Red Hat Decision Manager 7.0

Designing a decision service using DMN models

Red Hat Decision Manager 7.0 Designing a decision service using DMN models

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

Legal Notice

Abstract

This document describes how to design a decision service using Decision Model and Notation (DMN) models in Red Hat Decision Manager 7.0.

# 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.

CHAPTER 1. 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
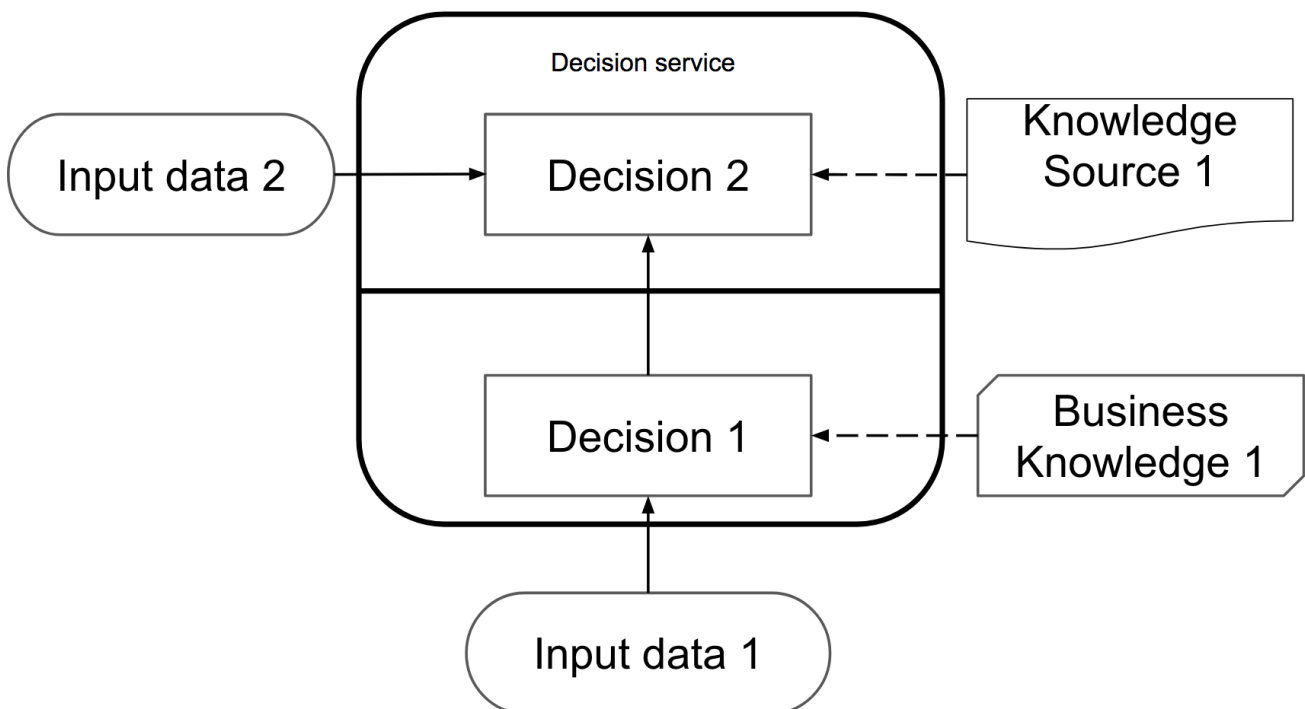
1.1. RULE EXPRESSIONS IN FEEL

Friendly Enough Expression Language (FEEL) is a new expression language defined by the DMN specification. It aims to bridge the gap between 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. FEEL expressions define the logic of a decision.

For more information about FEEL in DMN, see the OMG DMN Specification.

1.2. DECISION TABLES

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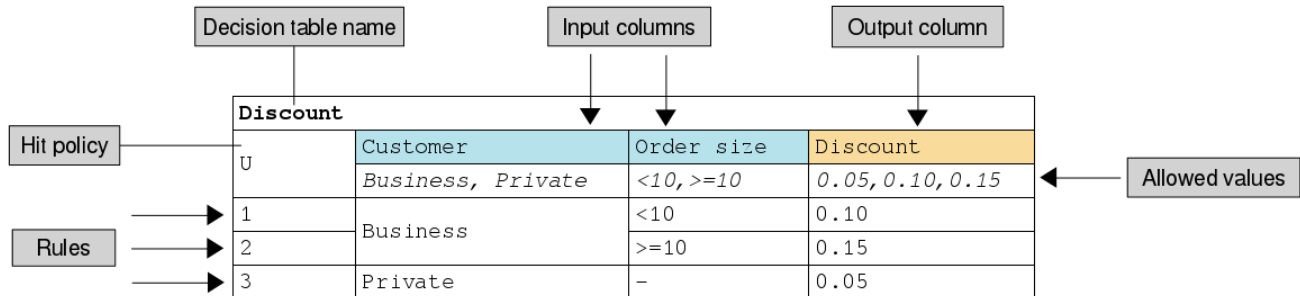
Decision tables are a popular way for modeling rules and decisions, and are used in many methodologies (such as DMN) and implementation frameworks (such as Drools used in Red Hat Decision Manager).

