New Light through Old Windows:
Delivering Real Time Information with Windows 7 Desktop Gadgets

David Shannon, Amadeus Software, Oxford, UK

ABSTRACT

In this paper the author demonstrates how to deliver SAS® information directly to the information consumer's desktop with Microsoft Windows 7 Desktop Gadgets.

Displaying headline analytics from continually updating data, such as an active marketing campaign, or simply the status of your SAS server are just two possibilities for exploiting the Gadget feature.

From Windows 7, Gadgets are an integral component in the operating system presentation. Real time (or near real time) information can be delivered by integrating a SAS session via web services or the SAS Integration Technologies product.

This paper presents the steps for integrating SAS and Windows Gadgets. Worked examples, with source code, are demonstrated along with options for deployment. Finally, the pros and cons of delivering information this way are considered.

Those attending this paper are not expected to have experience of creating Desktop Gadgets, however an appreciation of web pages, object orientated programming and the SAS Integration Technologies product will be useful to take the most from this paper.

INTRODUCTION

This paper examines a method of delivering short summaries of information directly to user desktops with Microsoft Windows Desktop Gadgets, referred to as Gadgets throughout this paper.

The paper begins by understanding what Gadgets are and how they work. The discussion then describes how to integrate Gadgets with SAS by using Microsoft Jscript to call a web service or SAS Integration Technologies.

Connections to SAS are made from a Gadget to submit PROC steps, retrieving data and displaying the results. Practical examples are provided to illustrate this.

The information displayed in Gadgets is intended to be top line summary data. Basic design principals are presented and references are drawn to existing well designed Gadgets and published design guidelines by Microsoft Corporation.

Finally, the paper positions the use of Gadgets for real time reporting against other Business Intelligence reporting tools.

WHAT ARE DESKTOP GADGETS AND HOW DO THEY WORK?

Gadgets are lightweight applications which are presented as translucent windows that float on the PC desktop. They are constructed from standard HTML, cascading style sheets and web scripting languages. In other words they are built in the same way as a web page is constructed.

Gadgets have an object model which intentionally exposes a limited range functions. To integrate with external services, Jscript is used which enables Windows installed or bespoke COM classes (deployed with the Gadget) to connect with a SAS session or server.

To create a basic Gadget, a minimum of two files are required:

1. GADGET.XML: This is an XML manifest about the Gadget with information such as how it will be displayed, which HTML file is displayed, who created the gadget, etc. The file is short and easy to create by modifying an existing example. The reader is directed to the documented from Microsoft, see Reference 1 for full details.
2. *NAME*.HTML: This is the page that is the physical presentation on the users on the desktop.

It is very easy to find examples of existing Gadgets and replicate their functionality by browsing the gadgets which exist on your Windows 7 installation. The source of gadgets on your PC is found in the following locations:

- C:\Program Files\Windows Sidebar\Gadgets (Windows Supplied Gadgets)

- C:\Program Files\Windows Sidebar\Shared (Gadgets Shared Gadgets)
- C:\Users\User-ID\AppData\Local\Microsoft\Windows Sidebar\Gadgets (User Added Gadgets)

Figure 1, below, is an image of the Weather Gadget that is provided with Windows 7. It connects to a service that provides current and future weather data for a configurable location:

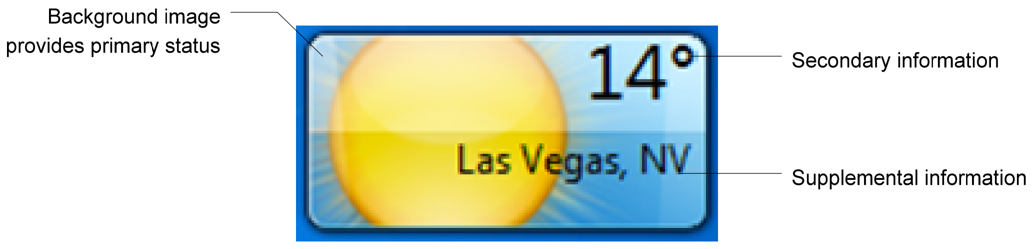


Figure 1: The Weather Gadget on a Windows 7 Desktop

There are three elements that together provide both a visual and textual summary. The information presented is without great detail, yet is sufficient to be informative. These design principles are appropriate for Gadget displays and can be summarised as being highly visual with minimal headline text information. Microsoft proposes quite detailed requirements about the construction of Gadgets. These can be found within the URL described in reference 1 of this paper.

INTEGRATING SAS WITH DESKTOP GADGETS

The implementation options for serving data from SAS are considered here. The following methods either expose SAS functionality through an API or publish data in a static structure:

1. **SAS Integration Technologies' Integration Object Model (IOM)**: The IOM provides several objects for creating and exploiting SAS sessions, stored processes and more from a custom client. See reference 4 of this paper for documentation of the IOM.

