# SAS® Server Pages
## <?sas and <?sas=

Richard A. DeVenezia, High Impact Technologies

## Abstract

Have you ever wanted to plunk down some SAS software output smack dab in the middle of a web page? Use a DATA Step to populate a Javascript array? How about dynamic control over page generation from within your HTML source code?

You can do the above and more using a SAS Server Page (.ssp). This concept aligns closely with other server page technologies such as ASP, PHP and JSP. Server pages let you transport your web development to a new creative plateau. This paper will discuss the page processor and demonstrate using the `<?sas` and `<?sas=` tags in a rich user interface web application; all served up by the SAS Stored Process Web Application. The processor is named "SASServerPageProcessor" and is a stored process.

## Environment

The SAS Management Console[1], a component of Base SAS software, is used to define the URL path to a stored process. The SAS Stored Process Web Application[2] is used to run SASServerPageProcessor. The application is part of the SAS Web Infrastructure Kit, a component of SAS Integration Technologies. A typical invocation looks like this URL:

http://myserver:8080/SASStoredProcess/do?parm=value&...&parm=value

The portion after the question mark (?) is the called the query string. A query string is a list of *name=value* pairs separated by ampersands. The name portion of the pair is often called a parameter. A stored process is run directly via **do** method by providing the parameter `_PROGRAM = <path of the stored process>`.

The **do** method converts parameters into global macro variables which may be used by the stored process. The values are macro quoted, however that does not mean complete security. Careless use of parameter values can be a vector for code injection or other black hat activities. There are numerous reserved macro variables[3] available to the stored process and their names are prefixed with an underscore. Reserved variables such as `_DEBUG` may be specified in the URL. The management console provides administrative settings to control or disallow actions requested via reserved macro variables.

## Genesis

The operating environment of a Cluster Management Utility (CMU) application was being changed from SAS/AF to intranet web. The utility allows a retailer to generate and maintain "Location Clusters" that may be used across their enterprise in solutions such as SAS Integrated Merchandise Planning, SAS Space Planning, SAS Business Intelligence Reporting and Analysis. Several demonstration pages were developed to show that the transition to a web solution was feasible, but would require some innovative thinking.

The SASServerPageProcessor (SSPP) was developed by the author to bring order to inherited code which had created HTML output using a mishmash of techniques that were difficult to understand and debug. Some of the problems included HTML segments stored in SAS data sets, partial HTML source stored in data lines or external files with special token markers for replacement or macro execution and on one occasion a global macro variable collision.

Several sandbox tests were done to examine the possibility of implementing SSPP using only the experimental `PROC STREAM`. This procedure has great features that leverage the code parsing and macro expansion features of the SAS executor; however, the differences in handling the ampersand symbol (&) in SAS and HTML were significant

[1] http://www.sas.com/technologies/bi/appdev/base/mgmtconsole.html
[2] http://support.sas.com/rnd/itech/doc9/dev_guide/stprocess/stpwebapp.html
[3] http://support.sas.com/documentation/cdl/en/stpug/61271/HTML/default/a003152603.htm

enough to vote down STREAM. Additionally, the test code showed that a STREAM based server page with submitted SAS code would have several blocks of code; some would be HTML surfaced out of STREAM, followed by various procs or data steps and returning to STREAM again. This type of layout would not be amenable to HTML editors or integrated development environments (IDE).

The tag-based server page solution was investigated with the goal of delivering a rich user interface application. A focus on Javascript and jQuery in particular was expected. The team was not versed in programming JSP pages but had some experience with PHP. PHP was not available due to an operational policy. The decision to mimic PHP server pages was made with the caveat that they would be SAS flavored. This scheme also means that any of several available HTML editors and IDEs can be used to code the pages.

