Paper 257-2011

Using SAS® Stored Processes with JavaScript and Flash to Build Web-Based Applications

Philip Mason, Wood Street Consultants Limited, Wallingford, Oxfordshire, UK

ABSTRACT

This paper describes techniques and ideas I have used for over five years as I built complex applications using the SAS® Stored Process Web application. The Web application enables the output from stored processes to be delivered in various ways (including HTML) by using the SAS® Output Delivery System. You can enhance the output by using JavaScript to add a lot of extra functionality, particularly interactivity. An easy way to add JavaScript functionality is to use some JavaScript programs written by experts. These programs can be easily found on the Web and are great for adding a feature or function here and there. However, a more powerful and structured way to add functionality is to make use of a JavaScript application framework. These frameworks are used by most of the best known Web sites in the world. This paper describes how to use the Ext JS framework, which is one of the best JavaScript application frameworks. It also describes how to use Flash objects from stored processes to deliver Flash based graphics.

INTRODUCTION

In this paper I will outline some of the most useful things that I have learnt while I have been building applications with SAS 9.1 using Stored Processes and the Stored Process Web Application. I have covered this in other papers and will emphasize some of the techniques that make use of JavaScript and Flash in the applications that can be built.

Building a web Based application

In this paper this is what I mean by a "Web Based Application":

An application that is used through a web browser that lets the user make choices, runs SAS code to produce results and delivers them back via the web browser.

With SAS 9 came the new metadata based architecture, including the Stored Process. SAS delivered a Stored Process Web Application with this release that meant that we could run stored processes from a web page and return results to it. In many ways this was very similar to SAS/Intrnet, but it was designed to be much more flexible and powerful. By using this web application and linking together a bunch of stored processes we can produce a Web based Application.

You can develop you application to different levels using stored processes. You could use them to produce an application using just standard HTML. You could extend it with a little JavaScript to provide some additional functionality. Or you could use quite a lot of JavaScript to provide a great deal of extra functionality. You can even look at including other web technologies such as Flash, Java or ActiveX into your HTML, which can provide an even higher level of functionality.

One way to build a web app – server side includes

One way to produce a web based application is to produce a stored process which takes a kind of pseudo code in. This pseudo code would define what a screen would appear as and what it is composed of.

For example, we could have a selection box that would let us choose one option and was filled by getting values for a variable from a dataset, then this could all be specified in a piece of pseudo code.

Example

```
   %select_box(dataset, variable, choice, 1)
```

The stored process would then call the macro that carries out the directive in and construct HTML to produce that component. This means that to produce a page you just need to write a list of pseudo code directives. By building up a library of macros to interpret different pseudo code functions, you can extend the functionality a great deal.

There is a book by Don Henderson called "Building Web Applications with SAS/Intrnet" that describes how to do this in great detail. Highly recommended!

ANOTHER WAY TO BUILD AN APPLICATION

This is the method I have been using recently. It makes use of the Stored Process Web Application to run stored processes through the web browser. One key technique is that when a stored process needs to prompt a user for input then it is written so that it can be run in two parts. The first part will build a web page which prompts the user for the required input and then calls the same stored process, passing in the selections that the user made. When the stored process is called the second time it can detect that it has been called with various parameters passed in, and it then runs to produce the output that is required. This basic idea was described in a SAS usage note and then developed some more by myself.

Example pseudo code

```
If expected parameters are passed in then
  Produce report
Else
  Produce web page
```

One key part of this technique is that a stored process needs to determine how to call itself. This can be done by constructing a URL from automatic macro variables that are provided to the stored process. The following code is taken from a stored process and it constructs a URL which will call that stored process, as well as passing some extra parameters to it. It makes use of the automatic macro variables: _program, _srvname, _srvport and _url. It also passes two parameters in: graph and x, whose values are &type and &x.

```
%let rev=%sysfunc(reverse(%superq(_program))) ;
%let index=%sysfunc(index(%superq(rev),/)) ;
%let rev2=%substr(%superq(rev),&index) ;
%let firstPart=%sysfunc(reverse(&rev2)) ;
%let html=http://&_srvname.:&_srvport.&_url.?_program=&_program.
%nrstr(&graph)=&type.%nrstr(&x)=&x ;
```

Once we have this URL built we can take it and put into an HTML form, which can then be used to prompt the user for various parameters and pass them to the web app using the URL. The URL can be put into the action field in the HTML form. When the submit button of the form is pressed then the action field is used to start building the actual URL. Other fields on the form are added to the end of the URL. These are then available to the stored process as macro variables.

HTML, RTF or PDF?

Another thing to consider when building your application is what you want your output to be produced as. You will most likely use a web browser for interacting with the user, but then reports that are produced may be required in HTML, RTF, PDF, EXCEL or other formats. Fortunately the web app makes this easy. You merely specify a value for _odsdest prior to calling the stpbegin macro, and it will then set the ODS options appropriately for the type of output you require. Since parameters passed in via the URL to the web app appear as macro variables, you can therefore pass a parameter on the URL called _odsdest, which will then set your output appropriately. That means that you can prompt the user for the type of output in your HTML and then pass that value through.

Want to do something tricky?

If you are building a nice interactive web page in HTML then it is likely you will be wanting to take control over exactly what HTML is produced. If you don't, then you can get a lovely HTML page produced for you by using the wizard in Enterprise Guide 4.1. You can define a stored process there along with various parameters and values they can have. EG then produces a web page to prompt you for those parameters and then call the stored process. Very easy. However should you want to do your own web page, there are other techniques.

You can use a web development IDE (Interactive Development Environment) to produce your web pages. This is reasonably easy, but reduces flexibility a little. You will need to code in to the page the URL to call your stored process for instance (and I should say I am not an expert web developer and no doubt there are 50 ways of automating this that I am not aware of). But once your URL is encoded you can use the full power of the IDE to build all the other bits and pieces of your page (or pages). I would recommend Aptana Studio for this, which is free and powerful.

The method I mostly use is to have SAS produce my web page for me. The key to this is knowing that you can write raw HTML code to the web page that the web app is creating from your stored process as it runs. This is done by writing the HTML code to the fileref called **_webout**. One trap to avoid is that you cant write to _webout unless it is free. It won't be free if %stpbegin has run, since it will be being used by ODS. So I usually only use %stpbegin when I want to use ODS to produce some kind of report, and then I turn it off with %stpend when I am finished. If you produce a stored process in Enterprise Guide then SAS helpfully puts and %stpbegin at the start and an %stpend at the end. I usually then remove these so I can just put them where I want them to be. That means that I can drop into a data step anytime and write some HTML or JavaScript to do something. For example:

```
Data _null_ ;
  File _webout ;
  Put '<h1>Make your choices and press submit to continue</h1>' ;
Run ;
```

Another very important thing to point out is that if you use %stpbegin to start writing to HTML, then use %stpend to stop so you can write some custom HTML, then the HTML produced by default is quite interesting and verbose. It will start with an <html> tag for instance, and end with an </html> tag. This is fine if you are just producing one report in a single lump. However if you want to nip in and out of writing custom HTML and have SAS produce reports around what you do, then I have found an incredibly useful undocumented result type. You will usually be using a result type of stream which streams the results to your web browser. However if you use a result type of streamfragment, then SAS will just produce the HTML for the reports and none of the extra tags required. That gives us much more control over what goes on in our HTML. To use this you just need to set the macro variable **_result=streamfragment** prior to running the stpbegin macro.

Stored Process Web Application

In SAS 9 we have the SAS Stored Process Web Application, which enables Stored Processes to be run from a web browser and then will stream the results back to the browser. This is the single most useful new facility that SAS have provided in the last decade – in my opinion. This is because it means that SAS code can be run from almost any place. For example, I can go into Microsoft EXCEL and enter a URL which runs the web application and produces a table – that table will then be imported into EXCEL automatically. Another example, at a previous client of mine we built a java application which simply constructed URLs to run the web application and then read the results that were streamed back.

