Red Hat Process Automation Manager 7.1

Designing a decision service using DMN models

Last Updated: 2018-10-10

Red Hat Process Automation Manager 7.1 Designing a decision service using DMN models

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

Legal Notice

Abstract

This document describes how to implement Decision Model and Notation (DMN) models in your decision services in Red Hat Process Automation Manager 7.1.

Table of Contents

PREFACE

As a business analyst or business rules developer, you can use Decision Model and Notation (DMN) to model a decision service graphically in a decision requirements diagram (DRD). This diagram traces business decisions from start to finish, with each decision node drawing logic from DMN model decision elements such as decision tables. Red Hat Process Automation Manager includes full runtime support for DMN 1.2 models at conformance level 3, but currently does not include a built-in DMN model editor. You can design your DMN models using a third-party DMN authoring tool and include them in your Red Hat Process Automation Manager projects for deployment and execution.

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

CHAPTER 1. 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 decision models can be shared between DMN-compliant platforms and across organizations so that business analysts and business rules developers are unified 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.

1.1. 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 are summaries of 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 Process Automation Manager includes full runtime support for DMN 1.2 models at conformance level 3, but currently does not include a built-in DMN model editor. Editors for DMN models will be added to the platform in the near future, but meanwhile you can use third-party DMN authoring platforms and implement DMN models in your decision services in Red Hat Process Automation Manager.

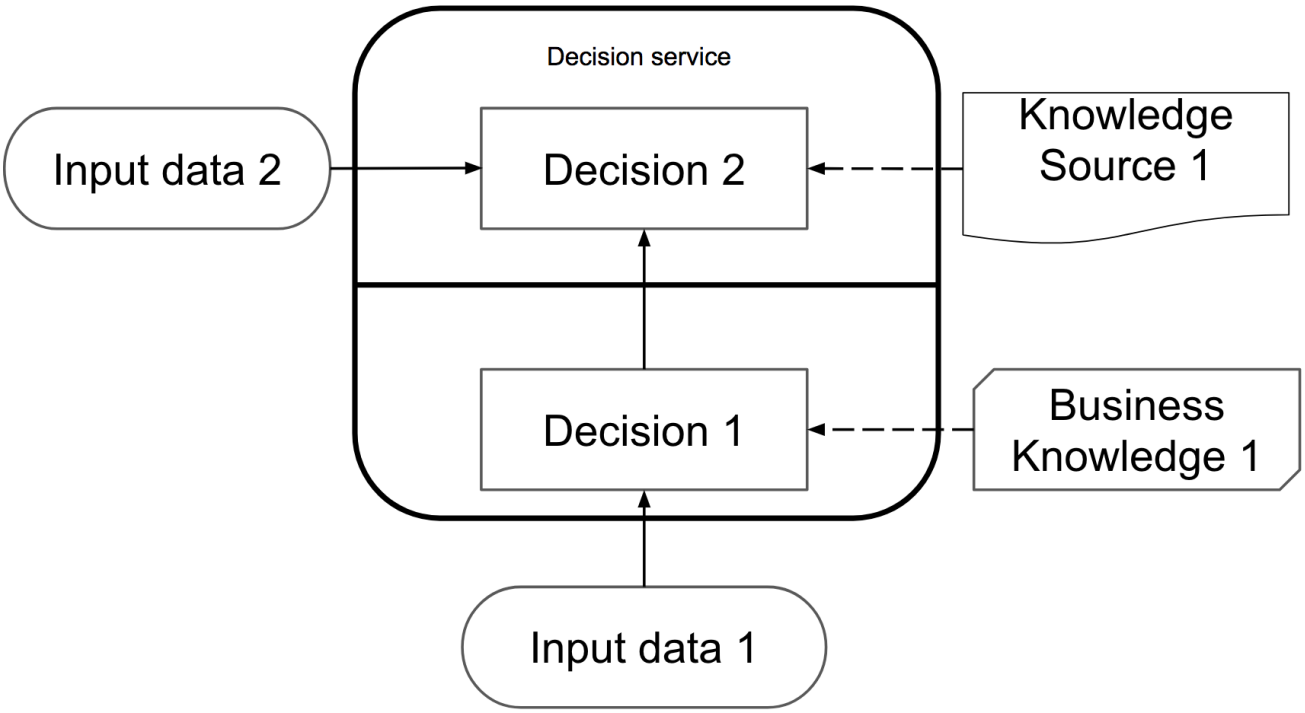
1.2. DMN ELEMENTS

DMN models consist of the following five elements:

- **Decisions:** Nodes in the model where one or several inputs determine an output based on decision logic.
- **Input data:** Information necessary to determine a decision. This information usually includes business-level concepts or objects relevant to the business, such as a restaurant's peak business hours and staff availability.

- **Business knowledge models:** Reusable pieces of decision logic. Decisions that have the same logic but depend on different sub-inputs or sub-decisions use business knowledge models to determine which procedure to follow.
- **Knowledge sources:** External regulations, documents, committees, policies, and so on that shape decision logic. Knowledge sources are references to real-world factors rather than executable business rules.
- **Decision service:** A decision service is a top-level decision, with well-defined inputs, that is published as a service for invocation. In the diagram it is represented by an overlay rectangle with round corners. The decision service can be invoked from an external application or business process (BPMN). For more information, see page 36 of the DMN specification document.