   The IOM is specifically designed for clients that need to integrate with SAS servers. Indeed it is how SAS Institute's own clients such as Enterprise Guide, Data Integration Studio and alike, communicate with the metadata and application servers. The advantages of this technology are a well defined and highly functional API and the author fully expected this technology to be the ideal solution.

   In the event, there were issues referencing multiple DLL's from a Gadget. Creating connections to SAS servers and facilitating calls to stored processes etc. cannot be performed directly in the Gadget object model. A custom class (DLL) must be written to do this. If that class references further external classes the Gadget has no mechanism to locate the additional DLL files and errors are returned. This is exactly what happens because the SAS Object Manager and IOM Type Libraries are external DLL's. It may be possible, however, to overcome this limitation by instantiating the IOM through a technique called Reflection.

   There were now two drawbacks to this technology choice. Firstly, the implementation effort has increased as (to the authors mind at least) programming .NET using reflection is relatively cumbersome for this solution. Secondly, the emphasis on managing the server requests and outputs is placed in the Gadget rather than the server.

2. **RSS Feeds**: SAS can be used to create and populate an RSS documents. For those reading this paper with only a Base SAS license, you can easily publish data to an RSS feed on a web server. This can be achieved with the data step or more tightly by via the libname statement, its XML engine and an XML MAP. A SAS Batch job can be scheduled to run at frequent intervals and so updating the RSS contents. Within a Gadget the script XMLHttpRequest object can be used to consume the XML document.

   This disadvantage of this route is that RSS feeds are updated periodically and the Gadget is independently updated of the RSS feed. The Gadget therefore has less control over the refresh rate and a reasonable amount of client implementation will be required to parse the returned XML for presentation in a pleasing visual style.

3. **SAS BI Web Services**: From SAS 9.2 creating and publishing a web service is astonishingly simple given a Stored Process on a SAS Business Intelligence server. Guidance for this process can be found within Reference 2 and is illustrated below.

   A web service will execute upon being called, therefore returning results in real time. Most processing overhead takes place on the server, rather than the Gadget. The call to a web service can be parameterised and results streamed to the client in XML format. Web services are not be appropriate for large volumes of data as streaming large amounts of XML can cause sluggish responses. This model therefore suits Gadgets well.

SAS BI Web Services were selected as the best option for providing data to desktop gadgets. Figure 2 summarises the flow of communications and data between the Gadget and the SAS Server:

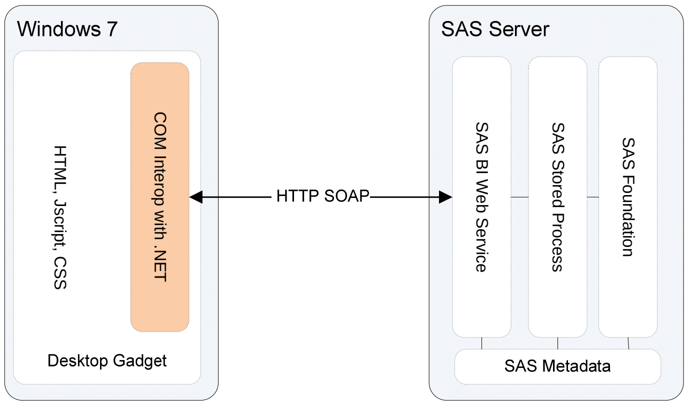


Figure 2: Communication flow between Desktop Gadget and SAS Code

The following list of tasks are the road map to implementing a Gadget that will consume information through a SAS BI Web Service, ultimately it is a SAS program that generates the information returned:

1. Write the SAS program that creates the output you wish to render in your Gadget;
2. Register your program as a SAS Stored Process;
3. Deploy your stored process as a web service;
4. Write a class in .NET (or other language) to consume the web service;
5. Write a Gadget that calls the class and renders the information received from the stored process;
6. Package and deploy.

This may appear like a lot of work, however some steps are simply a few mouse clicks. As with any project, spending a little time to plan what you want to achieve, before writing your code will be more conducive to a successful and timely outcome.

The steps above are best illustrated through the use of an example which, for the most part, makes up the remainder of this paper.

CREATING A GADGET TO QUERY SAS METADATA SERVER STATUS

The purpose of this Gadget is to display the SAS metadata server status obtained from PROC METAOPERATE and the number of active client sessions, calculated by a reading the current metadata server log file. Appendix A presents the SAS code used to query the server status and create one of the two return values.

Firstly, a design to present the information was required.

