Genesys Rules Authoring Tool 8.5.x Help

Exemples de développement de modèle

12/16/2025

Contents

Exemples de développement de modèle

Cette section fournit des exemples de règles configurables par un développeur de règle dans l'onglet Développement du modèle. Pour des informations spécifiques sur la façon dont les modèles de règles sont configurés pour être utilisés avec la solution Genesys intelligent Workload Distribution (iWD), reportez-vous au guide *iWD et Genesys Rules System*.

Exemple 1 : condition et action

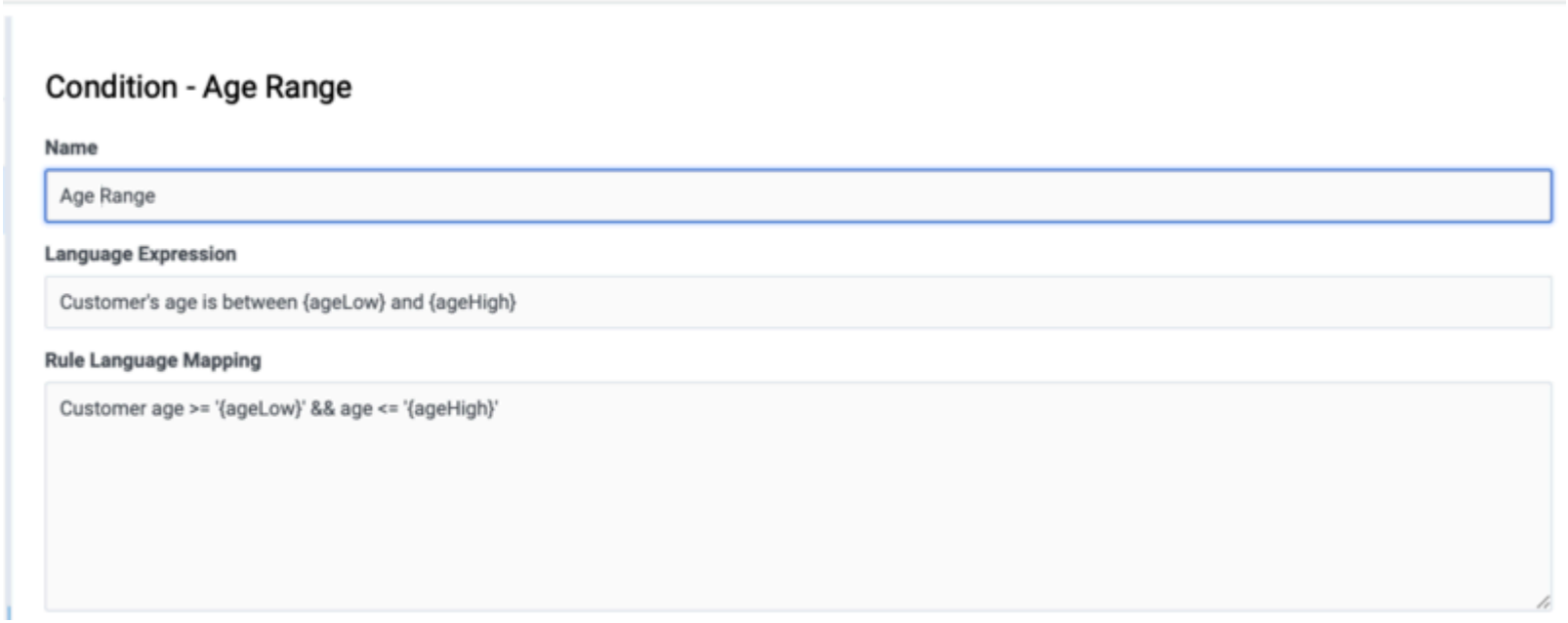
Condition de plage d'âges

Si l'âge du client est compris dans une plage spécifique, un groupe d'agents spécifique sera ciblé. Dans ce scénario, la condition détermine si l'âge du client est compris dans la plage. Dans l'outil Genesys Rules Development, les conditions seraient configurées comme suit :

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

Ne pas utiliser le mot « end » dans les expressions de langage de règle. Cela engendre des erreurs d'analyse de la règle.

La figure ci-dessous montre comment cette condition doit apparaître dans le développement de modèles GRAT.