A collection of useful macros

Any SAS application should have a nice collection of useful macros. If you are producing a web application then you will find yourself producing some macros which not only deal with the SAS environment, but also interact with HTML and JavaScript code. For example, the following macro will write a piece of JavaScript code to the HTML file being generated. That JavaScript will place a message into the status area at the bottom of the web page. Using this macro you can update the user on things that are happening in the background – for example "now generating graph 27 of 50".

```
%macro message_js(text) ;
  %if %symexist(_odsdest) %then
    %if %upcase(&_odsdest)=RTF or
        %upcase(&_odsdest)=PDF %then
      %return ;

  data _null_ ;
    file _webout ;
    put '<script type="text/JavaScript">' ;
    put "window.status = ""&text"";" ;
    put '</script>' ;
  run ;
%mend message_js ;
```

I have many other macros for use with web development which do things such as:

- Produce selection lists of various kinds based on variables in a dataset
- Produce messages of various types in HTML code
- Write JavaScript functions to HTML
- Read and write values of HTML cookies
- Produce pop-up dialogs
- Convert SAS data into a JSON data store
- Convert SAS data into XML for use by an object

There are also many other general purpose useful macros such as:

- Drop variables from a dataset that only have missing values

- Calculate _type_ values for use in a pseudo summary

I also have developed a range of macros which produce nice HTML reports. These could be used in other places, but each of them have been enhanced to be highly interactive by supporting tool tips, drill down and special links to other functionality. For example, some of these include:

- Heatmap which shows a graduated shading of colors combined with traffic lighting for exceeded limits
- Listing which has bars that indicate magnitude of values, along with colors that show traffic lighting
- Gantt charts, which use an innovative technique using Proc Gchart

A few useful macros appear here.

```
%*** create an annotate dataset which can then be used to add an icon on a graph,
which when clicked on
     will call the stored process again, but will toggle the value of the parameter
called zoom ;
%macro anno_info(parms) ;
 %* define the annotate macros ;
  %annomac ;

 %* build a link variable ;
  %let
link=http://&_SRVNAME:&_SRVPORT&_URL?_program=SBIP%3A%2F%2FFoundation%2F&env.%2Fbis
%2Fmedmon%2Fanno_info%28StoredProcess%29%nrstr(&parms)=&parms ;

 * create the annotate dataset to add the zoom icon with hyperlink ;
  data anno_info ;
    length function style $ 8
           html $ 240 ;
    retain when 'a' ;
    %system(5,3,4) ;
    %slice (99, 2, 0, 360, .30, purple, ps, 0);
    html="title=""Info"" href=""&link""" ;
    %LABEL (99.2, 2.2, "i", white, 0, 0, .4, swissb, +);
  run ;
%mend anno_info ;
/*%anno_info ;*/
/*proc gchart data=sashelp.class anno=anno_info ;*/
/*  vbar age / discrete raxis=axis1 ;*/
/*run ;*/
/*quit ;*/
```

The following macro is nice since it will work out what the URL of the current stored process is, and then if clicked on will call that stored process again, but adding the _odsdest=rtf parameter to the end, which will make the stored process produce RTF output.

```
%macro ods_button(ods=) ;
  %if %symexist(_odsdest) %then
    %if %upcase(&_odsdest)=RTF or
        %upcase(&_odsdest)=PDF %then
      %return ;

  %if &bypass=0 %then
    %stpend ;

  data _null_ ;
    file _webout  ;
    %* REF001: Remove trailing # from URL  - START;
    put '<a href="#"'
           'onClick="var loc = window.location ; loc = loc + ''&_odsdest=rtf'' ;
loc = loc.replace(''#'','''') ; window.open(loc) ;"'
           'style="background:black;color:white;font-size:large"><button>Send to
RTF</button></a>' ;
    %* REF001: Remove trailing # from URL  - STOP;
  run ;
```

```
   %mend ods_button ;
```
The following is a simple macro which will write out a JavaScript statement which will put a message in the message area at the bottom of the browser.

```
   %macro message_js(text) ;
     %if %symexist(_odsdest) %then
       %if %upcase(&_odsdest)=RTF or
           %upcase(&_odsdest)=PDF %then
         %return ;

     data _null_ ;
       file _webout ;
       put '<script type="text/JavaScript">' ;
       put "window.status = ""&text"";" ;
       put '</script>' ;
     run ;
   %mend message_js ;
```
This is a similar macro which does a JavaScript alert, which pops up a message box containing your message.

```
   %macro message_js_alert(text) ;

     %* Bring up a JavaScript message box;
     data _null_ ;
       file _webout ;
       txt = '<script type="text/JavaScript">alert("' !! "&text" !! '")</script>';
       put txt;
     run;

   %mend message_js_alert;
```
The following is a useful macro to lock a dataset and keep trying to lock the dataset if it is already locked by someone else. I have found this very useful for managing a central collection of parameters which can be shared and updated by a group of users.

```
   %macro locksave(_type=lock,
                   _member=,
                   _timeout=60,
                   _retry=0.01);

      %if &_type=lock %then %do;
         %* set start time;
         %local _starttime;
         %let _starttime=%sysfunc(datetime());
         %* try locking until lock is obtained or until timeout is exceeded;
         %do %until(&syslckrc=0 or %sysevalf(%sysfunc(datetime())>(&_starttime +
   &_timeout)));
            options noerrorabend;
            lock &_member;
            options errorabend;
            %* pause before retrying;
            %let sleep=sleep(&_retry.,1);
         %end;
      %end;
      %else %do;
         %* release lock;
         lock &_member clear;
      %end;
   %mend;
```
The following I have found incredibly useful in debugging stored process errors. Sometimes you will get an error and not be able to get any SAS log, except by going to stored process logs, which you may not have access to. Just calling this macro at the start of your stored process will write your log to a location you can access.

```
   %macro keep_work ;

     %global _keepwork;
```

```
    %let _keepwork=1;

    %if %symexist(env)=0 %then %return ;
    %if "%upcase(&env)" = "PROD" %then %return ;
    %if %sysfunc(fileexist(/export/home/&_metauser/work))=0 %then %return ;

    %* allocate user libref so that all work datasets now go to an alternate location
  ;
    libname user "/export/home/&_metauser/work" ;

    options mprint nosymbolgen nomlogic;

    proc datasets lib=user kill NOLIST ;
    run ;

    %* Reroute log to same name as STP, or to username if not found;
    %local proglog;

    data _null_;
      prog=symget('_PROGRAM');
      index=index(upcase(prog),'(STOREDPROCESS)');
      if index>0 then do;
        txt=reverse(substr(prog,1,index-1));
        txt=reverse(substr(txt,1,index(txt,'/')-1));
      end;
      else txt=symget('_METAUSER');
      call symput('proglog',strip(txt)!!"_&_keepwork");
    run;

    %let _keepwork=%eval(&_keepwork+1);

    proc printto log="%sysfunc(pathname(user))/&proglog..log" new;
    run ;

  %mend keep_work ;
```

The following is very useful since it will take any number of parameters passed to a stored process and convert them to a range of macro variables that are separated by something like a comma, so they can then be included in a where with an IN (for example).

```
  %macro html_parms_to_list(in,
                            out,
                            default=_:,   /* optional value to use as a default */
                            sep=%str( ),  /* optional one character separator */
                            quote=0,      /* 1=quote values, 0=dont quote values */
                            partstmt=0    /* 1=make part of where statement, 0=dont
  */
                            ) ;
    %global &out ;
    %let &out= ;
    %if &quote %then
      %let _q_=%str(%') ;
    %else
      %let _q_= ;
    %if %symexist(&in.0) %then
      %do ;
        %do j=1 %to &&&in.0 ;
          %let &out=&&&out..&sep.%superq(_q_)&&&in.&j%superq(_q_) ;
        %end ;
      %end ;
    %else
      %if %symexist(&in) %then
        %let &out=%superq(_q_)&&&in%superq(_q_) ;
      %else
```

```
        %let &out=%superq(_q_)&default%superq(_q_) ;
    %if %symexist(&in.0) %then
      %let &out=%qsubstr(%superq(&out),2) ;
    %if &partstmt=1 %then
      %do ;
        %if %symexist(&in) %then
          %do ;
            %if %superq(&in)=_ALL_ %then
              %let &out= ;
            %else
              %let &out=and &in in (%superq(&out)) ;
          %end ;
        %else
          %let &out=and &in in (%superq(&out)) ;
      %end ;
  %mend html_parms_to_list ;
  /*%let m0=2 ; %let m1=abc ; %let m2=def ; %html_parms_to_list(m,mlist) ; %put
  mlist=&mlist ;*/
  /*%let m0=2 ; %let m1=abc ; %let m2=def ;
  %html_parms_to_list(m,mlist,sep=%str(,),quote=1) ; %put mlist=&mlist ;*/
  /*%let m0=2 ; %let m1=abc ; %let m2=def ;
  %html_parms_to_list(m,mlist,sep=%str(,),quote=1,partstmt=1) ; %put mlist=&mlist ;*/
  /*%symdel m0 ; %let m=1 ; %html_parms_to_list(m,mlist,quote=1,sep=%str(-)) ; %put
  mlist=&mlist ;*/
  /*%symdel m0 ; %let m=1 ;
  %html_parms_to_list(m,mlist,quote=1,sep=%str(,),partstmt=1) ; %put mlist=&mlist ;*/
```

