Paper 235-30

# Catch the Stream: Stored Processes, ODS, and Java

Jeff Wright, ThotWave Technologies LLC, Cary, NC

## ABSTRACT

Wouldn't it be great if there was a straightforward way of taking SAS® programs and making their output available on demand via a web browser? Sure, you probably know that you can use ODS to format SAS output as static HTML pages that can be copied to a web server, but what if your program uses data sources that change every day, or every hour? Building on demand, web-based reports via SAS potentially requires an array of technologies ranging from HTML to web programming to SAS to databases. Some approaches require one programmer to master all these skills, but stored processes bring an appealing division of labor to this picture. A stored process is just a parameterized SAS program that is callable via the SAS® Integration Technologies Integrated Object Model (IOM) API. By encapsulating SAS logic in this way, the SAS programmer and web developer can each focus on the areas they know best.

This tutorial will introduce how SAS stored processes can be called from a web application, using Java as the example programming language. The IOM calls for stored process execution will be described, as well as the fundamentals behind using socket communication to stream back ODS results. Practical tips will be offered on developing, testing, and deploying applications using this technology. Stored processes have received a lot of publicity from SAS with respect to their support in SAS9, but are available and useful in SAS 8.2 as well.

## INTRODUCTION

This tutorial explores how SAS stored processes can be used to stream analytic results formatted with ODS back to another programming language such as Java. Let's define these terms:

- A *stored process* is a SAS program that is stored on a server and can be executed as required by requesting applications (quoting the documentation).
- *Streaming* is a communication technique where we make SAS think it's writing to a file, but the output is actually being sent back to the requesting application via a network socket.
- *ODS* is the Output Delivery System, the standard Base SAS facility for formatting procedure output.

This tutorial may be interesting to you if you

- … are looking for a good way to make SAS analytics available on-demand to a wide audience via a web interface.
- … have a SAS/IntrNet® application and want to migrate it to some more current technology.
- … want to get ready for SAS9, and know that stored processes are a part of that.
- … have seen SAS9 stored process demos using products like the Add-In for Microsoft Office, and want to know how that works.
- … just automatically devour all the information about SAS you can get your hands on.

We'll start by looking at a simple SAS report, and examine how to convert it to a stored process and run it under some of the pre-built SAS9 clients. Then we'll look at one of the Integration Technologies APIs for calling stored processes from Java, one that works under SAS 8.2 as well as SAS9. With that understanding to build on, we'll go back to the SAS side of things and talk about how ODS and streaming actually work. Then we'll wrap up by taking a quick tour of some advanced topics, and also provide some pointers to the new SAS9 API for stored processes.

## EXAMPLE REPORT

For this tutorial, we'll be referring to the sample *Shoe Sales by Subsidiary* report shown in Listing 1. This report starts life as a SAS program. ODS is used to format the two output tables as HTML. It uses standard sample data in the SASHELP library, so you can probably run it on your own system.

Note that a macro symbol &REGION is used to subset the shoe sales data. This &REGION parameter can be manually adjusted, the report can be run, and the output can be shared with others by copying it to a web server.

```
%let REGION=Pacific;

ods html body="shoe_sales.html";

title Shoe Sales by Subsidiary;
title2 Region: &REGION;

proc report data=sashelp.shoes(where=(region="&REGION")) nowd;
    column subsidiary sales;
    define subsidiary / group;
    define sales / analysis sum;
run;

title Shoe Sales Detail;
title2 Region: &REGION;

proc report data=sashelp.shoes(where=(region="&REGION")) nowd;
    column subsidiary product stores sales;
    define subsidiary / order;
    define product / order;
    define sales / analysis sum;
run;

ods html close;
```