Figure 1.1. Basic decision requirements diagram



1.3. RULE EXPRESSIONS IN FEEL

Friendly Enough Expression Language (FEEL) is an expression language defined by the 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 either table cells in decision tables or decision nodes.

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

1.3.1. 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 Process Automation Manager removes the limitation on the use of consecutive spaces. Red Hat Process Automation Manager supports variable names with any number of consecutive spaces, but normalizes them into a single space. For example, the two variable references **`First Name`** and **`First Name`** are both acceptable in Red Hat Process Automation Manager.

Red Hat Process Automation 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 Process Automation Manager FEEL engine perspective, all of these characters are normalized into a single white space before processing.

The keyword `in`

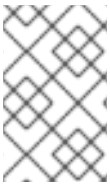
The keyword **`in`** is the only keyword in the language that cannot be used as part of a variable name. Although the specifications allow the use of keywords in the middle of variable names, the use of **`in`** in variable names conflicts with the grammar definition of **`for`**, **`every`** and **`some`** expression constructs.

1.3.2. 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

Functions, contexts, ranges, and lists are not explicitly supported in the DMN specification as data types, but they are supported by extension in Red Hat Process Automation Manager.

The following are descriptions of 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 Process Automation 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

FEEL does not have date literals, but you can use the built-in `date()` function to construct date values. Date strings in FEEL follow the format defined in the XML Schema Part 2: Datatypes document. The format is `"YYYY-MM-DD"` where `YYYY` is the year with four digits, `MM` is the number of the month with two digits, and `DD` is the number of the day.

Example:

```
date( "2017-06-23" )
```

Date objects have time equal to **"00:00:00"**, which is midnight. The dates are considered to be local, without a timezone.

Time

FEEL does not have time literals, but you can use the built-in **time()** function to construct time values. Time strings in FEEL follow the format defined in the XML Schema Part 2: Datatypes document. The format is **"hh:mm:ss[.uuu][(+-)hh:mm]"** where **hh** is the hour of the day (from **00** to **23**), **mm** is the minutes in the hour, and **ss** is the number of seconds in the minute. Optionally, the string may define the number of milliseconds (**uuu**) within the second and contain a positive (**+**) or negative (**-**) offset from UTC time to define its timezone. Instead of using an offset, you can use the letter **z** to represent the UTC time, which is the same as an offset of **-00:00**. If no offset is defined, the time is considered to be local.
Examples:

```
time( "04:25:12" )
time( "14:10:00+02:00" )
time( "22:35:40.345-05:00" )
time( "15:00:30z" )
```

Time values that define an offset or a timezone cannot be compared to local times that do not define an offset or a timezone.

Date and time

FEEL does not have date and time literals, but you can use the built-in **date and time()** function to construct date and time values. Date and time strings in FEEL follow the format defined in the XML Schema Part 2: Datatypes document. The format is **"<date>T<time>"**, where **<date>** and **<time>** follow the prescribed XML schema formatting, conjoined by **T**.
Examples:

```
date and time( "2017-10-22T23:59:00" )
date and time( "2017-06-13T14:10:00+02:00" )
date and time( "2017-02-05T22:35:40.345-05:00" )
date and time( "2017-06-13T15:00:30z" )
```

Date and time values that define an offset or a timezone cannot be compared to local date and time values that do not define an offset or a timezone.

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

FEEL does not have days and time duration literals, 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

FEEL does not have years and months duration literals, 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 supports **`function`** literals (or anonymous functions) that you can use to create functions. FEEL does not provide an explicit way of declaring a variable as a **`function`** in the DMN specification, but Red Hat Process Automation Manager extends the DMN built-in types to support 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.

IMPORTANT

A **`function`** datatype is an extension of the DMN specification and is subject to change if the DMN specification provides a standard way to declare functions in the future.

Contexts

FEEL supports **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. FEEL does not provide an explicit way of declaring a variable as a **context** in the DMN specification, but Red Hat Process Automation Manager extends the DMN built-in types to support 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 Process Automation 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**.

IMPORTANT

A **context** data type is an extension of the DMN specification and is subject to change if the DMN specification provides a standard way to declare contexts in the future.

Ranges (or intervals)

FEEL supports **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. FEEL does not provide an explicit way of declaring a variable as a **range** in the DMN specification (unless it is within another expression), but Red Hat Process Automation Manager extends the DMN built-in types to support ranges.
The syntax of a range is defined in the following formats:

```
range          := interval_start endpoint '..' endpoint interval_end
interval_start := open_start | closed_start
open_start     := '(' | ']'
closed_start   := '['
interval_end   := open_end | closed_end
open_end       := ')' | '['
closed_end     := ']'
endpoint       := expression
```

The expression for the endpoint must return a comparable value, and the lower bound endpoint must be lower than the upper bound endpoint.

For example, the following literal expression defines an interval between **1** and **10**, including the boundaries (a closed interval on both endpoints):

```
[ 1 .. 10 ]
```

The following literal expression defines an interval between 1 hour and 12 hours, including the lower boundary (a closed interval), but excluding the upper boundary (an open interval):

```
[ duration("PT1H") .. duration("PT12H") ]
```

You can use ranges in decision tables to test for ranges of values, or use ranges in simple literal expressions. For example, the following literal expression returns **`true`** if the value of a variable **`x`** is between **`0`** and **`100`**:

```
x in [ 1 .. 100 ]
```

IMPORTANT

A **`range`** data type is an extension of the DMN specification and is subject to change if the DMN specification provides a standard way to declare contexts in the future.