Figure 3 is a mock-up constructed in a HTML editor and is based closely on the Weather Gadget discussed earlier:

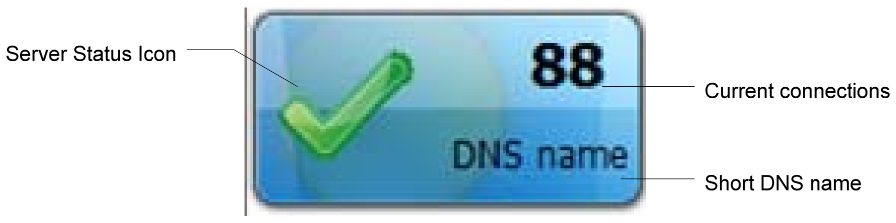
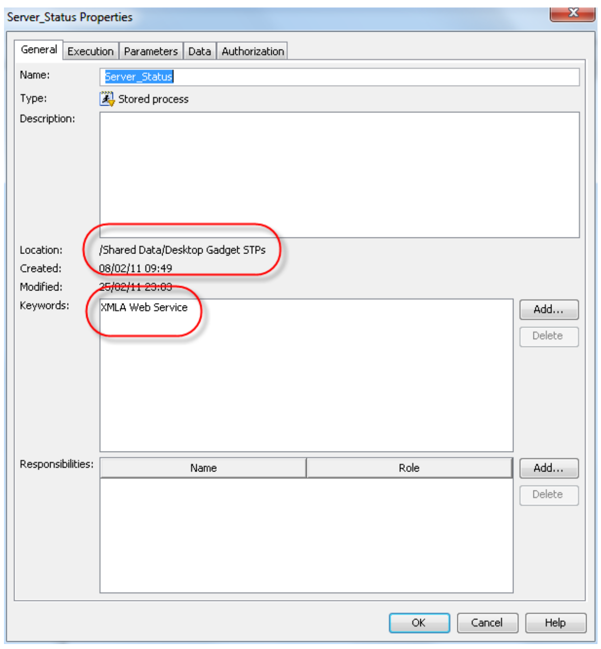


Figure 3: SAS Metadata Server Status Mock-Up

The information displayed is both visual and textual, offering high level summary information. The three elements are displayed within HTML DIV elements, therefore some rudimentary knowledge of HTML mark-up is assumed for this design. Appendix A presents the HTML, CSS and Jscript for the Gadget.

REGISTERING A SAS STORED PROCESS AND DEPLOYING AS A SAS BI WEB SERVICE

A stored process must be registered and configured before being deployed as a web service. The following figures illustrate the key properties of the stored process to ensure successful use as a web service.

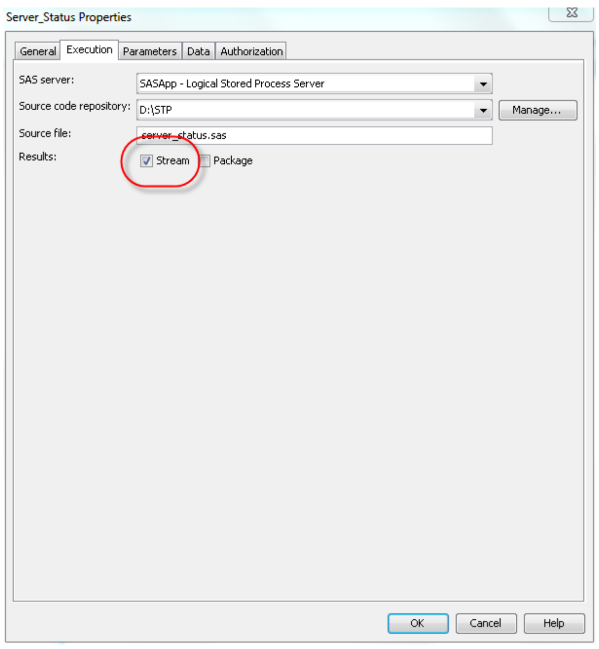


Name: Server_Status
Type: Stored process
Description:
Location: /Shared Data/Desktop Gadget STPs
Created: 08/02/11 09:49
Modified: 25/02/11 23:03
Keywords: XMLA Web Service
Responsibilities:

Name	Role

Figure 4: General Properties

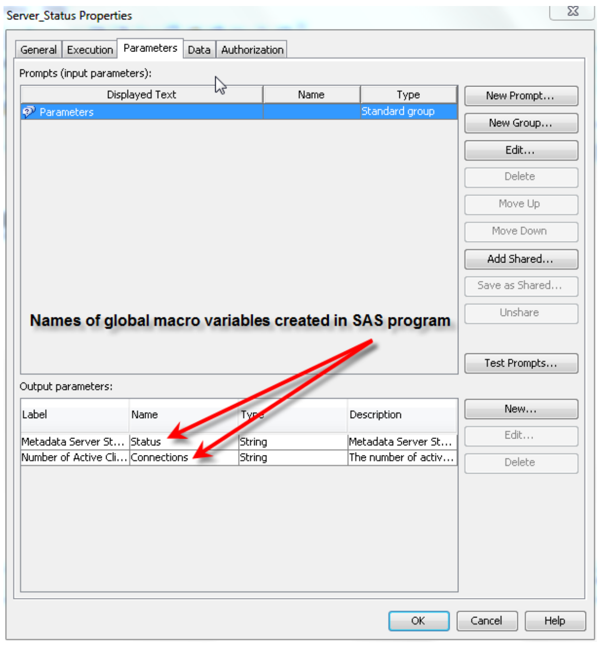
The stored process is located in the shared area. The keyword “XMLA Web Service” is added respecting the documentation that describes this requirement, see Reference 2.



