Paper 264-30

A Sample Web-Based Reporting Container for ODS XML Reports

Chris Olinger, d-Wise Technologies, Raleigh, NC

ABSTRACT

The SAS® system offers a wide variety of tools for reporting needs. From data _null_ to ODS PDF, there are many avenues to produce and surface SAS reports. What is less obvious is what to do with these reports once you have created them. This advanced tutorial describes techniques for building a simple Web container for ODS XML and other SAS reports. This tutorial covers various SAS technologies including ODS HTML/XML/Tagsets, SAS/Integration Technologies, Java, and XML transforms. This paper is for anyone who is interested in how abstract reports can be generated in SAS and how they can be displayed in a web infrastructure.

INTRODUCTION

The introduction of ODS in Version 7[1] of the SAS System helped transform the traditional notion of SAS output from ASCII line art to HTML, RTF, and PDF. Today, the SAS developer has many choices for producing large amounts of high fidelity reports. In some cases, this report generation is simple. The insertion of a few ODS statements into a SAS program can turn simple ASCII output into a high fidelity PDF report. However, oftentimes this conversion is not easily accomplished. In some cases the SAS user is looking to customize the report output into a format, or with content, that is not a SAS provided default.

This situation often arises when creating dynamic content for the web. HTML output from SAS uses ODS styles to help generate stylistic content. The HTML that is rendered by SAS contains all of the colors, fonts, and styling information that is needed to render the HTML in the format that you want. You can easily create and specify an ODS style for changing the look and feel of your output, and you can very simply use ODS to generate CSS files to abstract classes of style that are recognized by most current browsers. This flexibility *does* afford a layer of abstraction (because you can simply change out the CSS file to change the look and feel of your output without re-running SAS). However, there are times when this granularity is not enough. There are times when what we want is for SAS to act as a “result server” and hand off the actual rendering of our tables and content to an external system. Creating output that is “style free” allows us to seamlessly integrate our output into existing containers that already have schemes for rendering.

Our mission then is to create a simple web reporting container that treats SAS as a result server. SAS is great at data processing and analysis, and given some of the tools in ODS and Integration Technologies, great at helping us build applications to surface these results. What is interesting is that some of the techniques that we are going to discuss in this paper are being used in Version 8 and Version 9 of the SAS system. Stored processes, XML, ODS Tagsets, SAS/Web Report Studio… all of these technologies share in the tenets presented in this paper.

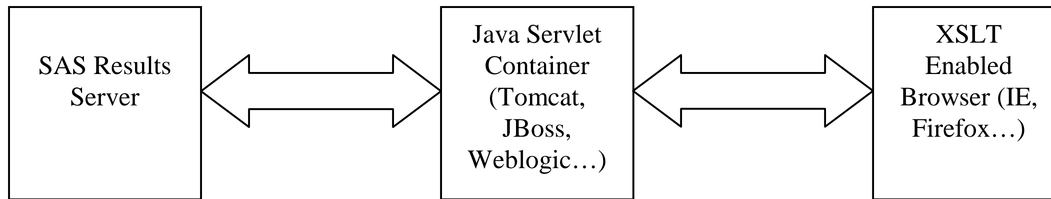
SEPERATION OF CHURCH AND STATE

Most people that live in the United States take our separation of church and state for granted. There are groups that argue that this separation goes overboard, and some that argue that it is not enough. But in general, most agree that keeping governance and worship separate is a good thing. The same goes for computer science. In computer science terms, the separation of church and state is called Model/View (or Model/View/Controller, aka MVC). Most people in computer science have heard of MVC as it was made popular in the 70’s and 80’s by Smalltalk 80 and other object-oriented systems. In MVC terms, the Model is the state, or the business logic. The Model has no notion of how to visually display its state – it leaves that responsibility to the View. The View (our church) takes a read on the state and decides whether it is relevant and deserves attention. If so, it renders the state in the manner it deems most fitting.

