Paper AD10

Leveraging SAS® Portal: Developing an Executive Dashboard with Customizable Data Source

Elibeth R. Carpenter, Comsys, Kalamazoo MI

ABSTRACT

Data is one of the biggest assets of corporations, and the ability to analyze it drives corporate goals, strategies and results. One of the latest trends in business intelligence is the use of Key Performance Indicators (KPI) to measure corporate performance, in an easy to read GUI. SAS provides a great deal of tools to generate this kind of interface; the question is how to create a flexible, attractive and meaningful Dashboard integrated into the SAS Portal.

This paper will step through the implementation of an Executive Dashboard as a Portlet Template (or Editable Portlet) that summarizes data and represents it with SAS Critical Success Factor graphs, using the SAS Java Connection Factory and the SAS Rangeview applet. Additionally, this Portlet illustrates the key benefit of SAS Editable Portlets by allowing the users to choose the data they want to diagram. They will be able to run stored processes and generate or modify the source data, all at the tips of their fingertips.

INTRODUCTION

SAS is widely known for providing useful libraries that greatly assist the programmer or developer in the generation of data representation with graphs and tables. At the same time, we have the SAS Portal that is a great leveraging tool in corporate business processes. Consider this: Wouldn't it be a great idea to combine these two resources and create an attractive Executive Dashboard integrated into the SAS Portal?

Dashboards seem to be under the spotlight in the area of Business Intelligence and Data Visualization. Designing a Dashboard for executives that accomplishes the mission of presenting a great deal of information in a limited amount of space and that at the same time is easy to read can be quite a challenge. It comes down to choosing the right layout, colors, but most importantly, the right graphs and diagrams. SAS comes to our rescue providing us an extensive amount of graphic tools to generate this kind of interfaces; one of them is the attractive SAS Critical Success Factor diagram that is generated by the DS2CSF macro. In this paper we will explore this diagram in particular.

Once we have nailed down a good design for our Executive Dashboard, a second challenge arises when we talk about integrating it with the SAS Portal. Something we know about SAS Portlets is that they require JSP and HTML fragments rather than complete pages, but the DS2CSF macro embeds the graphs into complete HTML pages. Now the question is: how can we integrate a Dashboard using SAS CSF diagram into the SAS Portal? Read on for the answer to this question.

For purposes of the example presented in this paper, you will first find an overview of the Executive Dashboard Portlet and the mechanics of it, and then will go through the implementation in four logical layers:

Layer 1 - Data Source: Summarize the data source into a SAS dataset that can be represented using a CSF diagram.
Layer 2 - Backend Connectivity: Establish a connection to the SAS Metadata Server to get a workspace connection to the SAS server.
Layer 3 - Middleware Architecture: Build the Editable Portlet which is comprised by a set of Java action servlets and utility java classes.
Layer 4 - Graphical User Interface: Present the data in a summary table as a list of performance indicators with actuals and goals, and represent each of one of them using the CSF diagram.

This paper does not intend to give an exhaustive presentation on the Java implementation of the Portlet, but rather to explain the elements necessary to use Critical Success Factors diagrams in the Portlet, and execute stored processes from an Editable Portlet.

EXECUTIVE DASHBOARD PORTLET OVERVIEW

Initially, the Executive Dashboard Portlet Template loads using the default information provided through a resource bundle file. This file provides the name of the initial source dataset, server connectivity information, and the repository or directory in the server where the stored processes to use will be stored. In the example used for illustration, Figure 1 shows the Portlet's appearance after loading the default data. The source data is created according to the guidelines previously discussed, and is represented using SAS CSF as a Rangeview applet. Given an indicator and a budget or target for the former, the diagram provides a simple means of displaying the percentage achieved for each indicator, based on the range 0 - 1.

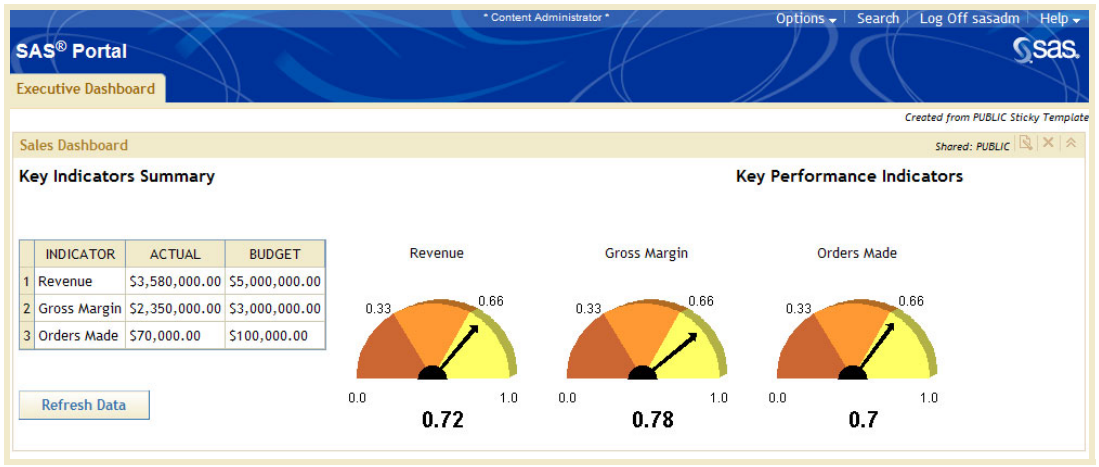


Figure 1 - Default data initially loaded

The "Refresh Data" button depicted in Figure 2 serves to load the data directly from the data source, and re-runs the stored process used for it, if any. In this particular example, there was no stored process run to generate the source dataset.

This Portlet has been developed as a Portlet Template. Hence, the data source which is used to represent the KPI's can be modified. Figure 2 shows the options that are given to the user in the edit page.