SAS server: SASApp - Logical Stored Process Server
Source code repository: D:\STP
Source file: server_status.sas
Results: [✓ Stream] Package

Figure 5: Execution Properties

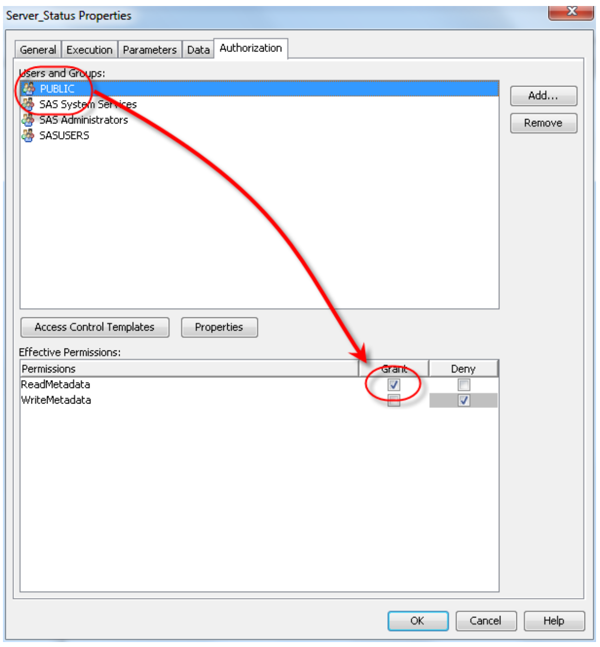
Results must be streamed back to the client. Data is streamed via the _WEBOUT libref and an XML engine. Constants are returned by macro variables.



Label	Name	Type	Description
Metadata Server St...	Status	String	Metadata Server St...
Number of Active Cli...	Connections	String	The number of activ...

Figure 6: Parameter Properties

The names of parameters must match macro variable names used in the SAS code.

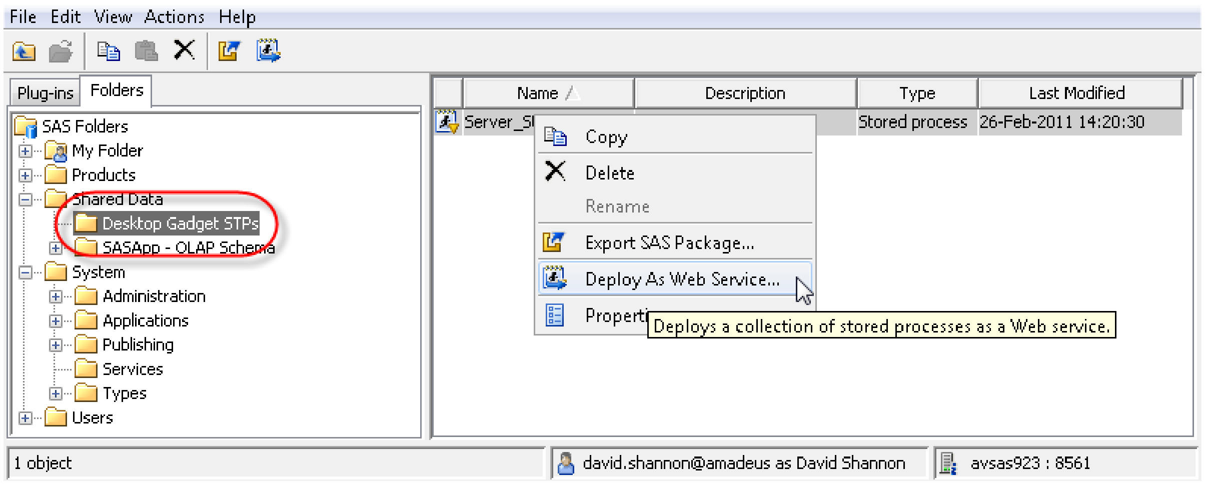


Permissions	Grant	Deny
ReadMetadata	✓	
WriteMetadata		✓

Figure 7: Authorization Properties

Finally, consider the security of the stored process (and web service when created). We allowed non-SAS users to access the web service.