## SAS Server Page specification

An HTML coded file with the extension .ssp will be a SAS Server Page having these features:

1. Each page will be constructed from a single SAS session. State information may be retained across page refreshes using a techniques such as hidden HTML input fields, HTML cookies and stored process sessions.
2. The tag
   `<?SAS= {source-code}?>`
   will be replaced by the server with the macro evaluation of *{source-code}*.
3. The tag
   `<?SAS {source-code}?>`
   will submit *{source-code}* on the server. Output sent to the fileref _WEBOUT, either directly or indirectly through a block bounded by %STPBEGIN and %STPEND, will become part of the page.
4. All other parts of the page would be output with no change.

Example1.ssp:

```
<html>
<title><?SAS= &TITLE?></title>
<body>Title is <?SAS= &TITLE?>.</body>
</html>
```

Example2.ssp:

```
<html>
<title><?SAS= &TABLE Listing?></title>
<body>
<?SAS
 %STPBEGIN; Proc Print data=&TABLE;run; %STPEND;
?>
</body>
</html>
```

## SSP Processor Invocation

The processor of an .ssp can be invoked in numerous ways. The team recognized that it would be possible to automatically invoke the processor when a .ssp URL was visited. However, the expertise to make the required settings in either an Apache server or JBOSS server was lacking. The management console made possible the use of a like-named stored process. Thus, each .ssp would have two parts:

*page*.ssp - the SAS server page
*page*.sas - the SAS stored process that would invoke the SASServerPageProcessor

Example1.sas:

```
%let sspp = &ROOT./StoredProcesses/SASServerPageProcessor.sas;
%let page = Example1.ssp;
%include "&sspp.";
```

Developers using SSP must be cautioned against using the parameters **sspp** and **page**. The requirement of each page having a distinct invoker allows for shorter URLs and control over the path via the SAS management console.

This scheme does not preclude the development of a dispatching process that accepts a page parameter. Such a process could be the basis of an Apache mod_rewrite rule-set that handles .ssp URLs directly.

## The SASServerPageProcessor

The processor evolved over several iterations. The early forms of the program attempted a single pass over the .ssp source file. Debugging was difficult, white space management was problematic and the possibility of macro name collision between the .ssp and the processor needed to be avoided. It didn't take long for the processor to become a program writer. The generated program would be included to generate the page sent back to the browser.

There are three phases:

- Parser
- Program writer
- Program runner

### Parser

A hash table is used to gather all the parts parsed from the ssp file. Each part will have an associated `index`, `type`, `value`, `rownum`, and `length`.

`type` indicates how `value` will be processed:

- 0 for as-is
- 1 for macro resolution
- 2 for code submission

`length` is the length of value.

`rownum` records the original line number containing value.

```
do until (missing(_infile_));
```

Find the position of three markers in each line of the ssp:

- p1 for `<?sas=`
- p2 for `<?sas`
- p3 for `?>`

The positions are used to link to various handlers:

```
      select;
            when (type=0 and p1=0 and p2=0) link copy_0;
            when (type=0 and p1>0 and p2=0) link p1_open_0;
            when (type=0 and p1=0 and p2>0) link p2_open_0;
            when (type=0 and p1<p2) link p1_open_0;
            when (type=0 and p2<p1) link p2_open_0;
            when (type=1) link p1_close;
            when (type=2) link p2_close;
            otherwise;
      end;
```

A handler may shift `_infile_`, set `type`, extract `value` and add elements to the hash. The following three sections demonstrate how the parsing states are processed:

```
copy_0:
  * no markers, store the entire _infile_ as current type;
  index+1;
  rc = parse.add(key:index,data:index,data:type,data:_infile_,data:rownum,data:length);
  _infile_ = '';
return;

p1_open_0:
  * <?sas= marker, store up to the marker as current type, set new type and shift _infile_;
  if p1 > 1 then do;
   value = substr(_infile_,1,p1-1);
```

```
   length = p1-1;
   index+1;
   rc = parse.add();
  end;
  type = 1;
  if p1+6 > length(_infile_)
   then _infile_ = '';
   else _infile_ = substr(_infile_,p1+6);
return;

p1_close:
  * ?> close marker, store up to the marker as current type, set new type and shift _infile_;
  index+1;
  if p3 then do;
   if p3 > 1 then do;
    value = substr(_infile_,1,p3-1);
    length = p3-1;
    rc = parse.add();
   end;
   type = 0;
   if p3+2 > length(_infile_)
    then _infile_ = '';
    else _infile_ = substr(_infile_,p3+2);
  end;
  else do;
   rc = parse.add(key:index,data:index,data:type,data:_infile_,data:rownum,data:length);
   _infile_ = '';
  end;
return;
```

