SAS1091-2017

Build Lightning Fast Web Apps with HTML5 and SAS®

Allan Bowe, SAS consultant

ABSTRACT

What do we want? Web applications! When do we want them? Well.. Why not today? This author argues that the key to delivering web apps ‘lightning fast’ can be boiled down to a few simple factors, such as:

- Standing on the shoulders (not the toes) of giants. Specifically, learning and leveraging the power of free / open source toolsets such as Google’s Angular, Facebook’s React.js and Twitter Bootstrap
- Creating ‘copy paste’ templates for web apps that can be quickly re-used and tweaked for new purposes
- Using the right tools for the job (and being familiar with them)

By choosing SAS as the back end, your apps will benefit from:

- Full blown analytics platform
- Access to all kinds of company data
- Full SAS metadata security (every server request is metadata validated)

By following the approach taken in this paper, you may well find yourself in possession of an electrifying capability to deliver great content and professional-looking web apps faster than one can say “Usain Bolt”.

AUDIENCE

This paper is aimed at a rare breed of SAS developer – one with both front end (HTML / Javascript) and platform administration (EBI) experience. If you can describe the object of object arrays, the object spawner and the Document Object Model – then this paper is (objectionably?) for you!

INTRODUCTION

You are about to receive a comprehensive overview of building Enterprise Grade web applications with SAS. Such a framework will enable you to build hitherto unimaginable things.. Such as converting that stalwart of a SAS/AF application, creation of highly specific drilldown reports, Data Editors, Release Management systems, crazy graph models, metadata managers, and geolocation based reports.

We start our journey by looking at a recommended folder setup and the generic toolsets needed on our mid-tier and application server(s). We then launch into a step by step guide to installing a simple web app in your SAS environment – this serves to illustrate the approach, as well as providing a ‘template app’.

A full section is dedicated to the open source Boemska Data Adapter (h54s), with an overview of how it interfaces with the Stored Process Server and the various logging facilities.

We finish with a whirlwind of additional tips to help with development, debugging & deployment of web apps, as well as advice for how you can optimize your system to ‘run faster’ – both server and client side.

SERVER SETUP

Before launching into the app itself, there are a few prerequisites to set up. Recommended file system locations are provided, but you may of course replace these with your own locations as necessary.

FOLDER STRUCTURE

SAS Mid-Tier

Web files (*.html, *.css, *.js) must be served from the web server (mid-tier), the location of which will vary depending on system topology and the specific web server in use. For instance, for JBOSS this is:

```
[jboss root]\jboss-as\server\SASServer1\deploy\jboss-web.deployer\ROOT.war
```

Any files saved in the root can be referenced directly in the URL, eg as follows:

```
  http://dev-sasmidtier.yourCompany.int:8080/somefile.html
```

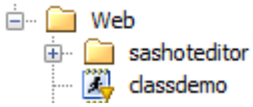
The number of files here will certainly grow, and so it is important to set up a folder structure to manage content. A recommended approach is as follows:

```
  /apps (to contain a subfolder for each distinct app)
  /css  (for shared CSS)
  /js   (for shared javascripts)
  /img  (for shared images)
```

To make it easier to work with this location, you may wish to ask your admin to set up a network share so you can save files there directly. If you are unable to get this level of access, it’s possible to develop on a local machine as described in Bypassing the Same Origin Policy. Another approach is to set up a remote process that continually pulls from your Version Control System (VCS), which has the added advantage of enforcing the use of version control.

SAS Metadata Folder (Tree)

To keep all web app STPs together in one place, it is recommended to create a new (top level) folder, eg /Web. If your app only has one STP then keep it here – if you need multiple STPs (for logic separation, or fine grained permissions control) then create subfolders as necessary.



Note that users will not be able to view (let alone execute) any STPs for which they do not have ReadMetadata permission.

INSTALLING RESOURCES

The great thing about web development is the number of great tools out there you can use for free! With browser caching you may even avoid the download ‘cost’ by referencing a CDN (Content Delivery Network), but if you do this be sure to use an SRI (Subresource Integrity) hash – this is a checksum the browser will use to validate that the external resource has not been tampered with.

Our demo app will make use of four tools in particular:

jQuery

jQuery makes it easy to programmatically traverse / modify your web page, as well as adding event handlers and animations. Furthermore, it handles many of the differences between browsers - avoiding ‘if chrome do this, if IE do that’ type logic.

Our app will take jQuery from a CDN (with SRI checking), so no download required.

Bootstrap

Bootstrap (often called ‘Twitter Bootstrap’ as it was built at Twitter) makes it super easy to style your web app. Your app becomes responsive, with a professional look and feel, by *simply adding standard class attributes* to your html tags. No more time spent twiddling CSS!

Our app references both the Bootstrap files (.js and .css) from the CDN with SRI, so no download here either.

HandsOnTable

One of many javascript libraries available for presenting table-like data, HandsOnTable is particularly noteworthy for it’s spreadsheet-like interface. Defaults are easy to configure, and advanced functionality can be implemented via hook scripts.

Our app will reference this as a local tool, so you will need to download the latest files from their github repo (https://github.com/handsontable/handsontable/tree/master/dist). To get the full distribution (all features) extract the following two files into the respective locations below:

1. `handsontable.full.min.js  ->  /js folder (midtier web root)`
2. `handsontable.full.min.css  ->  /css folder (midtier web root)`

The ‘.min’ part means ‘minified’ – ie the code has been generated in such a way as to minimise file size.

Boemska h54s Data Adapter

The HTML5 for SAS adapter is the final piece of the jigsaw. It handles all the ‘back and forth’ between (Stored Process) server SAS and your client front end, including:

- Super-efficient conversion of Javascript Object Arrays into SAS Datasets and vice versa
- Capture of SAS Logs, with ERROR parsing
- Management of SASLogon redirects

Installation requires two files to be saved in the SAS environment:

- `h54s.min.js -> /js folder (midtier web root)`
- `h54s.sas -> /SASEnvironment/SASCode/Programs (SASApp LevX folder)`
- These files can be found on the github repo: https://github.com/Boemska/h54s

In order for the macros in `h54s.sas` to be available, the following line of code should be placed in the `autoexec_usermods.sas` file for the relevant Stored Process server:

```
/* compile Boemska data connector macros */
%inc "/path/to/SASEnvironment/SASCode/Programs/h54s.sas";
```

The Stored Process server will then need to be restarted in order for the update to be applied.