Lists

Lists in FEEL are represented by a comma-separated list of values enclosed in square brackets. FEEL does not provide an explicit way of declaring a variable as a **`list`** in the DMN specification, but Red Hat Process Automation Manager extends the DMN built-in types to support contexts. 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]
```

IMPORTANT

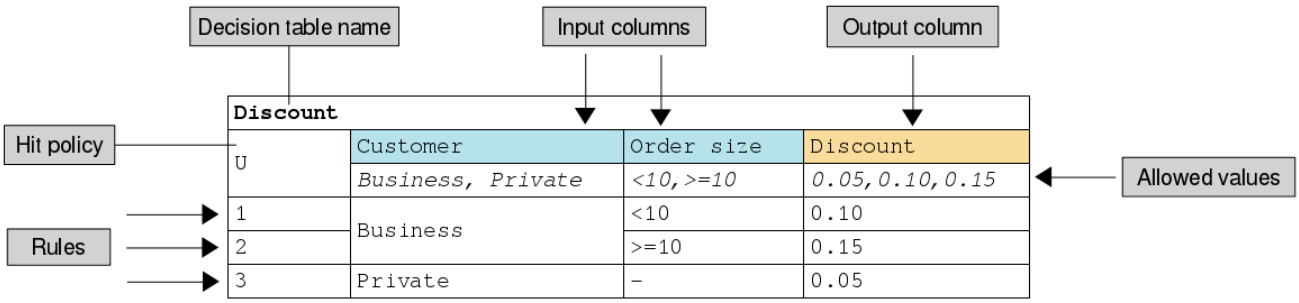
A **`list`** data type is an extension of the DMN specification and is subject to change if the DMN specification provides a standard way to declare contexts in the future.

1.4. DMN DECISION TABLES

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 and outcome for that particular row. The definition of each row is precise enough to derive the outcome using the values of the

conditions. For readability purposes, there is often a means to hide some of the more technical details when viewing the table.

Figure 1.2. Decision table example



Decision tables are a popular way for modeling rules and decisions, and are used in many methodologies (such as DMN) and implementation frameworks (such as Drools used in Red Hat Process Automation Manager).

IMPORTANT

Although the concept of decision tables is similar in DMN and Drools, DMN decision tables syntax and layout are defined by the DMN standard while Drools decision tables are defined by the Drools project. Red Hat Process Automation Manager supports both formats of decision tables, but they are not interchangeable. For more information about Drools decision tables, see Designing a decision service using uploaded decision tables.

1.4.1. Hit policies

Hit policies define how to reach an outcome when multiple rules match on a single decision table. Decision modelers select one of the following five policies for reaching an outcome and then specify that policy by placing an indicator in the table's upper-left corner. In the following list, the indicators are listed after the indicator type, in parentheses ().

- **Unique (U):** Permits only one rule to match. Any overlap raises an error.
- **Any (A):** Permits multiple rules to match, but they must all have the same output. If multiple matching rules do not have the same output, an error is raised.
- **Priority (P):** Permits multiple rules to match, with different outputs. The output that comes first in the *output values* list is selected.
- **First (F):** Uses the first match in rule order.
- **Collect (C+, C>, C<, C#):** Aggregates output from multiple rules based on an aggregation function.
