Adventures in Groovy – Part 8: Customizing Data Maps and Smart Pushes

Introduction

Groovy has the ability to execute Data Maps and Smart Pushes. Data Maps are objects that define the data movement between applications and are accessible globally. Smart Pushes are Data Maps that are attached to a Data Form and can override the dimensions settings. They are largely the same thing, but defined separately inside of PBCS.

So, why execute these in Groovy rather than assign to a form and call it a day?

1. The data push can be customized to push only data that has changed, improving performance.
2. When a form is saved, all calculations are executed first, and the Smart pushes are executed after all calculations are finished. If Groovy is used and the data push is done inside a Business Rule, the order of operation can be a
    1. calculation
    2. data push
    3. calculation
    4. data push
    5. etc
3. Since the Smart Push has a global limit of memory that can be used, and multiple people are running them through form saves, it is critical to make it as small as possible to reduce the probability of hitting that limit and increasing the odds of the Smart Pushes failing.

To date, I see the advantage of running a Smart Push (a Data Map on a form) in that most of the dimensions are already overridden and less overrides are required in the Groovy logic. There is no difference in performance or the size of what can be pushed between the two when executed from a Groovy calculation. The advantage of using a generic Data Map is that there is less effort in defining the form level Smart Pushes, and once one Groovy calculation is written to override all dimensions from a Data Map, it can be replicated easily to all required forms.

To understand the memory issues and explanation of how it differs from Data Maps and Smart Pushes, see PBCS Data Map / Smart Push Has Data volume Limits.

Data Map

Executing a Data Map is very simple and can be done in one line of code.

```
operation.application.getDataMap("Data Map Name").execute(true)
```

Calling execute() on a DataMap would execute the named Data Map (with no customization) and clearing the target location before pushing data. Changing the true to false, or removing it, would remove the clear and leave the data as is.

To override the the dimension settings and increase or decrease the scope of what is used, the member names need to be further defined. Every dimension's scope can be changed, or just the ones that require a change in scope can be altered. The following changes the scope for the account, scenario, version, and Entity dimensions.

```
operation.application.getDataMap("Data Map Name").execute(["Account":"Net Income, Income, Expense", "Scenario":"Budget", "Version":"Working", "Entity":"East Region"], true)
```

Smart Push

The Smart Push works exactly the same, except the object referenced is a Smart Push, and is obtained through the grid object, not the application. Again, the likelihood that the Smart Push is further scoped is high, so it is reasonable that the dimensional parameters would be fewer as many of them change based on the POV selected.

```
operation.grid.getSmartPush("Smart Push Name").execute(["Account":"Min Bonus, Min Salary"])
```

One additional option is to define a Smart Push from a Data Map in the Groovy script.

```
operation.application.getDataMap("Data Map Name").createSmartPush().execute(["Account":"Min Bonus, Min Salary"])
```

Error Trapping

When these are written, it is likely that the Smart Pushes and Data Maps exist. One extra step to ensure the calculation doesn’t fail is to verify their existence. For Smart Pushes, verify that it is attached to the form.

```
//Data Map Example
if(operation.application.hasDataMap("Data Map Name"))
       operation.application.getDataMap("Data Map Name").execute(true)
//Smart Push Example
if(operation.grid.hasSmartPush("Smart Push Name"))
  operation.grid.getSmartPush("Smart Push Name").execute()
```

Conclusion

Creating variables to use in these functions to change the scope to only the rows and columns that have been edited, the calculation would look like this. This is where I see the biggest bang for your buck. To understand more about using

the grid iterator, read Adventures in Groovy Part 3: Acting On Edited Cells. When a grid has hundreds of rows, only pushing the data that has been edited can make a huge difference in performance.

```
// Setup the variables to store the list of edited vendors and periods
def lstVendors = []
def lstPeriods = []
// Iterate through only the cells that were changed and create a list of changed vendors and periods
operation.grid.dataCellIterator({DataCell cell -> cell.edited}).each{
 lstVendors.add(it.getMemberName("Vendor"))
 lstPeriods.add(it.getMemberName("Period"))
}
// Convert the lists to a comma delimited string with quotes surrounding the members
String strVendors = """\"${lstVendors.unique().join('","')}\""""
String strPeriods = """\"${lstPeriods.unique().join('","')}\""""
// Execute the Smart Push with a change in scope to Vendor and Period
if(operation.grid.hasSmartPush("GP_SmartPush") && lstVendors)
operation.grid.getSmartPush("GP_SmartPush").execute(["Vendor":strVendors,"Period":strPeriods]
```

With any luck, you will see significant improvement, unless they change every month of every row!