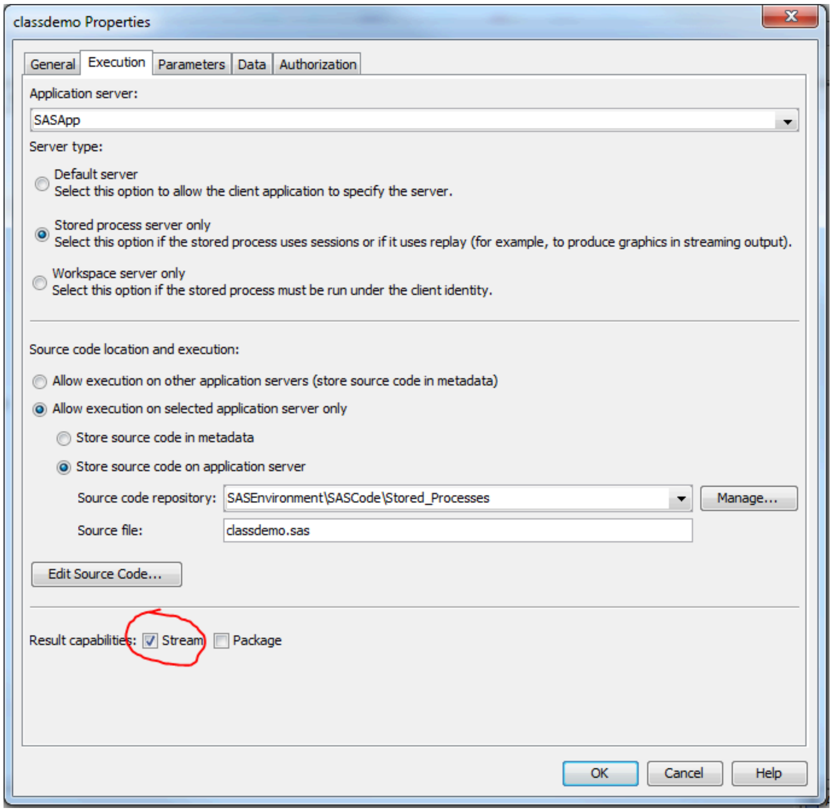
STEP BY STEP GUIDE TO BUILDING A SIMPLE WEB APP

Our web app will have a simple function – to allow a user to select sex (Male / Female) and use this to filter the ubiquitous sashelp.class dataset. This simplicity allows us to focus on the framework by which the app is built – as it is here that the true power of this development approach lies.

STEP 1 – CREATING THE STORED PROCESS

The SAS Stored Process will be the 'calling point' for our client web app. It is the place where permissions are configured, and where the SAS code (or a pointer to SAS code) is stored.

Figure 1 – Stored Process Configuration in SAS Management Console shows the properties of the classdemo Stored Process.



Application server: SASApp

Server type:

○ Default server
Select this option to allow the client application to specify the server.

● Stored process server only
Select this option if the stored process uses sessions or if it uses replay (for example, to produce graphics in streaming output).

○ Workspace server only
Select this option if the stored process must be run under the client identity.

Source code location and execution:

○ Allow execution on other application servers (store source code in metadata)

● Allow execution on selected application server only

○ Store source code in metadata

● Store source code on application server

Source code repository: SASEnvironment\SASCode\Stored_Processes

Source file: classdemo.sas

Result capabilities: ☑ Stream □ Package

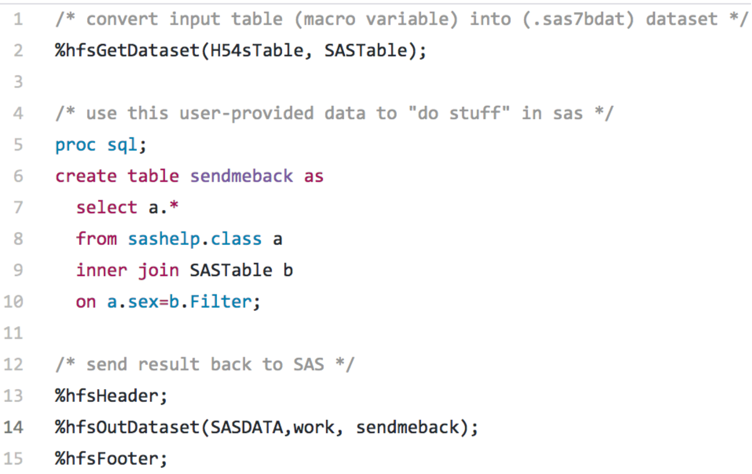
Figure 1 – Stored Process Configuration in SAS Management Console

It is important to set "Result capabilities" to "Stream". Without this, the special _webout fileref cannot be used to send data back to the browser.

Note that the Stored Process is saved under a top level folder (`/Web`) in the BIP metadata folder tree. Keeping web apps in one place can make administration easier, but is not required. Any root folder can be used, and can even be configured as a global parameter (`metadataRoot`) within the h54s adapter.

STEP 2 – WRITING OUR SAS CODE

Figure 2 – classdemo.sas file below shows the code that will execute when calling the `/web/classdemo` Stored Process.



```
1   /* convert input table (macro variable) into (.sas7bdat) dataset */
2   %hfsGetDataset(H54sTable, SASTable);
3
4   /* use this user-provided data to "do stuff" in sas */
5   proc sql;
6   create table sendmeback as
7     select a.*
8     from sashelp.class a
9     inner join SASTable b
10    on a.sex=b.Filter;
11
12  /* send result back to SAS */
13  %hfsHeader;
14  %hfsOutDataset(SASDATA,work, sendmeback);
15  %hfsFooter;
```

Figure 2 – classdemo.sas file

Notice that there are no `%stbegin` or `%stpend` macros - these are unnecessary, and would actually prevent data being sent back to the client (via the _webout fileref).

Regarding the rest of the file:

Line 2 executes the h54s macro (compiled in the autoexec) to convert the received data into a SAS dataset.

Lines 5-10 use this received data within the SQL Procedure to filter the sashelp.class dataset.

Lines 13-15 send this new dataset back to the client app (with metadata) in a specific JSON format.

Top Tip: in the `%hfsOutDataset()` macro, always make your javascript return object uppercase (in this case, “SASDATA”). This will avoid any contention with the additional metadata created by the `%hfsFooter` macro.