> **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 native decision tables are defined by the Drools project. Red Hat Decision 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.

1.2.1. 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.

1.3. 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 cancelled, 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

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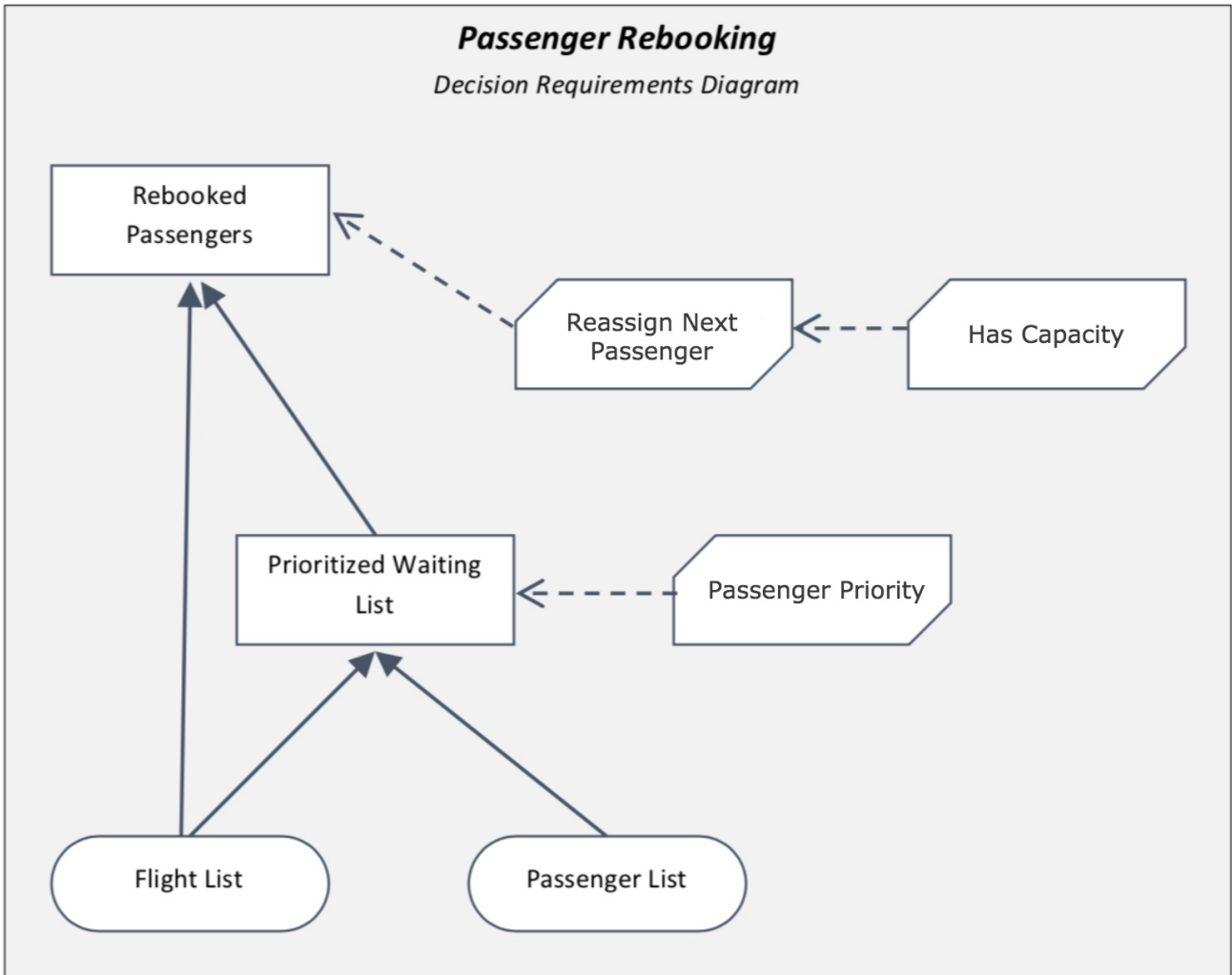
Similar to flowcharts, DRDs use shapes to represent the different elements in a process. Ovals contain the two necessary inputs, rectangles contain the decision points in the model, and rectangles with clipped corners contain reusable logic that can be repeatedly invoked.

