risk node, create the Risk predictor invocation function:

   a. Select the Transaction Dispute Risk node and click the edit icon on the left side of the node.

   b. Click Select expression and select Invocation from the menu.

   c. Enter Risk Predictor in the Enter function box.

   d. Click P1.

   e. In the Edit Parameter dialog, enter amount in the Name box, select number from the Data Type menu, and press the Enter key.
f. Click Select expression and select Literal expression from the menu.

g. Enter Transaction.transaction_amount in the box next to amount.

h. Right-click on 1 and select Insert below. The Edit Parameter dialog opens.

i. Enter holder_index in the Name box, select number from the Data Type menu, and press the Enter key.

j. Click Select expression on row 2 and select Literal expression from the menu.

k. Enter Transaction.cardholder_identifier in the box next to amount.

11. Create the Risk Threshold input data node with the data type number:

a. In the Model window of the DMN editor, drag an input data node to the DMN editor palette.

b. Rename the node Risk Threshold.

c. Select the node then click the properties pencil icon in the upper-right corner of the window.

d. In the Properties panel, select Information Item → Data type → number then close the panel.

12. Create the Can be automatically processed? decision node that takes as inputs the Transaction Dispute Risk and the Risk threshold nodes.
a. Drag a decision node to the DMN editor palette and rename it Can be automatically processed?.

b. Select the node, then click the edit icon on the upper-left side of the node.

c. Click Select expression and then select Literal expression from the menu.

d. Enter Transaction Dispute Risk.predicted_dispute_risk < Risk Threshold in the box.

e. Select the Transaction Dispute Risk node and drag the arrow in the top left of the node to the Can be automatically processed? node.

f. Select the Risk Threshold node and drag the arrow from the bottom left of the node to the Can be automatically processed? node.

13. Save the model and build the project:

a. In the DMN editor, click Save.

b. If necessary, correct any errors that appear.

c. To return to the project window, click Credit Card Dispute in the breadcrumb trail.

d. Click Build. The project should successfully build.
14. Add and run a test scenario:

   a. Click **Add Asset**

   b. Select **Test Scenario**.

   c. In the **Create new Test Scenario** dialog, enter the name **Test Dispute Transaction Check**, select **com** from the **Package** menu, and select **DMN**.

   d. Select **Dispute Transaction Check.dmn** from the **Choose a DMN asset** menu and click **OK**. The test template builds.

   e. Enter the following values and click **Save**:

   ![Table of test scenario parameters]

   **NOTE**

   Do not add a value to the **Transaction Dispute Risk** column. This value is determined by the test scenario.

**Table 91.1. Test scenario parameters**

<table>
<thead>
<tr>
<th>Description</th>
<th>Risk Threshold</th>
<th>cardholder_identifier</th>
<th>transaction_amount</th>
<th>Can be automatically processed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk threshold 5, automatically processed</td>
<td>5</td>
<td>1234</td>
<td>1000</td>
<td>true</td>
</tr>
<tr>
<td>Risk threshold 4, amount = 1000, not processed</td>
<td>4</td>
<td>1234</td>
<td>1000</td>
<td>false</td>
</tr>
<tr>
<td>Risk threshold 4, amount = 180, automatically processed</td>
<td>4</td>
<td>1234</td>
<td>180</td>
<td>true</td>
</tr>
</tbody>
</table>
Risk threshold 1, amount = 1, not processed

<table>
<thead>
<tr>
<th>Description</th>
<th>Risk Threshold</th>
<th>cardholder_identifier</th>
<th>transaction_amount</th>
<th>Can be automatically processed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk threshold 1, amount = 1, not processed</td>
<td>1</td>
<td>1234</td>
<td>1</td>
<td>false</td>
</tr>
</tbody>
</table>

f. To run the test, click the Play button, to the right of Validate. The results appear in the Test Report panel on the right of the screen.

91.2. CREDIT CARD TRANSACTION DISPUTE EXERCISE PMML FILE

Use the following XML content to create the dtree_risk_predictor.pmml file in the Section 91.1, “Using a PMML model with a DMN model to resolve credit card transaction disputes” exercise.