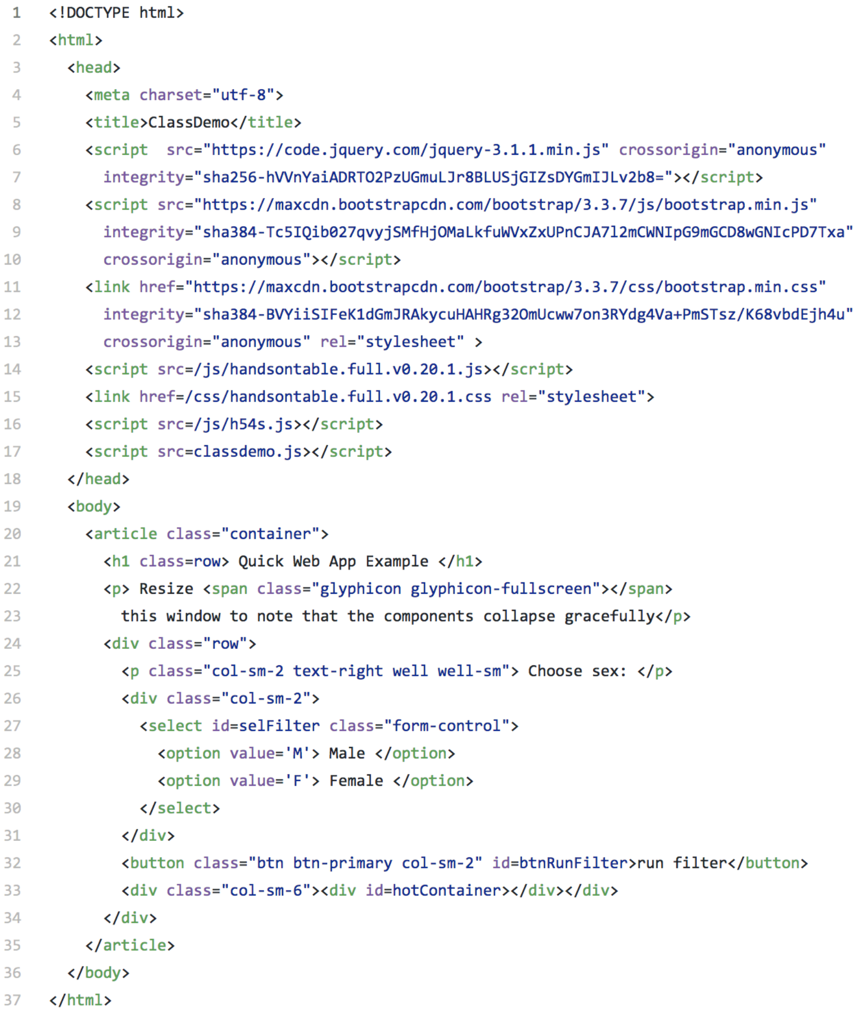
STEP 3 – CREATING YOUR HTML FILE

We are going to save our file as `index.html` under `/apps/classdemo` in the midtier web root location. The advantage of this approach is that we won’t need to reference the filename directly in our URL – if you point at a web directory with a browser it will always look for (and open) the `index.html` file if it exists[1]. So our URL will look like this:

```
http://dev-sasmidtier.yourCompany.int:8080/apps/classdemo
```

Figure 3 shows the entire `index.html` file - which can also found at github.com/rawsas/apps/tree/master/WebRoot/apps/classdemo.

[1] If you're working with Angular or React, index.html is always served with complete minified JS which is responsible for page



```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>ClassDemo</title>
    <script  src="https://code.jquery.com/jquery-3.1.1.min.js" crossorigin="anonymous"
      integrity="sha256-hVVnYaiADRTO2PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8="></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
      integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
      crossorigin="anonymous"></script>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
      integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
      crossorigin="anonymous" rel="stylesheet" >
    <script src=/js/handsontable.full.v0.20.1.js></script>
    <link href=/css/handsontable.full.v0.20.1.css rel="stylesheet">
    <script src=/js/h54s.js></script>
    <script src=classdemo.js></script>
  </head>
  <body>
    <article class="container">
      <h1 class=row> Quick Web App Example </h1>
      <p> Resize <span class="glyphicon glyphicon-fullscreen"></span>
        this window to note that the components collapse gracefully</p>
      <div class="row">
        <p class="col-sm-2 text-right well well-sm"> Choose sex: </p>
        <div class="col-sm-2">
          <select id=selFilter class="form-control">
            <option value='M'> Male </option>
            <option value='F'> Female </option>
          </select>
        </div>
        <button class="btn btn-primary col-sm-2" id=btnRunFilter>run filter</button>
        <div class="col-sm-6"><div id=hotContainer></div></div>
      </div>
    </article>
  </body>
</html>
```

Figure 3 – index.html file

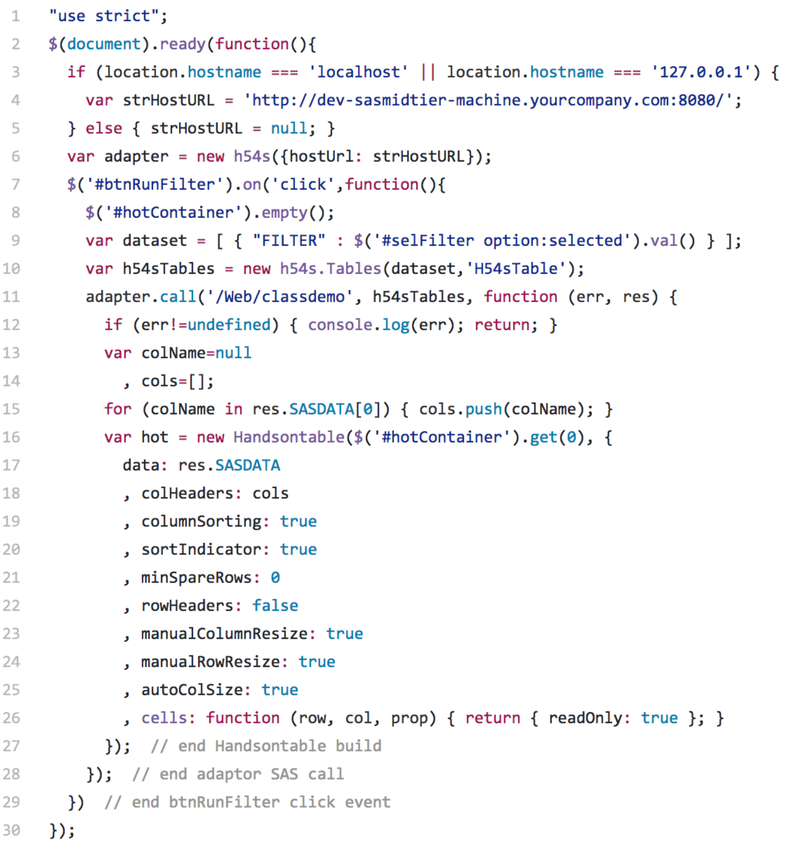
Key points about this markup:

- *All of the class attributes relate to Bootstrap. This library will make your app look good, instantly! Observe that we don't write a single line of CSS in this demo.*
- *The js / css files loaded from external sources (jQuery & Bootstrap) are checked with a subresource 'integrity' (SRI) hash. This guarantees that the code hasn't been tampered with. Where we don't have SRI, the files are hosted internally (HandsOnTable & h54s libraries)*
- *The areas that our App will interface with (selectbox / submit button / HandsOnTable container) all contain id attributes (selFilter / btnRunFilter / hotContainer). This makes them easy to reference.*

STEP 4 – ADDING SOME INTERACTIVITY

Ok, this is where things start to get interesting! The next step is to create our app-specific javascript file, which we will place in the same midtier location – eg: `/apps/classdemo/classdemo.js`. We could have saved it elsewhere but this approach keeps our app-specific files nicely grouped together.

