Featherman’s Data Management Adventures: Introducing Summary Arrays to Replace Global Running Variables.
*The classic Insert and Update Approach to keeping data current is presented using a Construction Project Management*

Arrays (the Visual Studio object is called data table) are used here in two ways,

|  |  |
| --- | --- |
| **Using Arrays Scenario #1: Transaction Data***1. To store detailed records of business transactions – here individual employee payroll records (figure 1).**2. To store compiled and summarized data that allows a managerial analysis of business performance, organized by a key dimension. Here payroll expenditures are grouped by labor grade (figure 2).*For scenario #1 here are the steps to creating an using the array:1. Create public shared array in memory
2. In Page\_Init procedure add the columns. Specify the data type for each column and set the first column to autonumber. Use the string datatype to store dates so that they are formatted nicely.
3. In your button\_click procedure the idea is save the values from the webpage’s controls into a row of data that is added to the array. So create a new blank row.
4. After performing the business calculations assign the values from the controls of the webpage into the columns of the new row. Here the rows record aspects of one labor record such as # of hours worked, etc.
5. Add the row of data to the array
6. Refresh the display of the gridview so that the new row of data is visible.
 |  |

|  |  |
| --- | --- |
| **Using Arrays Scenario #2: Summary data**The *main difference is that you have to add BOTH the columns and rows.* Use these steps.**To create the summary array using these 7 steps** 'step 1 create the array datatable public sharedIn the Page\_init procedure do the next 5 steps'step 2 add columns 'step 3 set default values = 0 for running totals (this is a property of the column)steps 4-6 can be accomplished with a ‘brute force method or loop'step 4 create new blank rows to store summary data'step 5 give a value to the first column (optional) - here we store the labor category'step 6 add the new rows to the array'step 7 add a gridview to the webpage to display the array. (you should see zero's in the numeric columns when you start the program) |  **More Info**1. Create the public shared array in memory
2. In Page\_Init procedure add the columns for the array. Specify the data type for each column. For metrics columns that are percentages, use the datatype string so that the column can be formatted nicely. Use decimal for all columns that are running totals (get incremented).
3. In Page\_Init set any column that keeps running totals to have a default value of 0. In the first table above, columns 2,4,5 are default =0 .
4. In Page\_Init add the necessary number of rows to the array so that it is ready for data entry. When adding the rows to store the summary compiled data, add the values for the dimension column. Here the first column is labor grade, which defines the usage of the array to be summarizing labor costs by labor grade. The choice of which dimension to use to aggregate data is based on system requirements. In the scenario to the right four rows were added to the first array and ine row was added to the second array. There was no need to specify a value for the first column for the second array.
5. **The Classic Insert Then Update Approach:** In the button click event that calculates the new labor record. First you add the transaction row to the scenario 1 array, then you update the running totals in the two summary tables. For example update the # hours worked by electricians, and the cost when a shift for electricians is recorded. The grandtotals array is also updated regardless of labor category specified. This is the classic ‘insert and update’ procedure.
6. Here other columns of analytics are introduced. The program uses a loop to calculate % of total for each labor grade
 |

**Differences in creating and using arrays – scenario 1 vs scenario 2**

|  |  |  |
| --- | --- | --- |
|  | Scenario 1 – saving new rows of transaction data | Scenario 2 – updating summary data |
| When to create columns? | Page\_init | Page\_init |
| When to add rows? | Button-click procedure | Page\_init |
| Need to set the default value of columns to zero to enable running totals | No | yes |
| Conceptual - # of rows | many | Few |
| Helps to make the first column auton-number | yes | no |

This project forms the basis of your final project. After you create this program and learn about arrays, its time to prototype your final project, which means think about the business process being performed, and the transactions being recorded. IF you set up the arrays needed for the insert and update data processing needed, you will have a great start to your final project.



'This program hopefully extends the purpose of working with calendars and performing calculations with timespan. The start of a labor timekeeping system is kept here to be used as an idea generator for a project management type of project. This project keeps track of the labor cost for different types of labor grades (e.g., electricians, plumbers, etc.). You could extend this webpage to keep track of multiple projects – you could monitor % completion of project as compared to % of $ spent on the project. The webpage then could be used as a cost control tool to keep projects on-budget).

'This is an upgraded version of the prior version. This version uses three datatables

'1. To record each timesheet entry for one employee's shift of work

'2. To record running totals (current labor on a project) and metrics for each of 4 labor grades (electricians etc.)

'3. To record grand total for all labor grades combined.

