Effective APIs for SAS Language Applications

Randy Betancourt, Altair

ABSTRACT

Firms typically execute a large number of SAS programs repetitively. Many programs run repeatedly with alterations to produce required output. A common improvement on this process is to pass parameters to validated autocall macros. Still, programs are often hand-edited, tested, then output is copy and pasted to distribute results. In many cases, multiple manual steps are required which are expensive, error-prone and, importantly in a regulated environment, hard to reproduce reliably.

The use of SAS language autocall macro libraries provides a method for effective reuse of SAS program logic. To make best use of available skilled workforce, firms need to fully control and automate this logic as well as expand software tooling and process automation to encompass Python, R, SQL, Perl, Powershell, Linux shell scripts, batch scripts and other programming technologies.

This proposes a fully automated distribution and invocation of SAS language programs along with Python, R, and SQL by enabling any consumer (browser, Excel, middleware applications, web portals, programming environments etc.) to make web-based REST API calls, with API arguments being passed through as macro parameters when executing program logic.

The paper also describes a novel, easy-to-implement architecture and workflow upgrading validated autocall macro libraries and exposing them as API calls passing parameters to auto-generated API endpoints. A related goal is to describe a scalable software architecture based on the OpenAPI Initiative (OAI) and its OpenAPI Specification (OAS) enabling these capabilities.

INTRODUCTION

Application developers will learn how to easily and quickly publish and track API endpoints integrating various programming tools. Application consumers benefit by having a single and consistent methods to acquire output and results. Consistent use of REST APIs enforces rigorous abstraction and artifact publishing and management aids effective, controlled application development.

We describe a workflow enabling SAS language program to be callable as REST APIs:

1. Validate the program is host-independent by executing it on Windows and Linux.
2. Push the program to a source code repository such as Git
3. Publish the program together with any supporting files or data a ssets as an artifact to a Hub. Altair's Hub software provides an enterprise repository for storing and managing analytics assets including SAS language programs; managing metadata; creating workflows (visual and code) enabling programs to be called through REST APIs; load balancing and orchestrating program execution across a grid or cluster of worker nodes; providing authentication services. The Hub software architecture is described in detail below.
4. Create a Deployment. Deployments enable client programs to call SAS language programs (or any other programs deployed in the Hub) though auto-generated URLs with defined, typed parameters (numerics, dates, strings, etc.) required for invocation of the API.
5. Define a pipeline to chain together deployed programs. A pipline uses a flow control graph based on success/failure of pipeline steps, integrating Powershell, Linux Shell scripts, Windows Batch scripts and control nodes Wait, Any, and All.
6. Illustrate browsers, Postman, R scripts, and SAS language PROC HTTP syntax as consumers calling the published REST API endpoint with a URL including parameters to execute the program's logic.

REST APIS

This section provides a brief description on the fundamentals of REST APIs, how URL strings are passed from the consumer (browsers, etc.) and how servers fulfil these requests. Using a REST API layer, SAS language programs are abstracted from the consumer through a “contract”. This allows different consumers to make the same API calls without needing to know the implementation or infrastructure used to fulfill requests, or even that SAS language logic was used. This allows, for example, an R or Python user to consume a SAS language program without any knowledge of SAS language syntax, and vice versa.

Representational State Transfer (REST) is a standardized method using web-based protocols (HTTP or HTTPS) to present remote resources to a local client. A server provisions an ‘endpoint’, a URL that services HTTP requests such as POST, GET, DELETE, UPDATE, etc. to accept client instructions to the server to operate on resources –application logic, data etc. - according to the request.

For example, a SAS language user wanting access to an already-written program may use a %INCLUDE statement with the path to a file containing a SAS language program in order to ruse the included program’s logic. The SAS language user must know details like where to find the program, what macro parameters etc. A common challenge is knowing if that path points to the most up-to-date version of the program. If the desired program is located on a remote file system, then they may need to use FTP or similar to copy locally, or use the SAS language FILENAME FTP access method, requiring the user to have detailed network knowledge and credentials.

Using a SAS language autocall macro library is a step towards abstracting file system and networking details allowing SAS language programmers easily to call program logic without knowing such details. However, SAS language autocall macro libraries provide abstration only to SAS language programmers. In today’s multi-language (polyglot) world that includes R and Python and potentially many more emerging languages such as Julia, LUA and Golang, organizations will benefit from SAS language assets and resources being readily consumable beyond the SAS language user community. By mapping critical SAS language programs to REST APIs, organizations can make SAS language code re-usability for any consumer throughout an organisation.

Later in this paper, we illustrate R programs calling a REST API to execute a SAS language program by importing the R library httr.

HUB SOFTWARE ARCHITECTURE

SLC Hub is the name of Altair’s enterprise anaytics management software. The software delivers a simple, modern, software stack that is easy to deploy, manage and control. The Hub software can be deployed on-premises, in the cloud or in a hybrid environment. The Hub software can run on Linux or Windows and provides a secure scalable platform that offers a range of services to support enterprise analytics processes including user security and management, user and data access provisioning, API definition, publishing and deployment, on-demand API execution, workload pipelines and scheduling, workload balancing for interactive and production workloads across a grid or cluster of worker nodes, auditing and logging.

