Paper AD10

Efficient Implementation and Applications of PROC HTTP in Analysis and Reporting

Simon Todd, Phastar, London, United Kingdom

ABSTRACT

Storing data in the cloud enables users to access information from any suitable device, provided sufficient authorisation is given, and to be able to work simultaneously on collaborative documents to dramatically increase efficiency. Being able to connect directly with cloud storage from within SAS® allows for automation of processes, both increasing the speed and accuracy of information reporting.

This presentation will explain the mechanisms required to enable several processes that use a communication link between SharePoint and SAS code. Examples of how these processes could be implemented as a "system" for use within analysis and reporting are provided to highlight the benefits that come with these applications.

INTRODUCTION

Cloud storage refers to the hosting of files on a remote web-based server rather than on a local device. This allows users to access files from any permitted device, from any location. It also allows files to be worked on simultaneously by multiple users, enabling multiple tasks to be completed in parallel, which can be hugely beneficial when considering the efficiency of a study team.

Microsoft SharePoint® is the cloud storage system that has been used throughout the setup explained in this presentation. SharePoint is a platform designed to integrate Microsoft Office software: Excel, Word, Teams, etc, to allow for improved efficiency [1].

However, because the data is stored on a remote web-based server, it cannot be directly referenced in the SAS code with a simple file path or accessed via a conventional libname. Instead, to communicate with the server in a way that it can interpret, "requests" are made which contain the instructions for what action to take, and a unique address to identify the specific resource. User authentication may also be included to permit access to certain content.

The first part of this presentation will describe how user authentication is controlled when accessing SharePoint via SAS, and the setup required to allow for this to take place. With authorisation in place, the coding setups required to locate and import files from SharePoint to SAS are explored, followed by a look at two different methods for exporting files (dependent on the file size).

Finally, a full explanation for the mechanism required to edit the content of specific cells within an excel spreadsheet stored in the cloud, directly from SAS, whilst it is still available for manual edit is then given. For each of the code setups explored, an example of how these could be used within the general operations of a study team are shared.

The set of code required for integrating SAS and SharePoint is presented as a "system" throughout this presentation. The idea here is to explore how study teams' current working practices can fundamentally be improved by allowing direct communication between SAS processes and documentation and data stored in the cloud. This system was developed to be as flexible and robust as possible for use by a wider study team and to facilitate improvements to accuracy and efficiency by automating processes.

COMMUNICATING WITH SHAREPOINT VIA SAS

Most modern cloud storage systems utilise some form of Application Programming Interface (API) to allow users to more easily set up code to access their content. A common metaphor for the way interaction between SAS and Sharepoint is that the SAS code being written is like a customer at a restaurant, the remote server (SharePoint) is the

kitchen, and the API acts as the waiter, taking the request to the kitchen and then returning with the order. In this context, the API is the part of the server that facilitates communication between the remote server and an external client request. Microsoft SharePoint makes use of a specific API for remote access called Microsoft Graph, which is an example of a "RESTful" web API.

REST API

Representational State Transfer (REST) is an architectural convention or standard for APIs that allow communication and exchange of information between systems on the web [2]. A "resource" within the context of REST APIs is an object or piece of information that is identified by a unique address, such as a Uniform Resource Identifier (URI). To control what is done with a resource, Hypertext Transfer Protocol (HTTP) requests containing a "method" are made to the API. The method is defined from a standardised set of "verbs" (PUT, POST, GET, and DELETE) that provide the instructions for what to do with a resource. For example, specifying a 'GET' method in a request only retrieves data about the specified resource and will not change it in any other way. Once a request has been made, the API will respond with an HTTP return code. "404 Not Found" is a well-known example of an HTTP return code when a requested resource cannot be located and is often seen when browsing the web.

MICROSOFT APPLICATION

To facilitate a request to the API, an application (app) is required, and one app can be used by all members of an organisation to access their data. The app is made using Microsoft Azure and requires logging in with an account specific to the organisation in question [3].

User authentication is required to access files and data stored in SharePoint via SAS in the same way that it would be for accessing via the web. However, rather than each request to the SharePoint API requiring a username and password, an access token specific to each user is used. This represents authorisation being granted to the app by a user to access their data on their behalf [Further reading – 1].

The app has a list of permissions that dictate what actions/requests a user can make to the data that they have access to. These permissions are unique to the app and not to each specific user. There are certain permissions that require administrator privileges to set for an app. For example, the 'Group.ReadWrite.All' permission allows the app to create groups, read all group properties, and also allows the app to manage and update group content for all groups the user is a member of.