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <Header copyright="Copyright (c) 2018 Software AG" description="Default Description">
    <Application name="Nyoka" version="4.3.0"/>
    <Timestamp>2020-10-09 14:27:26.622723</Timestamp>
  </Header>
  <DataDictionary numberOfFields="3">
    <DataField name="amount" optype="continuous" dataType="double"/>
    <DataField name="holder_index" optype="continuous" dataType="double"/>
    <DataField name="dispute_risk" optype="categorical" dataType="integer">
      <Value value="1"/>
      <Value value="2"/>
      <Value value="3"/>
      <Value value="4"/>
      <Value value="5"/>
    </DataField>
  </DataDictionary>
  <TreeModel modelName="DecisionTreeClassifier" functionName="classification" missingValuePenalty="1.0">
    <MiningSchema>
      <MiningField name="amount" usageType="active" optype="continuous"/>
      <MiningField name="holder_index" usageType="active" optype="continuous"/>
      <MiningField name="dispute_risk" usageType="target" optype="categorical"/>
    </MiningSchema>
    <Output>
      <OutputField name="probability_1" optype="continuous" dataType="double" feature="probability" value="1"/>
      <OutputField name="probability_2" optype="continuous" dataType="double" feature="probability" value="2"/>
      <OutputField name="probability_3" optype="continuous" dataType="double" feature="probability" value="3"/>
      <OutputField name="probability_4" optype="continuous" dataType="double" feature="probability" value="4"/>
      <OutputField name="probability_5" optype="continuous" dataType="double" feature="probability" value="5"/>
    </Output>
  </TreeModel>
</PMML>
```
<OutputField name="predicted_dispute_risk" optype="categorical" dataType="integer" feature="predictedValue"/>
</Output>
<Node id="0" recordCount="600.0">
  <True />
  <Node id="1" recordCount="200.0">
    <SimplePredicate field="amount" operator="lessOrEqual" value="99.94000244140625" />
    <Node id="2" score="2" recordCount="55.0">
      <SimplePredicate field="holder_index" operator="lessOrEqual" value="0.5" />
      <Node id="3" score="1" recordCount="145.0">
        <SimplePredicate field="holder_index" operator="greaterThan" value="0.5" />
        <ScoreDistribution value="1" recordCount="0.0" confidence="0.0" />
        <ScoreDistribution value="2" recordCount="55.0" confidence="1.0" />
        <ScoreDistribution value="3" recordCount="0.0" confidence="0.0" />
        <ScoreDistribution value="4" recordCount="0.0" confidence="0.0" />
        <ScoreDistribution value="5" recordCount="0.0" confidence="0.0" />
      </Node>
    </Node>
  </Node>
</Node>
<Node id="3" score="1" recordCount="400.0">
  <SimplePredicate field="amount" operator="greaterThan" value="99.94000244140625" />
  <Node id="5" recordCount="105.0">
    <SimplePredicate field="holder_index" operator="lessOrEqual" value="0.5" />
    <Node id="6" score="3" recordCount="54.0">
      <SimplePredicate field="amount" operator="lessOrEqual" value="150.4550018310547" />
      <ScoreDistribution value="1" recordCount="0.0" confidence="0.0" />
      <ScoreDistribution value="2" recordCount="0.0" confidence="0.0" />
      <ScoreDistribution value="3" recordCount="54.0" confidence="1.0" />
      <ScoreDistribution value="4" recordCount="0.0" confidence="0.0" />
      <ScoreDistribution value="5" recordCount="0.0" confidence="0.0" />
    </Node>
  </Node>
</Node>
<Node id="7" recordCount="51.0">
  <SimplePredicate field="amount" operator="greaterThan" value="150.4550018310547" />
  <Node id="8" recordCount="40.0">
    <SimplePredicate field="amount" operator="lessOrEqual" value="200.00499725341797" />
    <Node id="9" recordCount="36.0">
      <SimplePredicate field="amount" operator="lessOrEqual" value="195.4949951171875" />
      <Node id="10" recordCount="2.0">
        <SimplePredicate field="amount" operator="lessOrEqual" value="152.2050018310547" />
        <Node id="11" score="4" recordCount="1.0">
          <SimplePredicate field="amount" operator="lessOrEqual" value="151.31500244140625" />
          <ScoreDistribution value="1" recordCount="0.0" confidence="0.0" />
