This PDF is generated from authoritative online content, and is provided for convenience only. This PDF cannot be used for legal purposes. For authoritative understanding of what is and is not supported, always use the online content. To copy code samples, always use the online content.

Genesys Rules Authoring Tool Help

Examples of template development

4/25/2025

Contents

Examples of template development

This section provides some examples of what a rule developer might configure in the Template Development tab. For specific information about how rule templates are configured to be used with the Genesys intelligent Workload Distribution (iWD) solution, refer to the *iWD and Genesys Rules System* guide.

Example 1: Condition and Action

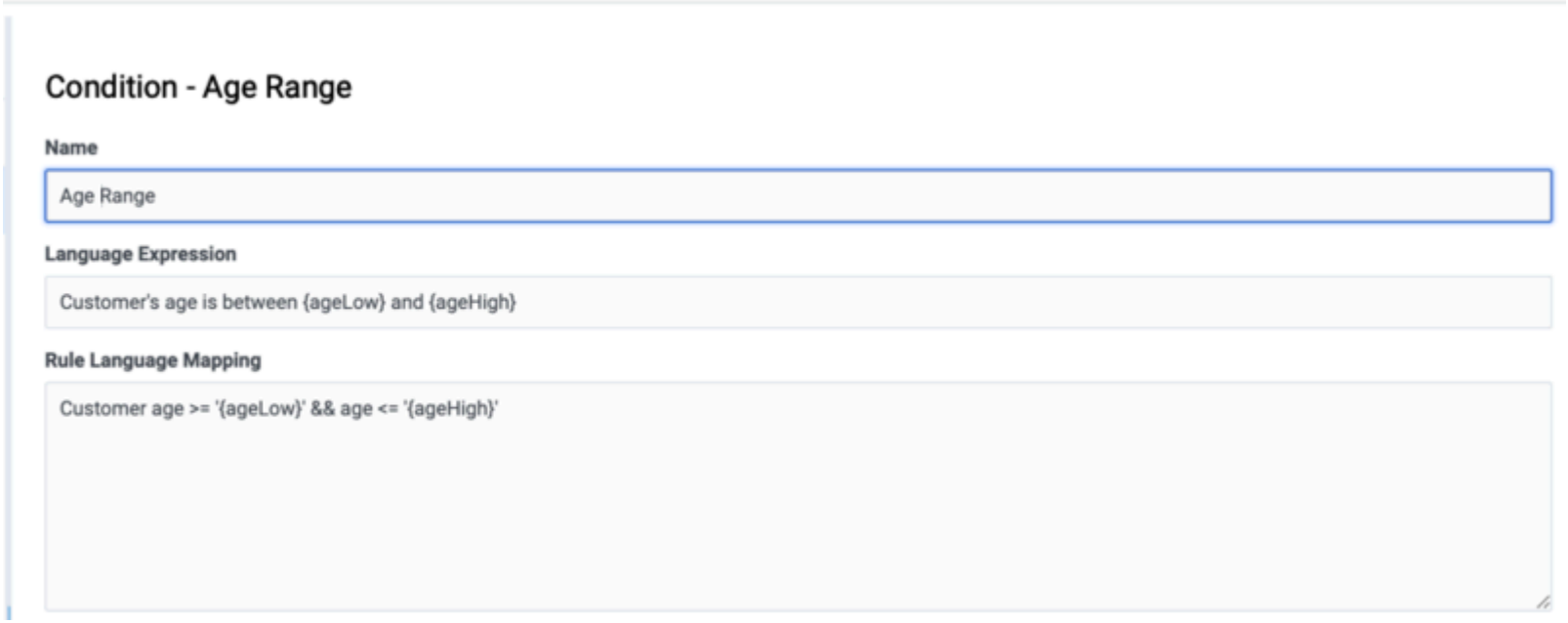
Age Range Condition

If a customer's age is within a specific range, a specific Agent Group will be targeted. In this scenario, the Condition is whether the customer's age falls within the range. In the Genesys Rules Development Tool, the conditions would be configured as follows:

```
Name: Age Range
Language Expression: Customer's age is between {ageLow} and {ageHigh}
Rule Language Mapping: Customer(age >= '{ageLow}' && age <= '{ageHigh}')
```

Do not use the word 'end' in rule language expressions. This causes rule parsing errors.

The figure below shows how this condition would appear in GRAT Template Development.