There are two key pieces of information that are attributed to the application that are required to generate a user access token. The "tenant" ID, also known as the "directory" ID, is used to identify the Azure Active Directory that the application is saved in (specific to the organisation). Secondly, the "client" ID is used to identify the specific application. Because these values are unique to the application and not the user, they can be stored in a file at the corporate level and accessed as required with a simple file import.

For the first stage of creating an access token, an authorisation URL is created using the two ID variables associated with the application, and then whilst a user is signed in to SharePoint, this URL is submitted. It is at this stage that the user is telling SharePoint that the app is permitted to access the user's files (because they are logged in, authorisation is confirmed) [4]. If authorisation was successful, an authorisation code can be copied out of the resulting updated redirect URL. This access code is temporarily stored in an external file in preparation for the next stage of the setup process (see the "Utility Macros" section below).

CODING SETUP

Once the authorisation code is stored, the non-code based 'setup' stage for this process is complete. PROC HTTP is the procedure in SAS that allows the sending of HTTP requests to the Microsoft Graph API directly from SAS.

Throughout this presentation, many different mechanisms and associated applications of this procedure are explored, but each of them builds off of the core setup below. In this example, a simple request is made to the public beta version of the NCI Cancer Clinical Trials API [Further reading – 2] .

```
%* Temporary filename for request response data *;
filename resp temp;

%* HTTP request to Clinical trials beta API *;
%* Returns top 5 terms that have the term metastatic in the official study
title*;
proc http
   url="https://clinicaltrialsapi.cancer.gov/v1/terms?term=metastatic&
        term_type=official_title"
   method="GET"
   out = resp;
run;

%* Set up libname for JSON file output *;
libname nciterms json fileref=resp;

%* Return dataset containing study information*;
data all_data;
  set nciterms.alldata;
run;
```

Throughout the rest of this presentation, this basic PROC HTTP code block is expanded and adapted for each mechanism, but the fundamental logic seen in this example remains consistent.

PROC HTTP

The working example above is a very simple setup for an HTTP request to a public API that does not require authorisation to access. However, when accessing content with Microsoft Graph, there are additional components of the procedure that are required. Significantly more detail is available in the SAS help documentation [Further Reading - 3].

```
proc http
    url=https://graph.microsoft.com/v1.0/groups/{resources}/{action}{query
    parameter}                                                    [1]
    method="GET"                                                  [2]
    oauth_bearer="&access_token"                                  [3]
    headerin=headin                                               [4]
    out = outfile;                                                [5]
    debug level = 2;                                              [6]
run;
```

1. The **URL** input is the destination for the request. This could be a location like a group folder or drive, or a specific file. These are all referred to as resources, and each 'level' of the URL must be defined.
    a. Each resource is identified by an ID value, and not by the name. As an example. *https://graph.microsoft.com/{Version}/groups/{GroupId}/drives/{driveId}/items/root/children* would be the URL needed to see all the folders and files within the specified group and drive.
    b. At the end of the URL, an additional action parameter can be added after a specific resource (file, drive) to add additional instructions to the request. For example, in a piece of code later, we'll see the phrase '/Content' used at the end of the URL.
    c. Finally, it is also possible to add query parameters to the end of the URL that can be used to control the responses received from the request. Adding $select='DisplayName' subsets the response dataset to only include the variable "DisplayName".
2. The **method** input is used to state which HTTP 'verb' is being used. Examples of these include: GET, PUT, POST, and DELETE.

3. **oauth_bearer** is where an access token is defined as part of the request. This is the approval for the app to use the request to access the protected data.
4. **Headers** allow additional information to be passed to the header of the HTTP request and can be stored in another file as a name and value pair (separated by a colon). This file format is known as a JSON file (see below.)
5. **Out** is the output file destination for the response from the HTTP request. The output is generally used for "GET" methods where content is being requested. In the URL above, if a "GET" request was submitted, the output file would contain information about all files and folders within the drive in the URL.
6. The **debug** statement has 3 levels, each corresponding to increasing levels of detail in the resulting log output. This option is an invaluable tool for understanding the request and response content but in general code operation, caution should be exercised as the access token used (via the oauth_bearer input) is printed to the log.