The sections p2_open_0 and p2_close are similar to the above. After the last line of source is processed the hash `parse` is output to dataset `work.parse`.

### Program Writer

The second step in the processor is to generate a program (`sspgm`) based on the parsed values. `sspgm` will have two alternating block types based on `chunk`; a data _null_ having put statements and as-is code.

```
set work.parse;
by chunk rownum notsorted;
file sspgm;

* codegen the start a data _null_ block;
if first.chunk then do;
  if chunk = 'put' then put 'data _null_; file _webout;';
end;

if chunk = 'put' then do;
  if type = 0 and length and not last.rownum
  then trailing = length - length(value); else trailing = 0;

  * codegen a put statement that uses a single quoted literal
  * which will prevent macro resolution;
  if type = 0 then do;
   if not last.rownum
    then value = "put '" || trim(tranwrd(value,"'","''")) || "'"
              || ifc(trailing,catt('+',trailing),' ') || "@;";
    else value = "put '" || trim(tranwrd(value,"'","''")) || "';";
  end;

  * codegen a macro resolving statement followed by a put that outputs it;
  if type = 1 then do;
   value = "resolved = resolve('" || trim(tranwrd(value,"'","''")) || "');
   _n_=length(resolved);";
   put value;

   if not last.rownum
    then value = 'put resolved $varying. _n_@;';
    else value = 'put resolved $varying. _n_;';
```

```
  end;
  put value;
end;

* add the as-is code to the sspgm;
if chunk = 'pgm' then do;
  L = length (value);
  put value $varying. L;
end;

* codegen the completion of a data _null_ block;
if last.chunk then do;
  if chunk = 'put' then put 'run;';
end;
```

### Program Runner

The `sspgm` is submitted using a %include:

```
%include sspgm;
```

### STREAM Macro

This macro was developed to deliver content from the macro environment. It uses the 9.3 experimental `PROC STREAM`. Typical use within the application was to report information back to the user or developer.

```
%macro STREAM(content);
proc stream mod outfile=_webout;
BEGIN
%superq(content)
;;;;
%mend;

%macro NoPage;
%STREAM(
<!DOCTYPE html>
<HTML><HEAD></HEAD><BODY>
<H1>SAS Server Page Processor</H1>
<p>The <code>page=</code> argument is missing! This should not happen.</p>
</BODY></HTML>
)
endsas;
%mend;

%macro Help;
%STREAM(
<!DOCTYPE html>
<HTML><HEAD></HEAD><BODY>
<H1>SAS Server Page Processor - Help</H1>
<p>A SAS Server page (.ssp) is similar to asp or php.</p>
<p>A .ssp is a normal HTML file that contains two SAS specific tags:</p>
<ul>
<li><code>&lt;?sas=<i>source code</i>?&gt;</code><p>Source code is RESOLVED and
replaced inline.  Typical use is to resolve macro variables.</p>
<li><code>&lt;?sas <i>source code</i>?&gt;</code><p>Source code is SUBMITTED to
SAS supervisor.  Typical use is to have code send output _WEBOUT.</p>
</ul>
</BODY></HTML>
)
endsas;
%mend;
```

The first two lines of the processor are these checks:

```
%if not %symexist(page) %then %NoPage;
%if %upcase(%superq(page))=HELP %then %Help;
```