          <ScoreDistribution value="2" recordCount="0.0" confidence="0.0" />
          <ScoreDistribution value="3" recordCount="0.0" confidence="0.0" />
          <ScoreDistribution value="4" recordCount="1.0" confidence="1.0" />
          <ScoreDistribution value="5" recordCount="0.0" confidence="0.0" />
        </Node>
      </Node>
    </Node>
  </Node>
</Node>
<Node id="12" score="3" recordCount="1.0">
  <SimplePredicate field="amount" operator="greaterThan" value="151.31500244140625" />
</Node>
<ScoreDistribution value="1" recordCount="0.0" confidence="0.0" />
<ScoreDistribution value="2" recordCount="0.0" confidence="0.0" />
<ScoreDistribution value="3" recordCount="1.0" confidence="1.0" />
<ScoreDistribution value="4" recordCount="0.0" confidence="0.0" />
<ScoreDistribution value="5" recordCount="0.0" confidence="0.0" />
</Node>
</Node>
<Node id="13" recordCount="34.0">
<SimplePredicate field="amount" operator="greaterThan" value="152.2050018310547" />
</Node>
</Node>
<Node id="14" recordCount="20.0">
<SimplePredicate field="amount" operator="lessOrEqual" value="176.5050048828125" />
</Node>
<Node id="15" recordCount="19.0">
<SimplePredicate field="amount" operator="lessOrEqual" value="176.06500244140625" />
</Node>
<Node id="16" score="4" recordCount="9.0">
<SimplePredicate field="amount" operator="lessOrEqual" value="166.6449966430664" />
</Node>
</Node>
<Node id="17" recordCount="10.0">
<SimplePredicate field="amount" operator="greaterThan" value="166.6449966430664" />
</Node>
<Node id="18" score="3" recordCount="1.0">
<SimplePredicate field="amount" operator="lessOrEqual" value="167.97999572753906" />
</Node>
<Node id="19" score="4" recordCount="9.0">
<SimplePredicate field="amount" operator="greaterThan" value="167.97999572753906" />
</Node>
</Node>
<Node id="20" score="3" recordCount="1.0">
<SimplePredicate field="amount" operator="greaterThan" value="176.06500244140625" />
</Node>
</Node>
CHAPTER 91. CREDIT CARD FRAUD DISPUTE USE CASE

</Node>
<Node id="21" score="4" recordCount="14.0">
  <SimplePredicate field="amount" operator="greaterThan" value="176.5050048828125">
    <ScoreDistribution value="1" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="2" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="3" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="4" recordCount="14.0" confidence="1.0"/>
    <ScoreDistribution value="5" recordCount="0.0" confidence="0.0"/>
  </SimplePredicate>
</Node>

<Node id="22" recordCount="4.0">
  <SimplePredicate field="amount" operator="greaterThan" value="195.4949951171875">
    <ScoreDistribution value="1" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="2" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="3" recordCount="1.0" confidence="1.0"/>
    <ScoreDistribution value="4" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="5" recordCount="0.0" confidence="0.0"/>
  </SimplePredicate>
</Node>

<Node id="23" score="3" recordCount="1.0">
  <SimplePredicate field="amount" operator="lessOrEqual" value="195.7649938964844">
    <ScoreDistribution value="1" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="2" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="3" recordCount="1.0" confidence="1.0"/>
    <ScoreDistribution value="4" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="5" recordCount="0.0" confidence="0.0"/>
  </SimplePredicate>
</Node>

<Node id="24" recordCount="3.0">
  <SimplePredicate field="amount" operator="greaterThan" value="195.7649938964844">
    <ScoreDistribution value="1" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="2" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="3" recordCount="1.0" confidence="1.0"/>
    <ScoreDistribution value="4" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="5" recordCount="0.0" confidence="0.0"/>
  </SimplePredicate>
</Node>

<Node id="25" score="4" recordCount="1.0">
  <SimplePredicate field="amount" operator="lessOrEqual" value="196.74500274658203">
    <ScoreDistribution value="1" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="2" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="3" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="4" recordCount="1.0" confidence="1.0"/>
