Paper 003-2013

Create Your Own Client Apps Using SAS® Integration Technologies

Chris Hemedinger, SAS Institute Inc., Cary, NC

ABSTRACT

SAS® Integration Technologies allows any custom client application to interact with SAS services. SAS® Enterprise Guide® and SAS® Add-In for Microsoft Office are noteworthy examples of what can be done, but your own applications do not have to be that ambitious. This paper explains how to use SAS Integration Technologies components to accomplish focused tasks, such as run a SAS program on a remote server, read a SAS data set, run a stored process, and transfer files between the client machine and the SAS server. Working examples in Microsoft .NET (including C# and Visual Basic .NET) as well as Windows PowerShell are also provided.

INTRODUCTION

SAS Integration Technologies is a product that allows you to build any application to integrate with SAS services. SAS Integration Technologies contains a set of application programming interfaces (APIs), and the components that support those APIs, which provide access to SAS features. These features include running programs, accessing data, working with metadata, and running stored processes.

ABOUT THE TERMINOLOGY

In some papers and documentation, SAS Integration Technologies is also known as Integrated Object Model (IOM). This informal name refers to the documented programming interfaces, which are designed for use as object-oriented components.

SAS Integration Technologies falls under a larger umbrella of the SAS Intelligence Platform, which refers to the products that provide compute services, data storage solutions, web infrastructure, and more. You might have SAS Integration Technologies as product within your SAS environment, or you might have it as just one component as part of a SAS Business Intelligence offering or any of several other SAS solutions.

For the purposes of this paper, it's not important to know which SAS solution or collection of products you have installed and configured. In general, if your organization uses SAS Enterprise Guide to access data or run programs on a SAS Workspace server, then you have SAS Integration Technologies.

ABOUT THE ARCHITECTURE

Figure 1 illustrates the main pieces of a SAS environment that includes SAS Integration Technologies. Desktop applications are on the client side (the top part of the diagram). These desktop applications include SAS products such as SAS Enterprise Guide, but they can also include any custom application that you build with desktop-based technology.

At a low level, client applications communicate with server applications (bottom part of the diagram) across the network using Transmission Control Protocol/Internet Protocol (TCP/IP). However, client applications are insulated from the low-level communications layer by using the SAS Integration Technologies Client for Windows. This component is automatically installed with all SAS desktop applications (such as SAS Enterprise Guide, SAS for Windows, and SAS Add-In for Microsoft Office). If necessary, you can download the client components from the **Demos and Downloads** section of support.sas.com.

The SAS Integration Technologies client components are compliant with the Component Object Model (COM), which means that you can use their APIs from most programming languages that are based on Windows. These include Visual Basic Script (VBScript), Windows PowerShell, Microsoft .NET, and Visual Basic for Applications (VBA) in Microsoft Office.

In this paper, we will look at examples in Windows PowerShell and Microsoft .NET.

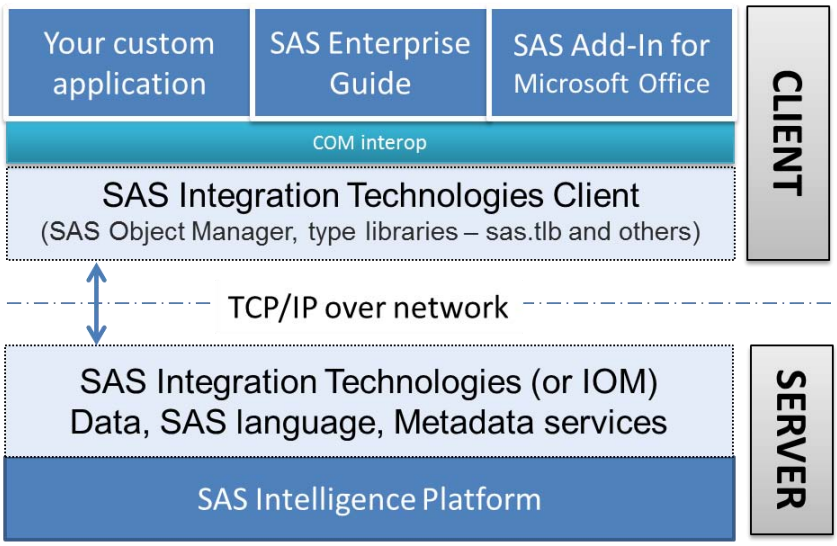


Figure 1. The SAS Integration Technologies Architecture

WORKING WITH WINDOWS POWERSHELL

The Windows PowerShell scripting engine is built into the most recent versions of Microsoft Windows, including Windows 7, Windows 8, Windows 2008 Server, and Windows 2012 Server. For Windows XP, you can download a compatible version of PowerShell from Microsoft.

If you are familiar with UNIX shells (such as Korn shell or its variants), you will probably be very comfortable with Windows PowerShell. Just like its UNIX predecessors, Windows PowerShell allows you to run commands and combinations of commands from an interactive console window. You can also write PowerShell scripts (saved as PS1 files), which allows you to combine the commands and programming logic to run more sophisticated operations.

You can run PowerShell commands in several ways:

- PowerShell has a command-line console (similar to a DOS prompt) where you can run commands and script files.
- PowerShell has a development environment, called the Interactive Scripting Environment (ISE), which enables you to write scripts, run commands, and see the script output. SAS users will find it familiar, because it is similar to a traditional SAS Display Manager session.
- You can create script files with a .PS1 file extension, and run them by invoking the `powershell` command.