Using HTML

Linking different stored processes

Another thing that you will often want to do in a web app, is to run one stored process and then have it automatically run another. The best method that I have discovered for doing this is to use some custom HTML. You can use the **onLoad** method on the **body** tag in an HTML page which will run some JavaScript after the current web page has fully loaded. This is exactly what we need to link stored processes. An example of this is in an application I have which links several stored processes together. This first produces a web page to choose a study and the user clicks on submit – that runs another which saves the selection to a parameter file – that runs another which loads a list of subjects in the study – which runs another that loads a list of favourites for that study – and so on. Here is some HTML taken from an application which will call refresh the contents of another HTML iFrame, which runs another stored process to update it.

```
    data _null_ ;
      file _webout ;
      put '</head>';
      put '<body' ;
      put " var x =
  window.parent.document.frames.main.location.href.indexOf('cookie_save') ; if (x==-
  1) "  ;
      put " { window.parent.document.frames.main.location.reload() ; } ;"  ;;
      put '" class="panel">' ;
    run ;
```

Get vs. Post method

HTML forms use one of two methods to pass parameters: get or post. I usually use the get method, since when the next page has been loaded you can see the entire URL in the address bar or properties, whereas if you use post then you cant see any of the parameters. When using an HTML form with a lot of parameters you may encounter a limit at which the get method can no longer pass parameters since it has a limit of 2083 characters. I encountered this when I wrote a stored process to build filters. After adding about 10 lines of filters it all stopped working. I eventually discovered that this was because I had hit the limit, and so parameters were just being truncated which produced unpredictable results. By switching to the post method the problem instantly went away.

What happens when you pass many parameters of the same name?

With the web application if you pass in a single parameter, then it becomes available to the stored process as a macro variable of the same name. e.g. "&name=phil" on the URL is equivalent to "%let name=phil ;". However if you pass two or more parameters in of the same name, then you get a series of macro variables created. One has a suffix of 0, and provides a count of how many parameters there are. Then the first one has a suffix of 1, the second a suffix of 2, and so on. For instance, "&name=phil&name=mike" is equivalent to "%let name0=2 ; %let name1=phil ; %let name2=mike ;". The following macro takes a list of HTML parameters and puts them into a macro variable where they can be used with the **in** operator and a **where** clause.

```
%macro html_parms_to_list(
                       in,
                       out,
                       default=_:,    /* optional value to use as a default */
                       sep=%str( ),  /* optional one character separator */
                       quote=0,      /* 1=quote values, 0=dont quote values */
                       partstmt=0    /* 1=make part of where statement, 0=dont */
                          ) ;
  %global &out ;
  %let &out= ;
  %if &quote %then
    %let _q_=%str(%') ;
  %else
    %let _q_= ;
  %if %symexist(&in.0) %then
    %do ;
      %do j=1 %to &&&in.0 ;
        %let &out=&&&out..&sep.%superq(_q_)&&&in.&j%superq(_q_) ;
      %end ;
    %end ;
  %else
    %if %symexist(&in) %then
      %let &out=%superq(_q_)&&&in%superq(_q_) ;
    %else
      %let &out=%superq(_q_)&default%superq(_q_) ;
  %if %symexist(&in.0) %then
    %let &out=%qsubstr(%superq(&out),2) ;
  %if &partstmt=1 %then
    %do ;
      %if %symexist(&in) %then
        %do ;
          %if %superq(&in)=_ALL_ %then
            %let &out= ;
          %else
            %let &out=and &in in (%superq(&out)) ;
        %end ;
      %else
        %let &out=and &in in (%superq(&out)) ;
    %end ;
%mend html_parms_to_list ;
```

Persistence

When I started developing web applications I looked at ways that I could have persistence of data, since I needed to be able to make some choices in one stored process and then use those choices in another (for example). I found that there were a range of ways that could be used to achieve this:

1) Passing parameters on URL. When building up a URL to call a stored process using the web application, you can add more and more parameters onto the URL to pass information from the current stored process to the next. If you build a form in the HTML to call the stored process, then you can have hidden values on it which will then pass those values to the next stored process.

2) Sessions. This is a method provided by SAS in order to pass parameters on from one stored process to another. The idea is that you put name all macro variables you want to save starting with "SAVE_", and you put all datasets to save into a libref of SAVE. You then use the function stpsrv_session to create a session.

You get two macro variables that identify this session and must be used to make use of the session in another stored process. One major drawback to all this is that a saved session must be used on the same stored process server that it was saved on – this can have performance implications. We find that sometimes a stored process server will hang, and that would mean the saved session would be inaccessible.

3) Cookies. One problem with using cookies is that you cant directly read or write a cookie from SAS. So you end up having to manipulate JavaScript which does the reading and writing for you. Then you have to get that information into SAS. Another problem is that a cookie is limited to 4096 bytes. This became a problem when I allowed users to build filters that returned lists of thousands of items which I then wanted to pass to other stored processes. I then had to split my data into chunks of less than 4096 bytes and stored in a series of cookies, which added more complexity. The final problem I found was that cookies just did not always work 100% of the time (using Internet Explorer 6). There were some cases when strange things would happen, yet my code looked OK – and it would work in a different web browser. This unreliability ultimately made me look at alternatives.

4) Saving data to files/datasets. I found this method to be the most reliable. I can write information to a dataset and then load it back in when I want it. A couple of key points that make this possible is that I save each users parameters in a different SAS dataset named as their userid. i.e. if the userid was U1234 then the dataset is called U123. This eliminates problems of file locking if I used a single dataset for writing everyones parameters to. Where I do have parameters that I want to share between people I do write them all to a single dataset, but I have implemented a locking macro since otherwise I would get locking errors.

Data Stores

You could also store data in a JavaScript data store. This can be useful if you are extensively using JavaScript objects, particularly for tabular data that will be displayed in grids or graphs. Such a data store can be written to a file on a server and then used to drive a JavaScript object directly.

Databases

You could write any kind of data to a database which would then be accessible later. Since the SAS stored process is so flexible it means that we can call a URL to write something to a database and then be calling another stored process we can retrieve information from a database and format it in whatever form is required.

The following macro has proved to be incredibly useful since it will get a lock on a dataset so that an update to a shared dataset can be made, and then the lock can be released for others to use it. Additionally it will keep trying to get the lock every .01 seconds for up to a minute. Calling it with _type=unlock, will release the lock.

```
%macro locksave(_type=lock,
                _member=,
                _timeout=60,
                _retry=0.01);

   %if &_type=lock %then %do;
      %* set start time;
      %local _starttime;
      %let _starttime=%sysfunc(datetime());
      %* try locking until lock is obtained or until timeout is exceeded;
      %do %until(&syslckrc=0 or
                 %sysevalf(%sysfunc(datetime())>(&_starttime + &_timeout)));
         options noerrorabend;
         lock &_member;
         options errorabend;
         %* pause before retrying;
         %let sleep=sleep(&_retry.,1);
      %end;
   %end;
   %else %do;
      %* release lock;
      lock &_member clear;
   %end;
%mend;
```

Stored Processes

Stored Processes were introduced in SAS 9 and are similar to a SAS macro, except they have some extra information attached. There are 2 parts to a stored process:

1) The SAS code, which is run when the stored process is executed
2) The metadata for the stored process which holds information about the following:
    a. Which server it will run on, which can be either a stored process server or workspace server.
    b. Which users are allowed to run it, as well as which users can change the metadata for the stored process.
    c. What parameters can be used, including any ranges, required parameters and default values.

When a stored process is run, it is actually run on behalf of a user by a special user id. If you have configured SAS in the recommended way then Stored Processes will usually be run under the SASSRV user-id. So if a user called PHIL tried to run a stored process, it would check whether that user was allowed to run that stored process and if so it would be run on the requested server (probably a stored process server) using the SASSRV user-id. This is an important fact to be aware of when designing applications particularly for UNIX systems which are very fussy about permissions.