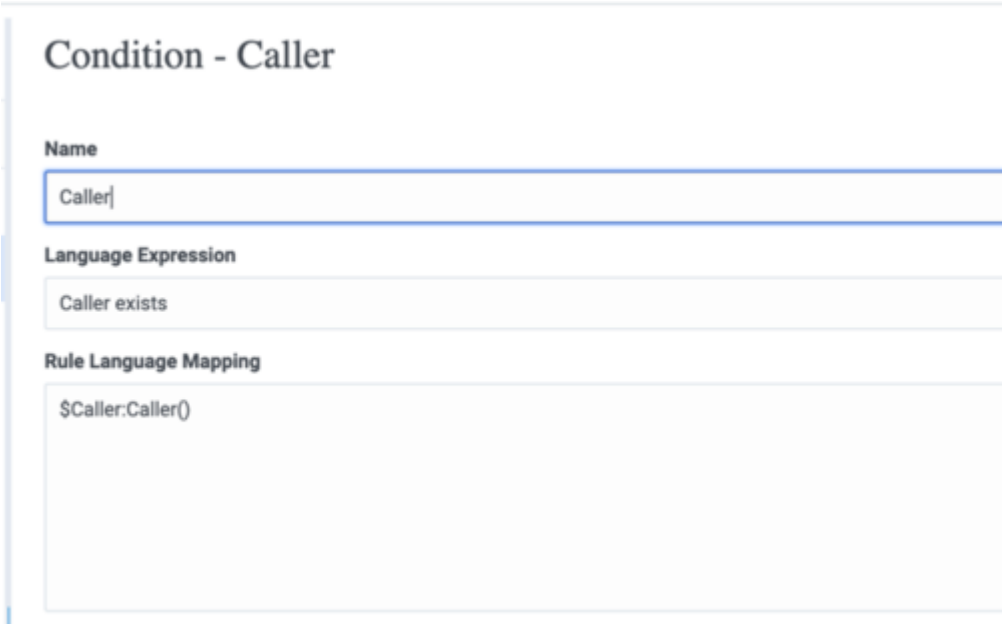
Caller Condition

In addition to testing that the Caller exists, the next condition also creates the `$Caller` variable which is used by actions to modify the Caller fact. The modified Caller will be returned in the results of the evaluation request.

You cannot create a variable more than once within a rule, and you cannot use variables in actions if the variables have not been defined in the condition.

```
Name: Caller
Language Expression:  Caller exists
Rule Language Mapping: $Caller:Caller
```

The figure below shows how this condition would appear in GRAT Rules Development.



Target Agent Group Action

The action would be configured as follows:

```
Name: Route to Agent Group
Language Expression: Route to agent group {agentGroup}
Rule Language Mapping: $Caller.targetAgentGroup='{agentgroup}'
```

The figure below shows how this action would appear in GRAT Rules Development.

Parameters

The condition in this example has two parameters:

- `{ageLow}`
- `{ageHigh}`

The action has the `{agentGroup}` parameter. The Parameters Editor screenshot shows a sample `{ageHigh}` parameter.

How it works

The way the preceding example would work is as follows:

1. The rule developer creates a fact model (or the fact model could be included as part of a rule template that comes out of the box with a particular Genesys solution). The fact model describes the properties of the `Customer fact` and the `Caller` fact. In this case we can see that the `Customer` fact has a property called age (probably an integer) and the `Caller` fact has a property called `targetAgentGroup` (most likely a string).
2. The rule developer creates the `ageLow` and `ageHigh` parameters, which will become editable fields that the business user will fill in when they are

authoring a business rule that uses this rule template. These parameters would be of type `Input Value` where the `Value Type` would likely be integer. The rule developer optionally can constrain the possible values that the business user will be able to enter by entering a minimum and/or a maximum.

3. The rule developer also creates the `agentGroup` parameter, which will likely be a selectable list whereby the business user would be presented with a drop-down list of values that are pulled from Genesys Configuration Server or from an external data source. The behavior of this parameter depends on the parameter type that is selected by the rule developer.
4. The rule developer creates a rule action and rule condition as previously described. The action and condition include rule language mappings that instruct the Rules Engine as to which facts to use or update based on information that is passed into the Rules Engine as part (of the rule evaluation request coming from a client application such as an SCXML application).
5. The rule developer publishes the rule template to the Rules Repository.
6. The rules author uses this rule template to create one or more business rules that utilize the conditions and actions in the combinations that are required to describe the business logic that the rules author wants to enforce. In this case, the previously described conditions and action above likely would be used together in a single rule, but the conditions and action could also be combined with other available conditions and actions to create different business policies.
7. The rules author deploys the rule package to the Rules Engine application server.
8. A client application such as a VXML or SCXML application invokes the Rules Engine and specifies the rule package to be evaluated. The request to the Rules Engine will include the input and output parameters for the fact model. In this example, it would have to include the age property of the Customer fact. This age might have been collected through GVP or extracted from a customer database prior the Rules Engine being called. Based on the value of the `Customer.age` fact property that is passed into the Rules Engine as part of the rules evaluation request, the Rules Engine will evaluate a particular set of the rules that have been deployed. In this example, it will evaluate whether `Customer.age` falls between the lower and upper boundaries that the rules author specified in the rule.
9. If the rule evaluates as true by the Rules Engine, the `targetAgentGroup` property of the `Caller` fact will be updated with the name of the Agent Group that was selected by the business rules author when the rule was written. The value of the `Caller.targetAgentGroup` property will be passed back to the client application for further processing. In this example, perhaps the value of `Caller.targetAgentGroup` will be mapped to a Composer application variable which will then be passed into the Target block to ask the Genesys Universal Routing Server to target that Agent Group.