JSON FILE TYPE

Many input and output files included in this process use a data structure called a JSON file (JavaScript Object Notation.) This a simple human-readable text data format primarily used for exchanging data between a server and application. A JSON data structure will consist of objects (name and value pairs, all enclosed in a set of curly brackets) and arrays (lists of values enclosed in square brackets) [5].

```
{
    "TENANT_ID": "123456789ABCD",
    "CLIENT_ID": "ABC-DEF-123-GHI-654"
}
```

CODING PRINCIPLES

When setting up the suite of macros to enable interaction between SAS and SharePoint, there are several coding principles that are followed to allow for easier validation and integration into a study team's existing code. Each coding process is split into functional macros that carry out specific tasks. This greatly improves the simplicity of reusing a piece of code that is required in many processes and prevents the need for complex input parameters to control case-based behaviour. For example, the URL for a specific file or folder often needs to be derived by taking the names of these as inputs and returning the related ID values. This specific task is wrapped in its own macro and so it is a straightforward process to make a call to this macro whenever a URL is needed.

This setup also makes designing test cases for validation a much simpler prospect as it is considerably more straightforward to control input and expected results with simple task related macros.

The full set of code for this system is held at a corporate level and initialised via an autoexec so that only a single production version of each piece of code ever exists. This greatly reduces the complexity of implementing the system within a wider study team as the exact mechanisms by which the code produces results does not need to be fully understood as there is no reason or opportunity to edit the code itself, only input parameters. This does of course place more importance on high-quality process documentation.

As a consequence of this, careful attention needs to be focused on the concept of 'graceful failure': whenever there is a problem during code execution, it is vital that this is captured and controlled by the system macros prior to termination. This means that at all steps of the code process, thorough error checks are included, with macro breakpoints as required.

There is also significant emphasis on thorough and comprehensive error reporting for the cases when code execution does encounter a problem. The only things that should be able to go wrong with the system are parameters controlled by the user, and so when they do go wrong, the user should be able to clearly understand why.

This system has multiple background processing steps that can be triggered automatically during a macro call. The aim of this project is to keep, where possible, as much of the processing behind the scenes, in order to reduce the impact of introducing such a system to an Analysis and Reporting Team. Where possible, information that is to be re-used is stored at the correct level and can be identified automatically from the executing piece of code, or the user ID running it.

HTTP RETURN CODES

Setting up user error catching and reporting within corporate macros is always a challenge that can often take longer than the development of the functional code itself. However, one invaluable resource for code development and error reporting is the use of HTTP return codes that are returned after each HTTP request. HTTP return codes give detailed responses that explain reasons why a certain request may have failed: for example, if the URL has not been constructed correctly, then "400 Bad Request" would be returned to the log after an attempted request. Using HTTP return codes in conjunction with custom user error reporting results in a significantly easier process for suggesting possible user input corrections. Such return codes are stored in the automatic SAS macro variables.

```
&SYS_PROCHTTP_STATUS_CODE.;
&SYS_PROCHTTP_STATUS_PHRASE.;
```

UTILITY MACROS

A suite of macros was set up by a couple of programmers from SAS (Joseph Henry and Chris Hemedinger) to enable initial creation and subsequent use of a user access token. Further information and source code for these macros can be found in a blog post [6] [Further reading - 4]. Below is a summary of each macro:

- **Process Token:**
  %process_token_file is the first macro and simply imports the JSON file containing the user access and refresh tokens and sets them to macro variables for reuse in the code.

- **Get Token:**
  %get_token is significantly more complex and contains the first example of a PROC HTTP step. Here an authorization "POST" request is made to the Microsoft login URL to get an access token returned. The tenant ID associated with the App forms a part of the URL, along with the token endpoint. This is also where the authorisation code (from the manual step defined previously) is submitted. The response from the request is both an access and refresh token which is then used to imitate a username/password login from the user.
  The access token is then valid until an expiry date is reached or the user password changes. The expiry date can be circumvented by refreshing the access token. As mentioned, the access token (and refresh token) do not need to be created with the manual process of generating an authentication code every time that a user wants to access data in this way. As such, this information can be stored in a file personal to the user and simply read into a SAS program whenever access is required.
  However, this, of course, raises several questions around the security of this process. The access token represents authorisation granted by the user with their username and password. A few considerations regarding security setup options are covered towards the end of the presentation, but in the setup featured in this presentation, the user access tokens are stored in a file within the system user area. As such, to access them someone would already have access to the users account details.