Figure 4 shows the entire file, also available at github.com/rawsas/apps/tree/master/WebRoot/apps/classdemo.



```
1  "use strict";
2  $(document).ready(function(){
3    if (location.hostname === 'localhost' || location.hostname === '127.0.0.1') {
4      var strHostURL = 'http://dev-sasmidtier-machine.yourcompany.com:8080/';
5    } else { strHostURL = null; }
6    var adapter = new h54s({hostUrl: strHostURL});
7    $('#btnRunFilter').on('click',function(){
8      $('#hotContainer').empty();
9      var dataset = [ { "FILTER" : $('#selFilter option:selected').val() } ];
10     var h54sTables = new h54s.Tables(dataset,'H54sTable');
11     adapter.call('/Web/classdemo', h54sTables, function (err, res) {
12       if (err!=undefined) { console.log(err); return; }
13       var colName=null
14         , cols=[];
15       for (colName in res.SASDATA[0]) { cols.push(colName); }
16       var hot = new Handsontable($('#hotContainer').get(0), {
17         data: res.SASDATA
18         , colHeaders: cols
19         , columnSorting: true
20         , sortIndicator: true
21         , minSpareRows: 0
22         , rowHeaders: false
23         , manualColumnResize: true
24         , manualRowResize: true
25         , autoColSize: true
26         , cells: function (row, col, prop) { return { readOnly: true }; }
27       });  // end Handsontable build
28     });  // end adaptor SAS call
29   })  // end btnRunFilter click event
30 });
```

Figure 4 – classdemo.js file

Some explanation:

Line 1 declares the “use strict” directive, which is kind of analogous to “option explicit” in VBA – it helps prevent sloppy programming, such as declaring variables in the global namespace.

Line 2 uses jQuery `.ready()` to ensure the inner function only executes once all the components of the page are loaded into the DOM (Document Object Model) and are ready to manipulate.

*Lines 3-5 are testing to see if our files are being served locally – if so, then we need to provide the location of the SAS web server. This is explained in the section “*Bypassing the Same Origin Policy*”.*

Line 6 instantiates a new instance of the HTML5 for SAS adapter, passing through the SAS midtier location if relevant.

Line 7 implements some interactivity – the click event on our “run filter” button. An anonymous callback function is set to run once the button is clicked.

Line 8 empties the hotContainer div element (in case it already contained a previous table)

Line 9 instantiates a javascript single column / row “dataset” as an object array, containing the value selected in the ‘selFilter’ dropdown

Line 10 instantiates the h54s Tables object, adding our dataset (it could contain several datasets).

Line 11 begins the call to SAS – specifically, to the classdemo Stored Process saved under /Web in the BIP folder tree. The dataset is sent, and the anonymous function receive two objects in return – err (containing any error details) and res (containing the data being sent back from SAS).

Line 12 has some basic error handling

Lines 13-15 create an array containing the column names from the returned SASDATA dataset

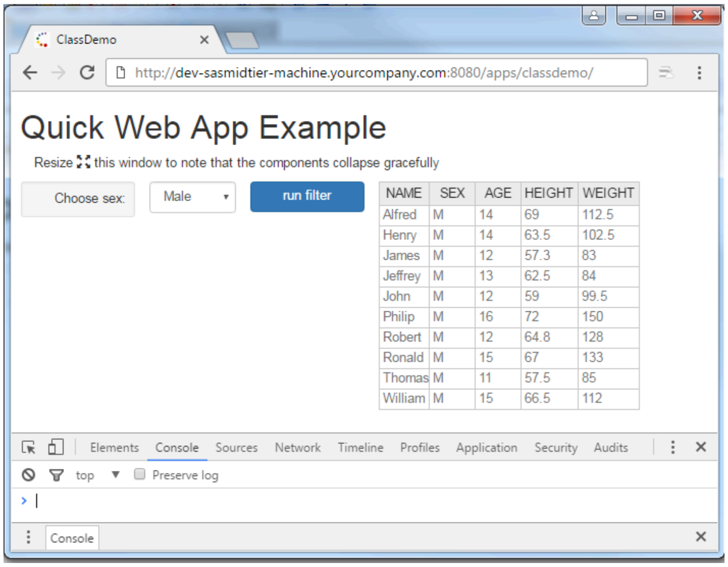
Lines 16-27 configure Handsontable by passing in the target location, the data itself, the array of column names, a number of configuration options, and a hook script to mark cells as read only.

STEP 5 – TESTING THE RESULT

We are finally ready to check out our new app! Enter the below into a browser (substituting your site's relevant midtier server name and port)

```
http://dev-sasmidtier.yourCompany.int:8080/apps/classdemo
```

Figure 5 - our classdemo web application. shows our finished application after selecting "Male" and hitting submit. The data has been returned, and displayed in a nicely formatted table. Developer tools is also open to verify that no javascript errors have been thrown.



Quick Web App Example

Resize this window to note that the components collapse gracefully

Choose sex: Male

run filter

NAME	SEX	AGE	HEIGHT	WEIGHT
Alfred	M	14	69	112.5
Henry	M	14	63.5	102.5
James	M	12	57.3	83
Jeffrey	M	13	62.5	84
John	M	12	59	99.5
Philip	M	16	72	150
Robert	M	12	64.8	128
Ronald	M	15	67	133
Thomas	M	11	57.5	85
William	M	15	66.5	112

Figure 5 - our classdemo web application.

Note that it's always worth testing your new app on a range of browsers - not just to identify functional issues, but also because they often give different feedback on the validity of your HTML.

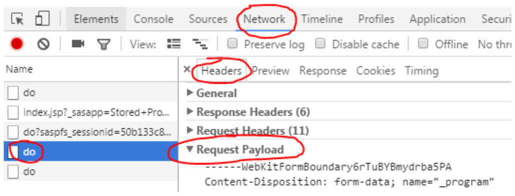
BOEMSKA H54S DATA ADAPTER

The h54s (HTML5 for SAS) adapter makes it super easy to connect to SAS! It provides a generic and transparent communication mechanism for both sending / receiving datasets as well as log handling and SASLogon redirects.