Example 2: Function

Functions are used for more complex elements and are written in Java. In this example, the function is used to compare dates. It would be configured as follows:

```
Name: compareDates
```

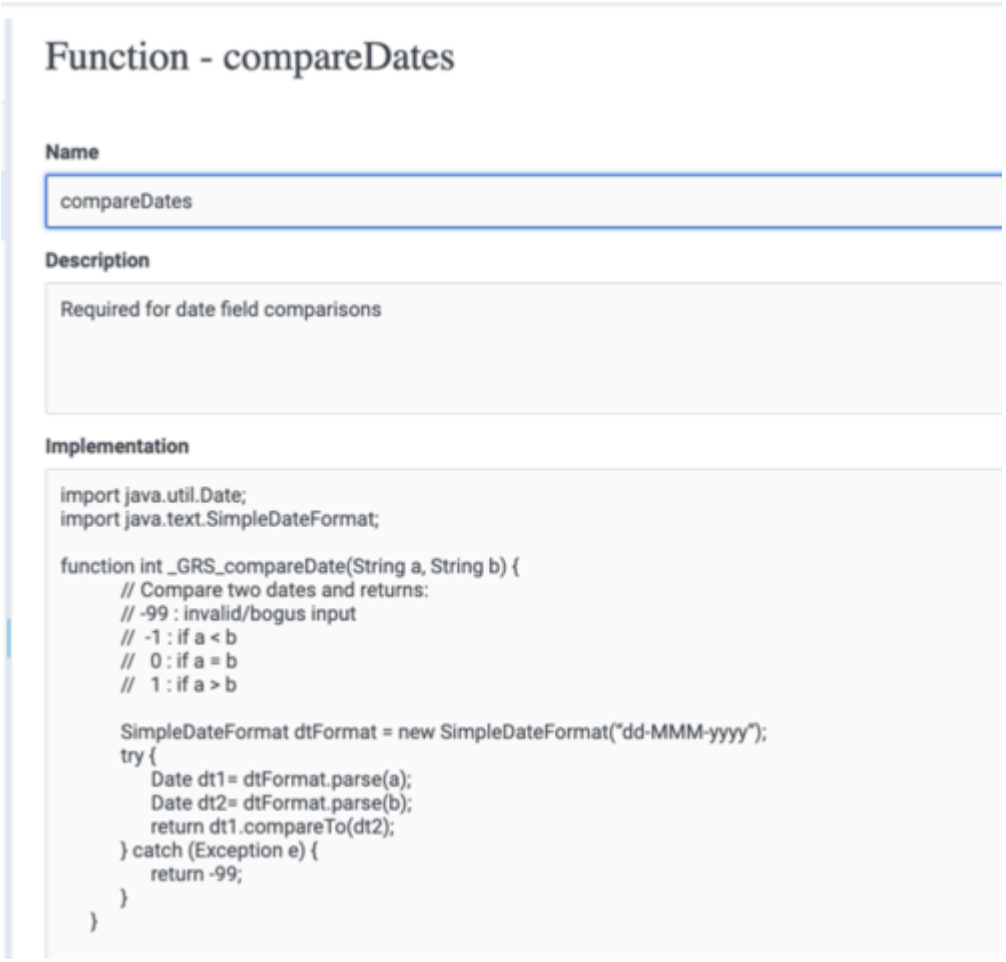
```
Description: This function is required to compare dates.
Implementation:
import java.util.Date;
import java.text.SimpleDateFormat;

function int _GRS_compareDate(String a, String b) {
            // Compare two dates and returns:
            // -99 : invalid/bogus input
            //  -1 : if a < b
            //   0 : if a = b
            //   1 : if a > b

            SimpleDateFormat dtFormat = new SimpleDateFormat(“dd-MMM-yyyy”);
            try {
                  Date dt1= dtFormat.parse(a);
                  Date dt2= dtFormat.parse(b);
                  return dt1.compareTo(dt2);
            } catch (Exception e) {
                  return -99;
            }
      }
```

For user-supplied classes, the .jar file must be in the CLASSPATH for both the GRAT and the GRE.

The figure below shows how this function would appear in GRAT Rules Development.