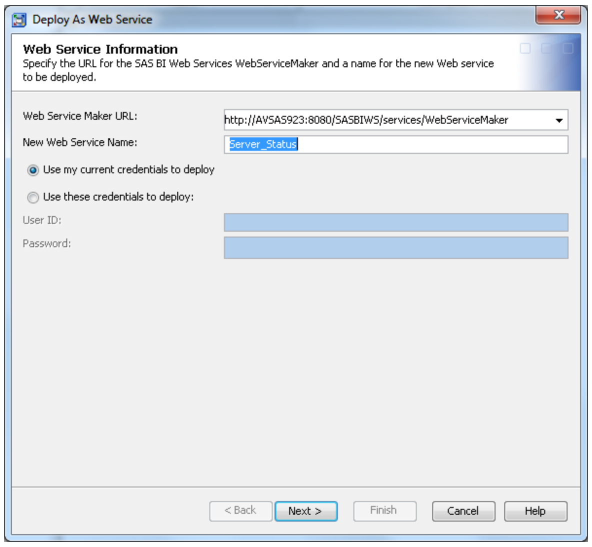
Now the SAS Stored Process is ready for use, it is deployed as a web service. This is completed from within the SAS Management Console as shown in Figure 8 below. Note for good housekeeping, the stored processes associated with Gadgets are organised into a specific metadata folder.



Name	Description	Type	Last Modified
Server_St		Stored process	26-Feb-2011 14:20:30

Figure 8: Deploying the Stored Process as a SAS BI Web Service

Selecting this option invokes a wizard that takes us through the following steps:



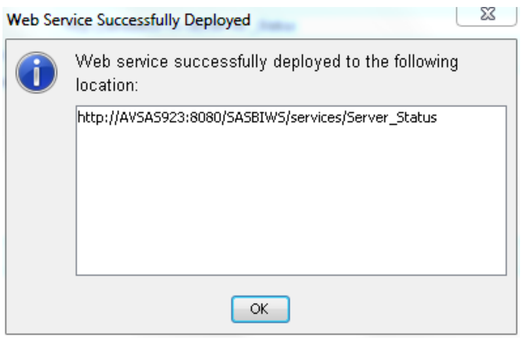
Deploy As Web Service

Web Service Information
Specify the URL for the SAS BI Web Services WebServiceMaker and a name for the new Web service to be deployed.

Web Service Maker URL: http://AVSAS923:8080/SASBIWS/services/WebServiceMaker
New Web Service Name: Server_Status
(●) Use my current credentials to deploy
( ) Use these credentials to deploy:
User ID:
Password:

Figure 9: The Case Sensitive Web Service Name

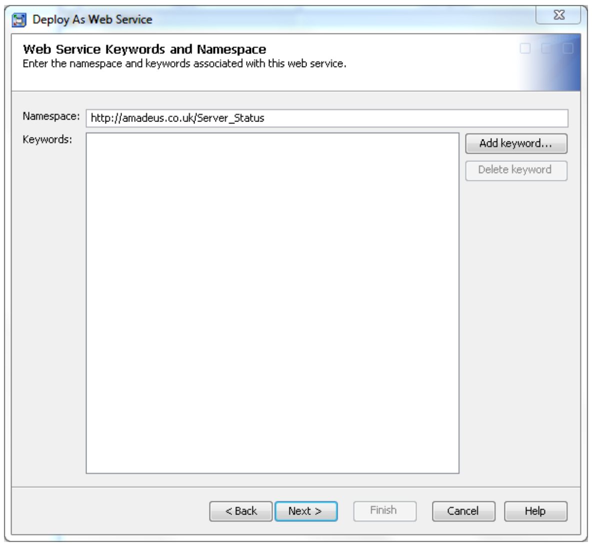
The name of the web service is case sensitive, to avoid problems consuming the web service , it is recommended to use lower case names (unlike this screenshot!)



Web Service Successfully Deployed

Web service successfully deployed to the following location:
http://AVSAS923:8080/SASBIWS/services/Server_Status

Figure 11: The Web Service URL



Deploy As Web Service

Web Service Keywords and Namespace
Enter the namespace and keywords associated with this web service.

Namespace: http://amadeus.co.uk/Server_Status
Keywords:

Figure 10: Adjusting the namespace URI

The namespace is modified to match the company domain. This allows the web service to be uniquely identified. It does not affect the address used to access the service.

Once the remaining steps are completed, the address of the web service is displayed. Be certain to copy or make a note, as it is the only place it is displayed.

The service can now be tested by suffixing ?wsdl onto the address in a web browser. The Web Service Description Language (WSDL) is displayed in the form of XML.