**Listing 1**

## CONVERTING THE REPORT TO A STORED PROCESS

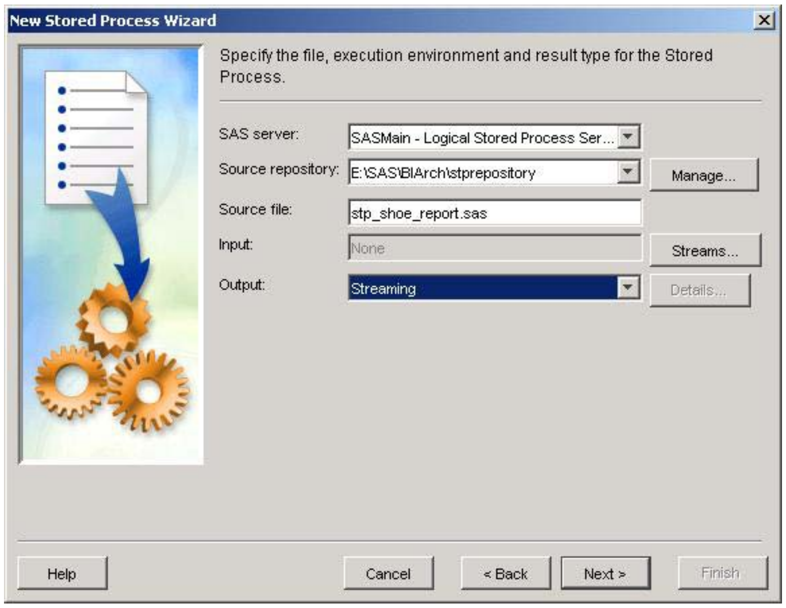
You may have heard that with SAS9, stored processes are all the rage. Could the shoe sales report be converted to a stored process? It certainly could, with these simple steps:

- Every stored process must have a special marker comment that identifies it as a stored process. The marker comment is this line, which should be included between the %LET and the first ODS statement:
  *ProcessBody;

- We can convert our existing macro symbol &REGION to a stored process parameter. The recommended coding practice is to replace the %LET with
  %global REGION;

- Our initial report contains starting and ending ODS statements. With stored processes, we let Integration Technologies take care of those details by replacing them with two macro calls:
  %stpbegin
  %stpend

Using SAS9 there are several pre-packaged ways to call our stored process, but before we can use those tools we have to register the stored process in – you guessed it – the Metadata Server. To do this, we use the SAS Management Console. The Metadata Server needs to know the location of our file, the type of output, and the definition of the parameters that it takes. This information is specified in a wizard (Figure 1).

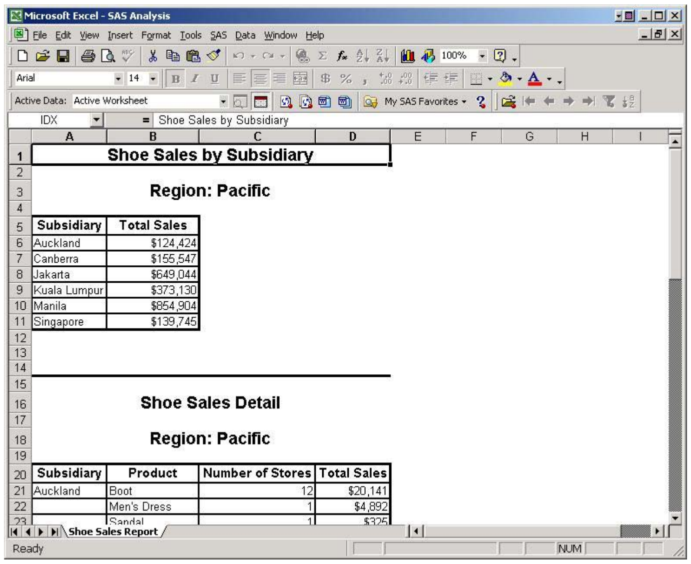


Specify the file, execution environment and result type for the Stored Process.

SAS server: SASMain - Logical Stored Process Ser...
Source repository: E:\SAS\BIArch\stprepository
Source file: stp_shoe_report.sas
Input: None
Output: Streaming

**Figure 1**

Once the registration is completed, the stored process is available to external tools and programs. One such tool is the Add-In for Microsoft Office (or AMO for short). AMO allows a Microsoft Word or Excel user to lookup stored processes, run them, and receive the results directly in an Office document (Figure 2).


