PhUSE 2017

Paper AD02

Turn your SAS® programs into Apps using HTML5 and H54S

Nikola Markovic, Boemska TS Ltd, UK

ABSTRACT

The open-source HTML5 Data Adapter for SAS® lets SAS programmers collaborate with UI developers to create modern, secure HTML5 driven application interfaces, using the SAS environments already at their disposal. These apps work across multiple platforms and devices, and allow end users to exploit SAS's data processing potential by means of a modern User Experience rarely offered by other platforms. Within most environments these will benefit from full metadata security, often with single sign-on authentication and out-of-the-box end-to-end encryption of any transmitted data. This presentation will outline how these apps work, what they're capable of, and how to get started with using SAS to develop them.

INTRODUCTION

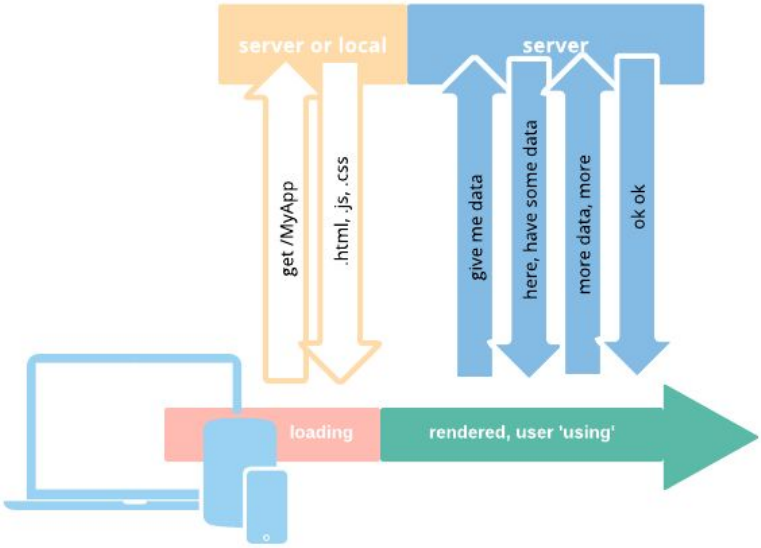
SAS has a long and successful track record of operating within the Pharmaceutical industry, and as a platform satisfies all of the industry's challenging compliance and regulatory demands whilst providing a powerful analytics solution. As a programming language it is the de-facto standard of the industry, and SAS programs exist for almost anything. However, they are *programs*, and while opening up a client application such as SAS Base or Enterprise Guide to run them may be second nature to programmers and developers, the User Experience that that workflow offers to end users leaves a lot to be desired.

In recent years Web Interfaces have come a long way, and the tooling and skillsets required to make them, such as AngularJS, ExtJS, is now mature and widespread. In addition, many SAS environments now come packaged with a Mid tier, containing a Web Server which can both host those Web Interfaces and facilitate their interaction with SAS through the SAS Stored Process Web Application. The last piece of the puzzle, the interface between the two, is the open-source HTML5 Data Adapter for SAS, maintained by Boemska and freely available on GitHub. The rest of this paper discusses the technical details of implementing those interfaces and provides some examples of the results possible with that capability.

WHAT IS A HTML5 APP?

HTML5 is the latest iteration of the HyperText Markup Language - a markup language familiar to most of us as one that's always been used to describe Web Pages. A browser interprets it alongside CSS files and JavaScript code, and its interpretation displayed as a modern web application.

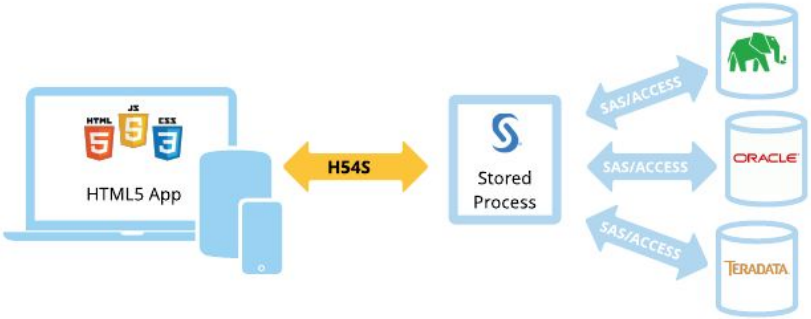
When trying to picture how HTML5 Apps work, it can be easier to just think of them as Web Pages. When an App runs, what really happens is the HTML, CSS and JavaScript files are loaded from a server to the local browser via a HTTP request, the browser's rendering engine renders the page, and the Javascript code on the page *executes,* possibly loading some more data from the server via the same mechanism. Generally, a user will subsequently interact with elements on that page, some other JavaScript code will handle the interaction events, data may be sent back to a server for writing, and new data will be loaded, updating what the user sees.