- **Refresh Token:**
  %refresh_token allows the validity of the access token to be refreshed if the expiry time has been reached. The HTTP request in this macro is very similar to the one seen in the *%get_token* macro call, but with a different grant type. In the first request, the grant type is set to auth_code, whereas to refresh access, a refresh token is included in the request and a new updated access token is returned in the response. A call to the *%refresh_token* macro is made prior to any interaction with SharePoint in all processes in this system.

AUTHORISATION CODE ACQUISITION AND STORAGE SETUP

The acquisition of an authorisation code can be directed using a SAS script, reducing the risk of user error. The authorisation URL is generated automatically using the app ID values (tenant and client) and printed to the log so that a simple copy and paste is all that is required to input this into a web browser.

When the URL is generated for a user, a blank JSON file is also created in their SAS user area with only a single object present (authorisation code) and a blank value for the authorisation code. The authorisation code can then be pasted as the missing value from the authorisation URL. During the call to *%get_token*, this JSON file is imported, the authorisation code submitted, and the JSON file deleted to reduce the amount of sensitive information stored for a user.

LOCATING AND IMPORTING A FILE

The process covers how a user can locate a specific file, where the file name and location are known. As mentioned previously, the URL included in an HTTP request is constructed using a series of IDs, so before any action can be carried out on the file (import for example) the full file URL needs to be generated.

For this process, a user must know the file path for the file in question so that the names of each group, drive, or sub-drive can be used to return the corresponding ID value.

The first step is for a GET request to be submitted to return a dataset containing the full set of groups that the user has access too. The request-response is output as a JSON file. This can be attributed to a temporary file reference as there is no need for it to be retained outside of this task. The JSON libname is used so that the response data can be accessed in SAS as a dataset [Further Reading – 5].

By including a query string as part of the request URL, the request is able to contain specific information to allow the user to subset what data is included in the response. This is done to prevent "overfetching." This is the concept of an HTTP request returning more data in the response than the user needs. As an example (below), in this request URL, the user wants only the ID associated with the group called "Example Project". The $filter and $select query strings are used in conjunction to ensure that the response value dataset contains only the DisplayName and ID variables for the specific named group. A full set of query strings are set out by the OData system query options [7].

```
"https://graph.microsoft.com/v1.0/groups?$filter=displayname eq 'Example
Project'&$select=DisplayName,ID"
```

At each level, the newly identified ID value is added to the URL and the process continued until the file itself is found and the file ID is returned.

If the item is going to be used repeatedly, it is prudent to store these ID values in a file at an appropriate level for future import.

The next stage is to use another PROC HTTP "GET" request, but with the "content" action parameter in the URL to indicate that the file data is wanted and not just metadata.

```
https://graph.microsoft.com/v1.0/groups/&groupid./drive/items/&fileid./con
tent
```

The output from this request is attributed to a fileref in the work library so that after the session is over the file is automatically cleared. The file is still currently in "xlsx" format and so it needs to be imported into SAS to convert to a dataset for viewing or data extraction.

FILE IMPORT APPLICATION

A "validation tracker" is a collaborative spreadsheet that allows a team to record the status of all outputs for a deliverable, monitor key dates and program accountability, and also to be the central store for holding the full set of titles and footnotes for each table, figure, or listing (TFL). The tracker is hosted in cloud storage to allow for synchronous editing by all team members.

The requirement here is to import the titles and footnotes from this tracker when executing the output code with minimal input required from the user. To achieve this, we need to acquire the user-specific access token to authorize the request, the URL associated with the project-specific validation tracker, and the program name to identify the row on the tracker with the relevant titles and footnotes.

By importing the titles and footnotes, it completely removes the need for hardcoding within the output SAS code itself and allows for more straightforward consistency checks and updates across all outputs in a delivery package. A key aspect of this process is that a file being open for manual edit by another user does not lock the file and prevent import, thus greatly improving the viability of using this system.

When running a piece of TFL production code, the user profile location is identified using:

```
%sysfunc(pathname(sasuser));
```

The user-specific access token is retrieved from the access JSON file stored within this location and a call is made to *%refresh_token* to refresh the access token in case of expiration.

The validation is held at the project level and so only needs to have the location ID variables derived once. The ID variables are then stored in a JSON file held at project level and so can be imported automatically during program execution. By following standardised corporate folder structures, the location for this file will remain consistent between projects and so the location can be inferred from execution path of the program itself.

```
%sysget(sas_execfilepath);
```

The program name also follows a corporate naming convention and is replicated in the validation tracker. This can be identified from the executing program too:

```
%sysget(sas_execfilename);
```

It is then a straightforward process to subset the imported spreadsheet and return only the required variables. From a user perspective, the only addition to a piece of reporting code is a macro call to trigger this process, the remainder is fully automated. Because this request is for file content, query strings are not applicable to pre-subset the data.

