Adventures in Groovy – Part 33: Mapping Members Between Plan Types

Groovy collections are used all throughout the ePBCS API. If you are not familiar with collections, you may want to take a look at Adventures in Groovy – Part 27: Understanding Collections before you continue. Maps, which are a type of collection, are very useful when moving data between different applications that have different member names representing the same data. In a the example below, data is moving from a product revenue cube to a financial cube. In the detailed cube, the member names are more descriptive, like Net Sales. In the financial application, the same data is a true account number from the GL, and names 42001. Mapping data between these two can easily be done with Groovy maps.

Introduction

There are two components to understanding the use of these maps. First, the map must be defined for use. The construction of the map is a delimited list of items. Each of the items is made up of an key and a value. These are separated by a colon.

```
//set account map
def acctMap = ['Units':'Units',
               '42001-Product Sales':'Net Sales',
               '50001-Cost of Sales':'COGS',
               '50015-Write-offs':'Write-offs',
                 '56010-Chargebacks':'Customer Satisfaction
Discount',
                  '50010-Sales and Discounts':'Sales and
Discounts',
                   '56055-Overstock Discount':'Overstock
Discount',
```

```
                        '56300-Customer Satisfaction
Discount':'Customer Satisfaction Discount',
              '56092-Multi-Purchase Discount':'Multi-Purchase
Discount',
              '56230-Open Box Discount':'Open Box Discount',
                '56200-Damage Container Discount':'Damage
Container Discount',
                '56205-Damaged Box Discount':'Damaged Box
Discount',
              '56090-Group Purchase Discount':'Group Purchase
Discount']
```

The second piece is retrieving the mapped value. The value on the left of the colon is referenced and the value on the right will be returned. The following would return 56230.

```
[acctMap.get("56230-Open Box Discount")]
```

A fully vetted example follows of moving data from one database to several others. The function's use is embedded in a loop, so rather than a hard coded value, the member of the account dimension is used as the accounts (rows in the form) are being iterated. It looks like this.

```
[acctMap.get(it.getMemberName('Account'))]
```

Working Use Case

The map above is used in several places for several reasons. First, the map is created. Second, the map is iterated and the key is used to create a data grid for all the values that will be copied, or synchronized, to the destination cube. Third, the map is used to lookup the converted value to create the grid connected to the destination. this is a complete working example. The items in red are specific to the map and its use.

```
//Dimension                  employeeDim                  =
operation.application.getDimension("Account")

//*******************************************************
```

```
****************
// Data Movement between Apps
//**************************************************************
****************

// Get POV
String                          sCompany                        =
operation.grid.getCellWithMembers().getMemberName("Company")
def                     sMaterialGroup                          =
operation.grid.getCellWithMembers().getMemberName("Material_Gr
oup")
String                          sChannel                        =
operation.grid.getCellWithMembers().getMemberName("Channel")

def lstProducts = []
operation.grid.dataCellIterator({DataCell    cell    ->
cell.edited}).each{
 lstProducts.add(it.getMemberName("Product"))
}

String                      strProducts                         =
"""\"${lstProducts.unique().join('","')}\"""
println "data push running for " + strProducts

if(operation.grid.hasSmartPush("Prod_SmartPush")     &&
lstProducts)
operation.grid.getSmartPush("Prod_SmartPush").execute(["Produc
t":strProducts,"Currency":'"USD","Local"'])

//set account map
def acctMap = ['Units':'Units',
               '42001-Product Sales':'Net Sales',
               '50001-Cost of Sales':'COGS',
               '50015-Write-offs':'Write-offs',
                 '56010-Chargebacks':'Customer Satisfaction
Discount',
                  '50010-Sales and Discounts':'Sales and
Discounts',
                   '56055-Overstock Discount':'Overstock
Discount',
                          '56300-Customer  Satisfaction
```

```
Discount':'Customer Satisfaction Discount',
              '56092-Multi-Purchase Discount':'Multi-Purchase
Discount',
              '56230-Open Box Discount':'Open Box Discount',
                '56200-Damage Container Discount':'Damage
Container Discount',
                '56205-Damaged Box Discount':'Damaged Box
Discount',
              '56090-Group Purchase Discount':'Group Purchase
Discount']


Cube lookupCube = operation.application.getCube("rProd")
DataGridDefinitionBuilder          builder          =
lookupCube.dataGridDefinitionBuilder()
builder.addPov(['Years', 'Scenario', 'Currency', 'Version',
'Company','Store_Type','Department','Source','Product','View']
, [['&v_PlanYear'], ['OEP_Plan'], ['Local'], ['OEP_Working'],
[sCompany],['Store_Type'],['Total_Department'],['Tot_Source'],
['Tot_Product'],['MTD']])
builder.addColumn(['Period'],                              [
['ILvl0Descendants("YearTotal")'] ])
for ( e in acctMap ) {
 builder.addRow(['Account'], [ [e.key] ])
}
DataGridDefinition gridDefinition = builder.build()

// Load the data grid from the lookup cube
DataGrid dataGrid = lookupCube.loadGrid(gridDefinition, false)
def povmbrs = dataGrid.pov
def rowmbrs = dataGrid.rows
def colmbrs = dataGrid.columns
def tmpColMbrs = []

//Fin Grid Setup
Cube finCube = operation.application.getCube("Fin")
Cube rfinCube = operation.application.getCube("rFin")
DataGridBuilder                finGrid                =
finCube.dataGridBuilder("MM/DD/YYYY")
DataGridBuilder               rfinGrid                =
rfinCube.dataGridBuilder("MM/DD/YYYY")
```

```
finGrid.addPov('&v_PlanYear','OEP_Plan','Local','OEP_Working',
sCompany,'Prod_Model')
rfinGrid.addPov('&v_PlanYear','OEP_Plan','Local','OEP_Working'
,sCompany,'Prod_Model','MTD')
def colnames = colmbrs[0]*.essbaseMbrName

String scolmbrs = "'" + colnames.join("', '") + "'"
finGrid.addColumn(colmbrs[0]*.essbaseMbrName as String[])
rfinGrid.addColumn(colmbrs[0]*.essbaseMbrName as String[])

dataGrid.dataCellIterator('Jan').each{ it ->

def sAcct = "${acctMap.get(it.getMemberName('Account'))}"
 def sValues = []
 List addcells = new ArrayList()
 colmbrs[0].each{cName ->
 sValues.add(it.crossDimCell(cName.essbaseMbrName).data)
 addcells << it.crossDimCell(cName.essbaseMbrName).data
 }

finGrid.addRow([acctMap.get(it.getMemberName('Account'))],addc
ells)
rfinGrid.addRow([acctMap.get(it.getMemberName('Account'))],add
cells)
}
DataGridBuilder.Status status = new DataGridBuilder.Status()
DataGridBuilder.Status rstatus = new DataGridBuilder.Status()
DataGrid grid = finGrid.build(status)
DataGrid rgrid = rfinGrid.build(rstatus)

finCube.saveGrid(grid)
rfinCube.saveGrid(rgrid)
```

Finishing Up

This is a relatively simple concept and not terribly difficult to implement. It is also something most will benefit from when synchronizing data with the dataGridBuilder. Have something to add? Post a comment and I will get back to you promptly.