HTTP REQUEST / RESPONSE

Adapter Request

It is possible to examine exactly what is sent to the SAS Mid-tier by the adapter in our demo web app. To do this in chrome, first open Browser Developer tools (F12), then make the request (hit 'run filter' in the app). You can find the request in the Network tab as per Figure 6 – examining payload



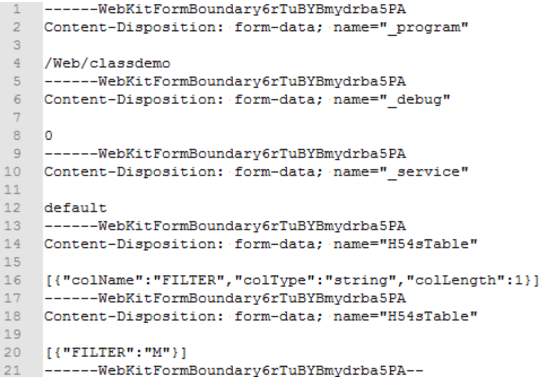
do
index.jsp?_sasapp=Stored+Pro...
do?saspfs_sessionid=50b133c8...
do
do

▸ General
▸ Response Headers (6)
▸ Request Headers (11)
▾ Request Payload

```
------WebKitFormBoundary6rTuBYBmydrba5PA
Content-Disposition: form-data; name="_program"
```

Figure 6 – examining payload

The “Request Payload” appears as follows:



```
1  ------WebKitFormBoundary6rTuBYBmydrba5PA
2  Content-Disposition: form-data; name="_program"
3
4  /Web/classdemo
5  ------WebKitFormBoundary6rTuBYBmydrba5PA
6  Content-Disposition: form-data; name="_debug"
7
8  0
9  ------WebKitFormBoundary6rTuBYBmydrba5PA
10 Content-Disposition: form-data; name="_service"
11
12 default
13 ------WebKitFormBoundary6rTuBYBmydrba5PA
14 Content-Disposition: form-data; name="H54sTable"
15
16 [{"colName":"FILTER","colType":"string","colLength":1}]
17 ------WebKitFormBoundary6rTuBYBmydrba5PA
18 Content-Disposition: form-data; name="H54sTable"
19
20 [{"FILTER":"M"}]
21 ------WebKitFormBoundary6rTuBYBmydrba5PA--
```

Lines 1, 5, 9, 13, 17 and 21 denote boundaries to data, which is essentially FORM data (submitted via POST request).

Lines 2-4 show how the _program parameter was passed (telling SAS which STP to execute)

Lines 6-8 contain the value for the _debug parameter (determines whether SAS will send back additional info with the response)

Lines 10-12 set the _service parameter. This has been configured to prevent JVM errors in some setups.

Lines 14-16 and 18-20 are used to pass our actual dataset. The adapter has examined the content, and determined the type and length of each variable. This metadata is passed first, and then the data, in what will become just two SAS macro variables[2]*.*

SAS Response

The Request Payload (form data) is received by the Stored Process SAS session as macro variables, and can be viewed as such in the SAS log. The pertinent ones for us are:

```
H54STABLE0=2

H54STABLE1=[{"colName":"FILTER","colType":"string","colLength":1}]

H54STABLE2=[{"FILTER":"M"}]

H54STABLE_COUNT=2
```

The above is converted into a SAS dataset (`work.SASTable`) by the following macro:

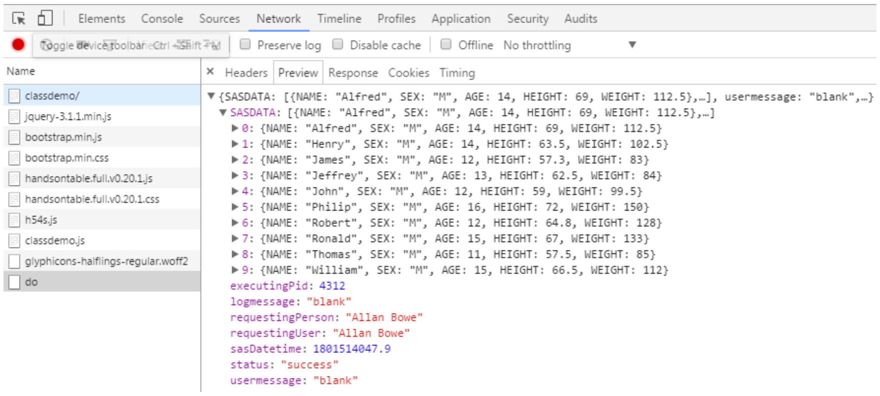
```
%hfsGetDataset(H54sTable, SASTable);
```

The response is sent with a couple of ‘wrapper’ functions that set up the JSON and add some additional metadata. Under the hood, this uses `put` statements against the _`webout` fileref.

```
%hfsHeader;

%hfsOutDataset(SASDATA,work, sendmeback);

%hfsFooter;
```

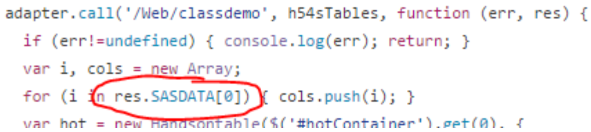
The SPWA streams the data back to the browser, which recognizes it as JSON:

[2] Boemska R&D are examining an even more efficient approach for passing datasets, using the file upload mechanism. Watch this (github) space!



```
▼{SASDATA: [{NAME: "Alfred", SEX: "M", AGE: 14, HEIGHT: 69, WEIGHT: 112.5},…], usermessage: "blank",…}
  ▼SASDATA: [{NAME: "Alfred", SEX: "M", AGE: 14, HEIGHT: 69, WEIGHT: 112.5},…]
    ▶0: {NAME: "Alfred", SEX: "M", AGE: 14, HEIGHT: 69, WEIGHT: 112.5}
    ▶1: {NAME: "Henry", SEX: "M", AGE: 14, HEIGHT: 63.5, WEIGHT: 102.5}
    ▶2: {NAME: "James", SEX: "M", AGE: 12, HEIGHT: 57.3, WEIGHT: 83}
    ▶3: {NAME: "Jeffrey", SEX: "M", AGE: 13, HEIGHT: 62.5, WEIGHT: 84}
    ▶4: {NAME: "John", SEX: "M", AGE: 12, HEIGHT: 59, WEIGHT: 99.5}
    ▶5: {NAME: "Philip", SEX: "M", AGE: 16, HEIGHT: 72, WEIGHT: 150}
    ▶6: {NAME: "Robert", SEX: "M", AGE: 12, HEIGHT: 64.8, WEIGHT: 128}
    ▶7: {NAME: "Ronald", SEX: "M", AGE: 15, HEIGHT: 67, WEIGHT: 133}
    ▶8: {NAME: "Thomas", SEX: "M", AGE: 11, HEIGHT: 57.5, WEIGHT: 85}
    ▶9: {NAME: "William", SEX: "M", AGE: 15, HEIGHT: 66.5, WEIGHT: 112}
  executingPid: 4312
  logmessage: "blank"
  requestingPerson: "Allan Bowe"
  requestingUser: "Allan Bowe"
  sasDatetime: 1801514047.9
  status: "success"
  usermessage: "blank"
```