FILE EXPORT

When exporting files to SharePoint, there is no file ID that needs to be located. Instead, only the IDs associated with the location where the file is to be placed need to be found.

A SAS dataset can be exported to SharePoint as an excel file, but this requires a temporary intermediate sheet to be created prior to the export request. The work library is used as the destination for these files to avoid the need to manually delete them after each export process is completed. There are many techniques in SAS to create excel files from SAS datasets, but in this setup, the “xlsx libname” is used so that separate datasets can be directed to different worksheets.

A ‘Drive’ resource represents the document library at the top level in a SharePoint area, in this case, the group identified with the group ID whereas a ‘DriveItem’ resource is an item within a drive. There are two ways to specify a ‘DriveItem’ in a URL, either directly with its ID value, or with its path relative to a known location. In the URL below, the path to the specific file is relative to the ‘Root’ resource’, where ‘Root’ is the top level of a drive hierarchy.

Because this file has not yet been uploaded and so has no ID assigned to it, the actual filename (including extension) is used in the URL. The colons are used to indicate that the ‘DriveItem’ is being referenced based on its relative location.

```
https://graph.microsoft.com/v1.0/groups/&groupid./drive/root:/example.xlsx
:/content
```

Once a file has been uploaded, the ID can be found either from the response from the initial request or from the same location process as with a file import.

In subsequent calls to this code, if the intention is to replace an existing file with a newer version, then the item ID can be used in the request URL instead of the filename and relative path. There would then be no need to include the colons.

```
https://graph.microsoft.com/v1.0/groups/&groupid./drive/items/&export_file
_ID./content
```

One key limitation of this process is a fill size limit for export. Currently, the “PUT” method only allows files up to a size of 4 megabytes. Within the file export process, the “FSIZE” function is used to determine the file size in bytes prior to an export request. This allows the principle of graceful failure mentioned earlier to be followed: the macro is forced to terminate early if a file is too large, prior to the HTTP request itself failing. However, during code development, the HTTP return code “413 -payload too large” would be returned in this case, to help identify what has occurred during the failed HTTP request.

LARGE FILE EXPORT

Fortunately, there is an alternative mechanism available for exporting files greater than 4MB. Rather than using the “PUT” method to upload a file in a single request, a resumable upload session is used. This allows smaller segments of the full file to be uploaded sequentially by submitting multiple HTTP requests with a byte-range declared in each. These file fragments are stored in a temporary location until the full file is uploaded. It is referred to as a resumable upload because once the session has been initiated, it can be returned to should there be a loss of connection.

In the first HTTP request of the process, the “CreateUploadSession” action parameter is used alongside the “POST” method. Once again, colons must be used within this URL. At this point, no file upload has been initiated; the request is simply to prepare the upload session itself.

```
https://graph.microsoft.com/v1.0/groups/&groupid./drive/root:/&export_file
.:/createUploadSession
```

If successful, the request-response returns the upload URL and the return code “200 OK”. This is the temporary file destination and will be the URL for all subsequent upload requests for each of the file fragments.

Once the upload URL is established, the upload of the file can begin, either as a series of file fragments uploaded sequentially or as the full file if the declared byte range is the total file size.

For this request, there are additional headers that must be submitted for the upload to be carried out successfully. The first of these is the content-length, which is just the file size in bytes: the “FSIZE” function can be used to determine this value. The content range determines the part of the file to be uploaded. As a note, the first byte declared in the content range starts at 0, so the upper limit will always be one less than the actual file size.

```
"Content-Length": 8228734,
"Content-Range": Bytes 0-8228733/8228734
```

If a fragment was uploaded successfully but there are still bytes remaining, the HTTP return code will be “202 Accepted” and the response will contain the next expected upload range. This can be used to populate the content range for the next request. If the file upload is complete a return code of 200 is printed and the file ID is returned. An upload session can be cancelled at any time by submitting a request to the upload URL with the “DELETE” method.

The “content-type” header uses the Multipurpose Internet Mail Extensions Type (MIME) convention to indicate the specific file type and format that is being uploaded. As an example, the content-type header when using an upload session for an xlsx file type is:

```
application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
```

It’s possible to find this information by making a “GET” request on an existing specific file type and looking at the content-type header in the response. However, this information is readily available online [7] and a library of these standard content-type values can be set up so that they can be assigned automatically based on the extension of the file being uploaded.