ENABLING POWERSHELL TO RUN

Here is the most baffling part about getting started with PowerShell: by default, you cannot run PowerShell scripts on your system. This capability is disabled out of the box. You can run script commands from the console, but you cannot execute scripts that are saved as PS1 files. If you try, you see an error message similar to this:

```
File C:\Test\TestScript.ps1 cannot be loaded because the
  execution of scripts is disabled on this system.
  Please see "get-help about_signing" for more details.

At line:1 char:23+ .\Test\TestScript.ps1 <<<<
  + CategoryInfo
 : NotSpecified: (:) [], PSSecurityException
 + FullyQualifiedErrorId : RuntimeException
```

Presumably, this default policy setting is for your own safety. Fortunately, you can easily change the policy by using the `Set-ExecutionPolicy` command:

```
Set-ExecutionPolicy RemoteSigned
```

Run this command from the PowerShell console, select **Y** to confirm, and you can now run local PS1 files on your PC. ("RemoteSigned" indicates that local scripts will run, but scripts that you download from the Internet will run only if they are signed by a trusted party.) You can read more about setting these policies for running scripts in the *Windows PowerShell Owner's Manual*. (See References.)

CREATING OBJECTS WITH SAS OBJECT MANAGER

When working with the SAS Integration Technologies client, you need a way to create the objects that represent the connections to the SAS services. For that, you must use the SAS Object Manager.

The SAS Object Manager includes a class named ObjectFactory. As the name implies, the ObjectFactory class is where your subsequent objects are created. In our examples, we use the ObjectFactoryMulti2 class to create the connection to the SAS server for use in our applications. After creating that connection, you can use methods on the connection object to access the other services we need.

To get started with the SAS Object Manager in Windows PowerShell, use the `New-Object -ComObject` command.

```
$objFactory = New-Object -ComObject SASObjectManager.ObjectFactoryMulti2
```

Note:
SAS Object Manager supports a class named ObjectFactory as well as ObjectFactoryMulti2. Both classes support the same methods, but the ObjectFactoryMulti2 class allows you create multiple instances of the class in the same process. If you want to manage your objects with pooling, which can result in more efficient use of client resources, use ObjectFactory instead.

Before you can connect to a SAS server, you must define its attributes to SAS Object Manager. A SAS server has several attributes: a host name, TCP port number, and a Class Identifier. The Class Identifier is a 32-character global unique identifier (GUID) that indicates the type of SAS server that you expect to connect to.

The ServerDef class allows you to collect all of these attributes into a single object. Here is a Windows PowerShell example of defining the attributes needed for a SAS Workspace:

```
$objServerDef = New-Object -ComObject SASObjectManager.ServerDef
$objServerDef.MachineDNSName = "server.mycompany.com" # SAS Workspace node
$objServerDef.Port           = 8591  # workspace server port
$objServerDef.Protocol       = 2     # 2 = IOM protocol
# Class Identifier for SAS Workspace
$objServerDef.ClassIdentifier = "440196d4-90f0-11d0-9f41-00a024bb830c"
```

FINDING THE CORRECT CLASS IDENTIFIER

If you search support.sas.com, you might be able to find a lookup table for the Class Identifier values to map to the specific types of SAS servers. However, the most reliable source for these values, and usually the easiest to access, can come from SAS itself by way of PROC IOMOPERATE. Here is an example program:

```
proc iomoperate;
   list types;
quit;
```

Here's an excerpt of the SAS log, which contains the values:

```
SAS Metadata Server
    Short type name  : Metadata
    Class identifier : 0217e202-b560-11db-ad91-001083ff6836

SAS Stored Process Server
    Short type name  : StoredProcess
    Class identifier : 15931e31-667f-11d5-8804-00c04f35ac8c

SAS Workspace Server
    Short type name  : Workspace
    Class identifier : 440196d4-90f0-11d0-9f41-00a024bb830c
```

CONNECTING TO A SERVER

With an instance of an ObjectFactory class and the ServerDef class, you can now establish a connection to the server:

```
# create and connect to the SAS session
$objSAS = $objFactory.CreateObjectByServer(
                "SASApp",      # server name
                $true,
                $objServerDef, # used server definition for Workspace
                "sasdemo",     # user ID
                "Password1"    # password
                )
```

When this statement completes successfully, the `$objSAS` variable contains a connection to the SAS Workspace. With this connection, you can run SAS programs, read SAS data, and transfer files between the SAS Workspace and your application.

RUNNING A SAS PROGRAM WITH LANGUAGESERVICE

The SAS Workspace provides an API called LanguageService, which enables you to run SAS programs and retrieve the text-based results, including the SAS log and the SAS listing output. Here is a simple example using the `$objSAS` variable from the previous example:

```
$program = "ods listing; proc means data=sashelp.cars; run;"

# run the program
$objSAS.LanguageService.Submit($program);
```

These statements run the program, but they do not retrieve your results. To see the contents of the SAS log, use the `FlushLog` method as in this example:

```
# flush the log - could redirect to external file
Write-Output "LOG:"
$log = ""
do
{
  $log = $objSAS.LanguageService.FlushLog(1000)
  Write-Output $log
} while ($log.Length -gt 0)
```

The `FlushLog` method requires that you specify the number of bytes that you want to retrieve in a single call. In the preceding example, we used 1000. But what if your SAS log is greater than 1000 characters? That's the reason for the `do..while` loop. In this example, the PowerShell statements continues to retrieve and emit the SAS log content until the `FlushLog` method no longer returns any content (that is, while the length of the `$log` variable is greater than 0).