Creating Stored Processes

When creating a stored process it is often easiest to use Enterprise Guide, since you can use wizards to create code or write your own, test it out and then save the code as a stored process. A wizard will guide you through the process and allow you to specify everything in an easy way.

Another way is to create the metadata for the stored process using the SAS Management Console. This allows everything to be specified, including where the source code is located. You then need to write the source code for the stored process separately and ensure that it is in place when you try to use the stored process. If doing this, then there are a few things you will need to know about the structure of stored processes.

The SAS code for a stored process can be as simple as a normal everyday SAS program. For instance I could have a data step and a proc print in a file called test.sas, and that would be all that was required. In my stored process metadata I would need to point to that code so that when the stored process was run it would load that SAS code in and execute it. However by making use of 3 other lines of code you can get a lot more power out of a stored process.

```
   *ProcessBody;
```
This comment should be placed at the start of a stored process since it will initiate input parameter processing, if there are any input parameters – otherwise it does nothing. Whatever input parameters are passed to the stored process, including any defaults, are inserted in the code at this point when it runs. So if you pass a value in for a parameter called MONTH as FEB, then it is just like having the statement "%let MONTH=FEB" in your code at that point. So you will get a global macro variable defined for every stored process parameter. If you don't include "*ProcessBody;" then values for parameters will be available.

```
   %stpbegin;
```
This macro initializes the Output Delivery System for use from a stored process. By setting various macro variables you can affect what this macro does. For example, by setting the _ODSDEST macro to RTF will cause the macro to produce RTF output.

```
   %stpend;
```
This macro finalizes the ODS output. For example if we had been writing HTML, it would write the final HTML tags such as </body> and </html>.

Of these 3 things, %stpbegin is the most complex to understand since it can make use of about 40 reserved macro variables to control what it does. Some of the more useful of these and ways to use them will be explained later.

Macro variables used with %stpbegin

Some of the following macro variables will be populated by the web application and you can look at the value to use it in your stored process (e.g. _metauser). Other values can be set by you prior to %stpbegin being called, and then the stpbegin macro will make use of the values you set.

1) **_ACTION** – an action for Web application to take (form, execute, properties, background, strip, index, data)
    a) *FORM* - displays custom input form if one exists.
    b) *EXECUTE* - executes the stored process.

   c) *PROPERTIES* - displays the property page, which enables you to set input parameters and execution options and to execute the stored process. This is really useful and flexible when you want to run an unfamiliar stored process
   d) *BACKGROUND* - executes the stored process in the background. Useful if your stored process runs for a long time, especially since browsers will usually timeout after about 3-5 minutes and if your stored process runs longer then you can lose track of it.
   e) *STRIP* - removes null parameters, used in combination with EXECUTE and BACKGROUND.
   f) *INDEX* - displays a tree of all stored processes. This is very useful if you just want to browse all the stored processes that are defined and then select which one you want to run.
   g) *DATA* - displays a summary of general stored process data.
   h) You can combine parameters as follows, e.g.
      i) *_ACTION=FORM,PROPERTIES* … displays a custom input form if one exists, otherwise displays the property page.
      ii) *_ACTION=FORM,EXECUTE* … displays a custom input form if one exists, otherwise executes the stored process.
2) **_DEBUG** – debugging flags. These have a range of possible values:
   a) *Log* – shows the SAS log after the stored process runs
   b) *Time* – shows the real time taken by the stored process at the end
   c) You can combine several _debug flags with commas like this: "_debug=log,time". You can also use SAS/Intrnet style numbers to specify these flags. My favorite is using "_debug=2179". This is a decimal converted from a binary, in which I set bits for various _debug options I want.
3) **_GOPT_DEVICE** – set the goption device parameter. I usually use sasemf for this, although other popular choices are java, activex and png. One nice thing about using sasemf on UNIX is that true type fonts are more easily used from it.
4) **_GOPT_HSIZE** – set the goption hsize parameter. Useful if you want to specify the horizontal graph size precisely.
5) **_GOPT_VSIZE** – set the goption vsize parameter. Useful for specifying the vertical graph size.
6) **_GOPT_XPIXELS** – set the goption xpixels parameter. I usually query my browser to work out the width, allow for any other things taking up space on the screen, and then set the width appropriately. I need to adjust this when I change destinations though, since producing a graph for an RTF document is best done by customizing its size for the page.
7) **_GOPT_YPIXELS** – set the goption ypixels parameter. In addition to the comments for _GOPT_XPIXELS, I use this parameter when I have lots of items I want to put on my y-axis. I can make the graph very long and then display them all clearly as the user scrolls the HTML page down.
8) **_GOPTIONS** – set some SAS/Graph options.
9) **_METAPERSON** – shows the real name of the person associated with the userid in the metadata. Will be *unknown* if there is no association.
10) **_METAUSER** – shows userid that was used to connect to metadata server.
11) **_ODSDEST** – Specifies the ODS destination (default is HTML). Can also be one of (CSV, CSVALL, TAGSETS.CSVBYLINE, HTML, LATEX, NONE (which produces no ODS output), PDF, PS, RTF, SASREPORT, WML, XML or any other tagset destination.
12) **_ODSOPTIONS** – specifies options that are added to the end of the ODS statement. One key use of this is if you want titles and/or footnotes to be included in graphs, since NOGTITLE and NOGFOOTNOTE are default options. You can override them by specifying GTITLE and/or GFOOTNOTE in _ODSOPTIONS.
13) **_ODSSTYLE** – sets ODS STYLE= option.
14) **_ODSSTYLESHEET** - Sets the ODS STYLEHEET= option.
15) **_PROGRAM** - Name of the stored process. This is really useful if you want to build up a link from the current stored process to itself, since this gives you the name of the stored process.

16) **_RESULT** – Specifies what kind of final result is produced by the stored process. It can be one of the following:
    a) *STATUS* – produces no output to client.
    b) *STREAM* - output is streamed to client through the _WEBOUT fileref.
    c) *STREAMFRAGMENT* – just like *stream* but kind of a cut down version. This is not documented but I find it really useful for producing HTML when I want to have more control over my HTML.
    d) *PACKAGE_TO_ARCHIVE* - package is published to an archive file.
    e) *PACKAGE_TO_REQUESTER* - package is returned to the client. The package can also be published to an archive file in this case.
    f) *PACKAGE_TO_WEBDAV* - package is published to a WebDAV server.
    g) *PACKAGE_TO_EMAIL* - package published to one or more e-mail addresses.
    h) *PACKAGE_TO_QUEUE* - package published to a message queue.
    i) *PACKAGE_TO_SUBSCRIBERS* - package published to a subscriber channel.
17) **_SRVNAME –** the host name of the server. This is very useful when you want to write stored processes that can build URLs for links
18) **_SRVPORT** – the port number on which this request was received. Also useful in building up a URL for links.
19) **_STPERROR** - Global error variable, 0 if everything worked properly, otherwise non-zero.
20) **_URL** - Specifies the URL of the Web server middle tier used to access the stored process. Also useful in building up a URL to use with links.
21) **_USERNAME** - the user name obtained from Web client authentication.

POTENTIAL PROBLEMS & SOLUTIONS

Tables with fixed headers

One major problem I have found when developing web applications with SAS is that the tables produced by ODS HTML don't fix the row and column headers. That means that if you have a very wide table and scroll to the right, then you no longer can see the row headers on the left. If your table is very long then as you scroll down you will lose the column headers and so can't tell what the columns are for. SAS have provided a few solutions to this problem which can be found in the SAS usage notes.

The first solution is to use a tagset provided by SAS which actually fixes the row and column headers. This tagset also does some other nice things, like allowing you to sort based on values in columns. I have found that this works well on small tables but once they get large it slows down a great deal and becomes almost unusable.

The next solution is to effectively insert some JavaScript into the HTML table which fixes the position of the row headers and column headers relative to the web page. This works, but the headers appear to jerk about as you scroll – which is not very pretty.

Handling different browsers

Different browsers behave in different ways and you can detect which browser you have and make allowances in your code. During development of my latest web application I tried to support a range of browsers, but discovered odd little differences between them in various areas. Something that worked in one browser would sometimes not work in another one – and to fix it in the other one would require some special work around coding. In the end I decided to only support the company standard web browser, which was Internet Explorer 6, and that made life somewhat easier.