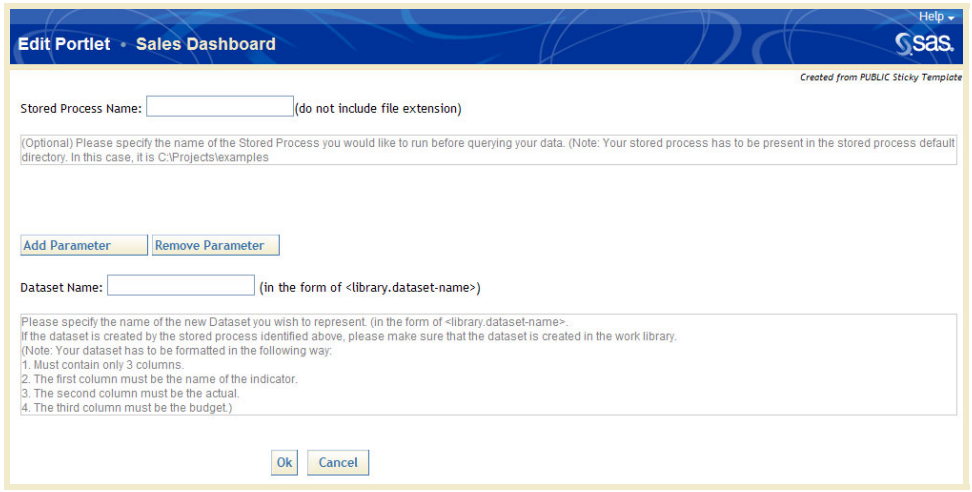


Figure 2 - Portlet editor page

As noticed in Figure 2, the user is given the capability to run a stored process, in which case he will need to indicate the file name containing it. The purpose of running a stored process is to be able to create or modify

programmatically the source dataset to use directly from the GUI. This file should be saved in the directory indicated by the resource bundle file when the Portlet is deployed. This will be explained more in details in the next section.

As illustrated in Figure 3, the user can add parameters passing name and its value, if the stored process requires them.

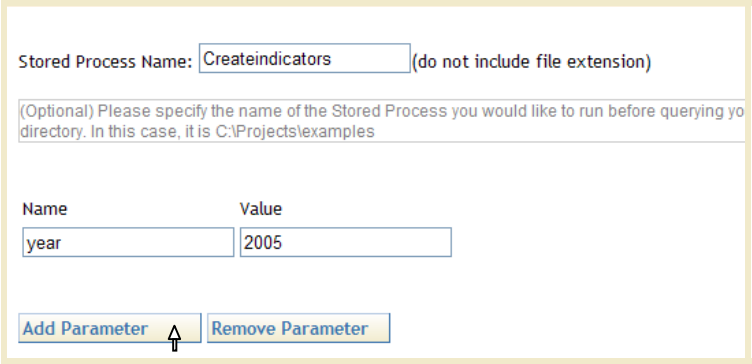


Figure 3 - Passing parameters to the stored process

In our example, we indicate the stored process contained in the file "CreateIndicators.sas", and pass three different parameters to it: year, quarter and table name. The latter is the name of the final dataset that will be created with the indicators. These parameters are particular to the stored process used in this example, so the number and names of parameters will depend on each stored process' requirements. CreateIndicators summarizes data and creates a final dataset which will then be used to represent in the Portlet. The last parameter "tablename" stands for the name of the final dataset to create. This name will be the same name provided in the "Dataset Name" field. In this field the user indicates the name of the dataset that contains the indicators to diagram. Figure 4 shows how the user would run the stored process specified in Appendix A, and assign the generated dataset as the Portlet data source.

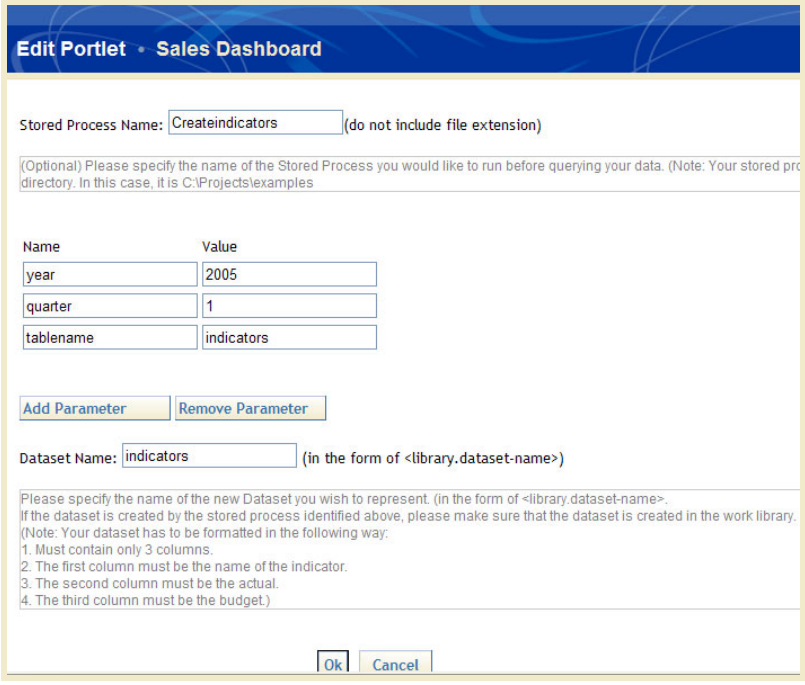


Figure 4 - Running the "CreateIndicators" stored process and assigning the resulting dataset as the Portlet's data source.

Once this information is fed into the Portlet, it runs the stored process with the indicated parameters, generates the indicators dataset, and loads the new data and updates the CSF diagrams, as shown in Figure 5.