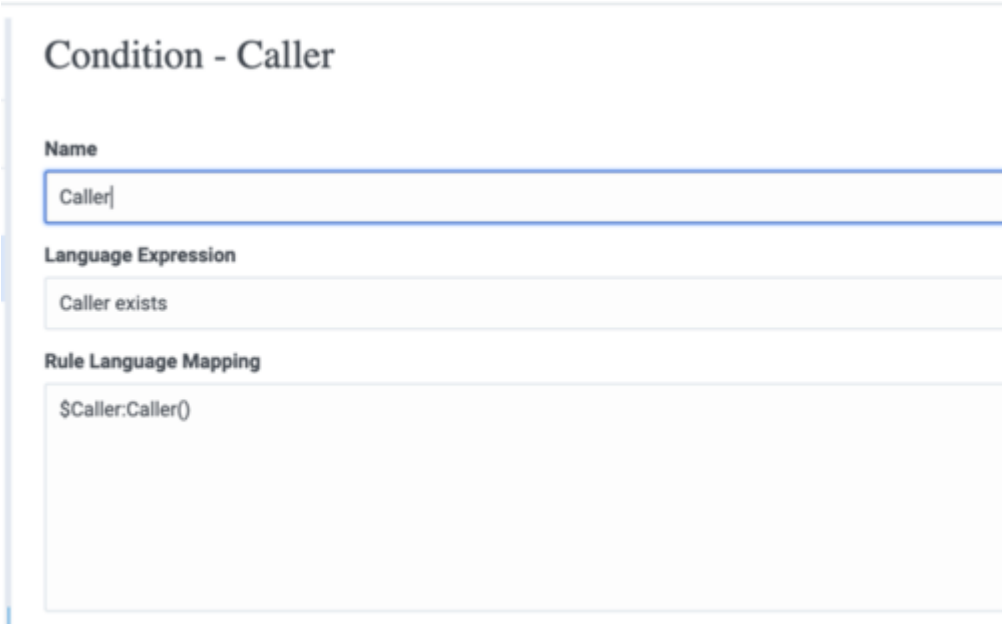
Condition de l'appelant

Outre le contrôle du fonctionnement de l'appelant, la condition suivante crée également la variable `$Caller` utilisée par des actions pour modifier le fait de l'appelant. L'appelant modifié sera repris dans les résultats de la demande d'évaluation.

Vous ne pouvez pas créer une variable plusieurs fois dans une règle; vous ne pouvez pas utiliser de variables dans les actions si celles-ci n'ont pas été définies dans la condition.

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

La figure ci-dessous montre comment cette condition devrait apparaître dans le développement des règles GRAT.



Action du groupe d'agents cible

L'action serait configurée de la manière suivante :

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

La figure ci-dessous montre comment cette action doit apparaître dans le développement des règles GRAT.

Paramètres

La condition dans cet exemple comporte deux paramètres :

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

L'action contient le paramètre `{agentGroup}`. La capture d'écran Éditeur Paramètres affiche un exemple de paramètre `{ageHigh}`.

Fonctionnement

La façon dont l'exemple précédent fonctionne est la suivante :

1. Le développeur de règles crée un modèle de fait (ou bien le modèle de fait peut être inclus dans un modèle de règle sortant du contenu d'une solution Genesys spécifique). Le modèle de fait décrit les propriétés du `fait Client` et du fait `Appelant`. Dans ce cas, le fait `Client` a une propriété âge (probablement un nombre entier), tandis que le fait `Appelant` a une propriété intitulée `targetAgentGroup` (probablement une chaîne).
2. Le développeur de la règle crée les paramètres `agedLow` et `ageHigh`, qui deviennent des champs modifiables que l'utilisateur renseignera lorsqu'il créera

une règle commerciale qui utilise ce modèle de règle. Ces paramètres sont de type `Valeur d'entrée` où le `type de valeur` serait probablement un entier. Le développeur de la règle peut limiter les valeurs possibles auxquelles l'utilisateur professionnel peut accéder en entrant un minimum et/ou un maximum.