This same notion – separation of responsibilities – is alive and well in most N-tiered web systems. For example, most DBA’s would not advocate a relational database directly creating PDF files based on stored data. The database has a responsibility. Its responsibility is to serve data in the most efficient manner possible. As such, the N-tiered system is comprised of separate “tiers” where each tier has a responsibility. The data tier serves data. The web tier manages the web application. The analysis tier analyzes and reports.
For our little project, SAS is going to function as a “results tier”. That is, SAS will produce results in an abstract form, and then our viewer client will be responsible for the rendering. This separation of rendering from result production will ensure us that we can render our results however we decide is most appropriate. Figure 1 shows us a logical picture of what we are trying to accomplish.

[1] Yes Virginia, there really *was* a Version 7 of the SAS System.

Figure 1



To build this system, we are going to need some help from the SAS supporting cast of components that come with most standard SAS installs[2]. These components include the following:

- SAS/Integration Technologies – IT is one of the coolest things about SAS (in the author's opinion, of course). IT bridges the gap between external clients (Web Servers, Application Servers, Desktop Applications) and SAS
- SAS/Base – Base SAS: don't leave home without it. We will use SAS/Base to create a simple version of a Stored Process.
- ODS Tagsets – something needs to put our results in a format that is consumable by our application. ODS Tagsets are extremely flexible in helping tailor custom SAS output.
- A web browser with an XML/XSLT engine. IE 6.0 and other browsers have this functionality built in by default. We will use IE 6.0.
- A Java/JSP/Servlet container such as Tomcat, JBoss, or Weblogic. The application will be Java based, although you can do the same sort of thing as a COM/.Net application.

INTEGRATION TECHNOLOGIES

SAS/Integration Technologies is a technology from SAS that allows a user to build external applications that can call directly into SAS. The external applications can be built using Java or COM. This technology affords the user an incredible amount of flexibility in that you are no longer required to build SAS applications using only SCL and the standard SAS programming language. Almost all of the V9 vertical applications use Integration Technologies to manage their client/server connection to SAS.

Our use of Integration Technologies is very simple. We need a way to launch SAS and run SAS programs from a web application. The SAS programs will produce results and stream them back to the web browser in an XML format. Our configuration of Integration Technologies is also going to be very simple. We are going to use it in its most basic form, foregoing security and other protections[3].

To use Integration Technologies we must first configure and launch the object spawner. The object spawner is a process that listens for requests to create a SAS session. The spawner will then create a SAS session and connect the calling client with the resulting SAS session. The object spawner can be launched by running the following command (this works for both V8 and V9, if you have Integration Technologies licensed):

```
"C:\Program Files\SAS Institute\SAS\V8\inttech\sasexe\objspawn.exe" -configFile
ObjectSpawner.cfg_82 -nosecurity
```

This code assumes that you have installed SAS in the default location on your machine. The configuration file that is referenced defines the parameters and options that control how Integration Technologies behaves. For our example we will use a simple configuration:

```
#
```

[2] Note, the techniques described in this paper can be leveraged with either V8 or V9 of the SAS system. For V9, there is a lot of additional infrastructure (Stored process server, Meta-data Server, and Workspace Server) that has been created for you out of the box.
[3] You can configure Integration Technologies security mechanisms to use LDAP, to use the OS, or to run with no security at all.

```
# Spawner Definition
#
dn: sasSpawnercn=sasSpawner
objectClass: sasSpawner
sasSpawnercn: sasSpawner
sasLogFile: "C:\Program Files\SAS Institute\SAS\V8\inttech\objspawn.log"
sasMachineDNSName: localhost
sasVerbose:
sasOperatorPort: 5306
description: SAS Object Spawner

#
# Object Server Definition listening on default service "sasobjspawn"
#
dn: sasServercn=sasServer
objectClass: sasServer
sasServercn: sasServer
sasCommand: "C:\Program Files\SAS Institute\SAS\V8\sas.exe" -config "C:\Program
Files\SAS Institute\SAS\V8\sasv8.cfg"
sasMachineDNSName: localhost
sasPort: 5307
sasProtocol: bridge
description: SAS Object Server
```

This configuration describes a server running on localhost (our current machine). When a request comes from an external application it will launch the command referenced in the “sasCommand” option. The communication pipe will use port 5307. The object spawner must be running before we can do anything else.