Function - compareDates

Name

compareDates

Description

Required for date field comparisons

Implementation

```
import java.util.Date;
import java.text.SimpleDateFormat;

function int _GRS_compareDate(String a, String b) {
        // Compare two dates and returns:
        // -99 : invalid/bogus input
        //  -1 : if a < b
        //   0 : if a = b
        //   1 : if a > b

        SimpleDateFormat dtFormat = new SimpleDateFormat("dd-MMM-yyyy");
        try {
            Date dt1= dtFormat.parse(a);
            Date dt2= dtFormat.parse(b);
            return dt1.compareTo(dt2);
        } catch (Exception e) {
            return -99;
        }
    }
```

Example 3: Using a JSON Object

Template developers can create templates that enable client applications to pass Facts to GRE as JSON objects without having to map each field to the fact model explicitly.

> **Important**
>
> Rules based on templates that use this functionality do not support the creation of test scenarios at present.

This example shows how to create a template containing a class (called `MyJson`) for passing a JSON object.

Start

1. Create the following class and import it into a rule template:

```
package simple;
 import org.json.JSONObject;
 import org.apache.log4j.Logger;

 public class MyJson {
         private static final Logger LOG = Logger.getLogger(MyJson.class);
         private JSONObject jsonObject = null;

         public String getString( String key) {
                         try {
                                         if ( jsonObject != null)
                                                         return jsonObject.getString( key);
                         } catch (Exception e) {
                         }
                         LOG.debug("Oops, jsonObect null ");
                         return null;
         }

         public void put( String key, String value) {
                         try {
                         if (jsonObject == null) {
```

```
                                jsonObject = new JSONObject();
                        }
                        jsonObject.put( key, value);
                        } catch (Exception e) {
                        }
        }
}
```

2. Create a dummy fact object with the same name (`MyJson`) in the template.
3. Add the `MyJson.class` to the class path of both GRAT and GRE.
4. Create the following condition and action:

```
Is JSON string "{key}" equal "{value}"       eval($MyJson.getString("{key}").equals("{value}"))
 Set JSON string "{key}" to "{value}"          $MyJson.put("{key}", "{value}");
```

5. Use this condition and action in a rule within the `json.test` package. The following will be generated:

```
rule "Rule-100 Rule 1"
 salience 100000
   agenda-group "level0"
   dialect "mvel"
   when
          $MyJson:MyJson()
          and (
          eval($MyJson.getString("category").equals("test"))
          )
   then
          $MyJson.put("newKey", "newValue");
 end
```

6. Deploy the `json.test` package to GRE.
7. Run the following execution request from the RESTClient:

```
{"knowledgebase-request":{
 "inOutFacts":{"anon-fact":{"fact":{"@class":"simple.MyJson", "jsonObject":
 {"map":{"entry":[{"string":["category","test"]},{"string":["anotherKey","anotherValue"]}]}}}}}}}
```

8. The following response is generated:

```
{"knowledgebase-response":{"inOutFacts":{"anon-fact":[{"fact":{"@class":"simple.MyJson","jsonObject":
```

```
{"map":{"entry":[{"string":["category","test"]},{"string":["newKey","newValue"]},
{"string":["anotherKey","anotherValue"]}]}}}}],
"executionResult":{"rulesApplied":{"string":["Rule-100 Rule 1"]}}}}}
```


End

Example 4: Developing Templates to Enable Test Scenarios

> **Important**
>
> Creation and editing of events is not supported in the initial 9.0.0 release of GRAT, so templates that support test scenarios cannot be developed. However, templates supporting events/test scenarios created in the Genesys Rules Development Tool in 8.5 can still be developed in GRDT, imported to GRAT 9.0 and used to build rules packages that support events/test scenarios.

For more information on this topic, please refer to *Developing templates to enable test scenarios* in the 8.1.3 GRDT Help.

Mapping Multiple Instances of a Rule Parameter to a Single Parameter Definition

At the point of creating parameters, instead of creating the `ageLow` and `ageHigh` parameters the rule template developer can create a single `{age}` parameter and use the underscore notation shown in the example below to create indices of it for scenarios in which multiple instances of parameter with the same type `(age)` are required (most commonly used with ranges). For example: `{age_1}, {age_2}....{age_n}` These will become editable fields. This feature is most typically used for defining ranges more efficiently.

Fact/Condition

Facts can be referenced in conditions and actions by prefixing the fact name by a `$` sign. For example, the fact `Caller` can be referenced by the name `$Caller`. GRS will implicitly generate a condition that associates the variable `$Caller` to the fact `Caller` (that is, `$Caller:Caller()`).

The condition `$Caller:Caller()` requires a `Caller` object as input to rules execution for this condition to evaluate to true.