Microsoft Excel - SAS Analysis
File Edit View Insert Format Tools SAS Data Window Help
Active Data: Active Worksheet
My SAS Favorites
IDX
Shoe Sales Report
Ready
NUM

Shoe Sales by Subsidiary

Region: Pacific

| Subsidiary | Total Sales |
|---|---|
| Auckland | $124,424 |
| Canberra | $155,547 |
| Jakarta | $649,044 |
| Kuala Lumpur | $373,130 |
| Manila | $854,904 |
| Singapore | $139,745 |

Shoe Sales Detail

Region: Pacific

| Subsidiary | Product | Number of Stores | Total Sales |
|---|---|---|---|
| Auckland | Boot | 12 | $20,141 |
| | Men's Dress | 1 | $4,892 |
| | Sandal | 1 | $325 |

**Figure 2**

Stored processes can also be surfaced via the web, using either the Web Infrastructure Kit or Information Delivery Portal. After the user selects a stored process, they are prompted for the parameters and the output appears right in their browser.

Compared to our initial process of manually publishing reports, this has a lot of advantages. End users can request the report whenever they want and get the latest results at that moment.

At this point, you may be wondering:

- How'd they do that?
- Can I call stored processes from my own custom program?
- Can I do it in SAS 8.2?

## BASIC STORED PROCESS CALL FROM JAVA

You can definitely use the Integration Technologies APIs to call stored processes from your own programs. You can even do this with SAS 8.2, using a simplified version known as *direct interface* stored processes. Knowing that direct interface stored processes can work in SAS 8.2, you won't be surprised to learn that they don't make use of the Metadata Server. In this section, we'll work through how to call our shoe sales report from Java using the direct interface API.

### OVERVIEW

Before we start looking at actual Java code, let's preview of what we need to do.

- The Integrated Object Model (IOM) is our API to remotely call SAS from Java (or other languages).
- When we connect to SAS, we get a Workspace object that represents our SAS session.
- We obtain other service objects from the Workspace.

*Streaming* will be used to return the output of SAS. What this means is that SAS will think it's writing output to a file, but the output will really be going over the network back to our Java program. Our program will have to set that up before calling the stored process.

### SAMPLE JAVA CODE

To call a stored process, you have to obtain a reference to an IStoredProcessService object and tell it:

- The repository location, which is the directory that holds the stored process. This is specified in URL format, as in "file:/some/directory".
- Whether to call the stored process asynchronously or synchronously. If we make the call asynchronous, Java will not wait for the stored process to finish.
- The stored process file name.
- The stored process parameters, a string in the form "name1=value1 name2=value2 …"

Listing 2 contains the Java code for a method that sets up and executes a single stored process call. This method is passed three arguments:

- *stpName* – The file name of the stored process.
- *params* – A java.util.Properties object holding (String) parameter names and values.
- *workspace* – The IOM IWorkspace object representing a SAS session.

```
private SocketListener executeImpl(
        String stpName,
        Properties params,
        IWorkspace workspace)
throws IOException, GenericError
{
    // create a socket for SAS to send result stream
    // SocketListener is an AppDev Studio utility
    SocketListener socket = new SocketListener();
    int port = socket.setup();
    socket.start();

    // get IOM objects from Workspace
    ILanguageService languageService = workspace.LanguageService();
    IStoredProcessService stpService =
    languageService.StoredProcessService();

    // get stored process call info
    String repository =
        LocalEnvironment.getProperty("stp.repository");
    String stpParams = buildParamString(params);
    Log.debug("repository = "+repository);
    Log.debug("stpName = "+stpName);
    Log.debug("stpParams = "+stpParams);

    // define fileref for socket and issue startup statements
    initWorkspace(languageService, port);
    // call stored process
    stpService.Repository(repository);
    languageService.Async(false);
    Log.debug("calling stp");
    stpService.Execute(stpName, stpParams);

    // send SAS log to web app log
    Log.debug("checking log");
    dumpLog(languageService);
    return socket;
}
```

**Listing 2**

The *executeImpl()* method in Listing 2 makes use of three IOM interfaces: IWorkspace, ILanguageService, and IStoredProcessService.