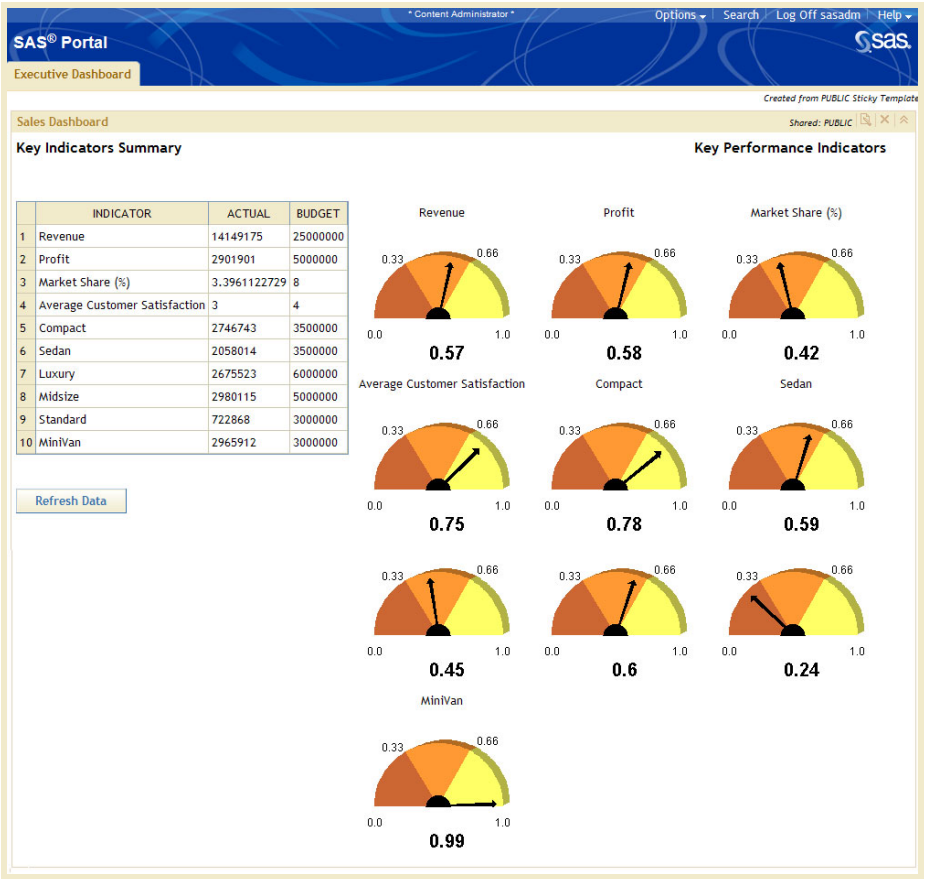


Figure 5 - Automotive Sales Dashboard (after editing the Portlet's instance)

This design allows the sales executive's data needs to change without requiring a deployment of a new Portlet or changing the implementation of the existing one. A SAS programmer that knows how to find the data that the sales executive is requesting can write a SAS stored process to summarize it in the right format, dump the source file in the Portlet default directory and give instructions to the executive on how to edit his Portlet's instance. Everything is done without too much hassle: neither on the programmer's nor on the executive's side.

LAYER 1 – DATA SOURCE: Getting the Indicators Dataset Ready

There is always a price to pay for flexibility, and the Portlet implementation presented in this paper is no exception to the rule. In order to make the Portlet able to feed off any data source, we need to create some guidelines in how this data is to be formatted. In this way, data in any kind of format and distributed in any number of datasets can be always merged into one single uniform dataset. This is one of the most powerful advantages of creating the Executive Dashboard as an Editable Portlet since virtually any data can be fed into the Dashboard as long as there is a stored process that will summarize it in the right format. Therefore, the final dataset that will be used as source for our diagrams has to have the variables "Indicator", "Actual", and "Goal". Each of the mentioned variables is explained in this section.

The first step is to identify the variable “Indicator”, which stands for the Key Performance Indicators (KPI) that we want to represent in the dashboard. Examples of these are profit, revenue, total orders, new customers, etc. These indicators are particular to each business nature and needs, and are meant to reflect the performance of the corporation over a period of time, in our case, quarterly. Our example is based on an automotive company that has chosen as Key Performance Indicators the following:

- Revenue
- Profit
- Market Share (%)
- Average Customer Satisfaction

And study the sales performance of their vehicle types:

- Compact
- Sedan
- Luxury
- Midsize
- Standard
- Minivan

The second step is to generate the variables “Actual” and “Goal” (or “Budget”), which represent for each indicator the actual number or sales achieved and the estimated goal that had been set for the quarter, respectively. Therefore, we need to pinpoint the SAS datasets that are going to be used to generate this kind of information. Generally, a dataset that reflects sales and dates is a good start to generate currency based indicators such as revenue and profit. Other datasets will be needed to produce actuals and budgets for numeric based indicators such as new customers, total orders, market share, etc.

After the indicators and the datasets needed to generate the information have been identified, a third step is to create a stored process that will create the final indicators dataset that summarizes and calculates actuals and budgets. Figure 6 shows the dataset “sales” used to generate all the indicators for our example.

	Product	Year	Quarter	MidWest	East	West	AlaskaHawaii	South	TOTALREVENUE	TOTALPROFIT	NationTotalAutoSales	MarketShare	CustomerSatisfaction
1	Compact	2005	1	712262	568158	412262	68158	985903	2746743	829609.16	24720687	11.1111111	4
2	Compact	2005	2	716134	568851	116134	68851	455826	1925796	631095.52	75106044	2.56410256	4
3	Compact	2005	3	808628	569097	108628	69097	785891	2341341	3280960.92	114725709	2.04081633	3
4	Compact	2005	4	266550	1074300	266550	974300	416483	2998183	359781.96	14990915	20	5
5	Sedan	2005	1	534046	311612	134046	511612	566698	2058014	246961.68	57624392	3.57142857	3
6	Sedan	2005	2	732316	313449	232316	413449	777667	2469197	296303.64	32099561	7.69230769	4
7	Sedan	2005	3	938201	218169	338201	318169	668623	2481363	297763.56	94291794	2.63157895	3
8	Sedan	2005	4	485860	395900	485860	395900	249759	2013279	241593.48	18119511	11.1111111	2
9	Luxury	2005	1	166110	716369	766110	416369	610565	2675523	321062.76	69563598	3.84615385	2
10	Luxury	2005	2	914010	118714	214010	118714	559422	1924870	230984.4	49276672	3.90625	3
11	Luxury	2005	3	819440	920214	319440	920214	411556	3390864	406903.68	200060976	1.69491525	3
12	Luxury	2005	4	272360	1981400	472360	981400	349762	4057282	486873.84	105489332	3.84615385	4
13	Midsize	2005	1	407265	427135	1407265	427135	311315	2980115	757613.8	29503138.5	10.1010101	4
14	Midsize	2005	2	802486	826565	1202486	826565	311568	3969670	676360.4	39299733	10.1010101	4
15	Midsize	2005	3	389452	135261	989452	135261	414731	2064157	5371548.26	41283140	5	5
16	Midsize	2005	4	201653	428878	1101653	428878	812937	2973999	956879.88	172491942	1.72413793	3
17	Standard	2005	1	531069	624276	931069	624276	212178	722868	350744.16	111068984	2.63157895	3
18	Standard	2005	2	506040	230483	806040	230483	310414	2083460	250015.2	47919580	4.34782609	4
19	Standard	2005	3	424214	232402	924214	232402	415364	2228596	267431.52	40114728	5.55555556	4
20	Standard	2005	4	764610	129215	1664610	129215	512876	3200526	384063.12	76812624	4.16666667	2
21	MiniVan	2005	1	404462	1722253	1404462	722253	712482	2965912	395909.44	124147800	4	2
22	MiniVan	2005	2	679816	820461	879816	820461	613051	3813605	457632.6	61017680	6.25	3
23	MiniVan	2005	3	364426	617250	964426	617250	712279	3275631	393075.72	55685727	5.88235294	4
24	MiniVan	2005	4	962398	815735	1962398	815735	991293	5547559	665707.08	99856062	5.55555556	4