This macro lists macro variable values:

```
%macro ReportMacroVars(whereclause);
%STREAM(<DIV STYLE='font-family:monospace; font-size:8pt'>)
ods html body=_WEBOUT (no_top_matter no_bottom_matter) ;
proc sql;
  create view macros as
  select scope, name, offset, value
  from dictionary.macros
  &whereclause.
  order by scope, name, offset
  ;
quit;
options nocenter;
proc print data=macros style(table)=[cellspacing=0 cellpadding=2];
var scope;
var name / style=[fontweight=bold backgroundcolor=lightgray];
var value;
run;
ods phtml close;
%STREAM(</DIV>)
%mend;
```

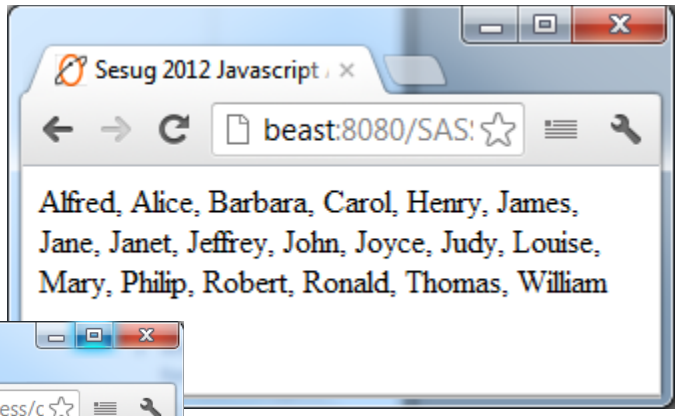
The macro had been used early on in development and remains commented as an artifact:

```
/* %ReportMacroVars (where lowcase(name) like '%ssp%') */
```

## Sample Page, Javascript Array

The following server page uses SAS to populate a Javascript array. The screen shots show the generated page and the corresponding source.

```
<html><head><title>Sesug 2012 Javascript Array</title>
<body>
<script>
names = [ <?sas
data _null_;
  set SASHELP.CLASS;
  file _webout;
  put name $quote. "," @;
run;
?> ];
document.write (names.join(", "));
</script>
</body></html>
```

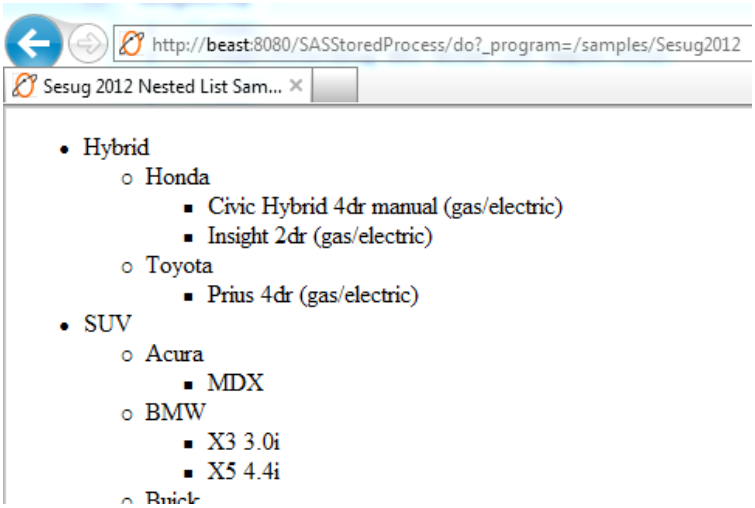



## Sample Page, Dataset By Groups shown as HTML Nested List

The following server page uses SAS to generate a nested unordered list based on the by groups of `TYPE MAKE MODEL` in the table `SASHELP.CARS`.