You can see the browser compatibility tables

which show which browsers support the official web standards. It is interesting to note that no browser supports all the standards.

Large graphs fail to be produced

We ran into some problems when trying to produce very large and complex graphs. Even though we had set the **memsize** option in SAS to allow 512meg of memory, we found that some graphs were running out of memory at a much lower level. I had decided to take advantage of the web page delivery by making some graphs very detailed and long, so that you would scroll down the web page to see them all. Anyway, after some investigation I discovered the **maxmemqry** SAS option, which sets the maximum amount of memory that a single SAS procedure can use. This had to be increased so I could produce my extra large and long graphs – then all was well.

Axis labels become too small

Early in development we were looking at a single clinical study, which didn't produce particularly large graphs. But eventually we developed graphs that were comparing many treatments between many different studies. This meant that I ideally want to fit a lot more onto my graph. Producing a horizontal bar chart with 60 items being compared resulted in tiny almost unreadable axis labels. But then it occurred to me that I was using a web browser in which I had a limited width, but an unlimited length. As long as people didn't have to scroll side-to-side, they would probably be happy scrolling down to see more details on a graph. So I set the graph ypixels to be very large and the x pixels to match the width of my screen (to allow for different resolutions). This let me make long graphs. To get the size of my screen (which may differ from user to user) I just used a simple bit of JavaScript which I then delivered to my stored process as a macro variable.

I put the following macro in reports where I am creating graphs. A lot of my graphs were not appearing properly on low res screens, so I adjust the resolution for that. Also, when I change the _odsdest to send the graph to RTF then I adjust the resolution so that the graph comes out making full use of the page in RTF.

```
%macro handle_low_res ;
 %* handle low resolution screen ;
  %if &_gopt_xpixels<450 %then
    goptions xpixels=650 ypixels=425 ;
  %else
    %if &_gopt_xpixels<800 %then
      goptions xpixels=875 ypixels=850 ;
  ;
 %* A4 sized graphs for RTF destination ;
/*A4 in Pixels - 300 dpi (print) = 2480 X 3508 pixels */
/*A4 in Pixels - 200 dpi (default for ODS RTF) =  8.3 x 11.7 inches = 1660 X 2340
pixels */
 %* PA4 is largest paper format that can fit on both A4 and US Letter without
resizing
    it is 210mm x 280mm, 8.26771654in x 11.023622, 1653pixels x 2204pixels (RTF),
ratio of 3:4 - which matches traditional TV screen ;
  %if %upcase(&_odsdest)=RTF %then
    %do ;
      goptions xpixels=1900 ypixels=1425 ;
      options orientation=landscape papersize=("280mm","210mm") ;
    %end ;
%mend handle_low_res ;
```

Making graphs look good on UNIX, while still being produced fast

When developing an application for the web you have to consider various factors. Response time is one important one. People ideally want to see pretty and detailed graphics, but also want their graphics to arrive quickly. The slowest part of producing a graph and delivering it to a web browser is usually the network speed – and so the size of the graph is a major factor. So it becomes a balancing act of weighing up the size of the graphic against the speed it takes to download.

So, select a device driver that produces high quality, yet small file sizes.

Another factor with selection of device driver is the support for fonts. Some device drivers support true type fonts, which usually look by far the best and lift the whole appearance of your graph. Other device drivers don't support true type fonts and so you are reduced to using SAS software fonts, which often don't produce very nice looking graphs.

So, select a device driver that can use true type fonts.

Getting True-type fonts to work in graphs

When I initially began to produce graphs on UNIX (Solaris and AIX) I began to ran into problems with the use of fonts. If you use a SAS software font such as swiss, then that will work in all graphic devices (PNG, JPEG, BMP, TIFF, etc.). However if you want your graphs to look really good and use true-type fonts then you find that they just don't work automatically in most devices. In fact they are not supported at all in most devices, but you can get them to work in certain ones. For instance you can define particular fonts to work with the PNG device, and I have done this but it is a little annoying. However if you use the SASEMF driver then any true-type fonts available on UNIX can be automatically used. At least this is true after they are imported into SAS. You must use **Proc fontreg** to import whichever fonts you want to use:

```
proc fontreg ;
  fontfile "/sas/general/fonts/arial.ttf" ;
run;
```

You can also just import all the fonts in a directory and then use any of them – which is a bit less efficient if you have lots of fonts.

```
proc fontreg ;
  fontpath "/sas/general/fonts" ;
run;
```

Using JavaScript frameworks

There are many JavaScript frameworks (or libraries) available on the internet, mostly which are free. They provide a collection of pre-written JavaScript controls which allow us to build web applications much quicker than we could otherwise do so. We also need much less knowledge to make use of these frameworks than we would to write the functionality from scratch. Common frameworks are: Prototype, script.aculo.us, jQuery, Ext and Dojo Toolkit. There are even libraries available from Microsoft and Yahoo!. You can Google these and find out all about them. You will find that each offer a similar but different range of controls to do all manner of things. For those readers who are familiar with SAS/AF you can think of these libraries as adding SAS/AF functionality for web development.

The one that we have most recently chosen to use is called Ext JS, and you can find it at http://sencha.com/.

This framework has many features available that make building applications much easier. One of the key things that I have used is a JavaScript layout. It lets me design a screen where I can have different parts of the screen used for different things. In the screen shot to the left you can see how we have two areas on the left, including the top one in which items can be expanded and collapsed. On the right we have tabs, and within tabs we have other tabs with various content.

All of this kind of layout can be used with your SAS application. We have a SAS macro that builds the overall layout with various elements we want defined. Then we populate various parts of the framework by calling stored processes within them. For instance, the area in the screen shot about called “north” could contain the output of a stored process. That might let you make some selections and submit them, which could then run a stored process in the area called “west” that would display the HTML generated. This makes for a very powerful and flexible layout which can produce all kinds of applications.

BUILDING INTERACTIVITY USING JAVASCRIPT

Popup windows

Sometimes you will want a secondary window to pop up so that you can make some selections before going back to a main window and applying those selections. This can be done from JavaScript by using window.open. The following line of code shows how we write some JavaScript which opens a stored process in a new window which is small like a popup window.

```
Data _null_ ;
  File _webout ;
  put "<a href='#'
onClick='window.open(""http://&_srvname:&_srvport/SASStoredProcess/do?_program=SBIP
%3A%2F%2FFoundation%2F&env.%2Fbis%2Fmedmon%2Flb_boxplotgroup%28StoredProcess%29"","
;
  put """Menu"",""menubar=no,width=430,height=360,toolbar=no"") ;'>" ;
 run ;
```

Providing status updates

At the bottom left of a web page there is a status area. You can write to this area using JavaScript. There was a macro that used this on a prior page.

```
window.status=”This is a message”.
```

Having multiple buttons on a screen to do different things

I have a filter builder which builds up a complex form for all the user selections. One of the nice features of this is the ability to click on a plus or minus icon to add a new filter line or remove one. These buttons are able to do different things by using the **onClick** attribute to specify some JavaScript to run when they are clicked on. The JavaScript goes and updates a hidden text fields to indicate if the plus or minus was clicked on, and which line it was on. The icon is also a submit button and so the form is submitted with the modified fields so that the stored process can then act on that information.

Example of simple JavaScript interactivity

The following sample code demonstrates a range of things that can be done using JavaScript to add some interactivity. I shall describe each of the features used and what it achieves.

1) In the body section we use onload, which will run a JavaScript function when the HTML page has completed loading. Very useful if you want something to be done once your page loads. We are just using an alert function here which pops up a box with the text in it.

2) Also in the body section we use onunload, which will run a function when the HTML page has been unloaded - which means when it has been closed.

3) We specify onkeypress, which will run the code every time a key is pressed. This will just write in the message area at the bottom of the screen what key was just pressed. This is usually used in case you want to intercept the press of a key and do something based on it. For instance if you displayed a menu then just by pressing ‘3’ you could link to the item at number 3.

4) The onkeyup will run the code specified every time a key that has been pressed is released - and so it comes up. So onkeypress can run code when you press the key down, and onkeyup when you release it. This gives you a lot of control. In this example we run a JavaScript function called checkkey. It will check the key that was released and display a message only for the ‘X’ key.

5) The onmouseover function is used when the mouse moves over the HTML item in which this was specified. Here we use it on a link, and so if we move the mouse over that link then the specified code will run. In this case it runs a function called open_popup, and displays a message at the bottom of the screen.