The data can finally be used by your client javascript in the response (callback) object:



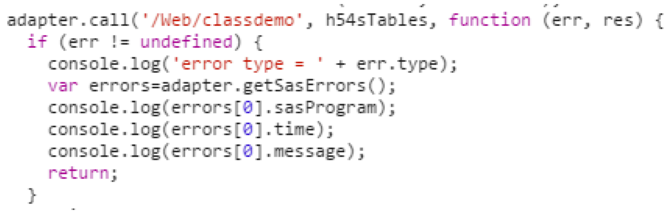
```
adapter.call('/Web/classdemo', h54sTables, function (err, res) {
  if (err!=undefined) { console.log(err); return; }
  var i, cols = new Array;
  for (i in res.SASDATA[0]) { cols.push(i); }
  var hot = new Handsontable($('#hotContainer').get(0), {
```

ADAPTER LOG HANDLING

There are a number of 'log handling' mechanisms available within the adapter, as follows:

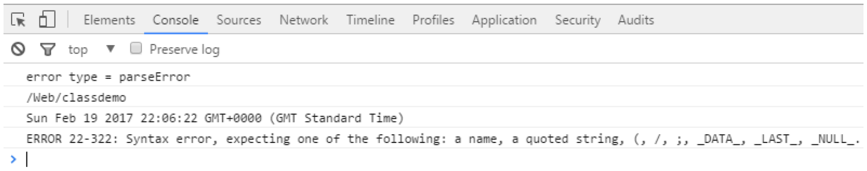
SAS Errors

Whenever there is a SAS program error, the Stored Process server will return the SAS logs by default. The adapter is able to parse these and return not just the error but the time at which the error occurred and the SAS "_program" the error occurred within. This can be surfaced within javascript as follows:



```
adapter.call('/Web/classdemo', h54sTables, function (err, res) {
  if (err != undefined) {
    console.log('error type = ' + err.type);
    var errors=adapter.getSasErrors();
    console.log(errors[0].sasProgram);
    console.log(errors[0].time);
    console.log(errors[0].message);
    return;
  }
```

Adding a syntax error on the SAS side (`classdemo.sas`) now returns the following:

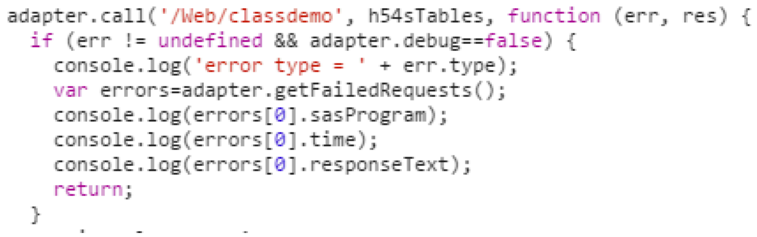


```
error type = parseError
/Web/classdemo
Sun Feb 19 2017 22:06:22 GMT+0000 (GMT Standard Time)
ERROR 22-322: Syntax error, expecting one of the following: a name, a quoted string, (, /, ;, _DATA_, _LAST_, _NULL_.
> |
```

Failed Requests

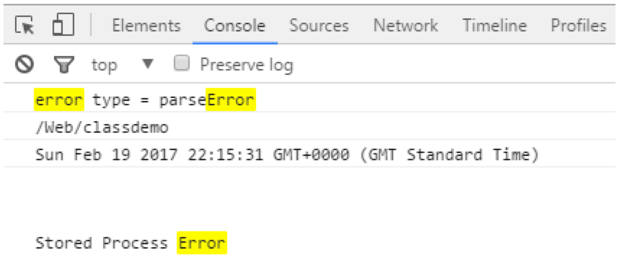
If more context is required, one can query the failed requests array – but only if the global debug option is set to false (which it is, by default)

The setup:



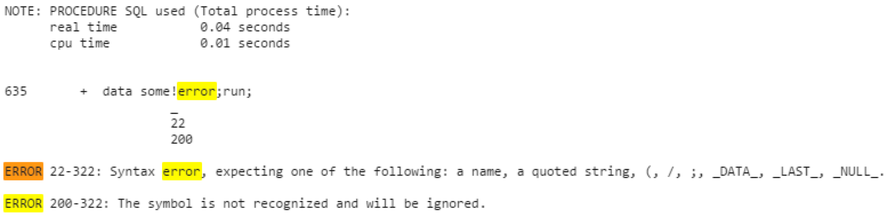
```
adapter.call('/Web/classdemo', h54sTables, function (err, res) {
  if (err != undefined && adapter.debug==false) {
    console.log('error type = ' + err.type);
    var errors=adapter.getFailedRequests();
    console.log(errors[0].sasProgram);
    console.log(errors[0].time);
    console.log(errors[0].responseText);
    return;
  }
```

The response (to my deliberate error):



```
error type = parseError
/Web/classdemo
Sun Feb 19 2017 22:15:31 GMT+0000 (GMT Standard Time)



Stored Process Error
```

……

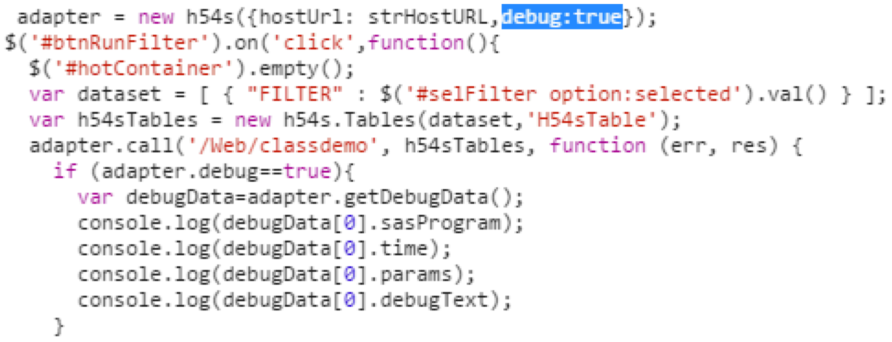


```
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.04 seconds
      cpu time            0.01 seconds


635       +  data some!error;run;
                      _
                      22
                      200

ERROR 22-322: Syntax error, expecting one of the following: a name, a quoted string, (, /, ;, _DATA_, _LAST_, _NULL_.

ERROR 200-322: The symbol is not recognized and will be ignored.
```

Debug Data

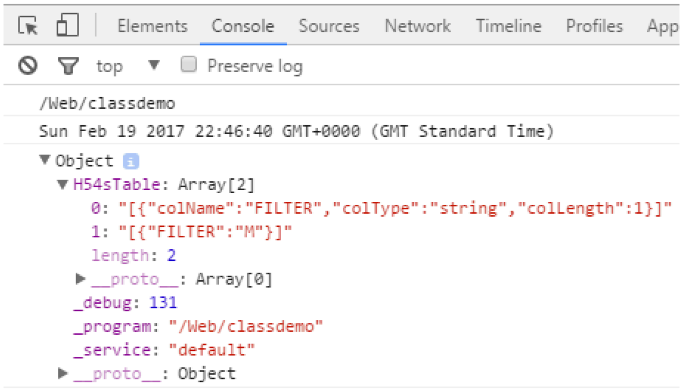
By setting debug to true in the adapter, it is possible to receive logs for every request. This is achieved at the back end by setting the SAS `_debug` parameter to 131 (equivalent to the value LOG,TIME,FIELDS). The adapter will not only store all these logs, it will even store the corresponding request parameters!