STORED PROCESSES

If you have been following SAS with the release of Version 9 then you have most likely have heard of a stored process. A stored process is a bit of SAS code that sits on the SAS server and returns a set of results to the caller. There is new infrastructure that has been added to the SAS system to control things like security, meta-data management, and Integration Technologies infrastructure, but in reality, a stored process is as simple as a standard SAS program and a hookup to call it from an external client[4]. You can configure the stored process with macro variables and you can return your results to the caller via interfaces in Integration Technologies.

For our application, we would like to do the following:

1. Allow the user to select a SAS job to run from a list of available SAS jobs
2. Provide a simple way to provide a front-end for the SAS job. That is, we want to be able to pass parameters to the SAS job via macro variables.
3. Stream the results of the SAS job back to the client in XML format.
4. Render the results in the web browser.

The format of our stored process will be similar to the new standard SAS definition of a stored process. There is a process preamble and the process main body. All results are written to _WEBOUT so they can be streamed back to the web browser. For instance, the following is a definition that prints a data set using ODS:

```
/*
Sample stored process
*/

*ProcessVariables;
```

[4] People that have been running SAS/Intrnet have been running what amounts to Stored Processes for years!

```
%let DSN = ;

*ProcessBody;

ods html file=_WEBOUT style=sasweb;
proc print data=&DSN; run;
ods html close;
```

The ProcessVariables section marks our stored process parameters section. The ProcessBody section defines the code to execute – in this case, an invocation of the PRINT procedure. _WEBOUT will have already been setup for us to use. A formal stored process in V8 and V9 has more complex mechanisms built into the system. For instance, the ability to build result packages containing multiple files and images. For our simple system we will restrict returned results to one file.

The &DSN reference in the code above is a macro variable that will need to be set as an option provided by the user. Given this, we must create some java code to read the process file and create the appropriate HTML form that will pass the user input to our stored process. The code above also returns ODS HTML output. We will use this initially for testing, but will change this to XML a little later on

ODS TAGSETS

The Output Delivery System is responsible for rendering the various types of output that SAS produces. It can produce a wide variety of types of output from HTML, RTF, and PDF to SAS data sets. The original version of ODS released in V7 of the SAS system relied on compiled C code to implement the various output destinations. Beginning (experimentally) in V8 and more formally in V9, you can configure the output that is generated by most ODS destinations. The mechanism for this control is an ODS Tagset, which is used primarily by the ODS Markup destination. Interestingly, Tagsets are also used by the SAS/XML Libname engine to produce XML from SAS data sets. And as SAS moves forward, I am sure that Tagsets will play an increasing role in SAS integration plans with other external systems.

ODS Tagsets are, at their core, fairly simple to understand. A Tagset is comprised of a set of text and instructions that should be executed when certain output events happen in the system. For instance, if you break down the running of a procedure into its component output events you will get a list that looks something like the following:

```
Procedure Start
    By Group Start
          Table Start
                Row Start
                      Cell Start
                            Value
                      Cell End
                      …
                Row End
                …
          Table End
          …
    By Group end
    …
Procedure End
```

This is not a correct list, nor is it 100% complete. However, it does show the types of events that are being generated by the system. For each of these events we want to output some text and possibly run other events. Tagsets can output arbitrary text, so you can use them to create any sort of text file that you wish. Our system requires XML output from the procedure so that we can render in the browser.