This mechanism of a web page sending requests for data to a server in the background without reloading the entire page is commonly referred to as AJAX (Asynchronous JavaScript and XML). When trying to 'demystify' AJAX it's worth noting that the way these requests are loaded is no different to how web browsers have always loaded pages when a user has typed something into the browser's address bar and hit enter; where they differ is in the structure of the data that is sent or returned, as there is no need to resend information about other elements of the page which are already displayed and haven't changed.

THE HTML5 DATA ADAPTER FOR SAS

The Boemska HTML5 Data Adapter for SAS (H54S) is a library which helps to facilitate that communication mechanism, acting as a wrapper to native browser AJAX functions and providing a familiar interface to both the HTML5 programmers responsible for creating the application front end interface, and the back-end SAS programmers wishing to make their SAS programs available to a wider audience, while taking advantage of SAS as a readily available Application Development Platform that already exists within their organisation.



The H54S library is therefore split in two parts: a JavaScript library that runs on the client and facilitates the sending and receiving of data, and a set of SAS macros that respond to its requests and handle the data that is sent.

Using this library, JavaScript developers can use a simple, familiar interface to asynchronously send data structures known as *object arrays* (essentially tables) back to SAS from within their apps. SAS-side, these are deserialised into *SAS datasets* by the aforementioned macros inside a Stored Process. SAS developers are then able to write native SAS code that uses those input datasets (as either control tables or actual data input) to service the application requests, returning SAS datasets (these again can be either control tables or actual data) back to the application front end as output. The datasets are returned by SAS as the output of a Stored Process, and then deserialised by the library's JavaScript counterpart to appear as JS object arrays at the front-end.

These *datasets* are intended to replace the macro-variable parameters method of communication used by the SAS Prompting Framework. Things like serialisation, deserialisation, type conversion, exception handling and response validation are all handled by the adapter. For those familiar with Web App Development, what we're doing here is enforcing *datasets* as a primary interfacing format. This provides a familiar interface to SAS programmers and allow the programmers to focus on rapid development of the data backend without worrying about the complex interface specifications mandated by some other technologies. The concept is simple: for each action, we have 'these datasets in, these datasets out' - a concept familiar to most SAS programmers Data Management professionals.

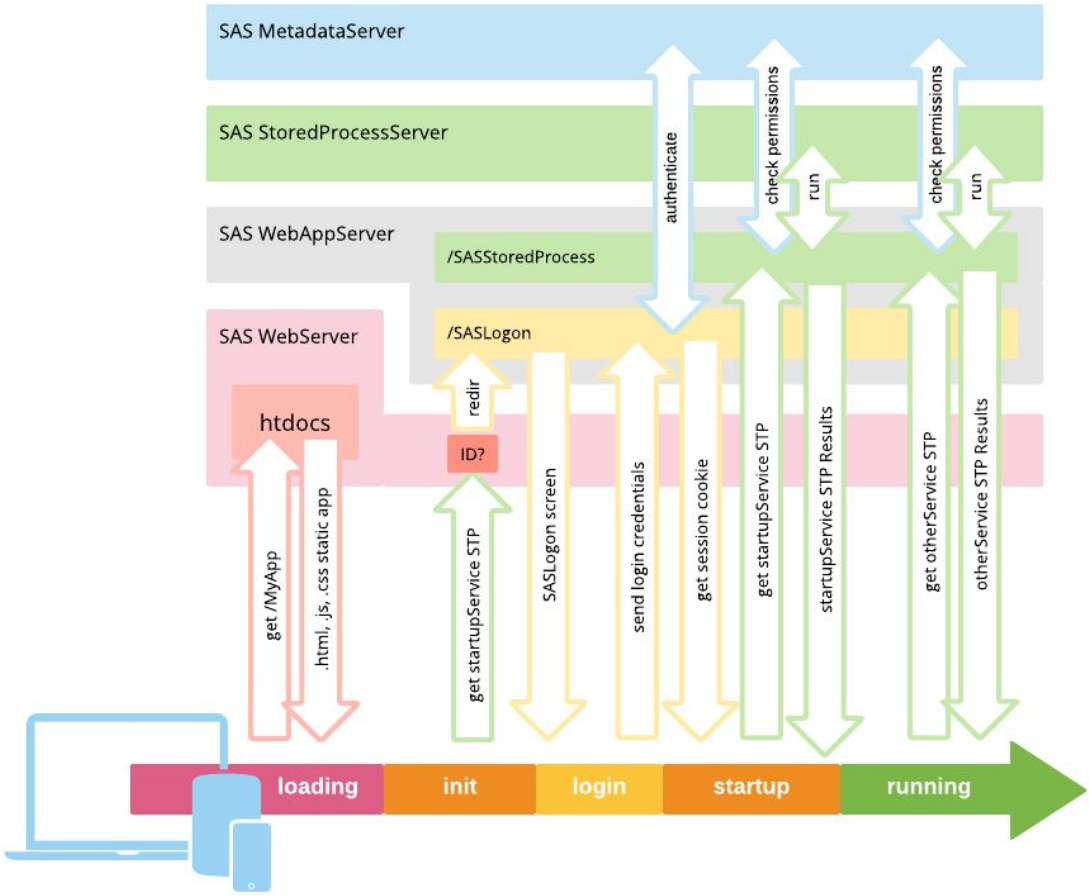
So, how does it work? To make their application contact SAS when it needs something from the server, the front-end JS programmer would write something like the following code:

```
// Instantiate adapter
var adapter = new h54s({hostUrl: 'http://myServer:8080/'});

// some data to send SAS (array of objects)
var myFirstTable = [
  { name: 'Allan', sex: 'M', weight: 101.1 },
  { name: 'Abdul', sex: 'M', weight: 133.7 }
];

// Instantiate a h54s tables object and attach object array to it
var tables = new h54s.Tables(myFirstTable, 'datain');

// make the call to a SAS STP called myFirstService
adapter.call('/Apps/myFirstService', tables, function(err, res) {
  if(err) {
    // if there was an error object returned, handle it here
    console.log(err);
  } else {
    // do what is needed with the tables returned by SAS
    console.log(res.myReturnedTable);
  }
});
```