  - **Collect (C):** Aggregates values in an arbitrary list.
  - **Collect Sum (C+):** Outputs the sum of all collected values. Values must be numeric.
  - **Collect Min (C<):** Outputs the minimum value among the matches. The resulting values must be comparable, such as numbers, dates, or text (lexicographic order).
  - **Collect Max (C>):** Outputs the maximum value among the matches. The resulting values must be comparable, such as numbers, dates or text (lexicographic order).

- **Collect Count (C#):** Outputs the number of matching rules.

1.5. BOXED EXPRESSIONS

Boxed expressions are tabular representations of contexts, function definitions, function invocations, and other expressions in a DMN model. For example, the following boxed expression defines the function **`Installment calculation`** that uses four parameters (**`Product`**, **`Rate`**, **`Term`**, and **`Amount`**) and calculates the monthly installment amount.

Figure 1.3. Boxed expression example

`Installment calculation`	
`(Product, Rate, Term, Amount)`	

`Monthly Fee`	`if Product Type = "standard loan"`
	`then 30.00`
	`else if Product Type = "special loan"`
	`then 20.00`
	`else null`

`Monthly Repayment`	`PMT(Rate, Term, Amount)`
`Monthly Repayment + Monthly Fee`	

CHAPTER 2. DMN USE CASE

This real-world DMN example demonstrates how you can use decision modeling to reach a decision based on inputs, circumstances, and company guidelines. The process in this section demonstrate how some of these components work together. 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:

Inputs

- A list of flights
- A list of passengers

Decisions

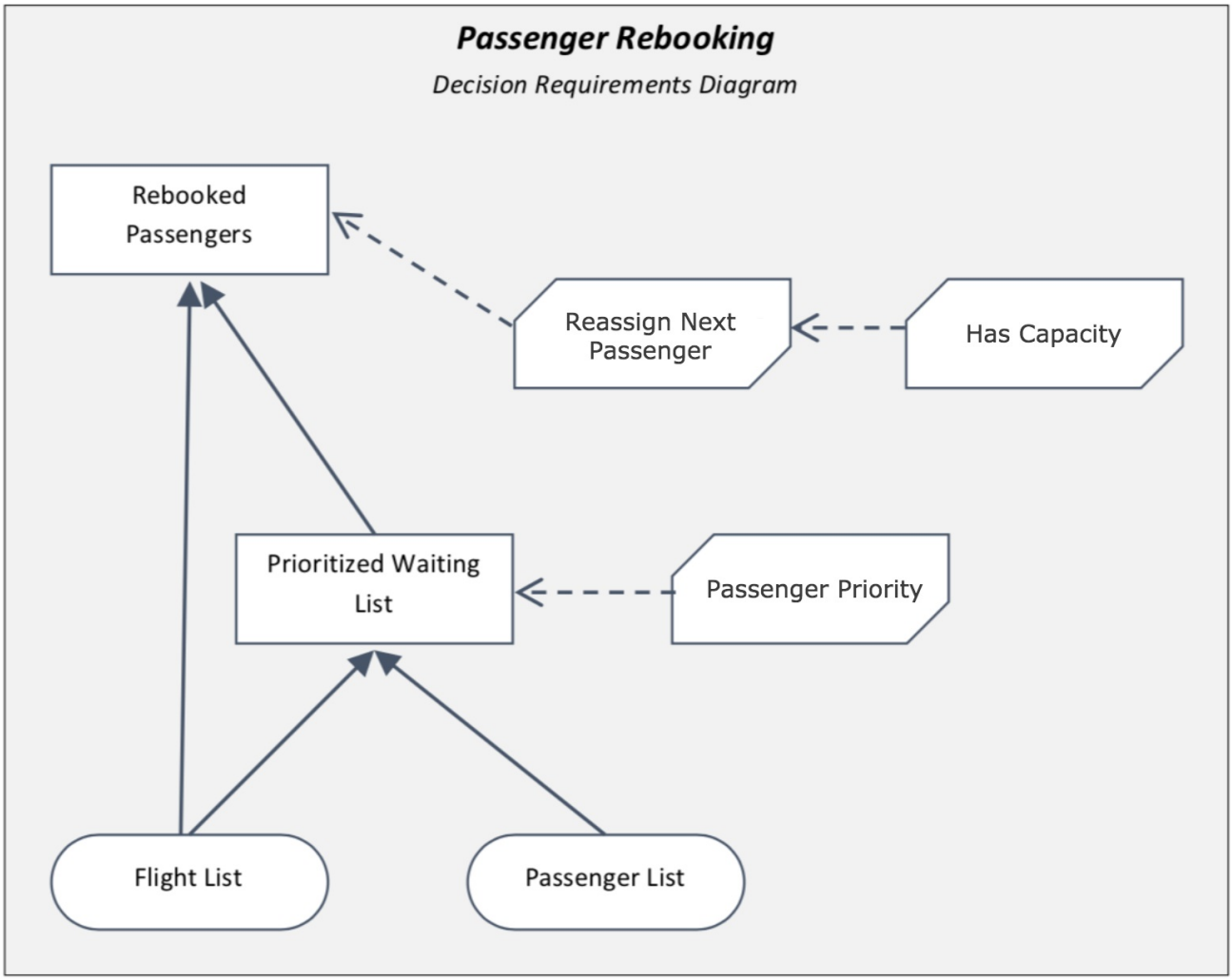
- Prioritizing the passengers who will get seats on a new flight
- Determining which flights those passengers will be offered

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 customers

Then, the airline uses the DMN standard to model its decision process in a decision requirements diagram (DRD), and creates the following diagram for determining the best rebooking solution:

Figure 2.1. Decision requirements diagram for passenger rebooking example



Similar to flowcharts, DRDs use shapes to represent the different elements in a process. Ovals contain the two necessary inputs, rectangles contain the decision points in the model, and rectangles with clipped corners contain reusable logic that can be repeatedly invoked.

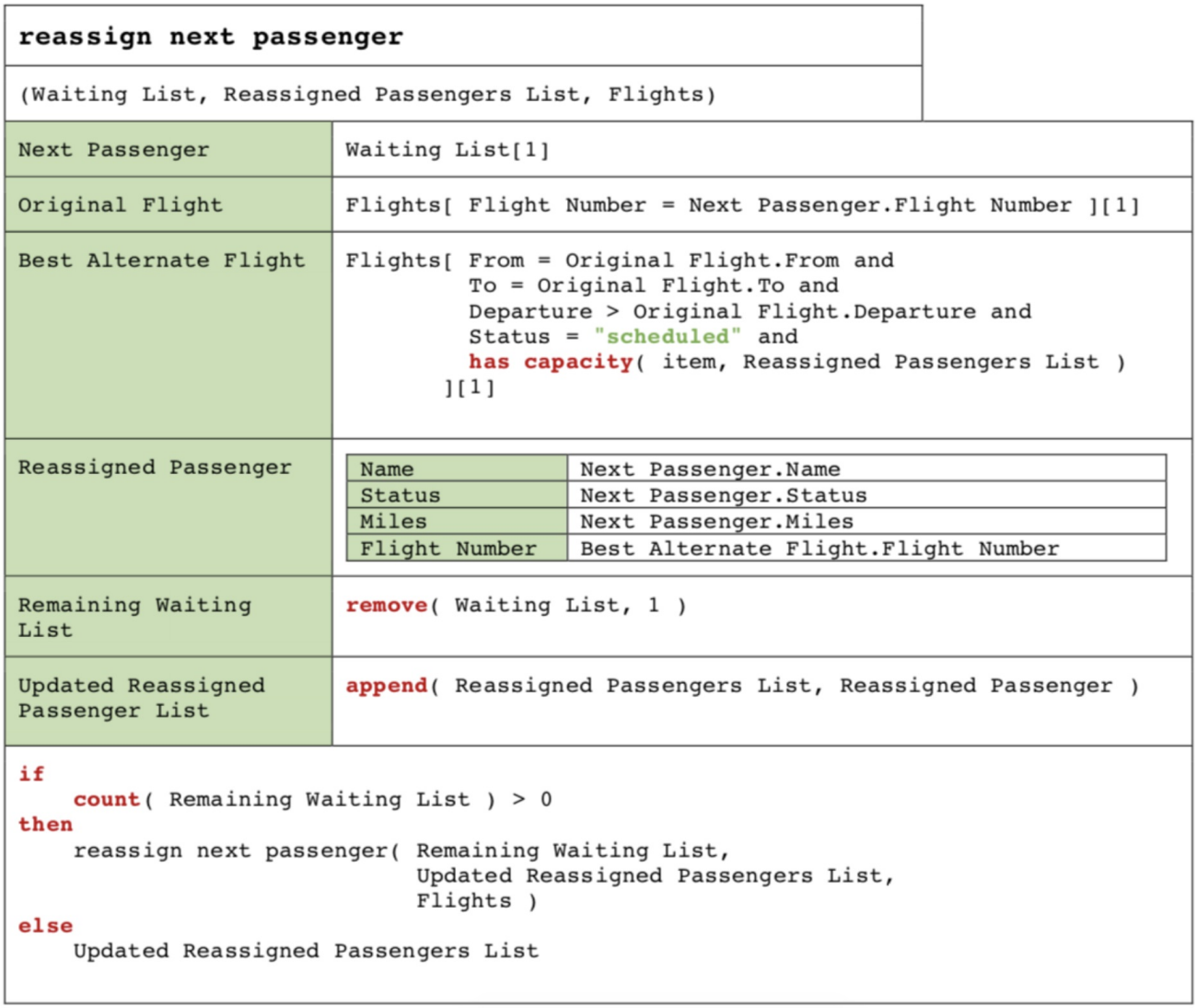
The DRD places details for each element into boxed content that provide variable definitions, again using FEEL expressions. Some content can be simple, such as the airline's decision process for establishing a prioritized waiting list.