Working with Tagsets can be challenging. You have to know the types of events that are occurring and you have to know the syntax of the Tagset language. The best piece of advice I can give you is to start small and use as many pre-defined Tagsets as possible. One of the first Tagsets that you should run is Tagsets.Event_Map[5]. This Tagset is your friend. It will dump all of the events and attributes that are generated from a SAS job. To invoke this Tagset, specify the following:

---

[5] To see the code behind the Tagset, RMB on the Results tree node in DMS mode. Select Templates, sashelp.tmplmst, Tagsets, and then Event_Map.

```
ods listing close;
ods tagsets.event_map file="events.xml";
proc print data=sashelp.class; run;
ods tagsets.event_map close;
```

This code will create a file named “events.xml” that contains all of the events and attributes that were generated in the running of the proc print step. The output looks like this:

```
…
<proc_branch event_name="proc_branch" trigger_name="attr_out"
class="ContentProcName"
    value="Print" colcount="1" name="Print" label="The Print Procedure" index="IDX"
    just="c" url="events.xml#IDX" hreftarget="body">
    <leaf event_name="leaf" trigger_name="attr_out" class="ContentItem"
        value="Data Set SASHELP.CLASS" colcount="1" name="Print"
        label="Data Set SASHELP.CLASS" index="IDX" just="c" url="events.xml#IDX"
        hreftarget="body">
           <page_anchor  event_name="page_anchor" trigger_name="attr_out"
                  class="PagesItem" colcount="1" index="IDX" just="c" />
           <output event_name="output" trigger_name="attr_out"
                  output_name="Print"
                  output_label="Data Set SASHELP.CLASS" colcount="1" name="Print"
                  label="Data Set SASHELP.CLASS" clabel="Data Set SASHELP.CLASS"
                  index="IDX" just="c">
…
```

This is just a small bit of the output from the job (there is quite a bit!). The trick is to figure out the events that matter for your application and then build just the events that we are interested in. For our application we are interested only in those things that directly apply to a piece of output. If we scan through the above XML file we can see that the events:

- Doc
- Doc_Head
- Doc_Body
- Table
- Table_Head
- Table_Body
- Row
- Header
- Data
- Verbatim
- Verbatim_Container
- Verbatim_Text

directly relate to the tables and batch output that are being created as part of a SAS run. There is a lot of meta-data that comes along with the tables, but for our application we are going to use the set of events that gets us what we want using the bare minimum.

To subset the XML we will need to define event text and code for each of the events that we are interested in. Using our rule of re-using available code, we can actually use the Event_Map Taget to get what we need. If you look closely at the Event_Map Tagset you will see that it specifies one event, “basic”, that is marked as the default for all events. If there is no event block defined for an event then the “basic” event is used instead. What we need then is a way to define an event that does what the “basic” event does, but just for the events that we are interested in. The final set of code defines specialized events for Output and Doc, but uses the default “basic” event for the other events that we are interested in. Please see the appendix for the source code (V9 source).

Using our customized Tagset gives us the following XML:

```
<?xml version="1.0" encoding="windows-1252" ?>
<doc operator="colinger" sasversion="9.1" saslongversion="9.01.01M3P11032004"
   date="2005-02-06" time="15:36:07" encoding="windows-1252">
 <doc_head  class="Body" />
 <doc_body class="Body">
```

```
<output name="Print" label="Data Set SASHELP.CLASS"
    clabel="Data Set SASHELP.CLASS"
    index="IDX">
  <table class="Table" type="table">
    <table_head>
      <row class="Table">
        <header class="Header" value="Obs" type="string" />
        <header class="Header" value="Name" type="string" />
        <header class="Header" value="Sex" type="string" />
        <header class="Header" value="Age" type="string" />
        <header class="Header" value="Height" type="string" />
        <header class="Header" value="Weight" type="string" />
      </row>
    </table_head>
...
```

This XML is much more succinct and to the point. We will use this format to be our results model.

You can get more information on Tagsets from the system documentation and from here: http://support.sas.com/rnd/base/topics/odsmarkup.

XML AND XSLT

XML stands for eXtensible Markup Language. The file format originated as an offshoot from HTML and SGML, and has become a standard for data interchange on the Web. Disparate systems now can communicate using XML, and with a little XSLT, can be massaged into any format that you wish. For our application, we are going to use XSLT to transform our format agnostic XML into HTML that can be displayed in a web browser.