You can retrieve the SAS listing output in a similar way by using the `FlushList` method:

```
# flush the output - could redirect to external file
Write-Output "Output:"
$list = ""
do
{
 $list = $objSAS.LanguageService.FlushList(1000)
 Write-Output $list
} while ($list.Length -gt 0)
```

Figure 2 shows the Windows PowerShell development environment with an example of the results:

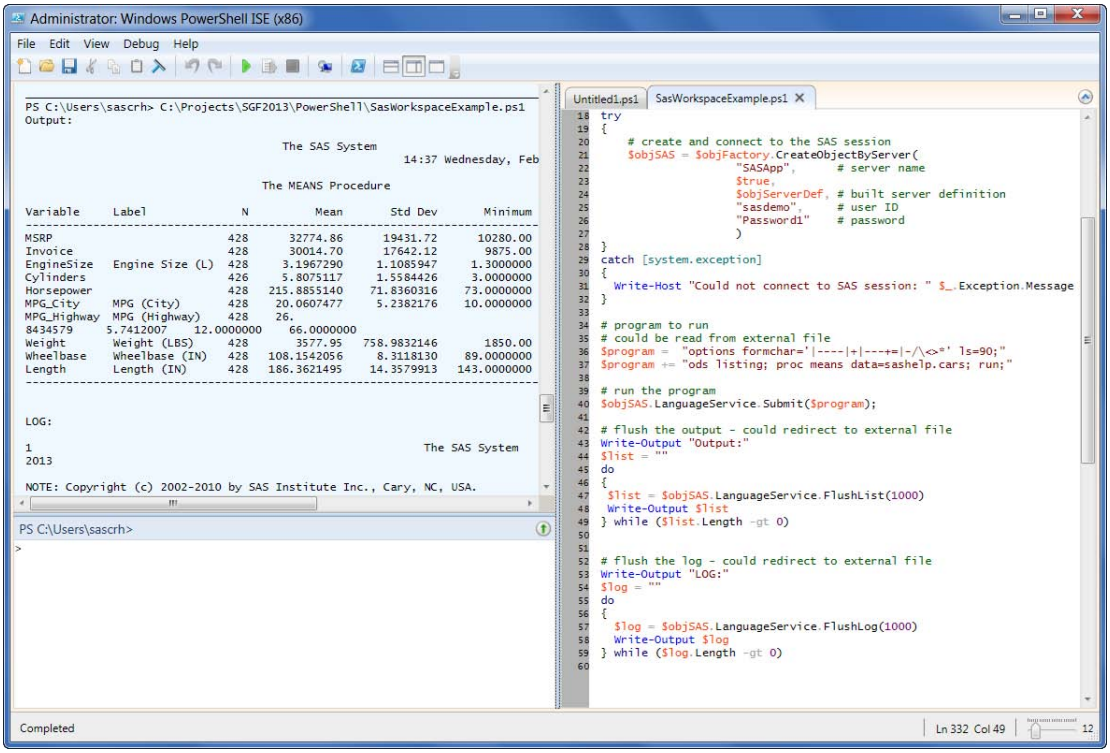


Figure 2. The Windows PowerShell ISE Application with Example Script and Results

DOWNLOADING A FILE WITH FILESERVICE

When you run a SAS program on a remote session, sometimes the expected result is more substantial than just a text-based listing. What if your program creates Output Delivery System (ODS) files such as HTML or graphics? To retrieve those results, you need to *download* the files from the SAS session to your local machine, where your application can access them. Consider this program, implemented in a PowerShell script, which creates an image with PROC SGPLOT:

```
# change these to your own SAS-session-based
# file path and file name
# Note that $destImg can't be > 7 chars
$destPath = "c:\outputFiles"
$destImg = "hist"

# local directory for downloaded file
$localPath = "c:\temp"

# program to run
# could be read from external file
$program =
       "ods graphics / imagename='$destImg';
        ods listing gpath='$destPath' style=plateau;
        proc sgplot data=sashelp.cars;
        histogram msrp;
        density msrp;
        run;"

# run the program
$objSAS.LanguageService.Submit($program);
```

When the program runs, it creates a file named `hist.png` in the `C:\outputFiles` folder within the SAS session.

The SAS Workspace provides the FileService API, which allows you to transfer file-based content between your local application and the SAS session.

As with most file-based operations in SAS, the FileService relies on the use of a SAS *fileref*, or the name that SAS uses to reference your file within a program. For a file download operation, these are the basic steps:

- Obtain a SAS reference to the file (FileService `AssignFileref` method).
- Tell SAS to open the remote file for reading (`OpenBinaryStream` method).
- Read the contents of the file into a local array of bytes (`Read` method, repeating in 1K increments).
- Write the contents into a local file (PowerShell objects).
- When completed, close the local and remote file handles, and unassign the SAS file reference (`Close` method and `DeassignFileref` method).