The setup:



```
  adapter = new h54s({hostUrl: strHostURL,debug:true});
 $('#btnRunFilter').on('click',function(){
   $('#hotContainer').empty();
   var dataset = [ { "FILTER" : $('#selFilter option:selected').val() } ];
   var h54sTables = new h54s.Tables(dataset,'H54sTable');
   adapter.call('/Web/classdemo', h54sTables, function (err, res) {
     if (adapter.debug==true){
       var debugData=adapter.getDebugData();
       console.log(debugData[0].sasProgram);
       console.log(debugData[0].time);
       console.log(debugData[0].params);
       console.log(debugData[0].debugText);
     }
```

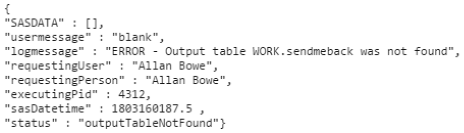
The response is below – note that the request parameters are returned in object format.



```
/Web/classdemo
Sun Feb 19 2017 22:46:40 GMT+0000 (GMT Standard Time)
▼Object
 ▼H54sTable: Array[2]
    0: "[{"colName":"FILTER","colType":"string","colLength":1}]"
    1: "[{"FILTER":"M"}]"
    length: 2
  ▶__proto__: Array[0]
  _debug: 131
  _program: "/Web/classdemo"
  _service: "default"
 ▶__proto__: Object
```

Log Strings

The adapter has some SAS side logic to identify obvious issues - such as the return dataset not being created - and will return helpful messages directly via the JSON response (`res.logmessage`):



```
{
"SASDATA" : [],
"usermessage" : "blank",
"logmessage" : "ERROR - Output table WORK.sendmeback was not found",
"requestingUser" : "Allan Bowe",
"requestingPerson" : "Allan Bowe",
"executingPid" : 4312,
"sasDatetime" : 1803160187.5 ,
"status" : "outputTableNotFound"}
```

CHARGING UP SAS SERVER PERFORMANCE

If we are aiming for an ‘immediate’ response time of 0.1 to 0.2 seconds[3], a few configuration tweaks may be necessary.   Ignoring hardware, there are a several things that can make SAS more responsive:

REDUCE LOGGING

If you don’t need to keep the SAS logs for every stored process that executes, you should generally keep server logging switched off. This is mainly an issue with the StoredProcessServer logs. Details of the config file can be found under Logging.

REDUCE STARTUP CODE

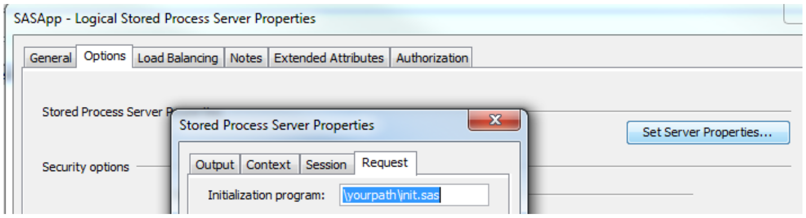
There are usually several SAS programs that are executed before your STP even begins. These should be kept as empty as possible, especially the initialization file, which is guaranteed to execute for every single request.

Autoexec files

The full list of autoexec files can be found in the sasv9.cfg file (check sasv9_usermods.cfg also). Long running code is only really a problem here though if you are not using pooled sessions.

Initialisation program

An initialisation program can be configured (via SAS Management Console) to execute for every single Stored Process request received. This can be useful (eg for dealing with the Locale Gotcha), but as this affects EVERY request, any code here should really be written such that it can execute in 0.01 seconds or less.



Initialization program: \yourpath\init.sas

[3] Miller, 1968: http://theixdlibrary.com/pdf/Miller1968.pdf

MULTIBRIDGE CONNECTIONS

A connection maps a server process to a port. By default, the stored process server comes with one Bridge connection, and three Multibridge connections. All stored process client requests are made initially to the Bridge connection. They are picked up by the object spawner, which determines which server process has the least load, and then redirects the client to the appropriate Multibridge connection.

Whilst each Multibridge connection can process up to five requests (configurable), you may quickly find that - with more than a few users, and any long running Stored Processes - this becomes a serious bottleneck!

It is recommended[4] to add up to 5 Multibridge connections per CPU core.

DEDICATED STP SERVER

In some environments you may find that you cannot avoid startup code as it is there for legacy reasons. You may also find that the permissions on the default STP OS account (`sassrv`) are too restrictive for your needs.

In this case you might consider setting up a Stored Process Server (eg `SASApp_Web`) which is dedicated to web apps! You can give it a special OS account (eg `sassrv_web`) via the SAS General Servers group. You can demonstrate to IT Security that this is safe, due to the rigorous controls you place on the Stored Processes and code that it executes. You can then minimize the startup code, maximize the multibridges, and configure logging as appropriate.

DEVELOPMENT TIPS

BYPASSING THE SAME ORIGIN POLICY

The Same Origin Policy is a security setting enabled in all browsers that prevents a javascript loaded from Server A interfacing with Server B. If this was possible, then a renegade www.dodgysite.com site could easily manipulate your concurrent www.myimportantbank.com session!

If you are modifying files directly in the root of the mid-tier, then this section does not apply to you (as you are serving from the same origin). As a developer though, it is quite possible that you will want to spin up a local web server for building your app. In this way you can make & test front end changes in isolation, without affecting other users. In some corporate environments, this may also be necessary if you are not granted direct write access to the SAS web root.

You know you are affected if you see the following in your console:

```
XMLHttpRequest cannot load http://SASMIDTIER:8080/SASStoredProcess/do.
No 'Access-Control-Allow-Origin' header is present on the requested resource.
Origin 'http://localhost:54048' is therefore not allowed access.
```

One way around this in Chrome is to launch the browser from the command line with this feature disabled:

```
"C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --user-data-
dir="C:/Chrome dev session" --disable-web-security
```

Another option is to download the "Web Server for Chrome" extension, which gives you a checkbox for this very feature ("Set CORS Headers" checkbox under "Advanced Settings").

