Adventures in Groovy – Part 30: Dynamically Identify Actionable Months

Almost every ePBCS application will require the need to run business logic and execute data movements on specific months based on a scenario selected. Almost every Data Form will require action to be taken dynamically based on the selection of the Scenario. Assuming a fiscal year of Jan to Dec and Actuals being final through Jun,

- processes on Actuals will run on Jan through Jun
- processes on Budget/Plan will run on Jan through Dec
- processes on Forecast will run on Jul through Dec

Prior to Groovy, this was done in an Essbase calculation by writing if statements based on the scenario selected in the POV and checking if the month was part of a variable range. It might have looked something like this.

```
IF((@ISMBR("Plan")   AND   @ISMBR(&v_BudYear)   AND
@ISMBR(&v_BudMths)) OR
    (@ISMBR("Forecast")  AND  @ISMBR(&v_FcstYr1)  AND
@ISMBR(&v_FcstMths1)))
 ...
ENDIF
```

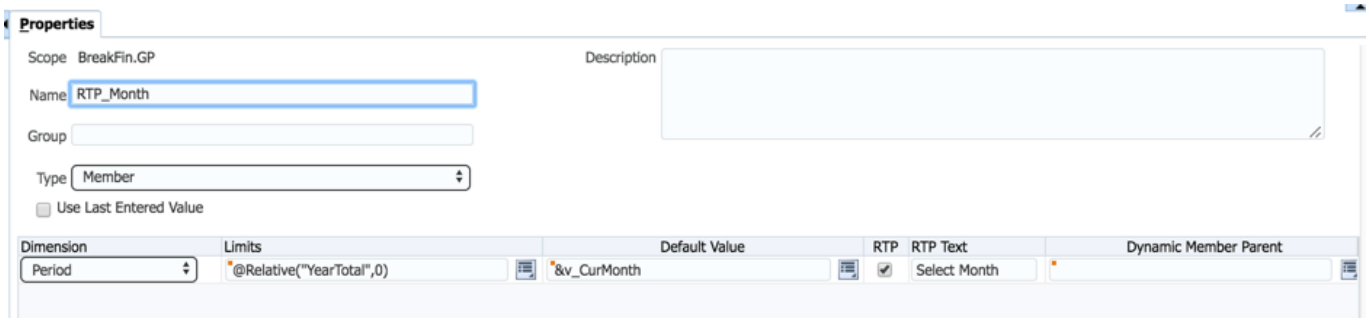
Groovy provides a much more efficient way to accomplish this. Although it may be different based on specific needs, the example below will assume that Forecast and Actuals can be identified based on a variable that is updated monthly identifying the last month of Actuals.

Setting Up The Variables and Run Time

Prompts

This application has a substitution variable named v_CurMonth used to identify the current reporting month of Actuals, or the last month Actuals are final. This will be used in the Run Time Prompt as a default value in several places and this is required.

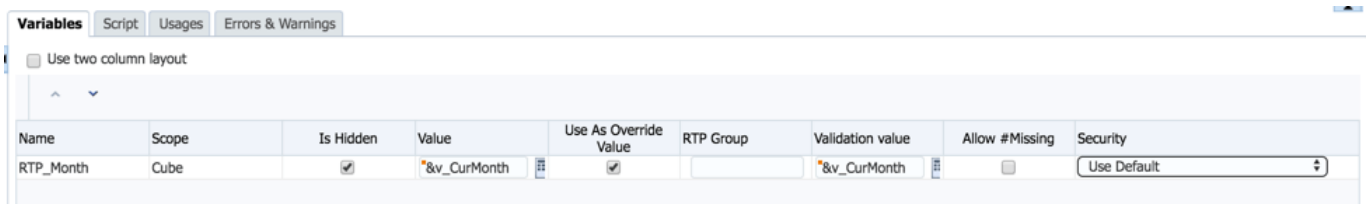
A Run Time Prompt is also required. There is nothing unique about the RTP for this use and the application may likely already have one. This will be a member Type, be connected to the Period dimension, and have a default value of the substitution variable above. It would look similar to this.