Once an upload session for a file has been started, a new request cannot be initiated by any user to create a file of the same name; a return code “409 – Conflict” would be returned. The upload session must be terminated with a “DELETE” request to the upload URL and a new filename declared before attempting another upload.

FILE EXPORT APPLICATION

Upon receipt of a new raw data delivery, a dataset review can be conducted, and the results exported to an external file. For example, a report could contain all occurrences of non-printable special characters, a set of observations where required variables contain missing values, or more complex RECIST data verification reports could be generated prior to SDTM programming activities. Automating these processes allow for the removal of all human error, speeds up the availability of these reports, and allows team members to simultaneously access them from any device.

SHARING AN EXPORTED FILE

As an extension of this, it is possible to set up automatic file sharing using an HTTP request from within SAS. If the raw data issue report was required to be shared with a data management team, an external JSON can be set up to contain a list of recipients. Within this JSON file, it is possible to control the access level permitted, whether a user sign-in is required, and even a message to accompany the invitation. This JSON file is then the input to the HTTP request, with a "POST" method and the following URL. Note the addition of the "Invite" action parameter.

```
https://graph.microsoft.com/v1.0/groups/&groupid./drive/items/&fileid./invite
```

Automating this file share prior to internal review could carry some risks and these should be carefully assessed before implementing this setup.

WORKBOOK MANIPULATION

As well as exporting complete files to SharePoint, it is also possible to edit individual or ranges of cells in an existing excel spreadsheet stored on SharePoint, whilst the file is being edited manually by other users. This setup requires several setup steps before specific cells can be targeted for an update. The key part of this mechanism is that the excel sheet is treated as a workbook which can be edited and navigated using an active workbook 'session'. This is the same principle as how a workbook stored on SharePoint can be edited directly using excel online, with updates saved automatically, rather than having to download the entire spreadsheet to make an edit before re-uploading it in full.

As with previous mechanisms described above, the first operation is to locate a specific existing spreadsheet by passing the key location names (group, site, drive etc.) to the macro setup for file location. Once a complete URL has been created, ending with the fileID, a "POST" request is needed to initiate the workbook session.

```
https://graph.microsoft.com/v1.0/groups/&groupid./drive/items/&fileid./workbook/createSession
```

The "workbook" and "CreateSession" actions are used at the end of the file URL to tell the API that the file is a workbook and that a workbook session is being initiated. The "PersistChanges" header is set to "True" and needs to be included in the request to ensure that changes that are made to the workbook are saved. At this stage, the content-type is simply "application/JSON" as only the session is being set up, no workbook access has been attempted.

If the request is successful, a session ID variable is returned in the response headers. This value is required as an input header (not URL as with the resumable file upload session) for all subsequent requests to this workbook for this session.

The session ID does have an expiry time associated with it which needs to be considered. During normal code execution post-development, this is not a concern as the processing required is already in place for the user. However, during code development, it is possible to use a "RefreshSession" action with a 'POST' method to refresh the expiration date of the workbook session, in a similar way to how a user access token needs to be refreshed. In this case, a successful request returns the HTTP code: "204 No Content".

The new value for the specific cell that is being updated must be included in the next request, passed to the HTTP procedure as an input JSON file. If a larger range than just a single cell was being updated, a JSON file would be created with a nested array structure so that each value in the cell range can be specified.

The URL included in the edit HTTP request requires specific identification of the sheet name. This can be done using a sheet name without the additional requirements of finding an ID value (although the sheet ID would also work).

The content-type header can also be determined from the result of a "GET" request on that worksheet during code development. However, this content type does not change, so once in place, a user of the system would not need to rederive this.

The action parameter used in the URL for this request is "range" which is then coupled with the address function parameter to allow identification of a specific cell range to edit. For a single cell edit, a cell reference is sufficient (A1 without needing to specify A1:A1.) The final component of this request is to use the "PATCH" method.

```
https://graph.microsoft.com/v1.0/groups/&groupid./drive/items/&fileid./wor
kbook/worksheets/&sheetname./range(address='A2')
```

Please see **Appendix 1** for a full coding setup for a workbook cell value update.

WORKBOOK MANIPULATION APPLICATION

The "validation tracker" referred to during the import application contains the status of all components of a deliverable recorded in a single location. The original version of this tracker was populated manually and included information such as the production and QC programmer names, process completion dates, and item status (complete, QC to review etc.)