The SocketListener class also comes from SAS, with AppDev Studio. This class is a key part of the streaming approach, because it actually open the network socket and listens for the streaming result. As Listing 2 shows, we can ask this object for its port number. Under the covers this object creates a thread to listen for data coming in to the port. Because this object has its own thread, we can call the stored process synchronously.

The code also makes use of two utility classes that were created as part of this sample application. *LocalEnvironment* is used to externalize environment settings into a properties file, information such as the location of the stored process repository. The *Log* class is a very simple logger.

Other details are implemented as methods; these will be shown next.

## BUILDING THE PARAMETER STRING

A stored process can take zero or more parameters. As mentioned previously, Java is responsible for concatenating these together in "name=value" form, with space delimiters.

What if a parameter value has embedded spaces, as *Eastern Europe* (a region in our data)? To guard against that, we will add double quotes around the values in our parameter string. Listing 3 shows the *buildParamString()* method.

```
private String buildParamString(Properties params)
{
    StringBuffer paramBuff = new StringBuffer();
    Iterator iter = params.keySet().iterator();
    while (iter.hasNext())
    {
        String key = (String) iter.next();
        paramBuff.append(key);
        paramBuff.append('=');
        paramBuff.append('\"');
        paramBuff.append(params.getProperty(key));
        paramBuff.append('\"');
        paramBuff.append(' ');
    }
    String stpParams = paramBuff.toString();
    return stpParams;
}
```

**Listing 3**

## INITIALIZING THE SAS SESSION

There are a couple of steps that need to be done to set up our SAS session before actually calling the stored process. In our first pass (Listing 2) these details were hidden in the workings of the *initWorkspace()* method. We'll now dig into those details.

The SocketListener we created is monitoring a socket, and we need to tell SAS its address. It turns out that a FILENAME statement can be used to define a fileref that corresponds to a network socket instead of a physical file. The code in Listing 4 below builds just such a FILENAME statement. Note that:

- The fileref is _WEBOUT, which is the fileref that SAS9 uses. This should look familiar to SAS/IntrNet programmers – SAS is trying to make it easy for you to migrate to stored processes!
- We've specified the network hostname for the socket as *localhost*. That works as long as our Java code is on the same computer as SAS. In a production application you'd want to code this a little more carefully. Some servers can even have more than one network interface, and hence more than one network address.

The FILENAME statement is not the only SAS code that this method puts together. Because the SAS 8.2 object spawner does not allow an autoexec to be defined, our code has the ability to submit an arbitrary startup statement. This is intended to accommodate the kinds of things people usually do in an autoexec: define the autocall path, define global symbols, and so on.

```
private void initWorkspace(ILanguageService languageService, int port)
throws IOException, GenericError
{
    // create fileref to my server socket
    // CHEAT ALERT: using localhost as host name for socket!
    StringBuffer block = new StringBuffer();
    block.append("filename _WEBOUT SOCKET ");
    block.append('\'');
    block.append("localhost");
    block.append(':');
    block.append(port);
    block.append('\'');
    block.append(';');
    block.append("\n");

    String startupStatement =
        LocalEnvironment.getProperty("stp.startupStatement");
    if (startupStatement != null)
        block.append(startupStatement);

    Log.debug("submitting "+block.toString());
    languageService.Submit(block.toString());
}
```

**Listing 4**

**GETTING THE SAS LOG**

The last remaining call from Listing 2 that has not yet been explained is the *dumpLog()* method. When you're programming Integration Technologies, you would do well to build diagnostics into your code from the very beginning. Not only will this be important when your code is in production – it's likely to solve a number of the problems you encounter during development as well.