Figure 6 - Sample Sales Data of an Automotive Company

Figure 2 shows the “Goals” dataset for each of the indicators that we wish to represent in the dashboard, and that is used to create the final indicators dataset.

	Indicator	Budget
1	Revenue	25000000
2	Profit	5000000
3	Market Share (%)	8
4	Average Customer Satisfaction	4
5	Midsize	5000000
6	MiniVan	3000000
7	Compact	3500000
8	Sedan	3500000
9	Standard	3000000
10	Luxury	6000000

Figure 7 - Goal or Budget for each Indicator

Finally, Appendix A shows the stored process that is used in our example to merge and generate the final indicators dataset. Note that the final dataset is created in the work library so it is accessible to the workspace created by the Portlet, and most importantly, to avoid concurrency problems between different sessions. It is advised that dynamic data used by the Portlet be always created in the work library.

LAYER 2 – BACKEND CONNECTIVITY

There are several ways to create a connection to a SAS Workspace server. In the example presented in this paper the Portlet makes use of a customized connection pool which connects to a SAS server using directly supplied Metadata Server attributes. Examples on creating server connections can be found in the SAS website under the documentation section [see reference 3]. In our particular application, we create and manage a connection pool that will create workspace connections on demand unless there are connections available in the pool. Further on the Connection Pool implementation is presented in the next section.

The sample stored process that is executed in our example through the Portlet edit page, accesses a dataset under the library “Chrysler”. Therefore, this library has to be accessible to the workspace server that is attempting to run the stored process. In order to access this SAS dataset, the SAS programmer needs to make sure two conditions are in place:

- The containing library needs to be registered in the SAS management console, as Figure 8 shows:

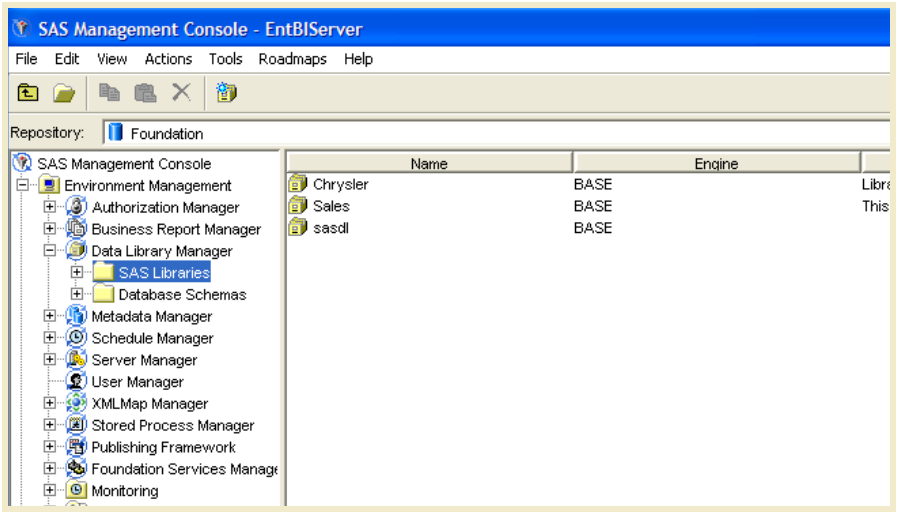


Figure 8 - Library registered via the Data Library Manager Plugin

- The library needs to be pre-assigned as shown in Figure 9:

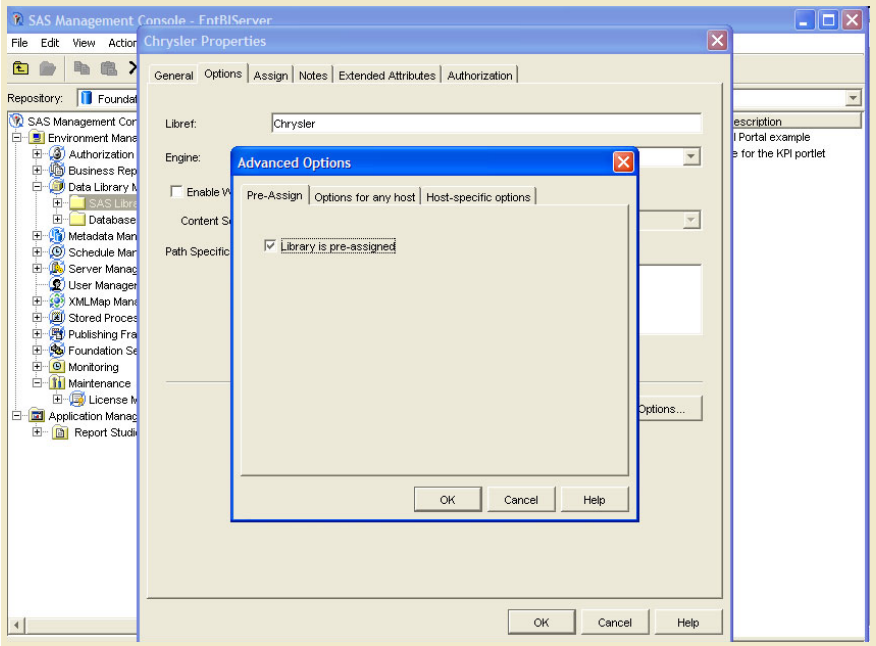


Figure 9 - Library Pre-assigned Option

LAYER 3 – MIDDLEWARE ARCHITECTURE: Building an Editable Portlet

As described in the SAS support documentation, an Editable Portlet is served by different action classes (or servlets) which define the flow in the Portlet. An Editable Portlet (or often called a Portlet Template) gives the capability to the user to edit the Portlet's instance. For more information on Editable Portlets refer to [5] in the References section. Each of the action classes used in our example are described below:

BaseAction: This class sets up the logger and implements the methods for checking errors in the Portlet. The explanation of this class falls out of the scope of this paper, but it is pertinent to mention that all action classes extend BaseAction, which implements PortletActionInterface.