By introducing a direct link between SAS and this tracking sheet, significantly more metrics can be recorded for validation purposes. Some examples include the last run date for a program and the creation date for any associated outputs, the user who last ran the code and, the last modification date of the code itself. Recording these values automatically prevents any user mistakes with dataset and output run orders which are critical for the traceability of data included in analysis for submission.

It is also possible to record basic summaries to assist with validating the success of a batch run. Metrics such as whether the log is free of errors and warnings and whether mismatches are present in the compare can also be summarised.

To determine which cell is to be edited, a mechanism for locating a specific value in the workbook is required. Simply importing the workbook using the "content" action parameter and "GET" method (see importing a file) is sufficient but requires an intermediate dataset and additional subset data step.

Continuing with the "workbook" approach, this could also be done using the "UsedRange" action to isolate the cells that have any content in them, and the "ValuesOnly" function parameter to control the data returned in the request-response .

```
https://graph.microsoft.com/v1.0/groups/&groupid./drive/items/&fileid./wor
kbook/worksheets/&sheetname./usedrange(valuesOnly=true)
```

However, the resulting set of values is not arranged in a layout that lends itself to subsetting, as the row values are arranged sequentially in a dataset, but without an associated variable to identify the row.

EXCEL WORKBOOK FUNCTIONS

The preferred technique for sub-setting the data is to make use of the excel workbook functions that are used commonly as formula when manually editing a spreadsheet. A full list of the functions supported by Microsoft Graph API can be found online [Further Reading - 6].

In this application, the "MATCH" excel function is used to return the row number from a cell value array that contains an exact match for a lookup string. Provided that the workbook (in this application, a validation tracker) has a consistent layout, the lookup array containing program identifiers can provided as a macro input.

	A	B	C	D	E
1	Title	Output Name	Production Programmer	Last Prod Run Date	Last Prod Mod Date
2	Patient Randomisation	DS200	AB		
3	Adverse Event Listing	AE100	ST		
4	Patient Demographics	DM200	CD		
5	Exposure Listing	EX100	EF		
6	Medications	CM200	ST		

Consider the validation tracker with column B identifying which output each row corresponds to. If the program AE100.sas was being executed at the time of the workbook macro call, a match function to identify the target row so that column D and E can be updated would be:

```
MATCH("AE100",B1:B6,0)
```

To set this function up using PROC HTTP, the "functions" action parameter is used, followed by the name of the function itself. The inputs to each function must be provided to the request in a separate JSON file.

```
https://graph.microsoft.com/v1.0/groups/&groupid./drive/items/&fileid./workbook/functions/match
```

With JSON input:

```
{
   "lookupvalue": "AE100",
   "lookuparray": {"Address": "Sheet1!B1:B6"},
   "matchtype": 0
}
```

By setting up this mechanism, the cell row can be automatically assigned based on the executing program name. To increase the flexibility one step further, the macro could be updated to take a column name as input and the column value for the lookup array derived too.

CONCLUSION

Integrating SAS code and cloud storage is an incredibly powerful tool. By setting up systems as described in this presentation, validation data can be far more accurately recorded and reports more easily shared. Much of the input to this system can be derived automatically if standard folder structures and naming conventions are followed, but even in scenarios where this is not possible, the only change would be the user having to specify additional information in the macro inputs.

In this presentation, only Microsoft SharePoint was explored, but there are multiple other viable web-based cloud storage providers available for use. Each of those will have its own API system and so the mechanisms presented here would need adapting accordingly. An important consideration in the future when choosing where to host data will be how simple is it to set up the code for users to access their content.

The security of the system also needs careful evaluation. In the setup created for this presentation, the user access tokens are stored as plain text within a file in the user area. However, additional steps may be taken to further improve on the security of this storage. Using a password protected SAS dataset to store the token, with an encoded password (using SAS's PWENCODE procedure) was explored, but ultimately deemed not necessary for the purposes here. Another option would be to look into encoding the access token itself rather than setting up password protection [Further Reading – 7] or other access token security measures [Further Reading – 8].

Finally, the applications shared here are purely to demonstrate some of the potential of this process. There is a wealth of information available about working with the Microsoft Graph API available on their website [Further Reading – 9].