3. Le développeur de la règle crée également le paramètre `agentGroup`. Il s'agira probablement d'une liste sélectionnable permettant de présenter à l'utilisateur une liste déroulante contenant des valeurs extraites de Genesys Configuration Server ou d'une source de données externe. Le comportement de ce paramètre dépend du type de paramètre sélectionné par le développeur de la règle.
4. Le développeur de règle crée une action de règle et une condition de règle, comme décrit précédemment. L'action et la condition incluent des mappages de langage de règles qui instruisent Rules Engine sur les faits à utiliser ou mettre à jour en fonction des informations transmises dans Rules Engine dans le cadre de la demande d'évaluation de règle émise par une application client, par exemple, une application SCXML.
5. Le développeur de règle publie le modèle de règle dans le référentiel de règles.
6. Le créateur de règles utilise ce modèle de règle pour créer une ou plusieurs règles commerciales utilisant les conditions et les actions des combinaisons requises pour décrire la logique commerciale que le créateur de règles souhaite appliquer. Dans ce cas, les conditions et l'action décrites précédemment seront probablement utilisées conjointement dans une seule règle, mais les conditions et l'action pourraient également être combinées avec d'autres conditions et actions disponibles pour créer des stratégies d'entreprise.
7. Le créateur de règles déploie l'ensemble de règles sur le serveur de l'application Rules Engine.
8. Une application client, telle qu'une application VXML ou SCXML, appelle Rules Engine et spécifie l'ensemble de règles à évaluer. La demande adressée à Rules Engine inclura les paramètres d'entrée et de sortie du modèle de fait. Dans cet exemple, il faudra que la propriété âge du fait Client soit incluse. Cet âge a pu être recueilli par le biais de GVP ou extrait d'une base de données clients avant l'appel de Rules Engine. Rules Engine évaluera un certain ensemble de règles déployées sur la base de la valeur de la propriété du fait `Customer.age` transmise au Rules Engine dans le cadre de la demande d'évaluation des règles. Dans cet exemple, il évalue si `Customer.age` se situe entre les limites supérieure et inférieure que le créateur de règles a spécifiées dans la règle.
9. Si Rules Engine détermine que la règle est true, la propriété `targetAgentGroup` du fait `Appelant` sera mise à jour avec le nom du groupe d'agents sélectionné par le créateur des règles commerciales lors de l'écriture de la règle. La valeur de la propriété `Caller.targetAgentGroup` sera renvoyée à l'application client pour poursuivre le traitement. Dans cet exemple, la valeur de `Caller.targetAgentGroup` sera peut-être mappée sur une variable d'application Composer qui sera ensuite transmise au bloc cible pour demander à Genesys Universal Routing Server de cibler ce groupe d'agents.

Exemple 2 : Fonction

Les fonctions sont utilisées pour des éléments plus complexes et sont écrites en Java. Dans cet exemple, la fonction permet de comparer des dates. Elle pourrait être configurée de la manière suivante :

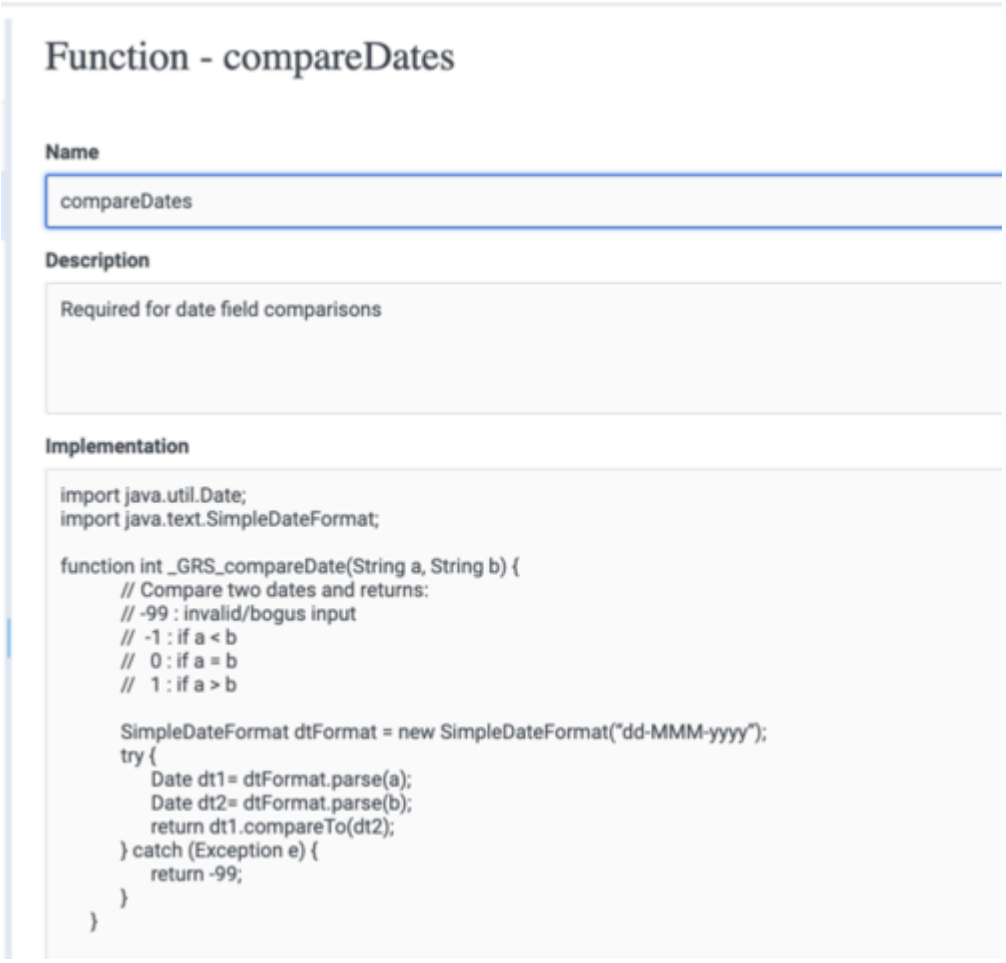
```
Name: compareDates
Description: This function is required to compare dates.
Implementation:
import java.util.Date;
import java.text.SimpleDateFormat;

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;
            }
      }
```