Figure 2.2. Boxed expression example for prioritized wait list

Prioritized Waiting List	
Cancelled Flights	`Flight List[ Status = "cancelled" ].Flight Number`
Waiting List	`Passenger List[ list contains( Cancelled Flights, Flight Number ) ]`
`sort( Waiting List, passenger priority )`	

Other elements can involve significantly greater detail and calculation. Consider the following business knowledge model for reassigning the next passenger:

Figure 2.3. Decision example for reassigning next passenger



reassign next passenger

(Waiting List, Reassigned Passengers List, Flights)

Next Passenger	Waiting List[1]
Original Flight	Flights[Flight Number = Next Passenger.Flight Number][1]
Best Alternate Flight	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]
Reassigned Passenger	Name: Next Passenger.Name Status: Next Passenger.Status Miles: Next Passenger.Miles Flight Number: Best Alternate Flight.Flight Number
Remaining Waiting List	remove(Waiting List, 1)
Updated Reassigned Passenger List	append(Reassigned Passengers List, Reassigned Passenger)

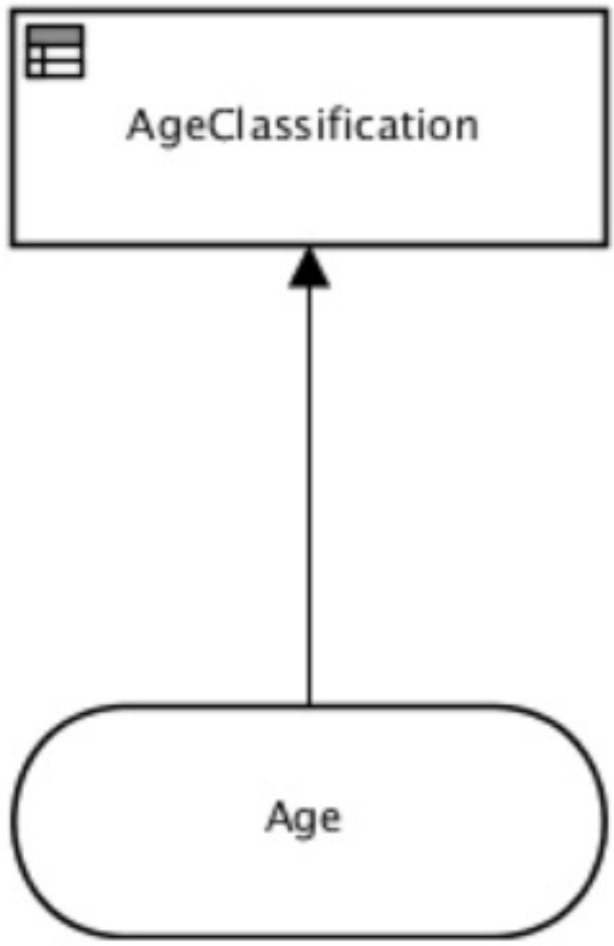
```
if
    count( Remaining Waiting List ) > 0
then
    reassign next passenger( Remaining Waiting List,
                             Updated Reassigned Passengers List,
                             Flights )
else
    Updated Reassigned Passengers List
```

CHAPTER 3. DMN MODEL EXAMPLE

DMN defines an XML schema that enables DMN models to be used between different DMN authoring platforms. The DMN specification enables multiple software platforms to work with the same file for authoring, testing, and production execution. You must use a third-party authoring platform such as Trisotech or Signavio if you require visual authoring capabilities.

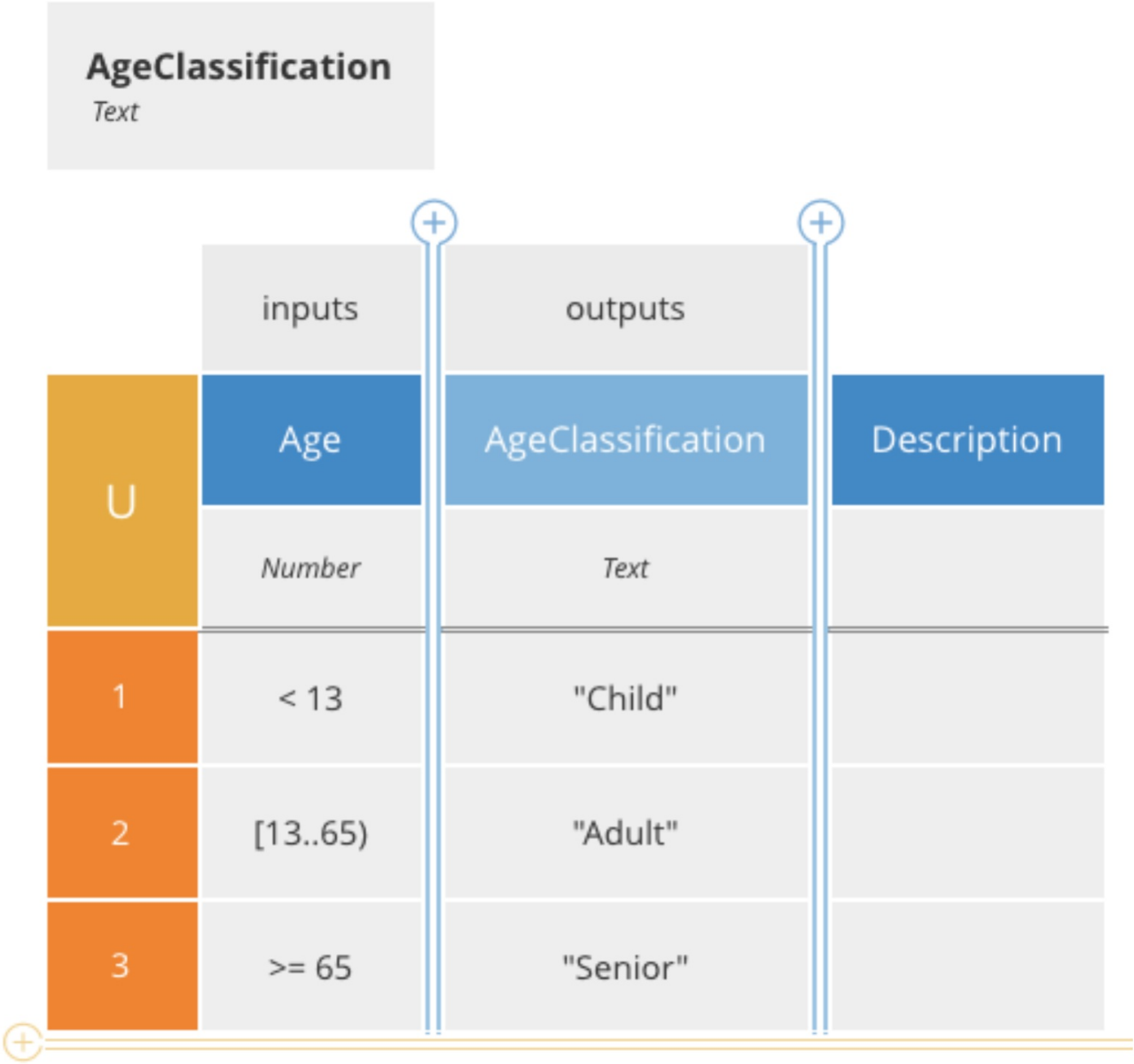
The following decision requirements diagram (DRD) example demonstrates a classification-type decision for the age categories of movie ticket purchases. This basic example demonstrates good form by creating classifications to avoid repeated calculations so that this mini-decision can be an input for other decisions.

Figure 3.1. Decision requirements diagram for the age classification decision



This example consists of a single numeric input value (**Age**), and produces a string output (**`AgeClassification`**). The inner workings of the **`AgeClassification`** decision is a basic table:

Figure 3.2. Decision table for the age classification decision