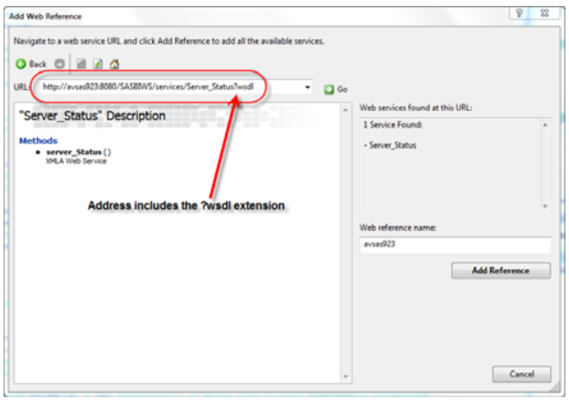
PROGRAMMING A DESKTOP GADGET TO CONSUME A WEB SERVICE

Now that the server side requirements are in place, attention is turned to writing the Gadget to display the information.

A Visual Studio 2008 project was created, and to this the following items were added:

1. A reference must be added to the web service WSDL. Developers in Visual Studio 2008 should be careful to add a Web Service rather than simply a Service (the latter is the default). Figure 12 illustrates adding a Web Reference.

2. A class was then written containing two functions, one to query each of the output parameters from the web service. The following snippet shows how to instantiate an object that talks to the web service. This is followed by a function that returns the status string generated by PROC METAOPERATE (note that error trapping has been removed for clarity):



Add Web Reference

Navigate to a web service URL and click Add Reference to add all the available services.

URL: http://avsas923:8080/SASBIWS/services/Server_Status?wsdl

"Server_Status" Description

Methods
- server_Status ()
  XMLA Web Service

Address includes the ?wsdl extension

Web services found at this URL:
1 Service Found:
- Server_Status

Web reference name: avsas923

Figure 12: Adding the Web Service Reference

```
// Declare s the web service object that will run the STP
avsas923.Server_Status s;

// Method exposed to Jscript and allows interop with SAS
public String Status()
{
  s = new Amadeus.avsas923.Server_Status();
  avsas923.server_StatusResponseServer_StatusResult rc = s.server_Status();
  return rc.Parameters.Status.Value.ToString();
}
```

Appendix C contains full code from the class.

3. The openly available Gadget.Interop .NET project was downloaded and added to the project. It contains a single class and JavaScript file. This allows a .NET class to be used as a COM object from within a Gadgets' Jscript function. A full discussion of this project and its location for download can be found in Reference 3.
4. The final task is to implement the Jscript functions that render information within the Gadget. Appendix B contains the contents of the HTML needed to implement a basic Gadget. The following snippet describes how the connection from the Gadget, via .NET, and a web service is made to SAS:

```
//Create an object and initialize the GadgetBuilder to allow .NET to be called via
//COM
var builder = new GadgetBuilder();
builder.Initialize();

//Create an object (SSS) from the Amadeus.ServerStatus .NET class that exposes the
//web service
var SSS =
builder.LoadType(System.Gadget.path+\\bin\\SSS.dll","Amadeus.ServerStatus");
```

The object SSS can now be used to call those methods. The following snippet, also from Appendix B shows how the contents of the oConnections DIV element are updated with the number of SAS client connections:

```
var connections = document.getElementById("oConnections");
 connections.innerHTML=SSS.Connections();
```

DEPLOYMENT

Gadgets are made up of a series of files in a folder structure.

For the SAS Server Status (SSS) Gadget the files are shown in Figure 13. The Visual Studio project maps to a folder structure on disk.

Deploying is as simple as zipping the folder structure into a zip file and changing the file extension from ".zip" to ".gadget". (see the highlighted file in Figure 14 below).

Installation is completed by opening the .gadget file and following the prompts.

With the Gadget installed users can add and remove the Gadget in the usual way; i.e. from the Desktop Gadgets item from the Control Panel.

Domain administrators can control distribution and usage through automated software delivery technologies.

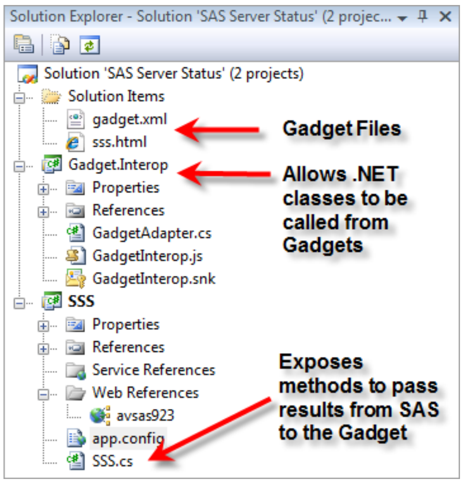


Figure 13: Gadget files from Visual Studio

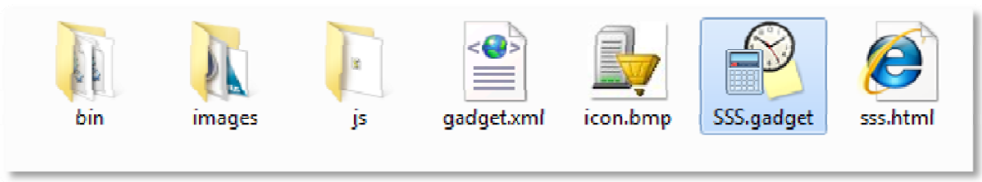


Figure 14: Folder Structure with the zipped Gadget file highlighted

Figures 15 and 16 illustrate the same Gadget, initially when connected to a server with eight current connections to the server and again when the laptop is disconnected from the network. When connected to the network, this status would also indicate a problem the server.