DisplayAction: This class is responsible for the generation of the right data to display in the CSF diagrams, and in the summary table. Its "service" method creates during the first load a pool of connections to the SAS server that will control and dispatch connections at every request from the Portlet. DisplayAction also creates a SalesData object which is a utility class explained in this section. This class does most of the work. Appendix B provides the main fragment of the service method in the DisplayAction class which performs the main operations.

RefreshAction: This action class is a summarized version of DisplayAction and is responsible for re-running the stored process, generating the datasets, and assigning the new loaded data to the table and the diagrams. This all is done in the generateTableView method in the SalesData object. Appendix C provides this class' "service" method code which performs the main actions.

EditorAction: This class is the responsible for performing utility actions like constructing the URLs for the OK and Cancel buttons in the editor page. Appendix D shows the code for the "service" method in this class, which performs the main actions.

In this example, the Portlet is initially loaded using default information provided through a resource bundle file. This file is read by a Java and gets the necessary information to connect to the SAS Metadata Server, the default stored process repository (or server directory), and the name of the initial dataset to be diagramed. Figure 8 shows the flow between action classes and JSP pages in the Editable Portlet.

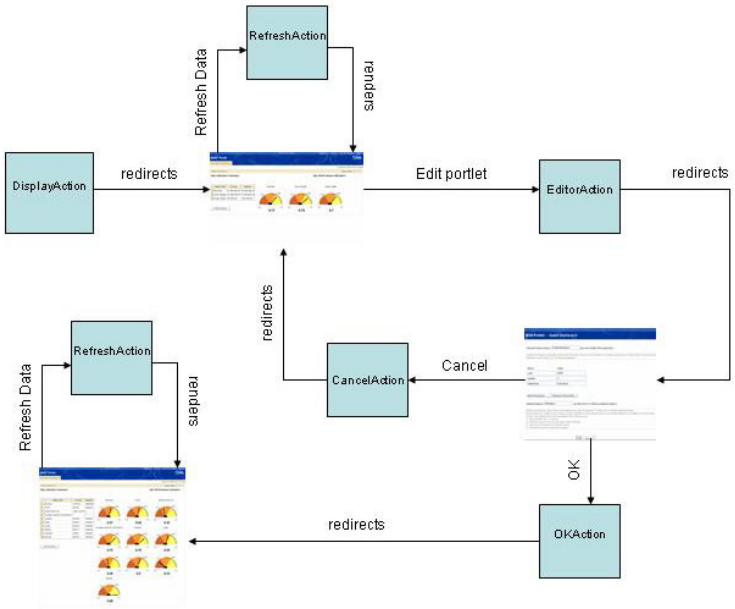


Figure 10 - Dashboard Portlet Diagram Flow

The utility classes are:

SalesData: This class is the responsible for abstracting the process of creating the TableViewComposite object, rendering the indicators table in the user interface, running the stored process, assigning the new data to the TableViewComposite, and calculating the percentages that indicate the ratio actual/budget for each indicator. For more information on TableViewComposite beans, please refer to [2] in the References section. The calculated percentages are used by the Rangeview applet to render the arrow in the right range of the CSF diagram. The main method that is called to generate the data and run the stored process is called generateTableView, which code is presented below:

```
public void generateTableView(HttpServletRequest req, HttpServletResponse resp,
PortletContext cont){

        request = req;
        response = resp;
        context = cont;

        HttpSession session = request.getSession();

        session.setAttribute("sales_sprocessPath", storedProcessName);

        sqlConnection = conPool.getConnection();
        context.setAttribute("sales_connection", sqlConnection);


        try{
                //Prepare action provider for the table
                initializeActionProvider();

                //run user's stored process
                runStoredProcess();

                assignTableView();

                //Get updated percentages and indicators' names

                Vector percentages = getPercentages();
                session.setAttribute("sales_percentages", percentages);
                Vector indicatorsNames = getIndicatorsNames();
                session.setAttribute("sales_indicatorsNames", indicatorsNames);
        }
        catch(Exception e){
                String message = e.getMessage();
                context.setAttribute(Initializer.PORTLET_EXCEPTION_KEY, message);
        }
}
```

This method calls five other methods in the SalesData class. `initializeActionProvider()` is a method that simply initializes the action provider associated with the TableViewComposite bean. For more information on initializing action providers, please refer to [1] in the References section.

`runStoredProcess()` is the method that runs the stored process indicated in the editor page of the Portlet. It uses the ConnectionPool object to obtain the IStoredProcessService object associated with the workspace where the current connection was generated from. In this example, all the connectivity process was abstracted into a ConnectionPool class using the Java Connection Factory API provided by SAS. For more information on how to get a workspace connection from a SAS server, please refer to [2] in the References section.