6) The onmouseout function is used to detect when the mouse moves away from the HTML item in which it was used. So in our example here we call close_popup, which closes the popup window which our open_popup function openned. This means that moving the mouse over the link opens a window, and moving it off that link closes the window.

7) Next we use the onmousedown function to open a window when the mouse button is pressed down.

8) Then we use the onmouseup function to close the window when the mouse button is released.

9) You will also see in the open_popup function that the function we use to open the window has a lot of parameters that will let us open windows with all kinds of things present or not.

10) Note also that we use a data _null_ step to read in all the cookies that are defined and to write them out onto the web page.

```
*ProcessBody;
data _null_ ;
  input ;
  file _webout ;
  put _infile_ ;
cards4 ;
<html>
<head>
<script type='text/JavaScript'>
var popwin = null ;

function open_popup()
{
  if (popwin == null)
    popwin =
window.open('http://xs103:8080/SASStoredProcess1/do?_program=SBIP://Foundation/cmis
/info(StoredProcess)','','top=150,left=350,width=250,height=50,status=0,toolbar=0,l
ocation=0,menubar=0,directories=0,resizable=0,scrollbars=0') ;
}

function close_popup()
```

```
{
  if (popwin != null)
    {popwin.close() ; popwin = null;}
}
function checkKey()
{
  var key=String.fromCharCode(window.event.keyCode) ;
  if (key == 'X')
    { alert("You pressed the X key") ; }
}

</script>
</head>
<body onload='alert("finished loading");' onunload='alert("finished unloading");'
      onkeypress='window.status="key pressed is: " +
String.fromCharCode(window.event.keyCode) ;'
      onkeyup='window.status="key up"; checkKey() ;'>
Pop-up a window with information by moving over <a href='#'
      onmouseover='open_popup(); window.status="Hovering over the link" ; return
true ;'
      onmouseout='close_popup(); window.status=" " ; return true ;'>here</a>.
<p>
Pop-up a window with information by clicking <a href='#'
onmousedown='open_popup();' onmouseup='close_popup();'>here</a>.
<p>
<a href='#' ondblclick='open_popup();'>Double click to open</a>, <a href='#'
onclick='close_popup();'>single click to close here</a>.
<p>
<a href='#' style='font-size="x-large"' onmousemove='open_popup();'>Move mouse over
this to open</a>, <a style='font-size="x-large"' href='#'
onmousemove='close_popup();'>move over this to close</a>.
<p>
Press <b>X</b> to make an alert window pop up.
;;;;
run ;

data _null_ ;
  file _webout ;
  put '<h1>Cookies</h1>' ;
  htcook=htmldecode("&_htcook") ;
  put htcook ;
  put '</body>' ;
  put '</html>' ;
run ;
%STPBEGIN;
%STPEND;
*';*";*/;run;
```

MAKING USE OF EXT JS FROM STORED PROCESSES

One of the most useful things that I have discovered in the last 2 years, since I last did a paper on this topic, is how to really use JavaScript and Flash objects from stored processes. By learning how to make SAS drive these web technologies is very powerful. Traditionally with SAS 8 and beyond we have been able to quite simply have SAS generate HTML to provide output for the web. Through the graphic device drivers available we have been able to add some interactivity very simply by taking advantage of the ActiveX and Java device drivers. These allow us to make a range of graphs which then provide some very good interactive functionality. This is great but has some problems. ActiveX is only supported on Windows platforms, which can be limiting. There is also a potential problem in that data can be edited using the ActiveX driver, and so output can be changed by the end user. There are a range of features available in each, which can be great for users but which you may not want to make available to users.

If users wanted to go beyond interactivity provided through the HTML ODS tag sets and ActiveX/Java drivers, then they would need to do some work and have a far greater understanding of further technologies in order to achieve this. A range of techniques can be used to do this including the following:

- Interfacing to another tool (such as EXCEL) and creating your output there via DDE, OLE, VBA or other techniques
- Creating a custom tagset, perhaps modified from an existing one, in order to implement further functionality
- Passing raw markup to your output in order to add functionality (e.g. passing raw RTF to MS Word)
- Creating custom JavaScript within HTML to implement any required functionality
- Creating HTML to make use of features like the HTML 5 Canvas element (see paper by Edwin van Stein, Astellas).

Two other methods that I have found which I prefer are: Generating Ext JS code and using Flash objects. Why these two? Well, Ext JS is a JavaScript framework which provides a large number of objects that can be used to build an application. There are many JavaScript frameworks, and so people will have their favourites, but I chose Ext JS by surveying the most popular ones and then looking at the application building functionality that each provided at that time. Ext JS provided far more than most others, and at the time of writing I think it still does. As for Flash, there are a lot of products (some which are free) that provide graph functionality in the form of a flash object. They then provide various ways to provide data to the flash object and to control its functionality with simple JavaScript calls. There are Flash objects that provide other functionality too, besdies graphs. Both Ext JS and Flash will give you a huge amount of functionality with fairly minimal coding.

Generating Ext JS code

In order to do something in Ext JS you will usually need some data and some directives telling Ext JS what you want to do with the data.

One simple way to provide data to Ext JS is by using arrays in JavaScript. For example you could use some lines like these ….

```
       // sample static data for the store
       var myData = [
           ['3m Co',                              71.72, 0.02,  0.03,  '9/1
   12:00am'],
           ['Alcoa Inc',                          29.01, 0.42,  1.47,  '9/1
   12:00am'],
           ['United Technologies Corporation',    63.26, 0.55,  0.88,  '9/1
   12:00am'],
           ['Verizon Communications',             35.57, 0.39,  1.11,  '9/1
   12:00am'],
           ['Wal-Mart Stores, Inc.',              45.45, 0.73,  1.63,  '9/1 12:00am']
       ];

       // create the data store
       var store = new Ext.data.ArrayStore({
           fields: [
              {name: 'company'},
              {name: 'price',      type: 'float'},
              {name: 'change',     type: 'float'},
              {name: 'pctChange',  type: 'float'},
              {name: 'lastChange', type: 'date', dateFormat: 'n/j h:ia'}
           ]
       });

       // manually load local data
       store.loadData(myData);
```

However a better and more flexible way would be to use a data store. The simplest data store is a JSON data store. You may be wondering what a JSON data store is. **JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

A simple JSON data store might look like this…

```
{'TotalCount':'2',
  'metaData':{'root':'rows', 'fields':["id","month","sales","wages"]   },
  'rows':[{"id":"1", "month":"1", "sales": 600, "wages": 900},
            {"id": "2", "month":"2", "sales": 800, "wages": 1300}      ]      }
```

A JSON data store can be created by a SAS program, written to a file on the server and then used by the stored process that makes the grid. As an example, lets look at how to make a grid/table in Ext JS. The following HTML (put together by Dimitri Woei) shows what is involved in doing this. All the lines are well commented, so if you work through this carefully you will see exactly what it is all doing.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
    <head>
       <title>ExtJS Grid - SAS Dataset Example</title>
       <!-- Ext JS libraries -->
       <link rel="stylesheet" type="text/css" href="/extjs/resources/css/ext-
all.css"/>
       <script type="text/JavaScript" src="/extjs/adapter/ext/ext-base.js"></script>
       <script type="text/JavaScript" src="/extjs/ext-all-debug.js"></script>

       <script type="text/JavaScript">
          Ext.onReady(function(){ /* When all Ext JS stuff is loaded we can start
defining our application */
             var sasdatasetStore = new Ext.data.JsonStore({ /* data store necessary
to load data from the server */
                id : 'sasdatasetStore' /* id is a unique id of the component so we
can later reference it using Ext.getCmp('sasdatasetStore') */
               ,url: 'dimitri_grid.json' /* stored process to retrieve data */
               ,reader: new Ext.data.JsonReader({ /* reader to process the json data
*/
                   root: 'rows', /* The property in the JSON data which contains an
Array of record objects */
                   id: 'name' /* name of the value which is used as id */
                }, ['name', 'sex', 'age']) /* names of the values */
             });

             sasdatasetStore.load(); /* load data from server */

             var mainwin = new Ext.Window({ /* Create a window where we put
everything, we can later reference this window with the variable name mainwin */
                xtype  : 'window' /* xtype defines the component type*/
               ,id     : 'main_win' /* id is a unique id of the component so we can
later reference it using Ext.getCmp('main_win') */
               ,title  : 'ExtJS Grid - Simple Example' /* title of the window */
               ,width  : 318 /* width of the window self explainatory */
               ,height : 480 /* height of the window self explainatory */
               ,resizable: false /* configure window not to be resizable
(default=true)*/
               ,layout : 'absolute' /* layout type absolute, we need to specify the
positions of the components. */
               ,closeAction: 'hide' /* when we close the window it becomes hidden,
so we don't need to create a new object next time we open it*/
               ,closable : true /* x mark in the top left corner to close the window
(default) */
               ,items  : [ /* here we define the components in the window */
                   { /* grid to show data */
```