REFERENCES

1. Products.office.com. (2019). *SharePoint, Team Collaboration Software Tools*. [online] Available at: https://products.office.com/en-us/SharePoint/collaboration [Accessed 21 Sep. 2019].
2. Restfulapi.net. (2019). *What is REST – Learn to create timeless REST APIs*. [online] Available at: https://restfulapi.net/ [Accessed 21 Sep. 2019].
3. Azure.microsoft.com. (2019). *Microsoft Azure Cloud Computing Platform & Services*. [online] Available at: https://azure.microsoft.com/en-gb/ [Accessed 21 Sep. 2019].
4. OAuth 2.0 Servers. (2019). *Access Tokens - OAuth 2.0 Servers*. [online] Available at: https://www.oauth.com/oauth2-servers/access-tokens/ [Accessed 21 Sep. 2019].
5. Json.org. (2019). *JSON*. [online] Available at: https://www.json.org/ [Accessed 21 Sep. 2019].
6. Hemedinger, C. and Henry, J. (2018). *Using SAS to access and update files on Microsoft OneDrive*. [online] The SAS Dummy. Available at: https://blogs.sas.com/content/sasdummy/2018/11/29/sas-programming-office-365-onedrive/ [Accessed 21 Sep. 2019].
7. Slick.pl. (2019). *Complete List of All MIME Types — Slick*. [online] Available at: https://slick.pl/kb/htaccess/complete-list-mime-types/ [Accessed 21 Sep. 2019].

ACKNOWLEDGEMENTS

I would like to thank Magnus Mengelbier for the guidance and advice that he provided whilst this project was under development.

FURTHER READING

1. https://www.oauth.com/oauth2-servers/access-tokens/
2. https://www.cancer.gov/syndication/api
3. https://documentation.sas.com/?docsetId=proc&docsetTarget=n0bdg5vmrpyi7jn1pbgbje2atoov.htm&docsetVersion=9.4&locale=en
4. https://blogs.sas.com/content/sasdummy/2018/11/29/sas-programming-office-365-onedrive/
5. https://blogs.sas.com/content/sasdummy/2016/12/02/json-libname-engine-sas/
6. https://support.office.com/en-us/article/Excel-functions-alphabetical-b3944572-255d-4efb-bb96-c6d90033e188
7. https://www.oauth.com/oauth2-servers/access-tokens/self-encoded-access-tokens/
8. https://blogs.sas.com/content/sasdummy/2018/01/16/hide-rest-api-tokens/
9. https://docs.microsoft.com/en-us/graph/api/overview?view=graph-rest-1.0

CONTACT INFORMATION

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

Simon Todd
Phastar
Unit 2, 2a Bollo Lane
Chiswick, London, W4 5LE
Email: simon.todd@phastar.com
Web: www.phastar.com

Brand and product names are trademarks of their respective companies.

APPENDIX 1: WORKBOOK MANIPULATION EXAMPLE CODE

```
* Refresh user access prior to tracker import *;
%refresh_token

* Utility macro to return resource ID values for request URL *;
%sharepoint_locate;

* Temporary request response location *;
filename resp temp;

* HTTP request to set up workbook session *;
proc http
   url="https://graph.microsoft.com/v1.0/groups/&groupid./drive/
        items/&fileid./workbook/CreateSession"
   method="POST"
   oauth_bearer="&access_token"
   out=resp
   headerin="PersistChanges: True";
   ct="application/json";
run;

* JSON libname to access JSON structure response content *;
libname wbook json fileref=resp;

* Assign session ID to macro variable for future workbook requests *;
data _null_;
  set wbook.root;
  call symputx('sessid',strip(id));
run;

* Header for each workbook session is now session ID *;
filename header "%sysfunc(pathname(work))/session.txt";

data _null_;
  file header;
  put "workbook-session-id: &sessid.";
  file print;
run;

* Filename for  *;
filename value "%sysfunc(pathname(work))/value.json";

* Find date and time *;
%let value = %sysfunc(date(),date9.) %sysfunc(time(),tod8.);

* Create JSON file containing the value item only *;
proc json out="%sysfunc(pathname(work))/value.json" pretty;
  write open object;
  write values "values" "&value.";
  write close;
run;
```

```
* Temporary file for http output *;
filename resp TEMP;

* HTTP Patch request to update cell A1 on sheet1 *;
proc http
   url="https://graph.microsoft.com/v1.0/groups/&groupid./drive/items/
       &fileid./workbook/worksheets/sheet1/range(address='A1')"
   method="PATCH"
   oauth_bearer="&access_token"
   in=value
   headerin=header
   ct="application/json;odata.metadata=minimal;odata.streaming=true;
       IEEE754Compatible=false;charset=utf-8"
   out = resp;
run;
```