XSLT stands for eXtensible Style Language Transforms and is used to transform XML into other formats. The language is XML based, and it follows all of the rules that other XML documents must follow. XSLT is a “templating” language. That is, instead of programmatically performing operations on the XML, you instead lay out what you want the resultant document to look like and then apply it to the XML to be transformed.

In order to transform XML using XSLT you must first have an XSLT engine that you will call to make the transformation happen. IE 6.0 has a built-in XSLT engine named MSXML 3.0. This engine will apply any XSLT that an XML document may reference and then display the results in the web browser. In fact, if you have ever clicked on an XML file under Windows chances are that IE was invoked and you saw your XML in a expandable/contractible format (using + and – for node operators). This XML format is the result of the default XSLT transformation that is built into IE.

Stylistic elements (colors, fonts…) can be applied to HTML using CSS. CSS stands for Cascading Style Sheets. A Cascading Style Sheet binds stylistic information to “classes” of HTML. If you look at the XML produced by our Tagset you will notice that we define a class that can be used to bind to a CSS style element. Or alternatively, we can just use the standard HTML classifications (tr, th, td …). Our CSS file will use the latter.

We are going to transform our simple XML so that it looks like the following:

Figure 2

Name: Print
Label: Data Set SASHELP.CLASS

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5

Obs	Name	Se x	Age	Height	Weight
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83.0
7	Jane	F	12	59.8	84.5
8	Janet	F	15	62.5	112.5
9	Jeffrey	M	13	62.5	84.0
10	John	M	12	59.0	99.5
11	Joyce	F	11	51.3	50.5
12	Judy	F	14	64.3	90.0
13	Louise	F	12	56.3	77.0
14	Mary	F	15	66.5	112.0
15	Philip	M	16	72.0	150.0
16	Robert	M	12	64.8	128.0
17	Ronald	M	15	67.0	133.0
18	Thoma s	M	11	57.5	85.0
19	William	M	15	66.5	112.0

In order to do this we must write some XSLT and then reference this file from the generated XML file. The XSLT code and CSS file that performs this transformation is located in the Appendix of this paper.

THE JAVA APPLICATION

The glue that makes the application work is the Java application. The Java application runs as a Servlet in a Servlet/JSP container (such as Tomcat, JBoss, or Weblogic). Our application is a simple example of how to build a reporting container. Our application must do the following:

1. Display a list of stored processes and allow the user to select them
2. Display a simple interface for each stored process that lets the user enter information
3. Run the stored process and stream the resultant XML back to the browser

To do this we will be creating a Servlet. A Servlet is the Java way to define a dynamic web application. The Servlet will communicate with SAS using the standard Integration Technologies Java interfaces. These interfaces allow a Java interface to “talk” to the SAS system including submitting SAS code, reading the SAS log, and transferring results back to the Java program. The Java code is somewhat lengthy for listing in this paper, so we will instead talk

about the SAS connection parts here. If you are interested in the full source you can download it at http://www.d-wise.com.

CONNECTING TO SAS

The connection to SAS is managed by what is known as a Workspace. In order to create a Workspace you must supply the Integration Technologies code a set of parameters that points to the host running the object spawner. The following is sample Java code (from the SAS web site) that creates a Workspace:

```
import com.sas.iom.WorkspaceConnector;
import com.sas.iom.WorkspaceFactory;
import com.sas.iom.SAS.IWorkspace;
import java.util.Properties;

Properties iomServerProperties = new Properties();
iomServerProperties.put("host",host);
iomServerProperties.put("port",port);
iomServerProperties.put("userName",user name);
iomServerProperties.put("password",password);
Properties[] serverList = {iomServerProperties};

WorkspaceFactory wFactory = new WorkspaceFactory(serverList,null,null);
WorkspaceConnector connector = wFactory.getWorkspaceConnector(0L);
IWorkspace workspace = connector.getWorkspace();

< insert workspace usage code here >

wFactory.shutdown();
connector.close();
```

SUBMITTING SAS CODE

This simple code will create a connection to SAS and allow you to submit SAS code. To submit SAS code you must get an instance of the ILanguageService interface:

```
ILanguageService sasLanguage = workspace.LanguageService();
sasLanguage.Submit("data a;x=1;run;proc print;run;");
```

This code will run a simple SAS job and then return. For our application we need to do a little more. We need to write some code that will read the file system to build a list of stored processes and also some code to stream back our ODS XML results. This code is lengthy and is omitted from the paper for brevity's sake.