Imports system.data

Partial Class PayrollwithTimespan2

 Inherits System.Web.UI.Page

 'notice we do not need a stack of global variables, the saved values are stored into these multi-column, multi-row arrays, it’s much easier and the code is more parsimonious. On a more important note, your learning of arrays and some ADO.NET objects (datatable, rows, columns, gridviews) prepares you to make the next jump to interracting your webpage with a SQL Server database.

 Public Shared gdtPayrollRecords, gdtLaborGradeTotals, gdtGrandTotals As New DataTable

#Region "Save labor record"

 'here is another start of a timecard keeping system, an accountant can perhaps complete it for a final project.

 Protected Sub Button1\_Click(sender As Object, e As EventArgs) Handles Button1.Click

 Dim decgrosspay, decPayrate As Decimal

'next the blank row is added to the bottom of the datatable that records each shift worked.You perform the calculations as before, and add the new functionality of storing values from the controls into the columns of the new datarow that we have created.

 Dim dr As DataRow = gdtPayrollRecords.NewRow

 'now some error checking

 If txtStart.Text = Nothing OrElse txtEnd.Text = Nothing OrElse txtDate.Text = Nothing Then

 txtOutput.Text = "Check entries for time and date"

 Exit Sub

 End If

 If rblLaborgrades.SelectedIndex = -1 Then

 txtOutput.Text = "Select a labor type"

 Exit Sub

 Else 'if a labor grade was selected then assign it to a column in the new row

 dr("LaborCategory") = rblLaborgrades.SelectedItem.Text

 End If

 If rblHRCategory.SelectedIndex = -1 Then

 txtOutput.Text = "Select an HR category"

 Exit Sub

 Else 'if a HR category was selected then assign it to a column in the new row

 dr("HRCategory") = rblHRCategory.SelectedItem.Text

 End If

 'here we use the newer textboxes that are rendered as calendars and have a time input. This next line is used to format a nice date for the output. We assign the date to a column (string) of the new row. We use a string because we will not do any calculations or metrics with the date (that's another day). Here using a string allows better formatting. The datetime.parse pulls the date out of the textbox, the toshortdatestring reformats the data nicely and also turns the date into a string.

 dr("LaborDate") = DateTime.Parse(txtDate.Text).ToShortDateString

 'In accordance with federal law in the USA the construction firm pays overtime. Hours worked on Saturday earn 1.5 times the base rate. Sunday labor hours are recorded as double time. PS. Managers should keep an eye out for employees that habitually work Sunday then call in sick Monday or Tuesday.

 'The Weekday function is an example of a very useful built-in time/date function (similar to DATEPART in SQL). Here we calculate the payrate depending on day of week. We also write a value to a column of the new row of data.

 Select Case Weekday(DateTime.Parse(txtDate.Text))

 Case 1 'sunday

 decPayrate = (rblLaborgrades.SelectedValue \* 2)

 dr("Day") = "SunnyDay"

 Case 2 To 6 'weekday

 decPayrate = rblLaborgrades.SelectedValue

 dr("Day") = "Weekday"

 Case 7 'saturday

 decPayrate = (rblLaborgrades.SelectedValue \* 1.5)

 dr("Day") = "Saturnday"

 End Select

 dr("Payrate") = decPayrate 'place the value that was just calculated into a column of the new row. Notice its better to add this line outside the select case, so you don’t have to write it three times.

 'Review: Timespan is an amazingly useful datatype that can be instantiated like any other variable. Timespan is used to hold the result of some date comparison. Such as the number of hours (or days) difference between two times (or dates). Here we create an instance of timespan with the name ts. We pull the date out of the textbox using datetime.parse(textbox.text). Once you have the results of the calculation in a timespan variable then you can format the result into # minutes, weeks, months, days, etc. very easily. ts.totalhours can output the number of hours with partial hours included (ie 8.5 hours, not rounded up or down).

 Dim ts As TimeSpan

 ts = DateTime.Parse(txtEnd.Text) - DateTime.Parse(txtStart.Text)

 dr("Hours") = ts.TotalHours

 'using timespan makes this next calculation eezie peezie. Multiply the number of hours worked times the labor rate, and store the value into a column of the datarow so that it can later be saved into the transactions table (array).

 decgrosspay = ts.TotalHours \* decPayrate

 dr("GrossPay") = decgrosspay

 'the construction business employees part-time, full-time and 1099 (seasonal) workers. HR and project managers when bidding jobs use the metric to add 40% to labor recorded to cover medical/dental/retirement/doggie daycare/spa benefits for full-time workers. HR provides a modest benefit for the 1099 workers (sub-contractors). 1099 workers can access medical benefits at a local clinic. To cover this expense they add a 15% to the gross pay calculation for the shift. The gross pay plus additional 15% or 40% create the 'fully burdened' cost for the shift. This 'fully burdened cost' is used by project managers and accountants to cost out projects. Part time workers do not receive medical benefits.