[4] http://support.sas.com/resources/papers/proceedings12/378-2012.pdf

Whichever way the same origin policy is bypassed, our app will need to know where to go to speak to SAS – as it can no longer derive it from the hostname. The following javascript logic detects if we are working locally, and configures the h54s adapter to react accordingly:

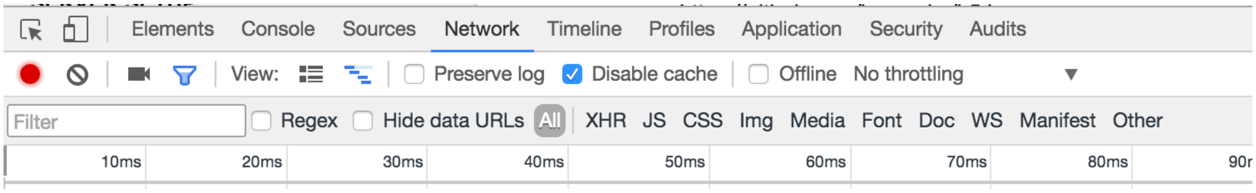
```
if (location.hostname === 'localhost' || location.hostname === '127.0.0.1') {
  var strHostURL = 'http://dev-sasmidtier-machine.yourcompany.com:8080/';
} else { strHostURL = null; }
 var adapter = new h54s({hostUrl: strHostURL});
```

A more permanent option (say, if you do need to serve your production files from a server other than SAS) is to actually enable CORS on the server itself – this can be achieved via SAS Management Console (Application Management -> Configuration Manager -> SAS Application Infrastructure) as described in the 9.4 mid-tier admin guide[5]

BROWSER DEVELOPER TOOLS

All modern browsers come pre-installed with a nifty interface to show you what exactly is going on 'under the hood'. Simply press F12 to fire it up (and F5 to refresh, so you can start logging events).

Figure 7 – Developer Tools



□ Preserve log ☑ Disable cache □ Offline No throttling

□ Regex □ Hide data URLs

The most useful parts are:

Network

This will show you every http request that is sent from the browser, and the corresponding response. If there are any errors in your SAS process, this is a good place to look, as the SAS log is usually returned here by default.

Console

This window is kind of a like a 'command line' for running javascript snippets, as well as a log for events in your main scripts - eg `console.log(someValue)`. Different browsers will also give you different feedback in this window regarding the validity of your HTML – another reason to test your code in different browsers!

DOM Explorer

This has different names depending on your browser, eg "HTML" (IE), "Elements" (Chrome) or "Inspector" (Firefox). The great thing about this is that you can see all parts of your page, as they are rendered, with all their corresponding attributes. So if you are wondering what your javascript is actually doing to the page, you can check it out here (and modify it directly if you wish).

[5] Page 299 (Whitelist of Websites and Methods Allowed to Link to SAS Web Applications) of SAS® 9.4 Intelligence Platform: Middle-Tier Administration Guide, Fourth Edition

CODING STANDARDS

Everybody has different styles for writing code (hey, we're artists right?!) but there are a couple of things worth mentioning:

Line length

Within reason, keep your lines of code to 80 characters or less. Not only does this keep it readable, but it also makes it much easier / quicker to identify differences in your preferred diff tool.

Indentation

Code should always be consistently indented! At risk of Holy War[6], I recommend that your editor (eg the Program Editor in Enterprise Guide, or your text editor) be configured to place 2 spaces when pressing the TAB key.

LOGGING

There are a number of "touch points" at which logging can be enabled. Be sure to switch this off afterwards as logging can be seriously detrimental for performance.

Log Type	Configuration	Info
Object Spawner	`/Lev1/`*`server-context`*`/`*`server-name`*`/logconfig.xml`	Useful for troubleshooting failed connections / permissions issues.
Stored Process Server	`/Lev1/SASApp /StoredProcessServer /logconfig.xml`	Whilst active, will write logs for ALL stored processes.
stpinit program	`Custom SAS code (proc printto)`	With a little conditional logic, one could write a switch to write individual logs for a particular user (&_metaperson) or stored process (&_program)
Stored process program	`Custom SAS code (proc printto)`	The failsafe – put your proc printto in the stored process itself.
Adapter logging	`Adapter setting (debug:true)`	The adapter can be configured to display logs directly via the client application itself!

STORING SAS CODE

Unless you need to do so for security or deployment reasons, I recommend that your actual SAS code is stored in a .sas file rather than within the Stored Process metadata. The main benefit of this is that you can track code changes in your VCS – but it also makes subsequent promotions easier (just a file copy), and your code is then accessible to third party text searching tools (like Agent Ransack).

[6] http://wiki.c2.com/?TabsVersusSpaces

LOCALE GOTCHA

It's important to be aware that the locale of your Stored Process session can switch according to the context of the client[7]. This can cause issues with dates, eg if your process uses the ANYDTDTE. informat.

To circumvent any problems, you can explicitly set your desire locale (so, in my case, `en_gb`) in the initialization program:

```
    options locale=en_gb;
```

VERSION CONTROL SYSTEM (VCS)

Not taking sides here – it really doesn't matter which VCS you use, so long as you do actually use a VCS! Popular choices are GIT, SVN and TFS – it is best to be consistent with what your organization already uses.

A VCS is way more than a complete revision history for your codebase – it can save a ton of time for those who need to maintain / debug / extend your applications, and provides auditability and security for your code. Don't leave `index.html` without it.

RELEASE MANAGEMENT

Every company has it's own approach to Release Management, and it's worth discussing here as it can often take longer to get an application into a Production environment than it does to develop it in the first place.

If you are serious about delivering Lightning Fast Web Applications, then it's important to dedicate resource to the automation of testing and deployment – moving towards a continuous delivery model.

As far as web apps are concerned, automated deployments are **entirely possible**! I'd say it is mostly true with BI / DI deployments too, with a few tricky exceptions. If you are looking to build a one-stop SAS deployment process, a good starting point is SAS® Release Management and Version Control (John Heaton , 2013[8])

CONCLUSION

For non trivial SAS web applications, code can quickly become complex and unmaintainable. This risk / complexity can be significantly reduced by following a standardized approach to app development (naming conventions, file locations, separation of HTML / CSS / javascript / SAS, leveraging the right external tools).

This paper has outlined an Enterprise-grade framework for app development that will not only help you develop fast web apps, fast – but to do so in a supportable and scalable fashion.

RECOMMENDED READING

- *https://github.com/boemska/h54s*
- *SAS® 9.4 Stored Processes: Developers Guide*

[7] http://rawsas.blogspot.co.uk/2016/12/look-out-locale-gotcha.html
[8] http://support.sas.com/resources/papers/proceedings13/467-2013.pdf