This table assigns a value to the **`AgeClassification`** output value using simple FEEL expressions to determine ranges on the age value. This decision model was created in the Trisotech DMN Authoring environment.

The following output is the XML source of this decision model:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<semantic:definitions
xmlns:semantic="http://www.omg.org/spec/DMN/20151101/dmn.xsd"
                     xmlns:feel="http://www.omg.org/spec/FEEL/20140401"

xmlns:tc="http://www.omg.org/spec/DMN/20160719/testcase"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                     namespace="http://www.redhat.com/_c7328033-c355-
43cd-b616-0aceef80e52a" (1)
                     name="dmn-movieticket-ageclassification" (2)
                     id="_99">
```

```
  <semantic:extensionElements/>
  <semantic:inputData displayName="Age" id="_1" name="Age">
    <semantic:variable id="_2" name="Age" typeRef="feel:number"/>
  </semantic:inputData>
  <semantic:decision displayName="AgeClassification" id="_3"
name="AgeClassification">
    <semantic:variable id="_4" name="AgeClassification"
typeRef="feel:string"/>
    <semantic:informationRequirement>
      <semantic:requiredInput href="#_1"/>
    </semantic:informationRequirement>
    <semantic:decisionTable hitPolicy="UNIQUE" id="_5"
outputLabel="AgeClassification">
      <semantic:input id="_6">
        <semantic:inputExpression typeRef="feel:number">
          <semantic:text>Age</semantic:text>
        </semantic:inputExpression>
      </semantic:input>
      <semantic:output id="_7"/>
      <semantic:rule id="_8">
        <semantic:inputEntry id="_9">
          <semantic:text>&lt; 13</semantic:text>
        </semantic:inputEntry>
        <semantic:outputEntry id="_10">
          <semantic:text>"Child"</semantic:text>
        </semantic:outputEntry>
      </semantic:rule>
      <semantic:rule id="_11">
        <semantic:inputEntry id="_12">
          <semantic:text>[13..65)</semantic:text>
        </semantic:inputEntry>
        <semantic:outputEntry id="_13">
          <semantic:text>"Adult"</semantic:text>
        </semantic:outputEntry>
      </semantic:rule>
      <semantic:rule id="_14">
        <semantic:inputEntry id="_15">
          <semantic:text>&gt;= 65</semantic:text>
        </semantic:inputEntry>
        <semantic:outputEntry id="_16">
          <semantic:text>"Senior"</semantic:text>
        </semantic:outputEntry>
      </semantic:rule>
    </semantic:decisionTable>
  </semantic:decision>
</semantic:definitions>
```