The following segment of a PowerShell script shows all of these steps, by example:

```
# now download the image file
$fileref = ""

# assign a Fileref so we can use FileService from IOM
$objFile = $objSAS.FileService.AssignFileref(
     "img", "DISK", "$destPath/$destImg.png",
     "", [ref] $fileref);

$StreamOpenModeForReading = 1
$objStream = $objFile.OpenBinaryStream($StreamOpenModeForReading)

# define an array of bytes
[Byte[]] $bytes = 0x0

$endOfFile = $false
$byteCount = 0
$outStream = [System.IO.StreamWriter] "$localPath\$destImg.png"
do
{
  # read bytes from source file, 1K at a time
  $objStream.Read(1024, [ref]$bytes)

  # write bytes to destination file
  $outStream.Write($bytes)
  # if less than requested bytes, we're at EOF
  $endOfFile = $bytes.Length -lt 1024

  # add to byte count for tally
  $byteCount = $byteCount + $bytes.Length

} while (-not $endOfFile)

# close input and output files
$objStream.Close()
$outStream.Close()

# free the SAS fileref
$objSAS.FileService.DeassignFileref($objFile.FilerefName)

Write-Output "Downloaded $localPath\$destImg.png: SIZE = $byteCount bytes"
```

CONNECTING TO A SAS METADATA SERVER

The SAS Metadata Server is used by most SAS applications to discover which SAS resources you can access. It serves as a central directory that lists all of the authorized users, the SAS application servers, the SAS libraries and tables for data, and much more. If you have worked with SAS metadata in programs before, you might have used PROC METADATA, which you can use to query metadata from within a SAS program. Any example that you have for PROC METADATA can also be adapted to run from Windows PowerShell.

The process for connecting to a SAS Metadata Server is similar to that for the SAS Workspace. You use the SAS Object Manager to create a ServerDef object, and then use the `CreateObjectByServer` method to establish the

connection. Compared to connecting to a SAS Workspace, the main differences are that in the ServerDef object, you name the SAS Metadata Server port (which is 8561 in a default installation) and the SAS Metadata Server value for the ClassIdentifier. Here is an example:

```
$objFactory   = New-Object -ComObject SASObjectManager.ObjectFactoryMulti2
$objServerDef = New-Object -ComObject SASObjectManager.ServerDef

# assign the attributes of your metadata server
$objServerDef.MachineDNSName  = "yournode.company.com"
$objServerDef.Port            = 8561  # metadata server port
$objServerDef.Protocol        = 2     # 2 = IOM protocol
# Class Identifier for SAS Metadata Server
$objServerDef.ClassIdentifier = "0217E202-B560-11DB-AD91-001083FF6836"

# connect to the server
# we'll get back an OMI handle (Open Metadata Interface)
try
{
$objOMI = $objFactory.CreateObjectByServer(
        "",
        $true,
        $objServerDef,
        "sasdemo",  # metadata user ID
        "Password1" # password
        )

Write-Host "Connected to " $objServerDef.MachineDNSName
}
catch [system.exception]
{
  Write-Host "Could not connect to SAS metadata server: " $_.Exception
  exit -1
}
```

CreateObjectByServer returns a connection to the SAS Metadata Server, sometimes called Open Metadata Interface (OMI), which is easily confused with IOM. In the preceding example program, the connection is in the PowerShell variable named `$objOMI`.

Most metadata operations require that you know the metadata ID for the repository in which your metadata resides. For most applications that's the Foundation repository. Even though it has a standard name ("Foundation"), the ID value can differ in each installation. So the next step is to use the `GetRepositories` method to find the ID value for the Foundation repository.

The `GetRepositories` method returns information within an XML-formatted structure, which you must then parse to get the information you need. Here is an example result from the `GetRepositories` method:

```
<Repositories>
 <Repository Id="A0000001.A5B4FV3C"
    Name="Foundation" Desc="" DefaultNS="SAS"/>
 <Repository Id="A0000001.A5Q23NT1"
    Name="BILineage" Desc="BILineage" DefaultNS="SAS"/>
</Repositories>
```

The nugget of information that you need from this example is `A0000001.A5B4FV3C`, which is the ID for the Foundation repository in this installation. Fortunately, Windows PowerShell provides an XML data type that makes it easy to filter and parse. The following code segment does the job:

```
# get list of repositories
$reps="" # this is an "out" param we need to define
$rc = $objOMI.GetRepositories([ref]$reps,0,"")

# parse the results as XML
[xml]$result = $reps

# filter down to "Foundation" repository
$foundationNode = $result.Repositories.Repository | ? {$_.Name -match "Foundation"}
$foundationId = $foundationNode.Id

Write-Host  "Foundation ID is $foundationId"
```

Now that you have the ID for the Foundation repository, you can begin to execute meaningful queries to retrieve objects. If you research PROC METADATA examples on support.sas.com, you will find that many queries have two components:

- a Type specification, indicating what type of object you want
- an XML template, which specifies the level of detail that you want to include in the XML result