Below is the code fragment from the `runStoredProcess` method:

```
private void runStoredProcess(){

  if (storedProcessName != null && !storedProcessName.equals("")){
    try{
       IStoredProcessService spService = conPool.getStoredProcess(sqlConnection);

       spService.Execute(storedProcessName, buildParameters(extraParams));

       ILanguageService lang = spService.Parent();


    }
    catch(GenericError ex){
       String message = ex.getMessage();
       context.setAttribute(Initializer.PORTLET_EXCEPTION_KEY, message);
    }
  }
}
```

`buildParameters` is a private method that gets all the parameters from the editor page as a Properties object and constructs a string appending the name of the variable with an equal sign and the value of the parameter, i.e.,
"<param-name1>=<param-value1> <param-name2>=<param-name2> …"

`assignTableView()` builds the SQL statement with the new dataset indicated in the Portlet editor page, creates a new JDBCTableModelAdapter object with the new query which then is associated with the TableViewComposite.

```
private void assignTableView() throws Exception{

       HttpSession session = request.getSession();

       //Assign grand total table
       synchronized (session) {

          //Setup the query for the connection
         String jdbcQuery = "select * from "+indicatorsTableName;


          //Setup the JDBC model adapter
          JDBCToTableModelAdapter adapter = null;
          if (session != null){
            adapter = (JDBCToTableModelAdapter)session.getAttribute(SALES_MODEL);
          }
          try{
              adapter = new JDBCToTableModelAdapter(sqlConnection, jdbcQuery);
               if (context != null){
                   context.setAttribute(SALES_MODEL, adapter);
               }
           }
           catch(Exception e){
               throw new Exception();
           }
```

```
        //Assign the tableView
        TableViewComposite sales_TableView = null;

        if (session != null){
                sales_TableView = (TableViewComposite)session.getAttribute(TABLE_VIEW);
        }

        if (sales_TableView == null){
                sales_TableView = new TableViewComposite();
        }
        sales_TableView.setModel(adapter);

        TableView tViewSummary = (TableView)
        sales_TableView.getComponent(sales_TableView.TABLEVIEW_TABLEDATA);


        session.setAttribute(TABLE_VIEW, sales_TableView);

  }

}
```

Finally, the `generateTableView` method in the SalesData object calls the methods getPercentages and getIndicatorsNames which return vectors with the percentages of the actual respective to the budget (or goal) for each indicator and each indicators names, respectively.

This is done by looping through the TableView object contained in the TableViewComposite object and getting the value for the actual and the budget for each indicator. Note that this method heavily relies on the guidelines provided for the source data, as it has to be in the format <Indicator> <Actual> <Budget> to be able to calculate the percentages. The following code fragment illustrates this.

```
private Vector getPercentages(){

        HttpSession session = request.getSession();

        Vector percentages = new Vector();

        TableViewComposite tvcSummary =
        (TableViewComposite)session.getAttribute(TABLE_VIEW);

        TableView tViewSummary = (TableView)
        tvcSummary.getComponent(tvcSummary.TABLEVIEW_TABLEDATA);

        TableModel modelSummary = tViewSummary.getModel();

        for (int j=0; j < modelSummary.getRowCount(); j++){

                String sactual = modelSummary.getValueAt(j, 1).toString();
                String sbudget = modelSummary.getValueAt(j, 2).toString();

                double actual=0;
                double budget=0;

                //Find the format of the variable
                try{
                Number nf = NumberFormat.getNumberInstance().parse(sactual.trim());
                actual = nf.doubleValue();

                nf = NumberFormat.getInstance().parse(sbudget.trim());
                budget = nf.doubleValue();
                }
```

```
                catch(ParseException e){
                        try{
                        //Find if the number is in currency format
                        Number nf =NumberFormat.getCurrencyInstance().parse(sactual.trim());
                        actual = nf.doubleValue();

                        nf = NumberFormat.getCurrencyInstance().parse(sbudget.trim());
                        budget = nf.doubleValue();
                        }
                        catch(ParseException ex){
                                String message = ex.getMessage();
                                context.setAttribute(Initializer.PORTLET_EXCEPTION_KEY,
                                "One of the values for actual or budget variables cannot be
                                parsed as a number. Please check your source data. Root
                                cause: " + message);
                        }
                }


                Double percentage = new Double(actual/budget);
                percentages.addElement(percentage);

        }
}
```

Also note that this method takes into account the possibility that the variables actual and budget in the SAS dataset have a currency informat and parses it adequately.

Similar to the `getPercentages` method is the `getIndicatorsName`. Both methods produce a vector with the percentages achieved by each indicator, and each indicator's name respectively. After these two methods have been called in the `generateTableView` method, the produced vectors are stored as a session attribute so they might be accessed in the JSP display page.

ConnectionPool: This class is an abstraction of all the methods related to getting a connection to the SAS Metadata server and getting a connection to a registered SAS Workspace server. Besides creating connections, this method is responsible for maintaining them once they are returned to the pool by the application, and cleaning up the resources when a connection has been dropped or closed. Explaining the step-by-step implementation of the connection pool is out of this paper's scope, but for more information on how to create connections a SAS server using the Java Connection Factory refer to [3] in the References section. The main methods from the ConnectionPool class utilized by the Portlet are:

- `getConnection()`: Returns a java.sql.Connection object obtained from the SAS server.
- `returnConnection()`: Returns the connection back to the pool to be reused by another Portlet request.
- `getStoredProcess()`: Returns a stored process service associated with the connection previously obtained from the pool.

LAYER 4 – GRAPHICAL USER INTERFACE

This section is one of the main focuses of this paper since it presents how the SAS CSF applet can be used from the Editable Portlet. But first, before we go through the implementation, let's go through the flow diagram of JSP pages in the Portlet:

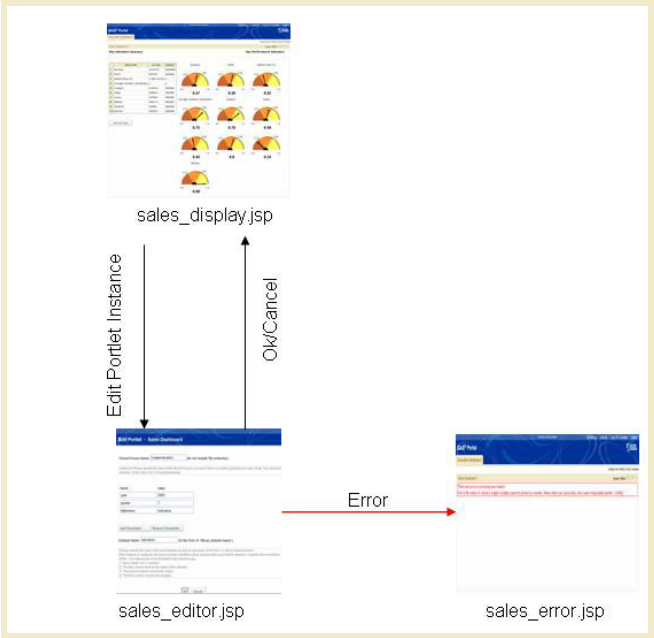


Figure 11 - Portlet's JSP pages Flow Diagram

The implementation of the sales_editor.jsp page, which is the one that displays the form for editing the Portlet's instance, is very straightforward HTML. The sales_error.jsp page is a simple JSP page that is redirected to by the Display Action class and prints the cause of the exception if there has been one during the execution.