    <ScoreDistribution value="5" recordCount="0.0" confidence="0.0"/>
  </SimplePredicate>
</Node>

<Node id="26" recordCount="2.0">
  <SimplePredicate field="amount" operator="greaterThan" value="196.74500274658203">
    <ScoreDistribution value="1" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="2" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="3" recordCount="1.0" confidence="1.0"/>
    <ScoreDistribution value="4" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="5" recordCount="0.0" confidence="0.0"/>
  </SimplePredicate>
</Node>

<Node id="27" score="3" recordCount="1.0">
  <SimplePredicate field="amount" operator="lessOrEqual" value="197.5800018310547">
    <ScoreDistribution value="1" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="2" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="3" recordCount="1.0" confidence="1.0"/>
    <ScoreDistribution value="4" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="5" recordCount="0.0" confidence="0.0"/>
  </SimplePredicate>
</Node>

<Node id="28" score="4" recordCount="1.0">
  <SimplePredicate field="amount" operator="greaterThan" value="197.5800018310547">
    <ScoreDistribution value="1" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="2" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="3" recordCount="0.0" confidence="0.0"/>
    <ScoreDistribution value="4" recordCount="1.0" confidence="1.0"/>
    <ScoreDistribution value="5" recordCount="0.0" confidence="0.0"/>
  </SimplePredicate>
</Node>
<Node id="40" recordCount="30.0">
  <SimplePredicate field="amount" operator="lessOrEqual" value="175.72000122070312" />
</Node>

<Node id="41" recordCount="19.0">
  <SimplePredicate field="amount" operator="lessOrEqual" value="168.69499969482422" />
</Node>

<Node id="42" recordCount="6.0">
  <SimplePredicate field="amount" operator="lessOrEqual" value="158.125" />
</Node>

<Node id="43" score="3" recordCount="5.0">
  <SimplePredicate field="amount" operator="lessOrEqual" value="157.85499572753906" />
</Node>

<Node id="44" score="2" recordCount="1.0">
  <SimplePredicate field="amount" operator="greaterThan" value="157.85499572753906" />
</Node>

<Node id="45" score="3" recordCount="13.0">
  <SimplePredicate field="amount" operator="greaterThan" value="158.125" />
</Node>

<Node id="46" recordCount="11.0">
  <SimplePredicate field="amount" operator="greaterThan" value="168.6999969482422" />
</Node>

<Node id="47" score="2" recordCount="1.0">
  <SimplePredicate field="amount" operator="lessOrEqual" value="168.69499969482422" />
</Node>

<Node id="48" recordCount="10.0">
  <SimplePredicate field="amount" operator="greaterThan" value="168.69499969482422" />
</Node>

<Node id="49" recordCount="4.0">
  <SimplePredicate field="holder_index" operator="lessOrEqual" value="1.5" />
</Node>

<Node id="50" score="2" recordCount="1.0">
  <SimplePredicate field="amount" operator="lessOrEqual" value="175.72000122070312" />
</Node>
<ScoreDistribution value="1" recordCount="0.0" confidence="0.0" />
<ScoreDistribution value="2" recordCount="1.0" confidence="1.0" />
<ScoreDistribution value="3" recordCount="0.0" confidence="0.0" />
<ScoreDistribution value="4" recordCount="0.0" confidence="0.0" />
<ScoreDistribution value="5" recordCount="0.0" confidence="0.0" />
</Node>
</Node>
</Node>
</Node>
</Node>
</Node>
</Node>
</SimplePredicate>
</Node>
</Node>
</SimplePredicate>
</Node>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
</SimplePredicate>
</Node>
<ScoreDistribution value="5" recordCount="0.0" confidence="0.0"/>
</Node>
</Node>
</Node>
</Node>
</TreeModel>
</PMML>
CHAPTER 92. ADDITIONAL RESOURCES

- Getting started with case management
- Getting started with decision services
- Designing a decision service using DMN models
- Getting started with Red Hat Business Optimizer
- Predictions 2019: Expect A Pragmatic Vision Of AI
APPENDIX A. VERSIONING INFORMATION

Documentation last updated on Monday, March 01, 2021.
APPENDIX B. CONTACT INFORMATION

Red Hat Decision Manager documentation team: brms-docs@redhat.com