```
                      xtype: 'editorgrid' /* this type of grid allows you to edit
the fields */
                     ,id: 'exampleGrid' /* id is a unique id of the component so we
can later reference it using Ext.getCmp('exampleGrid') */
                     ,title: 'People in my family' /* we can put the name of the
table here as a title */
                     ,height: 450
                     ,width: 304
                     ,store: sasdatasetStore /* data store containing the data */
                     ,columns: [ /* define the columns shown in the grid */
                          {
                              xtype: 'gridcolumn'
                             ,dataIndex: 'name' /* name of the variable used as an
index*/
                             ,header: 'Name' /*label of the column header*/
                             ,width: 100
                             ,editor: { xtype: 'textfield' } /* we want to use a
textfield as an editor */
                          }
                         ,{
                              xtype: 'gridcolumn'
                             ,dataIndex: 'sex' /* name of the variable used as an
index*/
                             ,header: 'Sex' /*label of the column header*/
                             ,width: 100
                             /* we didn't specify an editor, so this field is not
editable */
                          }
                         ,{
                              xtype: 'gridcolumn'
                             ,dataIndex: 'age' /* name of the variable used as an
index*/
                             ,header: 'Age' /*label of the column header*/
                             ,width: 100
                             ,editor: { xtype: 'numberfield' } /* we want to use a
textfield as an editor, which allows only numbers */
                          }
                      ]
                     ,sm: new Ext.grid.RowSelectionModel({ /* selection model
needed to handle event when data is selected */
                         singleSelect: true /* only one row can be selected at a
time */
                        ,listeners: {
                             rowselect: {
                                fn: function(sm, index, record) {
                                    /* add your code here what to do when a row is
selected*/
                                }
                             }
                            ,rowdeselect: {
                                fn: function (sm, index, record) {
                                   /* add your code here what to do when a row is
deselected*/
                                }
                             }
                          }
                      })
                  }
               ]
            });
            mainwin.show(); /* show the window */
         });
      </script>
   </head>
<body>

</body>
</Html>
```

Once you understand the basic way that you can use an object like this, then it is quite easy to translate this into a stored process, so that SAS can do the same. This can be very simply done by adding the following code in front of the HTML code that you are using…

```
   *ProcessBody;

   /* generate html code */
   data html_code;
      infile datalines4 length=l;
      input #1 htmlline $varying400. l;
   datalines4;
```

And then add the next code after the HTML code you are using…

```
   ;;;;
   run;
   /* stream html code to browser */
   data _null_;
      file _webout;
      set html_code;
      put htmlline;
   run;
   *';*";*/;run;
```

This then gives you the SAS Stored Process code required to run a Stored Process, putting all your HTML code into a dataset, and then writing it to the web browser. You can see how this technique can easily then be extended to create parts of the HTML and JavaScript code before it is written out.

An even better way to do this is to have another SAS stored process generate the JSON data store on the fly. A stored process to make a JSON data store could look like this.

```
   *ProcessBody;

   /* stream data to browser in JSON format */
   data _null_;
      file _webout;
      set sashelp.class end=last;
      if _n_ =1 then
         put "{ success:true, rows:["; /* return data retrieved was a success, rows is
   an array containing all the data */
      else
         put ","; /* seperator for each row in the data */
      put "{ name: '" name+(-1) "', sex: '" sex+(-1) "', age: '" age+(-1) "'}";
      if last then
        put "]}"; /* close array and JSON dataset */
   run;
   *';*";*/;run;
```

The Ext JS to use this would then look like this. It could replace the Ext JS for the sasdatasetStore variable that was shown in the previous complete version.

```
var sasdatasetStore = new Ext.data.Store({ /* data store necessary to load data from
the server */
                  id : 'sasdatasetStore' /* id is a unique id of the component so we
   can later reference it using Ext.getCmp('sasdatasetStore') */
                 ,url: 'do?_program=/CBA/dimitri_test2' /* stored process to retrieve
   data */
                 ,reader: new Ext.data.JsonReader({ /* reader to process the json data
   */
                     root: 'rows', /* The property in the JSON data which contains an
   Array of record objects */
                     id: 'name' /* name of the value which is used as id */
```

```
                }, ['name', 'sex', 'age']) /* names of the values */
            });
```

You could also create a macro to create JSON data stores, so then it could be used in various stored processes as require. All of this means that by building 2 stored processes you can make a single URL that can be called and will deliver a nice looking grid with all the functionality that the Ext JS grid provides as standard, including:

- column sorting, on host or client
- excluding columns
- paging long tables

There are many other features that can be added to Ext JS grids by adding a little more complexity to your JavaScript. This can then achieve things like traffic lighting, putting special objects into table cells and so on.

As an example, the following code shows how to implement traffic lighting with an Ext JS grid.

```
        /**
         * Custom function used for column renderer
         * @param {Object} val
         */
        function change(val) {
            if (val > 0) {
                return '<span style="color:green;">' + val + '</span>';
            } else if (val < 0) {
                return '<span style="color:red;">' + val + '</span>';
            }
            return val;
        }

    function pctChange(val) {
            if (val > 0) {
                return '<span style="color:green;">' + val + '%</span>';
            } else if (val < 0) {
                return '<span style="color:red;">' + val + '%</span>';
            }
            return val;
        }
```

The best place to find useful examples of Ext JS is in the samples section of their web site, which contains many samples along with full source code. http://www.sencha.com/products/extjs/examples/

Another great thing you can do with Ext JS is graphics. This is really a cross over area between Flash and Ext JS since Ext JS implements its graphs using flash technology. Ext JS provides some flash objects to implement their graphics in Ext JS version 3. However in version 4, which came out early 2011 they have changed to a completely JavaScript based graph technology. However Ext JS 3 provides a nice way to demonstrate how to do graphs in Ext JS with flash.

The following code is a very simple example (put together by Chris Brooks) which demonstrates almost the bare minimum required to produce an Ext JS Graph. The HTML (with JavaScript embedded) also required another file which has the JSON data.

```
    <html>
    <head>
        <title>Chart Example</title>
        <link rel="stylesheet" type="text/css" href="extjs/resources/css/ext-all.css"
    />
        <script type="text/JavaScript" src="extjs/adapter/ext/ext-base.js"></script>
        <script src="extjs/ext-all-debug.js"></script>

        <script type="text/JavaScript">

            Ext.BLANK_IMAGE_URL = "extjs/images/default/s.gif";
            Ext.chart.Chart.CHART_URL = "extjs/resources/charts.swf";

        Ext.onReady(function(){
            var figstore = new Ext.data.JsonStore({
                  url: 'figures.json',
                  root: 'rows',
            //     fields: ['id', 'month', 'sales', 'wages', 'red', 'amber',
'green'],

                  autoLoad: true
            });


            var pnl = new Ext.Panel({
                  title: "Monthly Figures",
                  renderTo: Ext.getBody(),
                  width: 500,
                  height: 300,
                  layout: 'fit',
                  items: {
                        xtype: 'linechart',
                        store: figstore,
                        xField: 'month',
                        padding:0,

                        yAxis: new Ext.chart.NumericAxis({
                              title: "Figures",
                              majorUnit: 500,
                              labelRenderer: Ext.util.Format.numberRenderer("0,0")
                        }),
                        xAxis: new Ext.chart.CategoryAxis({
                              title:'Month'
                        }),

                        series: [
                              {
                              // type: 'column',
                              displayName: "sales",
                              yField: 'sales',
                              style: {color: '#000000'}
                              },
                              {

                              displayName: "wages",
                              yField: 'wages',
                              style: {color: '#FF00FF',size:5}
                              }
                        ],
                        extraStyle:{
                              legend: {
                                    display: 'bottom'
                              }
                        }
                  }
            });
      });
      </script>
</head>

<body>

</body>
</html>
```

The following is the JSON data file.