THE RESULT

If all of the parts come together (and of course, they will ☺) then our application will look something like the following… The user will select a stored process from the left hand screen, and after entering in any information required by the interface, run a SAS job. The SAS job will produce results in an XML format and return them to the browser. The browser then renders the XML using XSLT and CSS and displays them for the user. We should be able to change the look and feel of the application by changing the CSS files. This means that the user will not need to rerun the SAS job to change how the system looks.

CONCLUSION

You may be asking yourself: "exactly what advantages does this process have over just using ODS to output my HTML?" This is a good question. It is simpler in the short term to just let ODS create your HTML files for you. If you are a SAS programmer only then you will want to create your SAS output this way. ODS Tagsets provide you ample tools to customize your output in any format that you want. However, the benefits to using this approach are the following:

- You can format the ODS output without specific stylistic information. This is important if you would like to embed ODS output into an existing portal where the style information has already been set.
- You can re-format the ODS output without having to run SAS again. For large processing jobs this is an important point. This reformatting can take on a variety of methods. For instance, there are XSLT transforms for turning XML into PDF files.

- You can easily change the look and feel of your HTML output using CSS, without having to define an ODS style and without having to rerun your SAS job.

ACKNOWLEDGMENTS

Special thanks to the folks at SAS R&D for all the cool stuff that they put out!

CONTACT INFORMATION

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

Chris Olinger
d-Wise Technologies, Inc.
3115 Belspring Lane
Raleigh NC 27612

Work Phone: 919 696 3276
Fax: 919 531 8391
Email: colinger@d-wise.com
Web: http://www.d-wise.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX

Code Sample 1

The following is the modified Tagset that defines what our XML will look like.

```
%macro event(name);
   define event &name;
   start:  trigger basic;
   finish: trigger basic;
   end;
%mend;

proc template;
define tagset Tagsets.mymap;
   notes "This is the custom event map definition";
   define event initialize; end;
   define event show_charset;
      set $show_charset "1" /if ^exists( suppress_charset);
   end;
   define event doc;
      start:
         trigger show_charset;
         put "<?xml version=""1.0""";
         putq " encoding=" encoding /if exists( $show_charset);
         put "?>" NL NL;
         put "<" EVENT_NAME;
         putq " title=" BODY_TITLE;
         putq " author=" AUTHOR;
         putq " operator=" OPERATOR;
         putq " sasversion=" SASVERSION;
         putq " saslongversion=" SASLONGVERSION;
         putq " date=" DATE;
         putq " time=" TIME;
         putq " encoding=" ENCODING /if exists( $show_charset);
         putq " language=" LANGUAGE;
         putq " trantab=" TRANTAB;

         trigger attr_out;
         put ">" NL;
         ndent;

      finish:
         xdent;
         put "</" EVENT_NAME ">";
   end;
   define event basic;
      start:
         put "<" EVENT_NAME;
         putq " hidden=" hidden / hidden;
         putq " asis=" asis / asis;
         putq " class=" HTMLCLASS;
         putq " text=" TEXT;
         putq " value=" VALUE;
         putq " rowspan=" ROWSPAN;
         putq " colspan=" COLSPAN;
         putq " name=" NAME;
         putq " label=" LABEL;
         putq " clabel=" CLABEL;
         putq " type=" TYPE;
         put TAGATTR;
         putq " precision=" PRECISION;
```

```
            putq " url=" URL;
            putq " hreftarget=" HREFTARGET;
            put "/" / if exists(empty);
            put ">" NL;
            break / if exists(empty);
            ndent;

         finish:
            break / if exists(empty);
            xdent;
            put "</" EVENT_NAME ">" NL;
      end;

      define event output;
         start:
            put "<" EVENT_NAME;
            putq " class=" HTMLCLASS;
            putq " text=" TEXT;
            putq " value=" VALUE;
            putq " name=" NAME;
            putq " label=" LABEL;
            putq " clabel=" CLABEL;
            putq " type=" TYPE;
            putq " index=" ANCHOR;
            put TAGATTR;
            put "/" / if exists(empty);
            put ">" NL;
            break / if exists(empty);
            ndent;

         finish:
            break / if exists(empty);
            xdent;
            put "</" EVENT_NAME ">" NL;
      end;


      %event(doc_head);
      %event(doc_body);
      %event(table);
      %event(table_head);
      %event(table_body);
      %event(row);
      %event(header);
      %event(data);
      %event(verbatim);
      %event(verbatim_container);
      %event(verbatim_text);

      output_type = "xml";
      indent = 2;
      split = " ";
      nobreakspace = " ";
      mapsub = %nrstr("/&lt;/&gt;/&amp;/&quot;");
      map = %nrstr("<>&""");
      stacked_columns;
   end;
   run;
```