Now, let's get into the heart of the matter and study the display page implementation, which contains the most acclaimed CSF diagrams.

The SAS Critical Success Factor diagram is commonly generated by the DS2CSF macro, which serves as a wrapper around the applet and is responsible for passing the arguments to the applet, generating a full HTML page, and generating the style tags for the look and feel of the HTML page which embeds the CSF diagram. Unfortunately, a simple call to this macro from the Portlet will not work, since a requirement for Portlets is that the JSP and HTML pages contained in them should not have <html> and <body> tags; and this is precisely what the DS2CSF macro does. A workaround for this is to embed the applet object by ourselves in our JSP page fragment, passing the appropriate arguments for its rendering.

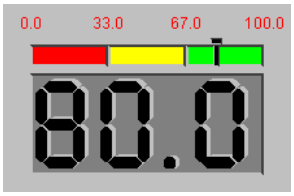
The following text presents the code fragment that embeds the CSF applet into the JSP page:

```
<OBJECT ALT="Percentage" HEIGHT="165" WIDTH="190" CLASSID="clsid:8AD9C840-044E-
11D1-B3E9-00805F499D931">
       <PARAM NAME="ARCHIVE" VALUE="rvapplet.jar">
       <PARAM NAME="CODE" VALUE="RVApplet.class">
       <PARAM NAME="JAVA_CODEBASE"
       VALUE="http://www2.sas.com/codebase/graph/v91">
       <PARAM NAME="TYPE" VALUE="application/x-java-applet;version=1.4">
       <PARAM NAME="GRAPHTYPE" VALUE="CLASSIC">
       <PARAM NAME="VALUE" VALUE="<insert-value-to-represent>">
       <PARAM NAME="VALUEPOSITION" VALUE="Bottom_Center">
       <PARAM NAME="BACKGROUNDCOLOR" VALUE="WHITE">
       <PARAM NAME="START" VALUE="0">
       <PARAM NAME="END1" VALUE="0.33">
       <PARAM NAME="COLOR1" VALUE="#CC6633">
       <PARAM NAME="END2" VALUE="0.66">
       <PARAM NAME="COLOR2" VALUE="#FF9933">
       <PARAM NAME="END3" VALUE="1">
       <PARAM NAME="COLOR3" VALUE="#FFFF66">
```

```
        There was a problem with the Java applet or Plugin in your browser.  The
        graphic cannot be displayed.
</OBJECT>
```

For those who are not acquainted with applets, these are applications written in Java that run inside a web page, executed within the browser itself. An applet downloads and executes on the end user's computer rather than executing on the server. The object tag embeds the applet in the generated HTML page. The ALT parameter specifies the alternate text to display. The CLASSID parameter is a URL indicating the implementation for the OBJECT. In some systems this is a class identifier. The applet is rendered in the dimensions specified by the HEIGHT and WIDTH parameters. Contained within the OBJECT tag, you find the list of parameters that are to be passed to the applet class. The required `NAME` attribute of `PARAM` gives the name of the parameter while the `VALUE` attribute gives the parameter's value. The parameters passed to the applet are explained below:

- `ARCHIVE`: specifies JAR files required to run the applet, in this case, `rvapplet.jar`. This file must exist at the location specified by the CODEBASE parameter.
- `CODE` (`URI`): Indicates the applet's class file, in this case RVApplet.class.
- `JAVA_CODEBASE`: Specifies the download URL for the applet class and all other required classes. In this case, the classes are downloaded from the SAS library repository directory: http://www2.sas.com/codebase/graph/v91
- `TYPE`: specifies the MIME type of the data referenced in the DATA attribute in advance of retrieving it.
- `GRAPHTYPE`: Specifies the style of the critical success factor diagram that can be either classic or digital. Our example displays the classic display, but the digital might be also used and has the following appearance:



- `VALUE`: Specifies the number that will be represented in the CSF diagram. In our example, it is dynamically calculated by the Portlet calling the `getPercentages` method from the SalesData object.
- `VALUEPOSITION`: Indicates the position where the number represented in the diagram will appear respective to the diagram itself.
- `BACKGROUNDCOLOR`: Indicates the color of the background of the diagram.
- `START`: Indicates the numeric value at the start of the first range in the Rangeview applet.
- `ENDX`: indicates the end of the x^{th} range in the Rangeview applet.
- `COLORX`: Indicates the color of the x^{th} range in the Rangeview applet.

There are some other parameters that can be used with this applet to make the CSF diagram drillable, indicating the drill target and the URL to display when clicked. This functionality was not implemented in this paper's sample Portlet, hence it is out of this paper's scope. SAS provides great documentation on the

macro that generates this applet. For more information refer to [4] in the References section.

CONCLUSION

Now with SAS Portal, SAS Integration Technologies and the extensive SAS graphic libraries, corporations are able to create Executive Dashboards with real time data and customizable data source, using very attractive and easy to read diagrams such as the SAS Rangeview applet. This paper illustrated the steps for the implementation of an Executive Dashboard Editable Portlet, in a way that executives are able to spot at a glance any problem/opportunity in the area of sales, allowing them to stay focused on mission critical issues and the success of their enterprise.