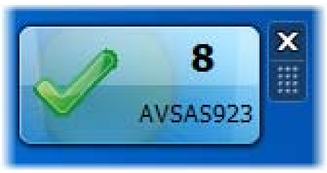


Figure 15: The Gadget when connected

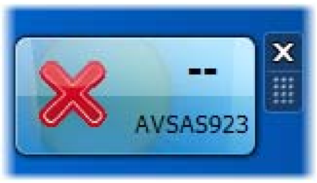


Figure 16: The Gadget with the Windows 7 laptop disconnected from the network

REAL TIME OR NEAR REAL TIME?

By now you have hopefully got an idea of the technical options and a thorough understanding of the steps required to implement a Gadget via SAS BI Web Services. However, we set out to deliver information in real time.

What does real time mean? Does it mean an instantaneous update, or is there an acceptable interval between updates from the operational data system? These answers vary between environments. In practice the rate at which the operational data source is updated and the refresh rate of the Gadget dictate what real time actually is. In this Gadget the refresh interval was defined as 60 seconds. Purists may argue this is near real time, rather than real time itself, where a server side event would trigger the update within a Gadget.

ALTERNATIVE METHODS FOR REAL TIME REPORTING

The latest release of SAS Add-In for Microsoft Office supports the monitoring of SAS reports and BI Dashboards within Microsoft Outlook gadget pane. Outlook is a commonly used e-mail client and a natural consideration for those whose function relies on much of their time being spent around their inbox.

As suggested above, architectures for true real time reporting would require an event based trigger from the source system itself, rather than the Gadget polling a server periodically. Most enterprise software vendors support such technologies. SAS Integration Technologies supports message queuing which could be an ideal method of consuming operational system event based triggers.

CONCLUSIONS

This paper has described how Windows Desktop Gadgets are appropriate for display high level summary information in a visual form.

The technology options for serving up real time information from SAS to Desktop Gadgets was explored. From RSS feeds, SAS Integration Technologies and SAS BI Web Services, the latter was selected. Web services place the emphasis of processing on the server and can be called from any client that consumes web services.

SAS BI Web Services are implemented using SAS Stored Processes, hence can use the power of the SAS programming language to access data, pass-through to operational systems, perform analyses etc.

A Gadget was implemented using a Visual Studio created class to consume a SAS BI Web Service. From within the gadgets HTML file, Jscript was used to instantiate the class and call the web service. The HTML object model was then used render graphics and text within the gadgets display.

Consideration was given to the meaning of real time. It was discussed that this solution may be considered as near real time. A genuine real time solution would require an event based trigger to refresh the Gadget. SAS Software supports this technology and it may be appropriate to explore such solutions. Gadgets are typically configured to refresh every few seconds or minutes.

REFERENCES

The following documentation has been referenced in this paper:

1. Microsoft Corporation. 2010. *Introduction to the Gadget Platform*. Available at: http://msdn.microsoft.com/en-us/library/dd370867(v=VS.85).aspx

2. SAS Institute Inc., 2009. *SAS® 9.2 BI Web Services Developer's Guide*. Cary, NC: SAS Institute Inc.

3. Lee, Wei-Meng. 2008. *Professional Windows Vista Gadgets Programming*. Indianapolis, IN: Wiley Publishing Inc.

4. SAS Institute Inc. 2009. *SAS® 9.2 Integration Technologies: Windows Client Developer's Guide*. Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

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

Name:	David Shannon
Enterprise:	Amadeus Software Ltd.
Address:	Mulberry House, 9 Church Green
City, State ZIP:	Witney, Oxfordshire, OX28 4AZ
Work Phone:	+44 (0)1993 848010
Fax:	+44 (0)1993 778628
E-mail:	david.shannon@amadeus.co.uk
Web:	www.amadeus.co.uk

APPENDIX A: PARTIAL SAS STORED PROCESS

```
proc metaoperate  server="localhost"
                  user="sasadm@saspw"
                  pw="-a-pwencoded-password-"
                  port=8561
                  action=status
                  out=work.status;

run;

data _null_;
set  status(where=(upcase(attribute)="STATUS"));
  call symputx('status',value);
run;
```