Code Sample 2

The following is some sample XSLT that transforms our XML document into HTML:

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/doc/doc_body">
  <html>
  <head>
    <title>My SAS Output</title>
    <link rel="stylesheet" href="render.css" type="text/css"/>
  </head>
  <body>
  <xsl:apply-templates select="output"/>
  </body>
  </html>
</xsl:template>

<xsl:template match="output">
  <br/> <br/>
  <b> Name:</b>
  <xsl:value-of select="@name"/>
  <br/>
  <b> Label: </b>
  <xsl:value-of select="@label"/>
  <br/> <br/>
  <xsl:apply-templates select="table|verbatim_container"/>
</xsl:template>

<xsl:template match="verbatim_container">
  <center>
  <table>
  <xsl:apply-templates select="verbatim"/>
  </table>
  </center>
</xsl:template>

<xsl:template match="verbatim">
  <tr>
  <td>
  <pre>
  <xsl:for-each select="verbatim_text">
     <xsl:value-of select="@value"/>
     <br/>
  </xsl:for-each>
  </pre>
  </td>
  </tr>
</xsl:template>

<xsl:template match="table">
  <center>
  <table>
    <xsl:apply-templates select="table_head"/>
    <xsl:apply-templates select="table_body"/>
  </table>
  </center>
</xsl:template>

<xsl:template match="table_head">
  <thead>
     <xsl:apply-templates select="row"/>
  </thead>
</xsl:template>

<xsl:template match="table_body">
  <tbody>
     <xsl:apply-templates select="row"/>
```

```
  </tbody>
</xsl:template>

<xsl:template match="row">
  <tr>
     <xsl:apply-templates select="header|data"/>
  </tr>
</xsl:template>

<xsl:template match="header">
  <th colspan="{@colspan}" rowspan="{@rowspan}">
     <xsl:value-of select="@value"/>
  </th>
</xsl:template>

<xsl:template match="data">
  <td colspan="{@colspan}" rowspan="{@rowspan}" align="right">
     <xsl:value-of select="@value"/>
  </td>
</xsl:template>

</xsl:transform>
```

Code Sample 3

The following is the CSS that we used to add color and font information to the output.

```
body {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 1em;
}

th {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 1em;
    padding: .25em .25em;
    background-color: #C1E2A5;
    color: #000000;
}

td {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 1em;
    padding: .25em .25em;
    color: #000000;
}

pre {
    font-family: Courier New;
    font-size: 1em;
    padding: .25em .25em;
    text-align: left;
    background-color: #C1E2A5;
    color: #000000;
}

body {
    color: #333333;
    background-color: #FFFFFF;
}

table {
    color: #FFFFFF;
}
```