APPENDIX A: Sample Stored Process used to merge and summarize the automotive sales data into a comprehensible indicators dataset.

```
/*************************************************************************
* STORED PROCESS NAME: CreateIndicators                                  *
* PURPOSE:    To create an indicators table called <tablename> to represent *
*             graphically via KPIs.                                      *
* PARAMETERS: <year>: represents the year for which the stored process will *
*             create the indicators table.                               *
*             <quarter>: represents the quarter of the selected year for    *
*             which the stored process will create the indicators table.    *
*             <tablename>: Name of the table to generate with the final     *
*             indicators. This table will have the form of:              *
*             Column 1: Indicator                                        *
*             Column 2: Actual                                           *
*             Column 3: Budget                                           *
*************************************************************************/

%global year quarter tablename;

*ProcessBody;

proc sql;
       create table indicators as
       select       sum(totalrevenue) as Revenue "Revenue",
                    sum(totalprofit) as Profit "Profit",
                    (sum(totalrevenue)*100)/sum(NationTotalAutoSales)         as
marketShare "Market Share (%)",
                    avg(customerSatisfaction)   as   AverageCustomerSatisfaction
"Average Customer Satisfaction"
       from Chrysler.sales
       where year=&year and quarter=&quarter
       group by quarter;
quit;

proc      transpose      data=indicators      out=indicators(rename=(col1=Actual
_label_=Indicators) drop=_name_);

run;

data indicators;
       length Indicator $30;
       set indicators;
       Indicator=Indicators;
       drop Indicators;
run;

** Create the table with the products sales actuals ;

proc sql;
       create table indProducts as
       select product, totalrevenue
       from Chrysler.sales
       where year=&year and quarter=&quarter;
quit;

data indProducts;
       length Indicator $30 Actual 8;
       set indProducts;
       Indicator=Product;
       Actual=totalrevenue;
```

```
       drop Product totalrevenue;
run;


** Join the two indicators tables  ;
proc append base=indicators data=indProducts;
run;

proc transpose data=Chrysler.goals
       out=goals(rename=(col1=Budget _name_=Indicators));
run;

data goals;
       length Indicator $30;
       set goals;
       Indicator=Indicators;
       drop Indicators;
       if (Indicator = 'marketShare') then
              Indicator = 'Market Share (%)';
       else if (Indicator = 'AverageCustomerSatisfaction') then
              Indicator = 'Average Customer Satisfaction';
run;



** Join the indicators table with the goals table;

proc sql;
       create table &tablename as
       select b.Indicator, Actual, Budget
       from indicators a, goals b
       where trim(a.Indicator) = trim(b.Indicator);
quit;
```

APPENDIX B: DisplayAction's "service" Method

```
public String service(HttpServletRequest request,
                      HttpServletResponse response,
                       PortletContext context) throws Exception
{

       super.service(request, response, context);

       HttpSession session = request.getSession();


       String    baseURL    =    NavigationUtil.buildBaseURL(context,    request,
"display");

       context.setAttribute("sales_baseURL", baseURL);

       String url = NavigationUtil.buildBaseURL(context, request,
           "refresh");

      context.setAttribute(Initializer.DISPLAY_REFRESH_URL_KEY, url);



       String             generateDefaultTable             =             (String)
       session.getAttribute("sales_generateDefaultTable");

if (generateDefaultTable == null || generateDefaultTable.equals("true")){


       //Get path to stored processes
       ResourceBundle rb = ResourceBundle.getBundle("application");
       String storedProcessPath = rb.getString("sasmacropath");


       //Get connection from server
       if (conPool == null) {
              conPool = new ConnectionPool();
              context.setAttribute("sales_connPool", conPool);
              session.setAttribute("sales_sprocessPath", storedProcessPath);
       }


       SalesData                data                =                (SalesData)
       session.getAttribute("sales_salesDataObject");

       if (data ==  null) {
              data = new SalesData(conPool);
              session.setAttribute("sales_salesDataObject", data);
       }

       //Assign table view if not assigned already
       data.generateTableView(request, response, context);

       session.setAttribute("sales_generateDefaultTable", "false");

}

errorCheck(context);

return (String) context.getAttribute("display-page");

}
```

APPENDIX C: RefreshAction's "service" Method

```
public String service(HttpServletRequest request,
                      HttpServletResponse response,
                      PortletContext context) throws Exception
{
        super.service(request, response, context);

         SalesData               data               =               (SalesData)
         request.getSession().getAttribute("sales_salesDataObject");


         //Assign table view if not assigned already
         if (data != null){
              data.generateTableView(request, response, context);
          }

          return (String)context.getAttribute("display-page");
    }
```

APPENDIX D: EditorAction's "service" Method

```
public String service(HttpServletRequest request,
                      HttpServletResponse response,
                      PortletContext context) throws Exception
{
      super.service(request, response, context);

      //create the URLs for the OK and Cancel buttons.
      String url;

      url = NavigationUtil.buildBaseURL(context, request,
           "ok");
      context.setAttribute(Initializer.EDIT_OK_URL_KEY, url);

      url = NavigationUtil.buildBaseURL(context, request,
               "cancel");
      context.setAttribute(Initializer.EDIT_CANCEL_URL_KEY, url);

      // the following call resets the mode back to display for the
      // next call
      context.resetMode();


        return (String) context.getAttribute("edit-page");
    }
```

REFERENCES

[1] SAS Institute Inc. Samples, “JSP TableView with ActionProvider”.
Available at: http://support.sas.com/ctx/samples/index.jsp?sid=1572

[2] SAS Institute Inc. SAS Custom tags, “The Custom Tag Handler Class for the TableViewComposite”. Available at:
http://support.sas.com/rnd/gendoc/bi/api/Components/taglibs/sas/sas_TableViewComposite.html

[3] SAS Institute Inc. SAS 9.1.3 Integration Technologies Developer’s Guide, “Connecting with Server Attributes Read from a SAS Metadata Server”. Available at:
http://support.sas.com/rnd/itech/doc9/dev_guide/dist-obj/javaclnt/javaprog/connfact_omr.html

[4] SAS Institute Inc. SAS 9.1.3 User Documentation, “Creating Critical Success Factor Diagrams”

[5] SAS Institute Inc. SAS 9.1.3 Integration Technologies Developer’s Guide, “Developing Custom Portlets”. Available at:
http://support.sas.com/rnd/itech/doc9/portal_dev/portlets/dg_portlets.html

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Elibeth Carpenter
Application Developer
Comsys SAS Practice
5278 Lovers Lane
Kalamazoo, MI 49002
Email: ecarpenter@comsys.com