The code to get the SAS log (Listing 5) is called after the stored process is executed, and appends it to the same log file as the Java code uses. It uses the ILanguageService method *FlushLog()*. In your code you may also want to explore a similar method *FlushLogLines()*, which is harder to use but returns a line type indicator that explicitly marks WARNING and ERROR messages in the log.

```
private void dumpLog(ILanguageService languageService)
throws GenericError
{
    int count = 0;
    String log = null;
    Log.sas("\nLOG START\n\n");
    do
    {
        log = languageService.FlushLog(BUFFSIZE);
        Log.sas(log);
        count++;
    }
    while (log.length() == BUFFSIZE);
    Log.sas("\nLOG END (" + count + ")\n");
}
```

**Listing 5**

#### WHAT YOU NEED

In order to run the code we've just seen, you need the following infrastructure:

- SAS 8.2 (or better).
- SAS Integration Technologies. You must also have the Object Spawner configured and running.
- You need Java. This sample application was created as a web application to run with Java 1.4 and Tomcat 4.1.30, a free application server.
- Your Java application must have the Java libraries for Integration Technologies in the form of JAR (Java Archive) files. Our sample used the most recent libraries from AppDev Studio 3.0: sas.core.jar, sas.svc.connection.jar, and sas.servlet.jar. Note that this last JAR contains the SocketListener class, which is part of AppDev Studio rather than Integration Technologies. We'll discuss this library more later in the paper.

## SAS STORED PROCESS CODE

Now that we've covered how stored processes are called using the direct interface stored process API, let's return to the SAS code for a few more aspects of that side of things.

#### UNLOCKING THE MYSTERIES OF %STPBEGIN AND %STPEND

When we converted our shoe sales report to run as a stored process in SAS9, we replaced our ODS statements with calls to the SAS-supplied macros %stpbegin and %stpend. These macros are supposed to bracket any code that produces output, to handle details of communications with the caller and configure ODS formatting.

That doesn't sound too hard! Our original report already handled ODS, and that was only a matter of two lines of code. Based on our Java work we know we'll be expecting Java to define a fileref _WEBOUT as our ODS "file". While we're at it, why don't we kick it up a notch? Let's assume the Java client can also provide a parameter _ODSDEST with a value of either HTML or PDF, because we know we can easily produce PDF output with ODS.

Listing 6 shows the resulting %stpbegin macro

```
%macro stpbegin;

%if &_ODSDEST eq PDF %then
%do;
    ods pdf
        file=_WEBOUT
        NOTOC
        ;
%end;
%else    /* default to HTML */
%do;
    ods html
        body=_WEBOUT(no_top_matter no_bottom_matter)
        stylesheet=(URL="include/site.css");
        ;
%end;

%mend stpbegin;
```

**Listing 6**

For PDF, we specified the NOTOC option, because we don't want SAS to create a table of contents for our report.

For HTML, we specified two options:

- *(no_top_matter no_bottom_matter)* causes the HTML to generate as a page fragment rather than a complete page. This is frequently useful when calling SAS from a web application. The web application will probably supply the HTML tags that begin and end the HTML page, and we just want SAS to supply some content for the middle.
- The *stylesheet* option tells ODS to create HTML that uses an external stylesheet. Of course, since we specified no_top_matter SAS will not actually create the link to the stylesheet, but this still changes the HTML that's generated. Instead of a lot of detailed formatting, that is left to the stylesheet to do.

The %stpend macro will be left to the reader's imagination!

### STORED PROCESS DEVELOPMENT

Let's now consider what the development process is like for stored processes. Stored processes share some characteristics with both SAS macros and SAS programs. Like macros, they are parameterized. Like a program, they are "open code".

One practice that is useful in stored process development is to create a *test harness* for each stored process. A test harness emulates the environment that a stored process will be called in for debugging purposes. The simplest possible test harness would be a file that has a %LET for each stored process parameter and a %INCLUDE of the stored process file. Some other things that might be useful to do in a test harness might be:

- Stub out the %stpbegin and %stpend macros with something simpler than the "real" versions.
- Supply a controlled subset of test data to the stored process rather than "real" data.