```
{
        'totalCount':'5',
        'metaData':{
               'root':'rows',
               'fields':["id","month","sales","wages"]
        },
        'rows':[
               {
                      "id":"1",
                      "month":"1",
                      "sales": 600,
                      "wages": 900
               },
               {
                      "id":"2",
                      "month":"2",
                      "sales": 750,
                      "wages": 1000
               },
               {
                      "id":"3",
                      "month":"3",
                      "sales": 800,
                      "wages": 1050
               },
               {
                      "id":"4",
                      "month":"4",
                      "sales": 850,
                      "wages": 1300
               },
               {
                      "id":"5",
                      "month":"5",
                      "sales": 1000,
                      "wages": 1300
               },
               {
                      "id":"6",
                      "month":"6",
                      "sales": 1200,
                      "wages": 1300
               },
               {
                      "id":"7",
                      "month":"7",
                      "sales": 1350,
                      "wages": 1300
               },
               {
                      "id":"8",
                      "month":"8",
                      "sales": 1450,
                      "wages": 1300
               },
               {
                      "id":"9",
                      "month":"9",
                      "sales": 1750,
                      "wages": 1300
               },
               {
                      "id":"10",
                      "month":"10",
                      "sales": 2000,
                      "wages": 1300
               },
               {
                      "id":"11",
                      "month":"11",
                      "sales": 2200,
                      "wages": 1300
               },
               {
                      "id":"12",
                      "month":"12",
                      "sales": 2500,
                      "wages": 1300
               }
         ]
   }
```

USING FLASH OBJECTS

To use flash objects from a web browser we need some HTML and JavaScript with a few key sections in it. There will usually be examples and documentation that make getting started quite easy. As an example we are going to look at using XML/SWF charts. The web site for this product is at http://www.maani.us/xml_charts/index.php - you can go there and see many useful examples, clear documentation and grab the code.

The following is some HTML which will enable a flash graph to be generated. The way this product works is that you specify certain things in the HTML, and then provide an XML file with directives and data to make your graph. A few useful things in the HTML are highlighted:

- Width will define how many pixels wide your graph is
- Height defines the height of your graph
- Bgcolor defines the background color to be used
- Xml_source defines where the XML file is which describes how to make your graph, and contains data

```
<HTML>
<script language="JavaScript">AC_FL_RunContent = 0;</script>
<script language="JavaScript"> DetectFlashVer = 0; </script>
<script src="AC_RunActiveContent.js" language="JavaScript"></script>
<script language="JavaScript" type="text/JavaScript">
<!--
var requiredMajorVersion = 10;
var requiredMinorVersion = 0;
var requiredRevision = 45;
-->
</script>
<BODY bgcolor="#FFFFFF">

<script language="JavaScript" type="text/JavaScript">
<!--
if (AC_FL_RunContent == 0 || DetectFlashVer == 0) {
    alert("This page requires AC_RunActiveContent.js.");
} else {
    var hasRightVersion = DetectFlashVer(requiredMajorVersion,
requiredMinorVersion, requiredRevision);
    if(hasRightVersion) {
            AC_FL_RunContent(
                    'codebase',
'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=10,0,4
5,2',
                    'width', '400',
                    'height', '250',
                    'scale', 'noscale',
                    'salign', 'TL',
                    'bgcolor', '#777788',
                    'wmode', 'opaque',
                    'movie', 'charts',
                    'src', 'charts',
                    'FlashVars', 'library_path=charts_library&xml_source=sample.xml',
                    'id', 'my_chart',
                    'name', 'my_chart',
                    'menu', 'true',
                    'allowFullScreen', 'true',
                    'allowScriptAccess','sameDomain',
                    'quality', 'high',
                    'align', 'middle',
                    'pluginspage', 'http://www.macromedia.com/go/getflashplayer',
                    'play', 'true',
                    'devicefont', 'false'
                    );
      } else {
            var alternateContent = 'This content requires the Adobe Flash Player. '
            + '<u><a href=http://www.macromedia.com/go/getflash/>Get Flash</a></u>.';
            document.write(alternateContent);
      }
}
// -->
</script>
<noscript>
    <P>This content requires JavaScript.</P>
</noscript>

</BODY>
</HTML>
```

The simplest possible XML file you can specify is as follows.

```
<chart>
</Chart>
```

This file will use some sample data, and make a sample graph.

In this following piece of code we define data for 3 regions (A, B & C) over 4 years. We then use a chart_type XML element to say that we want the first region to be a line chart, and the other two to be column charts (vertical bars). You can see how easy it is to define the data and type of graph.

```
<chart>
   <chart_data>
      <row>
         <null/>
         <string>2006</string>
         <string>2007</string>
         <string>2008</string>
         <string>2009</string>
      </row>
      <row>
         <string>Region A</string>
         <number>5</number>
         <number>10</number>
         <number>30</number>
         <number>63</number>
      </row>
      <row>
         <string>Region B</string>
         <number>40</number>
         <number>20</number>
         <number>65</number>
         <number>90</number>
      </row>
      <row>
```

```
            <string>Region C</string>
            <number>56</number>
            <number>35</number>
            <number>40</number>
            <number>90</number>
         </row>
      </chart_data>

      <chart_type>
         <string>line</string>
         <string>column</string>
         <string>column</string>
      </chart_type>
   </chart>
```

This has shown how to make a very simple graph. You can imagine how with SAS it is very easy to generate an XML file in which we can take a SAS dataset and produce all the data in this XML format. We can then add any other XML tags in to make the graph that we desire. We end up with a high quality and highly interactive Flash based graph.

RISKS

When an innovative new solutions is proposed at a company good managers will usually ask what the risks involved are. For instance, if you press ahead with building an application that uses Ext JS, JavaScript, Flash, HTML and SAS then what can go wrong? Well, based on my experience using this technology at 4 companies I would suggest the following:

- Browser lacks functionality or has bugs. IE6 is still the standard for many companies although it is no longer supported by Microsoft and has many bugs and vulnerabilities. Supporting this has proved to be a real headache at several of my clients.
- Flash unavailable to users and they cant install it.
- Lacking proper web development tools such as JavaScript debuggers and IDE.
- Costs involved with various tools. For some large companies, it doesn't matter how expensive a tool is but merely that there is a cost.
- Introducing new tools/products to a controlled environment. In some companies there is a long and involved process involved in getting a new product accepted. Sometimes there are also support implications such as increased costs.
- Tools such as Ext JS which are free if you make your code public. Some companies have a problem with making their code available to others, although they like the idea of having their software for free.

CONCLUSION

This paper attempts to give some insight into a very large topic – web development using SAS - more specifically it aims to show some techniques for using JavaScript and Flash in your SAS applications. It is not only a large topic, but a fast evolving one. That means that by the time you read this, it may not be totally up to date. I hope it has given you some insight into what can be done and how. To the uninitiated it may seem that SAS is not the best tool to develop a web application. In fact I have run into experienced java developers who have refused to believe this until the evidence was collected and beat over their heads – but we had java developers come to love SAS.

The key to the power of SAS as a web application enabler is the Stored Process web application. Using that it is very easy to make a web based application. By adding some JavaScript to your application you can add more interactivity and automation. By adding some Flash graphics to your application you can get powerful interactive graphs. Combining these techniques you can build a web application as good as anything on offer today.

ACKNOWLEDGMENTS <HEADING 1>

I would like to thank several of my co-workers who have contributed some material that I have used in this paper. They are:

- Chris Brooks
- Rafal Gagor
- Dimitri Woei

Each of these SAS experts also have a strong expertise in web applications development using the Ext JS framework and have been involved in creating innovative and powerful web based SAS applications that have been incredibly well received by their users.

References

There are some very useful sections in the documentation on the SAS web site.

Creating Stored Processes - http://support.sas.com/rnd/itech/doc9/dev_guide/stprocess/program.html

Building a Web Application - http://support.sas.com/rnd/itech/doc9/dev_guide/stprocess/webapp.html

Recommended Reading

Visual Quickstart Guide: CSS, DHTML & AJAX, by Cranford & Teague

Visual Quickstart Guide: HTML, XHTML & CSS, by Castro

Contact Information

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

Name:	Philip Mason
Enterprise:	Wood Street Consultants Limited
Address:	16 Wood Street
City, State ZIP:	Wallingford, Oxfordshire, OX10 0AY, England
E-mail:	phil@woodstreet.org.uk