Pour les classes fournies par l'utilisateur, le fichier .jar doit être dans la variable CLASSPATH pour GRAT et GRE.

La figure ci-dessous montre comment cette fonction apparaîtrait dans le développement des règles GRAT.



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;
        }
    }
```

Exemple 3 : utilisation d'un objet JSON

Les développeurs de modèles peuvent créer des modèles qui permettent aux applications client de transmettre les faits à GRE en tant qu'objets JSON sans avoir à explicitement associer chaque champ au modèle de fait.

> **Important**
>
> Les règles basées sur des modèles qui utilisent cette fonctionnalité ne prennent pas en charge la création de scénarios de test pour le moment.

L'exemple montre comment créer un modèle contenant une classe (nommée `MyJson`) pour transmettre un objet JSON.

Démarrer

1. Créez la classe suivante et importez-la dans un modèle de règle :

```
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. Créez un objet fait factice portant le même nom (`MyJson`) dans le modèle.
3. Ajoutez `MyJson.class` au chemin d'accès de GRAT et GRE.
4. Créez la condition et l'action suivantes :

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

5. Appliquez cette condition et cette action à une règle de l'ensemble `json.test`. Les éléments suivants seront créés :

```
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. Déployez l'ensemble `json.test` dans GRE.
7. Exécutez la demande d'exécution suivante depuis le RESTClient :

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

8. La réaction suivante est générée :

```
{"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"]}}}}}
```


Fin

Exemple 4 : Développement de modèles pour permettre des scénarios de test

> **Important**
>
> La création et la modification d'événements ne sont pas prises en charge dans la version initiale 9.0.0 de GRAT; les modèles qui prennent en charge les scénarios de test ne peuvent donc pas être développés. Cependant, les modèles prenant en charge des événements/scénarios de test créés dans Genesys Rules Development Tool dans la version 8.5 peuvent tout de même être développés dans GRDT, importés dans GRAT 9.0 et utilisés pour élaborer des ensembles de règles prenant en charge des scénarios d'événement/de test.

Pour plus d'informations, veuillez vous reporter à la rubrique *Développement de modèles pour permettre des scénarios de test* dans l'aide de GRDT 8.1.3.

Mapper plusieurs instances d'un paramètre de règles avec une définition de paramètre unique

Au moment de créer les paramètres, le développeur de modèles de règles peut créer un seul paramètre `{age}` au lieu des paramètres `ageLow` et `ageHigh`, et utiliser la notation de barre de soulignement indiquée dans l'exemple ci-dessous pour en créer des indexes dans lesquels plusieurs instances de paramètres du même type `(age)` sont requises (plus généralement utilisées avec des plages). Par exemple : `{age_1}, {age_2}....{age_n}` Ces champs deviennent accessibles. Cette fonction est généralement utilisée pour définir plus efficacement les plages.

Fait/Condition

Il est possible de référencer les faits dans les conditions et les actions en préfixant le nom de fait par un signe `$`. Par exemple, le fait `Appelant` peut

être référencé par le nom `$Caller`. GRS génère implicitement une condition associant la variable `$Caller` au fait `Appelant` (c'est-à-dire `$Caller:Caller()`).

La condition `$Caller:Caller()` exige un objet `Caller` comme entrée pour l'exécution de règles pour cette condition, afin qu'elle soit vraie (true).

Template:BestPractices