Furthermore, 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

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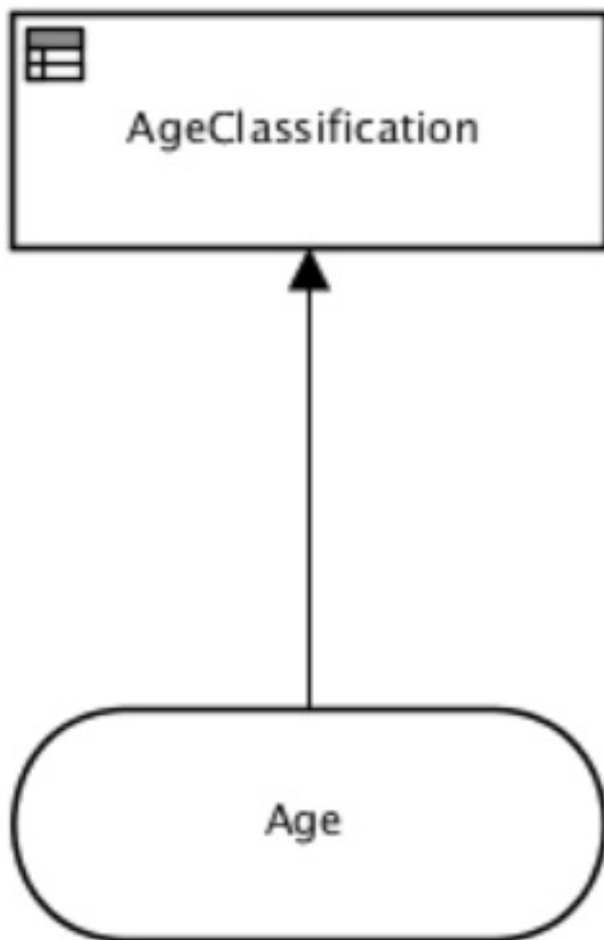
This example consists of a single numeric input value (**Age**), and produces a string output (**AgeClassification**). The inner workings of the **AgeClassification** decision is a basic table:

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" ❶
                      name="dmn-movieticket-ageclassification" ❷
                      id="_99">
```

```xml
    <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. OPTIONS FOR INVOKING A DMN MODEL

To invoke a decision that is defined in a DMN file, you must first package the file in a KIE container. A specific version of knowledge components comes in a knowledge JAR (KJAR), which you can either deploy to Decision Server for remote access, or manipulate directly as a dependency of the calling application. Covering all options for creating and deploying these knowledge packages is outside the scope of this document, although most are similar to other knowledge assets (for example, a Drools rule file or jBPM process definition). After you have packaged or deployed the decision, you have several options for invoking it.

4.1. EMBEDDING A DMN CALL DIRECTLY INTO THE 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 should 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 should only take effect after you have intentionally updated and redeployed the application. One potential benefit of this approach is that proper operation does not rely on any external dependencies to the runtime, 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 runtime.

Prerequisites

- A KJAR containing the DMN model to execute has been created.

- The following dependencies have been added to the **pom.xml** file of the project:

```xml
<!-- Required for the DMN runtime API -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-dmn-core</artifactId>
  <version>${drools-version}</version>
</dependency>

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

> **NOTE**
>
> **${drools-version}** is a Maven property that should resolve to the precise version used for other KIE / Drools dependencies at runtime.

Procedure

1. 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();
```

2. 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);
```

3. 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() +
"' : " + 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
```

4.2. EXECUTING DMN SERVICES REMOTELY ON DECISION 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 Decision Server. This approach reduces the number of runtime dependencies necessary to interact with a knowledge base. Decoupling the calling code from the decision definition also increases flexibility by enabling them to iterate independently at the appropriate pace.

Prerequisites

- Decision Server is installed and configured, including a known user name and credentials for a user with the `kie-server` role.

- A KIE container is deployed in Decision Server in the form of a KJAR that includes the DMN model.

- 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.

- The following dependency is added to the **pom.xml** file of the project:

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

> **NOTE**
>
> **${drools-version}** is a Maven property that should resolve to the precise version used for other KIE / Drools dependencies at runtime.

Procedure

1. 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.

2. Obtain a **DMNServicesClient** from the KIE server Java client connected to the related Decision 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 Decision Server.

3. 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() + "'
: " + 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.
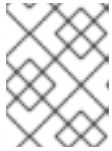
> **NOTE**
>
> You can also execute only a specific DMN Decision in the model, by making use of alternative methods of the **DMNServicesClient**.

TIP

If the KIE container only contains one DMN Model, you can omit **$modelNamespace** and **$modelName** because the KIE API will select it by default.

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

Directly interacting with the REST endpoints of Decision 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

- Decision Server is installed and configured, including service user accounts to allow access.

- A KIE container is deployed in Decision Server in the form of a KJAR that includes the DMN model.

- 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 REST 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 Decision 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 Decision 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 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
   <?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 5. RELATED INFORMATION

Packaging and deploying a decision service

APPENDIX A. VERSIONING INFORMATION

Documentation last updated on: Monday, October 1, 2018.