The following example requests the complete collection of registered data tables (PhysicalTable objects) and includes details such as the SAS library, table names, column names, and column types:

```
$libTemplate =
     "<Templates>" +
     "<PhysicalTable/>" +
        "<Column SASColumnName=`"`" SASColumnType=`"`" SASColumnLength=`"`"/>" +
        "<SASLibrary Name=`"`" Engine=`"`" Libref=`"`"/>" +
    "</Templates>"

$libs=""

# Use GetMetadataObjects method
# Usage is similar to PROC METADATA, so you
# can look at PROC METADATA doc to get examples
# of templates and queries

# 2309 flag plus template gets table name, column name,
# engine, libref, and object IDs. The template specifies
# attributes of the nested objects.

$rc = $objOMI.GetMetadataObjects(
      $foundationId,
      "PhysicalTable",
      [ref]$libs,
      "SAS",
      2309,
      $libTemplate
  )

 # parse the results as XML
[xml]$libXml = $libs

Write-Host "Total tables discovered: " $libXml.Objects.PhysicalTable.Count
```

Once again, you can use PowerShell's ability to parse XML to reshape the result into native PowerShell objects, which can then be used as output from your script:

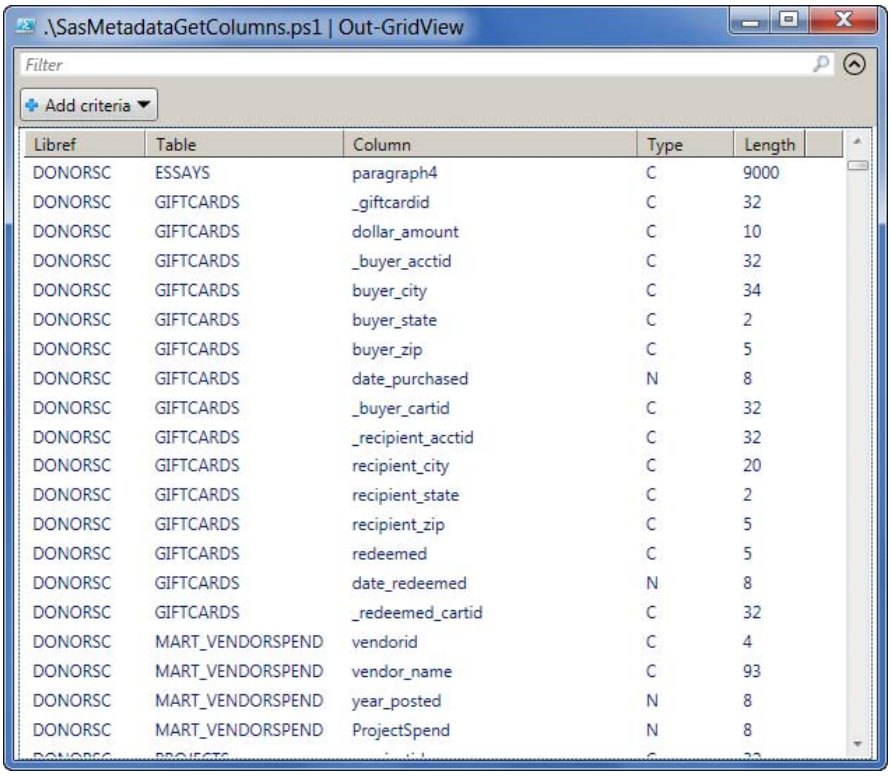
```
# Create output, which you can pipe to another cmdlet
# such as Out-Gridview or Export-CSV

# for each column in each table, create an output object
# (named $objCol here)
```

```
for ($i=0; $i -lt $libXml.Objects.PhysicalTable.Count; $i++)
{
  $table = $libXml.Objects.PhysicalTable[$i]
  for ($j=0; $j -lt $table.Columns.Column.Count ; $j++)
  {
    $column = $table.Columns.Column[$j]
    $objCol = New-Object psobject
    $objCol | add-member noteproperty -name "Libref"
                                      -value $table.TablePackage.SASLibrary.Libref
    $objCol | add-member noteproperty -name "Table" -value $table.SASTableName
    $objCol | add-member noteproperty -name "Column" -value $column.SASColumnName
    $objCol | add-member noteproperty -name "Type" -value $column.SASColumnType
    $objCol | add-member noteproperty -name "Length" -value $column.SASColumnLength

    # emit the object to stdout or other cmdlet
    $objCol
  }
}
```

When you run this script, the `$objCol` value is sent to **stdout**. You can easily pipe this output to other PowerShell "cmdlets" such as Out-Gridview or Export-CSV, which you can then use to study the data further. Figure 3 shows an example of tables and columns metadata within the PowerShell grid view component:



Libref	Table	Column	Type	Length
DONORSC	ESSAYS	paragraph4	C	9000
DONORSC	GIFTCARDS	_giftcardid	C	32
DONORSC	GIFTCARDS	dollar_amount	C	10
DONORSC	GIFTCARDS	_buyer_acctid	C	32
DONORSC	GIFTCARDS	buyer_city	C	34
DONORSC	GIFTCARDS	buyer_state	C	2
DONORSC	GIFTCARDS	buyer_zip	C	5
DONORSC	GIFTCARDS	date_purchased	N	8
DONORSC	GIFTCARDS	_buyer_cartid	C	32
DONORSC	GIFTCARDS	_recipient_acctid	C	32
DONORSC	GIFTCARDS	recipient_city	C	20
DONORSC	GIFTCARDS	recipient_state	C	2
DONORSC	GIFTCARDS	recipient_zip	C	5
DONORSC	GIFTCARDS	redeemed	C	5
DONORSC	GIFTCARDS	date_redeemed	N	8
DONORSC	GIFTCARDS	_redeemed_cartid	C	32
DONORSC	MART_VENDORSPEND	vendorid	C	4
DONORSC	MART_VENDORSPEND	vendor_name	C	93
DONORSC	MART_VENDORSPEND	year_posted	N	8
DONORSC	MART_VENDORSPEND	ProjectSpend	N	8

Figure 3. SAS Metadata about Libraries, Tables, and Columns in the PowerShell GridView

WORKING WITH MICROSOFT .NET

Microsoft .NET, when accessed through Microsoft Visual Studio, offers a full application development platform. This might be the best choice if you want to build an end-user application, complete with a user interface. You can download Express editions of Microsoft Visual C# or Visual Basic .NET for free from the Microsoft website. However, if you want the best productivity with the tool, consider investing in one of the professional editions.