This code spins up an instance of the adapter, creates a 'tables' object to which one or more object arrays can be attached (from here on also referred to as *datasets*), and using the call() function makes the AJAX call to the named SAS Stored Process. To handle this data, the SAS programmer would write the following code:

```
%include '/pub/sasautos/h54s.sas';

Receive dataset from the client ;
%hfsGetDataset(datain,work.additions);

* Do something with that dataset. Standard SAS Merge and Sort used as an example here;
data mydata;
  set sashelp.class (obs=3 keep=name sex weight) work.additions;
run;

proc sort data=mydata;
  by name;
run;

* Return the output of the above SAS code to the client ;
%hfsHeader;
  %hfsOutDataset(processed, work, myData);
%hfsFooter;
```

The background processing that this library handles actually looks something like this:



The order of events outlined by the above diagram can also be described as:

1. A user requests the app and its .html, .js and .css files are loaded from the server
2. The app *runs.* As it runs, it makes its initial call to SAS (a Stored Process typically called startupService) to get the data it needs before it is ready to be used.
3. If a user has not logged in or their SAS session has expired, that request to the SPWA is redirected to the SASLogon application (just as if a user had attempted to run a Stored Process without being logged in).
4. The app prompts the user for their SAS username and password.
5. This is sent back to the SASLogon application, and upon successful authentication the a new Session ID cookie is returned.
6. The adapter tries to load that initial call again. SAS now checks whether the authenticated user has sufficient Metadata permissions to access the STP that was requested. If so, the STP will be permitted to run. The STP itself may then use some Metadata based authorisation sub-mechanism, such as Metadata-bound libraries or Metadata DATA step functions.
7. The output of the startupService STP is returned to the application front end. Usernames, dropdowns, and other data-driven components are initialized and the user begins to use the app.
8. As the user interacts with the application, steps 6 & 7 above are repeated, reverting to 3 if a user allows their session to expire.

The SAS Stored Process Web Application itself was initially built by SAS with Application Developers in mind (and in many ways was well ahead of its time). As such has many useful features intended for debugging and optimisation. The H54S library abstracts some of these features, making them available as callbacks in the JavaScript frontend; this is done in order to make the developed applications as user-friendly and 'supportable' as possible.

SESSION EXPIRY AND CAPTURE OF LOGON REDIRECTS

With traditional 'linked Stored-Process' based SAS applications, if a user spent too much time entering data into a page of their application, the next time they interact with the the server their request would be redirected to the SASLogon application for re-authentication, in the process losing the data their app was sending to the server, and therefore the context of what they were doing.