 Select Case rblHRCategory.SelectedIndex

 Case 0 'part-time employee

 dr("FullyBurdoned") = dr("GrossPay")

 Case 1 ' full-time employee

 dr("FullyBurdoned") = dr("GrossPay") \* 1.4

 Case 2 '1099 employee

 dr("FullyBurdoned") = dr("GrossPay") \* 1.15

 End Select

 'now that the columns of the new row are populated with values, we can now add that row to the datatable, and display the data

 gdtPayrollRecords.Rows.Add(dr)

 GridView2.DataSource = gdtPayrollRecords

 GridView2.DataBind()

' would you rather work with tables, columns and rows or long concatenated strings?

'ok now we update the two summary tables that you see on the left. IN the first array one row of the summary table is updated, the row that corresponds to the item selected in the labor grades radiobuttonlist. The += code lets you know the columns on the left of the = sigh are being incremented (increased) by the values pulled from the form. First we update the array that records labor by category, then we update the array with the grand totals.

‘Using a radiobutton list is preferred to record the selection of labor grade because the 0-indexed item in the radiobutton list effortlessly corresponds to the 0-indexed row of the array. Here electrician is the 0-index of both the radiobutton list and array.

‘You refer to columns using the syntax
‘tablename.rows(row #).item(“column name”)

 With gdtLaborGradeTotals.Rows(rblLaborgrades.SelectedIndex)

 .Item("#Shifts") += 1

 .Item("TotalHours") += ts.TotalHours

 .Item("TotalCost") += dr("FullyBurdoned") ‘the fully burdoned cost of the labor includes medical/dental program costs

 End With

 'here we update the grand totals table. We have a table with 1 row in it, which is referred to as row 0.

 With gdtGrandTotals.Rows(0)

 .Item("#Shifts") += 1

 .Item("TotalHours") += ts.TotalHours

 .Item("TotalCost") += dr("FullyBurdoned")

 End With

'now to implement a metric for the labor grades table. We want to know the labor % of whole each labor grade has spent. For example the current project may be 40% framers and electricians at an early part of the project. This is an example of calculating a metric for the entire table, based on some value in some other table. Generating useful analytics for your project greatly add to its usefulness. Metrics also server as the datasource for charts. You must add columns of metrics to your projects, to finish the job, score high marks, and turn this into a business programming class. This is the value added activity for business programmers.

 For Each drow As DataRow In gdtLaborGradeTotals.Rows

 drow("%Total") = FormatPercent(drow("TotalCost") / gdtGrandTotals.Rows(0).Item("TotalCost"))

 Next

 'above we loop each row of the labor grades array and calculate the % of total cost for the labor grade. Notice we can use a column from another array in the calculation, the grand totals array has the TotalCost field for all the labor so why not use that value? Now we finish this calculatin’ by displaying the labor grades table totals on view #2. Remember the data is in the datattable and the gridview is the object that only displays the datatable, and offers no further funcitonality.

 GridView1.DataSource = gdtLaborGradeTotals

 GridView1.DataBind()

 'finish by updating the display of the grand totals table...would you rather have 3 global variables or 1 row in a table with 3 columns?

 GridView3.DataSource = gdtGrandTotals

 GridView3.DataBind()

 End Sub

#End Region

#Region "Load Datatable"

 Private Sub PayrollwithTimespan\_Init(sender As Object, e As EventArgs) Handles Me.Init

 'all the prior functionality is great and useful, but it work because the array data structures are already created. Here we create the arrays in the page\_init procedure which runs once when the webpage is started. We start by ensuring a view is not shown yet when the program is started.

 MultiView1.ActiveViewIndex = -1

 'let’s make sure we do not add the same columns to the same tables more than once (system would crash). Theoretically the .init event should execute just once when the webpage first starts up. Well its necessary to add this safeguard, especially use when developing the project and pressing refresh on your webpage, or adding a new tab to the webpage you already have open (webpages have amazing memory). When developing webpages with arrays and data connections its best to test changes in a new webpage not refreshing the current.