Creation of a test harness may seem like extra work, but it can easily pay for itself in time and mistakes saved.

## ADVANCED TOPICS

This section touches on an assortment of advanced topics.

### NON-HTML FORMATS

When we wrote our own %stpbegin macro, we built in support for PDF formatting as well as HTML. One of the appeals of ODS is that it supports a number of useful formats for output. Let's look at some code on the Java side to take advantage of this.

Listing 7 below is an excerpt from a Java servlet that calls the stored process logic we previously discussed in Listing 2. Based on a request parameter from an HTML form, it sets the stored process parameter _ODSDEST to either HTML or PDF. For HTML output, it forwards to a JSP page that will include the SAS-generated HTML page. For PDF, it will send the PDF bytes directly back to the browser. The content-type and content-disposition are set in the response so the PDF is treated as an attachment, and the browser will prompt the user to either open or save the document.

```
private void doView(HttpServletRequest request,
        HttpServletResponse response)
    throws ServletException, IOException
{
    Properties params = new Properties();
    String region = request.getParameter("REGION");
    params.setProperty("REGION", region);
    String outputType = request.getParameter("_ODSDEST");
    params.setProperty("_ODSDEST", outputType);
    Log.debug("output type = "+outputType);

    String stpName = "stp_shoe_report";

    if (outputType.equals("HTML"))
    {
        CharArrayWriter resultBuff = new CharArrayWriter();
        mySTP.execute(stpName, params, resultBuff);
        String odsOutput = resultBuff.toString();

        request.setAttribute("odsOutput", odsOutput);
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("result.jsp");
        dispatcher.forward(request, response);
    }
    else if (outputType.equals("PDF"))
    {
        response.setContentType("application/pdf");
        response.setHeader("Content-disposition",
                "attachment;filename=output.pdf");
        mySTP.execute(stpName, params, response.getOutputStream());
        response.getOutputStream().flush();
    }
}
```

**Listing 7**

**CHART OUTPUT**

Chart output from stored processes presents a special challenge because of the way that image tags work in HTML. When a web server returns a web page with images, it actually initiates a conversation between the browser and web server. Each image tag causes the browser to go back to the server and make a separate request for the image.

Contrast that with the way SAS would produce a report that contains text and graphics. Both the text and graphics are produced at the same time. This means that if a web application is requesting a stored process to create mixed text and graphics, the web application needs to temporarily hold onto the graphics until the browser comes back and asks for them.

There are a number of ways to approach this situation:

- You could split your stored process into multiple parts, one to handle the text output and one to handle each chart. This will work, but it's likely to be inefficient.
- You could always use a format like PDF, which doesn't require this kind of back and forth because it's self-contained. However, users are likely to complain about the delay in opening PDFs.
- You could write the graphics to a temporary directory. For this to work, the web server and SAS must be running on the same computer, or there must be a networked file system to which both have access. Also, you'll want to develop some cleanup logic for this directory.
- You could use the graphics catalog to store the chart output, and PROC GREPLAY to render it.
- You could use client side graphics. SAS/Graph includes ActiveX controls and Java applets for client-side charting, as opposed to server-generated images. All the parameters to these charts are returned in a single HTML file, so there is no back and forth. However, there are some support issues with client-side charts running in the browser which may make this unattractive to some organizations.

**DRILLDOWN REPORTS**

The shoe sales report includes a summary table and a detail table. Now that we've made this an on-demand report, it would be appealing to make the details available as a drilldown. The report could be broken into two stored processes, and the summary report could have drilldown links to the detail report.

This can be done, but a protocol must be worked out so that the SAS side has knowledge of the URL address in which it's being called. Usually you would have the Java side supply a base URL, and have the SAS code add additional query string parameters to form the drill down links. Many SUGI papers have been written on ODS and drilldown, so we won't cover that any further here.