Note: This discussion assumes that you are familiar with Microsoft .NET and Microsoft Visual Studio as a development tool. If you want to learn more about that topic, Microsoft offers many resources on their website: http://msdn.microsoft.com/beginner/.

To create a Microsoft .NET project that can connect to SAS Integration Technologies, follow these steps in Microsoft Visual Studio:

1. Create a new project of any type, appropriate for your objective (for example, a Windows Forms application, Console application, or Windows Presentation Foundation application).
2. In your Visual Studio project, add references to the following DLL files:
   - SASOManInterop.dll (for SAS Object Manager classes)
   - SASInterop.dll (for SAS Workspace classes)
   - SASOMIInterop.dll (for SAS Metadata Server classes)
   - SASIOMCommonInterop.dll (supports SASInterop.dll and SASOMIInterop.dll; not used directly in these examples)

   You can find these files in the installation folder for the SAS Integration Technologies client. For example: `C:\Program Files\SAS\Shared Files\Integration Technologies` or `C:\Program Files\SASHome\x86\Integration Technologies`.

Figure 4 shows an example of a Microsoft Visual Studio project with the necessary references to use a SAS Workspace. (Note: This example project does not include use of a SAS Metadata Server.)

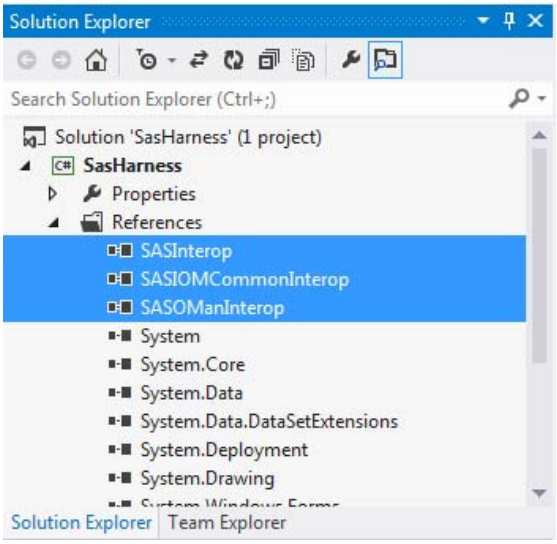


Figure 4. The Solution Explorer View of a Project with SAS Interop References

ABOUT THE SAMPLE APPLICATION

As a companion to this paper, I have prepared a full sample application with these features:

- connects to a SAS Workspace session using a server that you define in a dialog box.
- allows you to connect to a local instance of SAS – no configuration required.
- features three windows: a Program Editor, a log viewer, and a listing viewer. (Does that seem familiar?)
- allows you to run a SAS program on a background thread, keeping the main user interface responsive.
- retrieves the SAS log and listing output, and colors each line of output as appropriate (errors, warnings, notes, page boundaries).

Figure 5 shows a screen capture of the application, named SAS Program Harness. It shows a program, the SAS log, and the listing output:

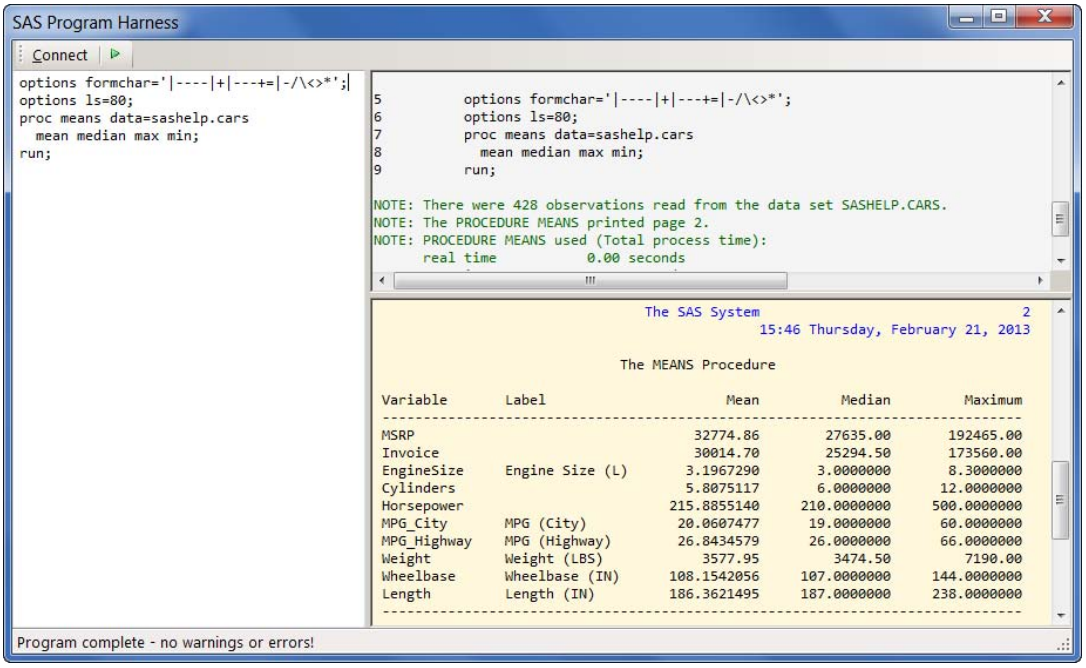


Figure 5. Example of the SAS Program Harness Application in Action

The sample application was built using Microsoft Visual Studio 2012. You can also use Microsoft Visual C# Express (free) to view, modify, and build the project. See the section titled "AccessING the Code Examples" for links to download the project files.

CONNECTING TO A SAS WORKSPACE

The sample application provides a login window in which you can supply the necessary values for the SAS Workspace server definition. Figure 6 shows an example of the login window:

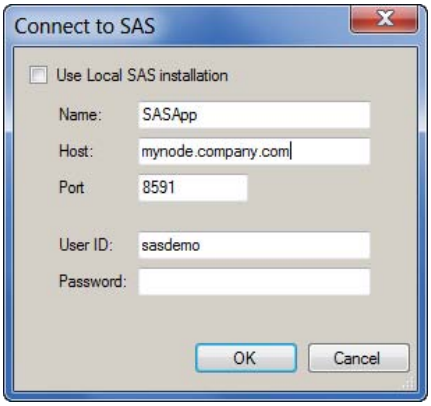


Figure 6. Login Screen for Sample Application

When you complete the fields in this form and click **OK**, the application uses those values to create a `ServerDef` object and then connect to a SAS Workspace. Here is the code segment in C#:

```
// Connect using the IOM Bridge (TCP) for remote server
 SASObjectManager.IObjectFactory2 obObjectFactory =
     new SASObjectManager.ObjectFactoryMulti2();
 SASObjectManager.ServerDef obServer =
     new SASObjectManager.ServerDef();
 obServer.MachineDNSName = Host;
 obServer.Protocol = SASObjectManager.Protocols.ProtocolBridge;
 obServer.Port = Convert.ToInt32(Port);
```

```
obServer.ClassIdentifier = "440196d4-90f0-11d0-9f41-00a024bb830c";
_workspace = (SAS.Workspace)obObjectFactory.CreateObjectByServer(
    Name, true,
    obServer,
    UserId,
    Password);
```

Note that these statements are very similar to those that we used in the Windows PowerShell examples. The C# syntax is slightly different than PowerShell, but the SAS Integration Technologies objects are the same.

RUNNING A SAS PROGRAM ON A BACKGROUND THREAD

As in the PowerShell example, you can use the `Submit` method on the LanguageService class to run a program. Some SAS programs can take a long time to run. In this event, it's better to take steps that keep the application responsive to mouse clicks and other interactions, even as the program runs. To achieve this, you must submit the SAS program on a *background* thread – that is, a different thread in the Windows process that is not dedicated to updating the UI.

This C# code shows how to run a SAS program in the background with the .NET BackgroundWorker object:

```
private void RunProgram()
{
    if (activeSession != null && activeSession.Workspace != null)
    {
        // if we don't use a background thread when running this program
        // we'll BLOCK the UI of the app while a long-running
        // SAS job completes.
        // This allows us to keep the UI responsive.
        BackgroundWorker bg = new BackgroundWorker();
        bg.DoWork += bg_DoWork;
        bg.RunWorkerCompleted += bg_RunWorkerCompleted;

        statusMsg.Text = "Running SAS program...";
        bg.RunWorkerAsync();
    }
}

void bg_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    FetchResults();
}

void bg_DoWork(object sender, DoWorkEventArgs e)
{
    activeSession.Workspace.LanguageService.Submit(txtProgram.Text);
}
```

The `RunProgram` method defines the `BackgroundWorker` object and tells it which method (`bg_DoWork`) contains the instructions to run. The `RunWorkerAsync()` call kicks off the program asynchronously, leaving the UI free to process other commands while the end user waits.

When the `Submit` method finishes processing, the `BackgroundWorker` raises an event that is handled in `bg_RunWorkerCompleted`. At that point, you know that the SAS program is complete and you can retrieve the results.

This is a very simple example, and it lacks some precautions that you should consider for a production application:

- Even while the UI remains responsive, the sample application does not contain logic that prevents you from trying to run *another* program while a program is still running. You should add checks to prevent the `RunProgram` code from being reentered inappropriately.
- In a Windows application, you cannot update the user interface from code that is running on a background thread. Note that in the sample, the `bg_DoWork` method does not contain any statements that disable or enable buttons, issue status messages, or interact with the UI at all. If you do want to issue status updates while the program runs, you need to marshal those instructions to the UI thread using the .NET `BeginInvoke` method. That topic

is beyond the scope of this paper, but you can find documentation about threading and `BeginInvoke` on the Microsoft website.

FETCHING THE COLOR-CODED SAS LOG AND LISTING

In the PowerShell examples we used the `FlushLog` and `FlushList` methods to retrieve the text output from the SAS program. The LanguageService class offers alternative methods to retrieve a richer set of output, which contains enough information to provide coloring cues for how to highlight certain lines, such as the parts of the log that shows a SAS error or warning.