1. Model namespace
2. Model name

This basic file captures enough information to encapsulate the business logic, the input and outputs of the overall decision, and enough detail to enable software tools to graphically represent the relationships consistently.

The **namespace** and **name** attributes of the root **definitions** tag uniquely identify this decision model.

Like much XML, the **namespace** value appears as a unique URL associated with the organization or individual that authored the document.

CHAPTER 4. DMN SUPPORT IN RED HAT PROCESS AUTOMATION MANAGER

Red Hat Process Automation Manager includes full runtime support for DMN 1.2 models at conformance level 3, but currently does not include a built-in DMN model editor. Editors for DMN models will be added to the platform in the near future, but meanwhile you can use third-party DMN authoring platforms to create DMN models and then integrate the DMN models in your decision services in Red Hat Process Automation Manager. You can import DMN files into your project in Business Central (**Menu** → **Design** → **Projects** → **Import Asset**) or package the DMN files as part of your project knowledge JAR (KJAR) file without Business Central. In addition to all DMN conformance level 3 requirements, Red Hat Process Automation Manager also includes enhancements and fixes to FEEL and DMN model components to optimize the experience of implementing DMN decision services with Red Hat Process Automation Manager.

From a platform perspective, DMN models are like any other business asset in Red Hat Process Automation Manager, such as DRL files or uploaded decision tables, that you can include in your Red Hat Process Automation Manager project and deploy to Process Server in order to start your DMN decision services. This enables you to use your preferred DMN authoring tool to design your DMN models that you then deploy with your existing Red Hat Process Automation Manager assets. For example, you might have BPMN models that directly invoke DMN decision services from their decision task nodes.

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

4.1. CONFIGURABLE DMN PROPERTIES IN RED HAT PROCESS AUTOMATION MANAGER

Red Hat Process Automation Manager provides the following DMN properties that you can configure when you execute your DMN models on Process Server or on your client application:

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 process 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 process 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.KieDMNSignavioProfil
e
```

CHAPTER 5. OPTIONS FOR INVOKING A DMN MODEL

You can import DMN files into your Red Hat Process Automation Manager project using Business Central (**Menu** → **Design** → **Projects** → **Import Asset**) 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 Process Automation Manager project, you can invoke the DMN decision service by deploying the KIE container that contains it to Process Server for remote access or 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 more information about including DMN assets with your project packaging and deployment method, see *Packaging and deploying a Red Hat Process Automation Manager project*.

5.1. 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.

Prerequisite

A KJAR containing the DMN model to execute has been created. For information about project packaging, see *Packaging and deploying a Red Hat Process Automation Manager project*.

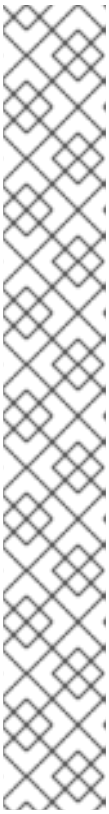
Procedure

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

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

The **<version>** is the Maven artifact version for Red Hat Process Automation Manager currently used in your project (for example, 7.11.0.Final-redhat-00002).



NOTE

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

Example BOM dependency:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.1.0.GA-redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

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

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

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

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

Alternative option:

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

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

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