 If gdtLaborGradeTotals.Columns.Count > 0 OrElse gdtPayrollRecords.Columns.Count > 0 Then Exit Sub

 With gdtPayrollRecords 'add columns to store the data records for individual labor records

 .Columns.Add("LaborID", GetType(Integer))

 .Columns.Add("LaborDate", GetType(String))

 .Columns.Add("Day", GetType(String))

 .Columns.Add("LaborCategory", GetType(String))

 .Columns.Add("HRCategory", GetType(String))

 .Columns.Add("Hours", GetType(Decimal))

 .Columns.Add("Payrate", GetType(Decimal))

 .Columns.Add("GrossPay", GetType(Decimal))

 .Columns.Add("FullyBurdoned", GetType(Decimal))

 End With

 'to simplify data entry we make the first column autonumber. Each new labor record recieves its own sequenced id#. The seed value starts at 1 and the stp value specifies to increment each additional record’s ID# by 1.

 With gdtPayrollRecords.Columns("LaborID")

 .AutoIncrement = True

 .AutoIncrementSeed = 1

 .AutoIncrementStep = 1

 End With

 'this is the table that holds the grand total - we create 3 columns then set their initial value to 0.

 With gdtGrandTotals

 .Columns.Add("#Shifts", GetType(Decimal))

 .Columns.Add("TotalHours", GetType(Decimal))

 .Columns.Add("TotalCost", GetType(Decimal))

 .Columns("TotalHours").DefaultValue = 0

 .Columns("#Shifts").DefaultValue = 0

 .Columns("TotalCost").DefaultValue = 0

 End With

 'When you create a numeric column the default value is NULL. NULL + 10 = Crash the computer

 ' but 0 + 10 = 10

 'tablename.rows(0).item("TotalCosts") = 10

 'tablename.rows(0).item("TotalCosts") += 10

 ‘we need to add a new row to the table, so that it is ready for data entry. You need a row to store data.

 Dim dr As DataRow = gdtGrandTotals.NewRow

 gdtGrandTotals.Rows.Add(dr)

 GridView3.DataSource = gdtGrandTotals ‘used to see if the array is created and available

 GridView3.DataBind()

 'Now we add the columns to the labor grade summary table and set some default values. If calculations are going to be made in columns (such as running totals) then the values need to start at 0. When a new row is created its numeric fields are set to NULL (empty). We can't add a number to a NULL (empty) value, so for those columns that we will use to keep running totals, we need to set their initial values to 0. Notice the %Total field is a string so that it can be formatted nicely.

With gdtLaborGradeTotals

 .Columns.Add("LaborGrade", GetType(String))

 .Columns.Add("#Shifts", GetType(Integer))

 .Columns.Add("%Total", GetType(String)) 'not a running total so default value of 0 not needed

 .Columns.Add("TotalHours", GetType(Decimal))

 .Columns.Add("TotalCost", GetType(Decimal))

 .Columns("#Shifts").DefaultValue = 0

 .Columns("TotalHours").DefaultValue = 0

 .Columns("TotalCost").DefaultValue = 0

 End With

 Next we need to add 4 rows to the laborgrade totals table to get them ready for data editing/updates to keep the running totals by labor category (which is the business requirement). We need to ensure this code only runs once so look at this next code that will stop the procedure if the rows were already created/added to the table.

 If gdtLaborGradeTotals.Rows.Count > 0 Then

 Response.Write("Already have" & gdtLaborGradeTotals.Rows.Count & " records")

 Exit Sub

 End If

'We could write this code and add repeat it once for each labor category (using dr1, dr2 for Framers, dr2 for PLumbers, etc)

Dim dr1 as datarow = gdtLaborGradeTotals.NewRow
dr1.item("LaborGrade") = “Electrician”
gdtLaborGradeTotals.Rows.Add(dr1)

 'Rather than that hard coding we can loop the control that has the labor grades and insert those values into the first colunm of new row added to the array. We need to create,'initialize' and add four rows of data to allow later data entry and updates (ie. total labor for each plumbers, electricians, etc.). A radiobutton list exists for the labor grades, so we loop the items list of that control, placing the values into the first column of the new row. Because the newrow is added within a loop it can be re-created over and over.

 For Each li As ListItem In rblLaborgrades.Items

 Dim dr2 As DataRow = gdtLaborGradeTotals.NewRow

 dr2.Item("LaborGrade") = li.Text

 gdtLaborGradeTotals.Rows.Add(dr2)

 Next

 'should now see tables on view #2 - with rows of data ready to be updated

 GridView1.DataSource = gdtLaborGradeTotals

 GridView1.DataBind()

 End Sub

#End Region

#Region "Utilities"

 'switch between views

 Protected Sub LinkButton1\_Click(sender As Object, e As EventArgs) Handles LinkButton1.Click

 MultiView1.ActiveViewIndex = 0

 End Sub

 Protected Sub LinkButton2\_Click(sender As Object, e As EventArgs) Handles LinkButton2.Click

 MultiView1.ActiveViewIndex = 1

 End Sub

 #End Region

End Class