Finally, a variable in the Groovy business rule must be created. This variable will be set as an override value with the default value equal to the substitution variable created above. In a Groovy business rule, a variable is instantiated by adding the following line at the top of the rule. It looks like a comment, but it acts differently when it starts with RTPS:.

```
/*RTPS: {RTP_Month}*/
```

After this is added, the Variables tab of the business rule will show all the variable defined. Set the variable so that it is hidden and is used as an override value.



The Code

At this point, all of the required variables are setup. The first step is to define the months. This can be hard coded since this is not something that will change frequently. If this is added to a script and included in all the business rules it is needed, changing it would be easy if the months did change since it is in one place and shared throughout.

```
// Hard coded months
def months = ['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']
```

Optionally, this can be set dynamically. Either option, in my opinion, is acceptable. The result of the following would produce the exact same value for months.

```
// Dynamically get the list of months based on the application's fiscal year
Cube cube = operation.application.getCube("GP")
Dimension productDim = operation.application.getDimension("Period", cube)
def months = productDim.getEvaluatedMembers("ILvl0Descendants(YearTotal)", cube)
```

The next piece of this rule is to get the month from the variable. This will be the key to identify which months are used in calculations that run on Actuals and Forecast.

```
// Get the month from the hidden RTP
def actMonth = rtps.RTP_Month.toString()
```

The last step is to get the appropriate months based on the Scenario selected. If Actuals is selected, all the elements in the months list from the first to the index of the month in the RTP are selected. If Plan is selected, all the months are included. This can be altered based on needs. Some companies do mid year plans, for example, and may need to only execute the logic on the last 6 months. If Forecast is selected, all

the elements in the months list AFTER the RTP value to the end of the list are selected.

```
// Create dynamic list of months based on user selected Scenario
def useMonths = []
if(operation.grid.pov.find{it.dimName ==='Scenario'}.essbaseMbrName.toString().toLowerCase().contains('plan')){
  // This will create a list of all months - assuming the the plan is for 12 months
  useMonths = months
  }
if(operation.grid.pov.find{it.dimName ==='Scenario'}.essbaseMbrName.toString().toLowerCase().contains('forecast')){
  // This will create a list all months after the actMonth variable - or the out months
  useMonths = months[months.findIndexOf{it == actMonth}+1..-1]
  }
else{
  // This will create a list of all actual months
  useMonths = months[0..months.findIndexOf{it == actMonth}]
  }

// Create delimited list for methods that don't require a List object
def useMonthsString = """\"${useMonths.join('", "')}\""""
```

Using useMonths and useMonthsString

The hard part is over. Using the variable is the easy part. It can be used anywhere the months needs to be change from all months to the identified months.

In a DataMap/SmartPush

```
operation.grid.getSmartPush("GP_SmartPush").execute(["Period": useMonthsString])
```

Same concept but overriding the entire POV (something more realistic in a real world example)

```
// This creates a variable for the povMap that includes all members in the existing POV
// This is used as a possible scenario but is not directly related to the content of this post
def povMap = operation.grid.pov.each{povMbrs["pov$it.dimName"] = "$it.essbaseMbrName"}
// Add the months to a map for the Period dimension
povMap['Period'] = useMonthsString
operation.grid.getSmartPush("GP_SmartPush").execute(povMap)
```

In a GridBuilder

```
builder.addColumn(['Period','Currency'],           [ useMonths,['Local','USD'] ])
```

In an Essbase FIX statement

```
essCalc << """
FIX($useMonthsString)
"""
```

Finally

This could be added to a script and included in all the Business Rules that require such a variable. It creates a variable once that can be used in many classes. It improve maintainability and reduces replicating the same logic for each class. If you have any suggestions, or have something you would like to share, please post a comment.