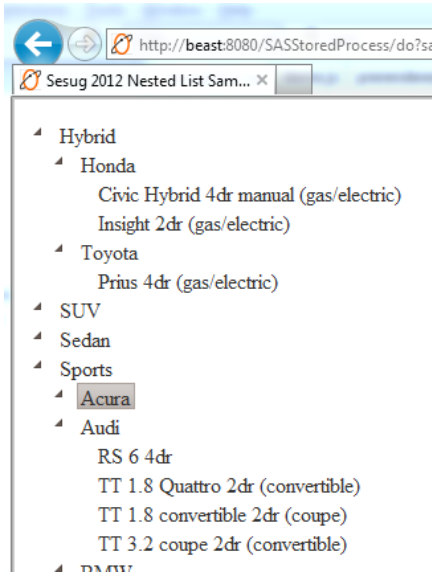
```
<html><head><title>Sesug 2012 Nested List Sample</title></head>
<body>
<?sas
proc sort data=sashelp.cars out=list;
  by type make model;
run;
data _null_;
  file _webout;
  put '<ul>';
  do until (end);
   do until (last.type);
    do until (last.make);
      set list end=end;
      by type make model;

      if first.type then put '<li>' type '<ul>';
      if first.make then put '<li>' make '<ul>';
      put '<li>' model '</li>';
    end;
    put '</ul></li>';
   end;
   put '</ul></li>';
  end;
  put '</ul>';
run;
?>
</body></html>
```




A rich user interface web application runs in a browser and uses HTML, CSS and Javascript to realize its design. One of the most popular third party Javascript libraries is jQuery®[4]. Another library for user interfaces is Kendo UI Web™ . The following example uses these libraries to transform the nested list into a treeview. The yellow code shows what was modified.

```
<html><head><title>Sesug 2012 Nested List Sample</title>
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js" type="text/javascript"></script>
<script src="http://cdn.kendostatic.com/2012.2.710/js/kendo.all.min.js"></script>
<link href="http://cdn.kendostatic.com/2012.2.710/styles/kendo.common.min.css" rel="stylesheet"/>
<link href="http://cdn.kendostatic.com/2012.2.710/styles/kendo.default.min.css" rel="stylesheet"/>
<script type="text/javascript">
$(document).ready(function() {
  $("#treeview").kendoTreeView();
});
</script>
</head>
<body>
<?sas
proc sort data=sashelp.cars out=list;
  by type make model;
run;
data _null_;
  file _webout;
  put '<ul id="treeview">';
  do until (end);
   do until (last.type);
    do until (last.make);
     set list end=end;
     by type make model;
     if first.type then put '<li>' type '<ul>';
     if first.make then put '<li>' make '<ul>';
     put '<li>' model '</li>';
```




[4] http://jquery.org/

```
    end;
    put '</ul></li>';
   end;
   put '</ul></li>';
  end;
  put '</ul>';
run;
?>
</body></html>
```

## Developing Rich Internet Applications

A simple text editor such as Windows Notepad is sufficient to write ssp pages. The author recommends a more robust text editor such as Ultraedit, Notepad++ or Programmers Notepad. Additionally, the use of a source version control system is strongly encouraged. Some examples are SVN, CVS, or Git.

All the major browsers provide an integrated developer mode (invoke it with F12 key) that greatly assists in examining source code, debugging Javascript and observing network traffic. The website jsFiddle.net is superb 'sandbox' for testing out a RIA situation before attempting to code it as part of a .ssp. Outside the scope of this paper are discussions of the palette of UI components available, making AJAX calls to SAS stored processes and reading and writing JSON data streams.

## Conclusion

SAS Server Pages can transport your web application development to new levels. The <?sas tag provides a SAS programmer easy access to page generation, in part or in whole. The similarity to other server page technologies means you can take full advantage of skill-sets in your organization by teaming CSS designers, jQuery and RIA coders and SAS programmers.

## Author Information

Richard DeVenezia is an experienced programmer and has been a using SAS for over 20 years. He has presented previously at NESUG, SESUG and SGF. He has an active interest in learning and applying new technologies. He is a contributor to SAS-L, where he looks for new techniques and offers some of his own.

The author may be reached at rdevenezia@hitactics.com.

## Trademarks