The adapter provides a way of mitigating this problem by interrupting background redirects while keeping a cached copy of the data request initially sent to the server. The `call()` method will return an `err` object, with a type property explaining that `err.type = 'notLoggedinError'`. This allows for the session timeout to be handled by the front-end programmer, for example with a modal Login window that prompts the user to re-enter their username and password without losing their application context. The programmer can then pass those credentials to an `adapter.login()` method, which logs the user back into SAS, resuming the request queue and the normal operation of the application.

APPLICATION LOGS AND USER MESSAGES

The Adapter provides facility for the SAS user to maintain a client-side application log for each user (to help support large userbases in production), as well as a way for them to communicate ad-hoc messages such as imminent maintenance outages to the application front end alongside the standard data transmission.

To log application activity to the client-side log, a SAS macro variable named `&logmessage` is simply set from within the code of the SAS Stored Process. This is then added to a client-side log, the retrieval of which can be programmed into the application front-end (under a 'support' or 'debug' menu). The log can be loaded into a displayable object by calling the `adapter.getApplicationLogs()` Javascript function.

By setting a macro variable named `&usermessage` in a similar fashion, a message is transmitted to the front end (again alongside the data that was requested). In the event that it is set, the application can then display that ad-hoc message, within a modal popup alert for example. The usermessage is plaintext, and developers are able to agree conventions on the format of different types of user message, to be handled differently by the front end application (ie. whether informational or important).

DEBUG MODE

The SPWA allows programs to be run in *debug mode*, displaying the SAS logs alongside standard program output. The H54S library abstracts this functionality, allowing the programmers to implement a 'debug mode' into their application front-end. This is intended to make productionised applications both easier to develop and easier to remotely support than using a server-side logging model.

Debug mode within an instance of the adapter is enabled either at instantiation, or by calling `adapter.setDebugMode()`. Similarly to the getApplicationLogs() call above, an array of logs for all calls made to the Server is then accessed by calling `adapter.getDebugData()`.

More documentation on this can be found on GitHub. For examples of what an implementation of this can look like, it is worth looking at some of the demo projects available on GitHub, such as the H54S Angular Seed App.

SECURITY CONSIDERATIONS AND ROLE-BASED ACCESS

A useful benefit of this approach is that, by default, any communication between the application front-end and server back-end is done via secured (HTTPS) channels, to a standard that has otherwise been approved for the secure transmission of sensitive reporting data.

However, one of the greatest benefits here is the ready availability of the in-built SAS Metadata Security layer. Every time an instance of an app makes a call to the SPWA, SAS ensures that that user has *current* SAS Metadata permissions to access the STP service that was requested. This not only means that access to SAS-based applications can be entirely managed using Groups via SAS Management Console, but that, by segregating services into sub-directories with differing Metadata permissions, developers are able to separate functionality within their Apps into different roles - again, managed simply by using SAS Metadata Group permissions.

One of the purposes of the startupService convention mentioned above is to communicate *available functionality* to the application front-end, so that portions of the interface can be disabled (improving UX). This can be done by grouping the functionality for different roles within the application into respective Metadata subfolders, and then creating a 'Permissions' table at startup using the following SAS Data Step-based Metadata lookup:

```
data permissionsOutput(drop=id type);
    length functionPermitted 8.;
```

```
    length id $20;
    length type metadataRoot $256;
    metadataRoot="&_METAFOLDER";
    id=""; type="";
    set listOfRoles;
    functionPermitted=max(0,metadata_pathobj("",metadataRoot,"",type,id));
run;
```

This example code should iterate through a table named `listOfRoles` and validate whether the user currently logged onto SAS has *readMetadata* permissions on any of the subfolders. A `functionPermitted` column is added to the input table, with a value of 1 for available functionality and a 0 for anything that is not currently permitted. This table can be sent to the front end upon application startup, allowing any unavailable functionality to be disabled by the front end prior to any user interaction.

This is a simple, SAS standards-compliant way of solving the often complex problem of managing role-based access to enterprise applications. It allows both top-level access and sub-application role-based access to be managed entirely from SAS Management Console using a standard SAS Metadata Security based approach.

CONCLUSION

SAS programs can, and do, almost anything. With the techniques outlined in this paper, those same programs can be adapted so that non-programmers and business users can interact with them with little to no experience.

CONTACT INFORMATION

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

Nikola Markovic
https://www.linkedin.com/in/nikmarkovic
Email: nik@boemskats.com
Boemska
20 Nugent Road, Guildford
Surrey, GU2 7AF, UK
t: (+44) 01483 331113 (ex. 1000)