The methods are `FlushLogLines` and `FlushListLines`. Instead of retrieving a specified number of bytes, these methods retrieve a specified number of text lines in an array of strings. In addition, these methods retrieve arrays of values that indicate whether the line was a warning, error, note, or other special line. By matching up the array indices for each line, you can write code that provides the special color treatment that SAS users are accustomed to.

The following C# code shows this for the log window in the sample application. The output of FlushLogLines contains three arrays that are synchronized by index value. That is, the value of `lineTypes[0]` corresponds to the treatment for `lines[0]`, and so on.

```
bool hasErrors = false, hasWarnings = false ;

// when code is complete, update the log viewer
Array carriage, lineTypes, lines;
do
{
    activeSession.Workspace.LanguageService.FlushLogLines(1000,
        out carriage,
        out lineTypes,
        out lines);
    for (int i = 0; i < lines.GetLength(0); i++)
    {
        SAS.LanguageServiceLineType pre =
            (SAS.LanguageServiceLineType)lineTypes.GetValue(i);
        switch (pre)
        {
            case SAS.LanguageServiceLineType.LanguageServiceLineTypeError:
                hasErrors = true;
                txtLog.SelectionColor = Color.Red;
                break;
            case SAS.LanguageServiceLineType.LanguageServiceLineTypeNote:
                txtLog.SelectionColor = Color.DarkGreen;
                break;
            case SAS.LanguageServiceLineType.LanguageServiceLineTypeWarning:
                hasWarnings = true;
                txtLog.SelectionColor = Color.DarkCyan;
                break;
            case SAS.LanguageServiceLineType.LanguageServiceLineTypeTitle:
            case SAS.LanguageServiceLineType.LanguageServiceLineTypeFootnote:
                txtLog.SelectionColor = Color.Blue;
                break;
            default:
                txtLog.SelectionColor = txtLog.ForeColor;
                break;
        }
        // add the line to the log window, using the current color
        txtLog.AppendText(string.Format("{0}{1}",
            lines.GetValue(i) as string, Environment.NewLine));
    }
}
while (lines != null && lines.Length > 0);
```

Note:
When determining how to highlight SAS log output, it's not adequate to simply search the log text for the words

"ERROR" or "WARNING". When you run SAS in non-English languages, those keywords might be translated into a local language. Also, those keywords can occur in the text of other messages, even if the message is not part of an error or warning.

CLOSING THE SAS SESSION

It's important to manage the lifetime of the SAS session in a responsible way. If you create a SAS Workspace connection and use it within your application, you should call the `Close` method on the SAS.Workspace object when your work is complete. This is especially true if your application is designed to stay running for a long time, even when the SAS session is not in use.

It is also a good practice to set the SAS.Workspace object to **null** (or **Nothing** in Visual Basic) when you are no longer using it. This allows the .NET run time to clean up any associated SAS Object Manager connections and objects even if your application continues to stay active.

Every SAS Workspace object that remains active represents a SAS process on the host where the SAS sessions are run. SAS administrators usually do not like to see "zombie" SAS sessions, consuming resources while they are not being used.

The following C# code provides an example of closing the SAS Workspace:

```
/*  defined as
      private SAS.Workspace _workspace;
*/

if (_workspace != null)
    _workspace.Close();
_workspace = null;
```

ACCESSING THE CODE EXAMPLES

The complete code samples, including Windows PowerShell scripts and Microsoft .NET projects, are available from the sasCommunity.org article for this paper:

http://www.sascommunity.org/wiki/Create_Your_Own_Client_Apps_Using_SAS_Integration_Technologies

The Microsoft .NET examples are organized into projects, which can be built with Microsoft Visual Studio. Microsoft offers express editions of its development tools: Microsoft Visual C# Express and Microsoft Visual Basic Express. These are available from the Microsoft website: http://www.microsoft.com/visualstudio.

CONCLUSION

SAS Integration Technologies provides flexible access to the SAS servers that are configured in your enterprise. It's a proven and supported technology, and has been used for many years within SAS Enterprise Guide and SAS Add-In for Microsoft Office. Windows PowerShell and Microsoft .NET are two approaches to integrate those capabilities into your own custom applications. With a small investment in development tools and a bit of imagination, you can build impressive applications that bring the power of SAS to your desktop users.

RECOMMENDED READING

Hemedinger, Chris. 2012. *Custom Tasks for SAS® Enterprise Guide® Using Microsoft .NET.* SAS Institute Inc., Cary, NC. This book provides a guide for creating your own custom apps to host within SAS Enterprise Guide, which provides a number of APIs and services that make the job easier.

REFERENCES

Désilets, Karine. 2012. "SAS® IOM and Your .NET Application Made Easy." *Proceedings of the SAS Global Forum 2012 Conference*. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/proceedings12/017-2012.pdf.

Microsoft Corporation. *Windows PowerShell Owner's Manual.* Available at http://technet.microsoft.com/en-us/library/ee221100.aspx.

CONTACT INFORMATION

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

Chris Hemedinger
SAS Campus Drive
SAS Institute, Inc
Chris.Hemedinger@sas.com
Blog: http://blogs.sas.com/content/sasdummy/
Twitter: http://twitter.com/cjdinger