APPENDIX B: SAS METADATA SERVER STATUS GADGET HTML, CSS & JSCRIPT

```
<html><head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="/js/GadgetInterop.js"></script>
    <style type="text/css">
        body
        {
            margin: 0; width: 130px; height: 67px; font-family: Tahoma;
            font-size: 12px; background-repeat: no-repeat;
        }
        #oMain
        {
            width: 130px; height: 67px;
        }
        #oConnections
        {
            position: absolute; font-size: 14pt; font-weight: bold;
            text-align: left; top: 7px; left: 91px;
        }
        #oServer
        {
            position: absolute; float: left; height: 20px; width: 61px;
            margin-top: 70px; text-align: right; vertical-align: bottom;
            top: -30px; left: 62px;
        }
        #oIcon
        {
            position: absolute; top: 8px; left: 8px; width: 48px; height: 48px;
        }
    </style>
    <script type="text/jscript">

    var asyncCallback = null;

    //This function invokes a background call to the DLL that calls the web service
    function getStatus(callback)
    {
        // Save a reference to the callback function
        asyncCallback = callback;

        // Call Function Asynchronously
        window.setTimeout("asyncGetStatus();", 1);
    }

    //This runs on a separate thread and actually does the work
    function asyncGetStatus()
    {
        var builder = new GadgetBuilder();
        builder.Initialize();
        var SSS = builder.LoadType(System.Gadget.path +
                  "\\bin\\SSS.dll","Amadeus.ServerStatus");
        try
          {
              var connections = document.getElementById("oConnections");
              switch(SSS.Connections())
              {
                case -1:
                  connections.innerHTML="--";
                  break;
                case -2:
                  connections.innerHTML="--";
                  break;
                default:
                  connections.innerHTML=SSS.Connections();
              }
          }
          catch(e)
          {
              connections.innerHTML="--";
          }

          try
          {
              switch (SSS.Status())
              {
                  case "Running":
document.getElementById("oIcon").style.backgroundImage="url(images/ok.png)";
                    break;
                  case "Paused":
document.getElementById("oIcon").style.backgroundImage="url(images/query.png)";
                    break;
                  default:
document.getElementById("oIcon").style.backgroundImage="url(images/error.png)";
              }
          }
          catch(err)
          {
document.getElementById("oIcon").style.backgroundImage="url(images/error.png)";
          }
       }

        function getStatusCallBack(result)
        {
        //When the aynchronous call finishes code execution lands here.
        //This could be used to receive results of calculations etc.
        }

        function initiateRefreshes()
        {
          getStatus(getStatusCallBack);
        }

       function init()
       {
       //Set the background image
     document.getElementById("oIcon").style.backgroundImage="url(images/unknown.png)";
          //Perform initial load
          getStatus(getStatusCallBack);
          //Then refresh every 60s
          window.setInterval(initiateRefreshes,60000);
       }
    </script>
</head>
<body onload="init()">
    <g:background id="imgBackground" src="url(images/10.png)"> </g:background>
    <div id="oMain" >
       <div id="oIcon"></div>
       <div id="oConnections">--</div>
       <div id="oServer" >AVSAS923</div>
    </div>
</body>
</html>
```

APPENDIX C: SERVER STATUS C# .NET

The following code is the contents of the SSS.cs (C# class). The Visual Studio project must also reference:

1. The Gadget.Interop project available from Codeproject.com
2. A Web Reference (not a service reference) must be added which points to the SAS BI Web Service. This takes the object name avsas923 below.

```
namespace Amadeus
{
    [ComVisible(true)]
    public class ServerStatus : IDisposable
    {
        // Declare s the web service object that will run the STP
        avsas923.Server_Status s;

        // Method exposed to JScript and allows interop with SAS
        public String Status()
        {
            try
            {
                s = new Amadeus.avsas923.Server_Status();
                avsas923.server_StatusResponseServer_StatusResult rc =
s.server_Status();
                return rc.Parameters.Status.Value.ToString();
            }
            // Likely to catch here when the server is off-line, bad credentials etc.
            catch (System.Web.Services.Protocols.SoapException soapEx)
            {
                if (soapEx.Detail.InnerXml.Contains("paused") == true)
                {
                    return "Paused";
                }
                else
                {
                    return "Error";
                }
            }
            // Will catch here in all other situations
            catch (Exception e)
            {
                return "Cannot connect to server";
            }
        }

        // Method exposed to JScript and allows interop with SAS
        public int Connections()
        {
            try
            {
                s = new Amadeus.avsas923.Server_Status();
                avsas923.server_StatusResponseServer_StatusResult rc =
s.server_Status();
                return rc.Parameters.Connections.Value;
            }
            // Likely to catch here when the server is off-line, bad credentials etc.
            catch (System.Web.Services.Protocols.SoapException soapEx)
            {
                return -1;
            }
            // Will catch here in all other situations
            catch (Exception e)
            {
                return -2;
            }
        }

        public void Dispose()
        {
            //---do nothing---
        }
    }
}
```