```
DMNRuntime dmnRuntime =
kieContainer.newKieSession().getKieRuntime(DMNRuntime.class);

String namespace = "http://www.redhat.com/_c7328033-c355-43cd-b616-
0aceef80e52a";
String modelName = "dmn-movieticket-ageclassification";

DMNModel dmnModel = dmnRuntime.getModel(namespace, modelName);
```

4. Execute the decision services for the desired model:

```
DMNContext dmnContext = dmnRuntime.newContext();   (1)

for (Integer age : Arrays.asList(1,12,13,64,65,66)) {
    dmnContext.set("Age", age);   (2)
    DMNResult dmnResult =
        dmnRuntime.evaluateAll(dmnModel, dmnContext);   (3)

    for (DMNDecisionResult dr : dmnResult.getDecisionResults()) {
(4)
        log.info("Age: " + age + ", " +
                 "Decision: '" + dr.getDecisionName() + "', " +
                 "Result: " + dr.getResult());
  }
}
```

(1) Instantiate a new DMN Context to be the input for the model evaluation. Note that this example is looping through the Age Classification decision multiple times.

(2) Assign input variables for the input DMN context.

(3) Evaluate all DMN decisions defined in the DMN model.

(4) Each evaluation may result in one or more results, creating the loop.

This example prints the following output:

```
Age 1 Decision 'AgeClassification' : Child
Age 12 Decision 'AgeClassification' : Child
Age 13 Decision 'AgeClassification' : Adult
Age 64 Decision 'AgeClassification' : Adult
Age 65 Decision 'AgeClassification' : Senior
Age 66 Decision 'AgeClassification' : Senior
```

5.2. EXECUTING DMN SERVICES REMOTELY ON PROCESS SERVER (JAVA)

The KIE remote API client provides a lightweight approach to invoking a remote DMN service either through the REST or JMS interfaces of Process 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.

Prerequisites

- Process 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 Process Automation Manager installation*.
- A KIE container is deployed in Process Server in the form of a KJAR that includes the DMN model. For information about project packaging and deployment, see *Packaging and deploying a Red Hat Process Automation Manager project*.

- You have the container 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 dependencies to the relevant classpath of your Java project:

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

The **<version>** is the Maven artifact version for Red Hat Process Automation Manager currently used in your project (for example, 7.11.0.Final-redhat-00002).

NOTE

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

Example BOM dependency:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.1.0.GA-redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

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

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

```
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 Process Server by invoking the method **`getServicesClient()`** on the KIE server Java client instance:

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

   The **`dmnClient`** can now execute decision services on Process Server.

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

```
for (Integer age : Arrays.asList(1,12,13,64,65,66)) {
    DMNContext dmnContext = dmnClient.newContext(); (1)
    dmnContext.set("Age", age);  (2)
    ServiceResponse<DMNResult> serverResp =  (3)
        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.

4 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**.

TIP

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

5.3. CALLING A DMN SERVICE ON A REMOTE SERVER USING REST APIS

Directly interacting with the REST endpoints of Process 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.

Prerequisites

- Process 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 Process Automation Manager installation*.
- A KIE container is deployed in Process Server in the form of a KJAR that includes the DMN model. For information about project packaging and deployment, see *Packaging and deploying a Red Hat Process Automation Manager project*.
- You have the container 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 Process 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/server**)

   Local deployment example URL:

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

2. Determine user authentication requirements.

When users are defined directly in the Process Server configuration, BasicAuth is used which 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 Process 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] /containers/*CONTAINER_ID*/dmn

   Example curl request:

   ```
   curl -u krisv:krisv -H "accept: application/xml" -X GET
   "http://localhost:8080/kie-
   server/services/rest/server/containers/MovieDMNContainer/dmn"
   ```

   Sample XML output:

   ```
   <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
   <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>
               <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:

```
{
  "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] `/containers/`*`CONTAINER_ID`*`/dmn`

   Example curl request:

```
curl -u krisv:krisv -H "accept: application/json" -H "content-type:
application/json" -X POST "http://localhost:8080/kie-
server/services/rest/server/containers/MovieDMNContainer/dmn" -d "{
\"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 JSON request:

```
{
  "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 style):

```
<?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>
```

NOTE

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

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

Example JSON response:

```
{
  "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 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-b616-0aceef80e52a</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>
                  <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"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">Senior</result>
                              <messages/>
                              <status>SUCCEEDED</status>
                        </value>
                  </entry>
            </decisionResults>
      </dmn-evaluation-result>
</response>
```

CHAPTER 6. ADDITIONAL RESOURCES

- Decision Model and Notation specification
- DMN Technology Compatibility Kit
- *Packaging and deploying a Red Hat Process Automation Manager project*

APPENDIX A. VERSIONING INFORMATION

Documentation last updated on Friday, October 12, 2018.