Adventures in Groovy – Part 40: Eliminating Data Sources With The Groovy Calendar Class

I am currently working on a migration of on-premise to cloud project (going GREAT by the way). One of the things we are working on is the change with the data integration. One of the processes loads the number of working days to the application from a source that provides it. "Why not use Groovy," I ask? It turns out to be a great question.

There are two concepts to cover. A few lines of Groovy using a calendar class and the gridbuilder to save the data. This could also be a dynamically generated calculation that updates the data. Since this is used on an ASO cube, the example below uses the gridbuilder option.

The Code Explained

This example will have a run time prompt for year, will calculate the values for each month in the year selected and store them to the appropriate account. The prompt and the variables required are declared here.

```
/*RTPS {rtpYear} */
int iYear = rtps.rtpYear.getEnteredValue().toString().substring(2).toInteger() + 2000
def sYear = rtps.rtpYear.getEnteredValue().toString()
Def values = []
Calendar calendar = GregorianCalendar.instance
```

Once this is setup, the calendar class will be used to identify the working days. Working days here is defined as any weekday in the month. These days are used to calculate payroll accruals and monthly averages. Holidays are not considered in the calculations that use this.

The following uses the calendar class to create a list variable that will be passed to the gridbuilder later in the script.

```
for (int currentMonth = 1; currentMonth <= 12; currentMonth++)
{
   Calendar startCal = new GregorianCalendar(iYear,
currentMonth,
calendar.getActualMinimum(GregorianCalendar.DAY_OF_MONTH))
  Calendar endCal = new GregorianCalendar(iYear, currentMonth,
calendar.getActualMaximum(GregorianCalendar.DAY_OF_MONTH))
  int workDays = 0
  startCal.upto(endCal) {Calendar it ->
    if( (it[Calendar.DAY_OF_WEEK]).toString().toInteger() > 1
&& it[Calendar.DAY_OF_WEEK].toString().toInteger() <7 )
       workDays += 1
     }
    def sMonth = Date.parse( 'MM', "$currentMonth" ).format(
'MMMM' )
   values << workDays
   println "for the month of $sMonth we have $workDays working
days."
}
```

At this point, a list object has 12 values, one for each month. This will be loaded to a specific POV that won’t change. If the requirement was more dynamic, this example could certainly be expanded to account for it. The last step is to store the data back to the database. There are many examples on the gribuilder in previous articles, so it won’t be explained in detail.

```
Cube cube = operation.application.getCube("Plan1")
DataGridBuilder builder =
cube.getDataGridBuilder("MM/DD/YYYY")
builder.addPov($sYear, 'Local', 'Working', 'Plan', 'No
Entity')
builder.addColumn('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
'Jul', 'Aug’, 'Sep', 'Oct', 'Nov', 'Dec’ )
builder.addRow(['Working Days'], values )
DataGridBuilder.Status status = new DataGridBuilder.Status()
builder.build(status).withCloseable { grid ->
      println("Total number of cells accepted: $status.numAcceptedCells")
      println("Total number of cells rejected: $status.numRejectedCells")
  println("First 100 rejected cells: $status.cellsRejected")
  // Save the data to the cube
  cube.saveGrid(grid)
 }
```

The Results

The log will provide some descent information. This can be expanded for POV, user info, the time it took to process, but as it is, this is what the log would produce. Remember, the security of the user that runs this is used. If this is for a forecast, for example, and the start month is April, the accepted cells would only be nine. 3 cells would be rejected because they are not writable.

```
for the month of January we have 21 working days.
for the month of February we have 21 working days.
for the month of March we have 22 working days.
for the month of April we have 22 working days.
for the month of May we have 20 working days.
for the month of June we have 22 working days.
for the month of July we have 22 working days.
for the month of August we have 21 working days.
for the month of September we have 22 working days.
for the month of October we have 21 working days.
for the month of November we have 21 working days.
for the month of December we have 22 working days.

Total number of cells accepted: 12
Total number of cells rejected: 0
```

The Calendar Class

Although not required in this example, there are all kinds of things that this can be used for. Have you ever needed to calculate the week of the year? The day of the week? How about the days between two dates? You likely have come across these things for WFP or CapEx at least. The calendar object itself has a ton of useful cases and the object that is returned is basically an array of information. If the object is sent to the log with a println, all the values are exposed. Of course, you can always google it, but it looks like this.

```
java.util.GregorianCalendar[time=-62130585600000,areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="UTC",offset=0,dstSavings=0,useDaylight=false,transitions=0,lastRule=null],firstDayOfWeek=1,minimalDaysInFirstWeek=1,ERA=1,YEAR=1,MONTH=2,WEEK_OF_YEAR=10,WEEK_OF_MONTH=1,DAY_OF_MONTH=2,DAY_OF_YEAR=61,DAY_OF_WEEK=4,DAY_OF_WEEK_IN_MONTH=1,AM_PM=0,HOUR=0,HOUR_OF_DAY=0,MINUTE=0,SECOND=0,MILLISECOND=0,ZONE_OFFSET=0,DST_OFFSET=0]
```

It would take pages to explain and provide examples of all that can be done, which won’t be in this post. Here are some ideas of uses.

- use copyWith to duplicate an instance
- use the format method to format the date as a month, year, long date, etc.
- use minus to subract two calendar dates from each other (datediff)
- use next and previous to increment days, months, or years
- use set to set a specific date

Last Call

Will this change the world? No. Is it useful? To some, absolutely. The goal here is to just provide some information on how to use the calendar class and give you some ideas of what it could be used for. Manipulating dates and time has always been a challenge in Essbase.   It is better now with some of the custom functions that have been exposed (and thank goodness they did this on the cloud or we wouldn’t be able to register custom functions). But, performance, complexity, rolling through the same blocks multiple times, all can be minimized with the ability to calculate this outside of Essbase, pass it to a calculation, let Groovy do what it is good at, and let Essbase handle what its strengths are.

To read about more uses, Google “Groovy Calendar” and take a look at all the methods it has – pretty useful stuff.

You have any other thoughts? Post a comment. You know we would love to hear from you.