**MORE ON PARAMETERS**

There are actually a lot of considerations related to parameter handling, above and beyond what was discussed above. Here is a survey of some of the issues:

- Work with your users to determine what kind of default logic would make sense for their usage. If they return to a given report, do they want to see the same parameters as the last time they ran it? Or do they want common parameter values to be shared across reports? Or do they want a way to bookmark their most common parameters?
- In our basic code above, we assumed parameters were single valued. What if we wanted to pass a list of values as a parameter? The SAS9 convention is to use a numeric suffix, and the "0th" parameter gives a parameter count. For example, if the caller wants to pass the values "Egg Drop Soup" and "Sesame Chicken" to the LUNCH parameter, Integration Technologies will set the following symbols:
  ```
  %LET LUNCH=Egg Drop Soup;
  %LET LUNCH0=2;
  %LET LUNCH1=Egg Drop Soup;
  %LET LUNCH2=Sesame Chicken;
  ```
- In SAS 8.2, you need to be wary of special characters in parameter values. In SAS9 parameter values are quoted with the equivalent of *%nrstr()*.
- On a related note, any time you expose information resource through a web interface, and especially when you are publishing to the public Internet, you need to prepare for possible security attacks. Do some research on "SQL injection attacks" and consider how that applies to SAS code.
- Almost all applications have a requirement for authorization applied to parameter values. For example, the shoe

sales report takes a REGION parameter. In many organizations, some users would only be able to put in their region, but higher level managers would be able to run the report for any region.

#### SCALABILITY

In Listing 2 above, we used an instance of the com.sas.servlet.util.SocketListener class to receive the streamed ODS output from SAS. This class is convenient to use, but has some characteristics that limit scalability in high usage applications:

- SocketListener creates and destroys a thread on each usage.
- SocketListener creates and destroys a socket on each usage.

When scalability is important, you probably will need to write your own replacement for this class, and use a pooling approach. This will also give you better control over what ports are in use by your application, which will help you to stay friends with your network administrators.

If scalability is important, you may also want to give consideration as to how you manage the binary stream coming back from SAS. If you can send that stream directly back to the browser you will reduce your overall memory requirements.

## SAS9 STORED PROCESSES

We have spent the majority of this paper exploring the details of programming the direct interface stored process API, which is available in both SAS 8.2 and SAS9. In SAS9 Integration Technologies, there is also a new Foundation Services API which leverages the Metadata Server's information about the stored process. SAS9 Foundation Services includes other generally useful application services, such Security, Session management, and Logging.

The principles we've presented on the SAS side are relevant for both SAS 8.2 and SAS9.

## CONCLUSIONS

There are a few limitations you should keep in mind:

- Chart output is just not as easy as you would wish.
- For a really robust production system, not to mention friendly, you'd like to be able to interrupt a cancel a long-running stored process. You can't.
- SAS 8.2 Integration Technology and stored processes have some limitations that are addressed in SAS9, such as the lack of an autoexec for server SAS sessions and special character escaping in parameters.
- Keep in mind, if you use Workspace pooling, that your stored process will not get a clean slate. The WORK library and global symbols from the previous stored process will be left lying around. This begs for more programming discipline.

The great thing about stored processes is that they open up the power of SAS to the outside world. They make it possible for a SAS programmer to share his or her work with co-workers that don't use SAS, but want to see the results on demand. The sample code we reviewed enables a Java programmer and a SAS programmer to collaborate to build powerful web-based reports, and the two of them don't have to fully understand each other's worlds.

## REFERENCES

Apache Jakarta Tomcat. <http://jakarta.apache.org/tomcat/index.html>.

SAS Integration Technologies, Release 8.2 Documentation. <http://support.sas.com/rnd/itech/library/library82.html>.

SAS Integration Technologies, Version 9 Documentation. <http://support.sas.com/rnd/itech/library/library9.html>

## ACKNOWLEDGMENTS

The author would like to recognize Michael D. Thomas and Danny Grasse, who are bright and talented consultants, but more to the point, colleagues on the projects that led to this paper and responsible for some of the insights that it contains.

## CONTACT INFORMATION

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

jwright@thotwave.com