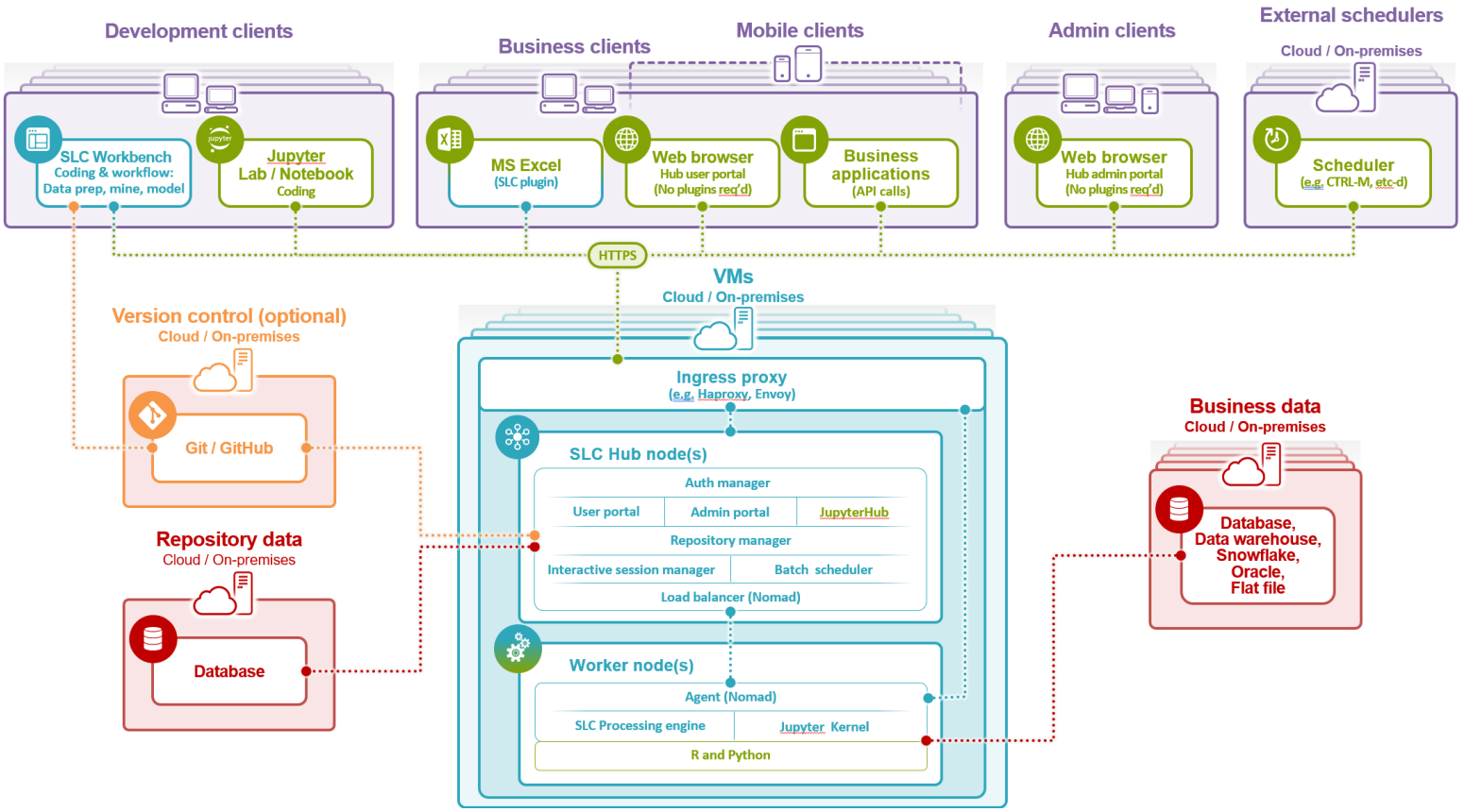

Figure 1. SLC Hub Architecture

The SLC Hub architecture is built to be an elastic and scalable platform. It is composed of a Hub node as the control plane, and a collection of worker or compute nodes managed by the Hub(s) to which workloads are distributed by the Hub node for execution a according to workload profile rules. Results are returned to the Hub and returned to end-users and machine-to-machine clients (browsers, SLC Excel plug-in, business applications making REST API requests).

The software has five major sub components described below.

1. **Eclipsed-based Analytics Workbench** is a rich GUI/IDE used to develop analytics applications both through visual drag-and-drop Workflows and coding. Programmers compose SAS language programs, SQL, Python, and R. Non-programmers use a palette abstracting coding to visually design workflows.

   During development, programs and workflows can be executed interactively both locally and remotely via the Hub. The Workbench has an interactive wizard to wrap programs in paraterised package ready for testing and deployment into a Hub. The parameterized package can be executed and tested locally in the Workbench without need to be published to a Hub. Once a package is ready for publishing it can transmitted to a Hub as a Hub package (described next) using a simple point-and-click wizard in the Workbench.

   A Hub administrator defines profiles that map resources to specific worker nodes. For example, requests for access to invoke a program that reads data from a particular Oracle database may be a profile defined for worker node instances 1 and 2 running Linux. Access to SQL/Server requirements is met by worker node 3 running Windows etc.

2. **Hub Console**. This management portal is accessed by a web-based GUI providing services to manage services such as auth domains, libref definitions, execution profiles (fine-grain control for

multiple SAS language program execution configurations), control of repositories, building workload pipelines (chained programs), job scheduling, REST API deployment and publishing.

All configuration management and change control activities are performed through the Hub Console.

3. **User Portal.** This portal is accessed in a similar way to the Hub Console, but is intended for end users to run deployed programs and access results.
4. **Workers:** Deployed programs are orchestrated for execution by the Hub node onto one of multiple worker nodes using Hashicorp's opensource Nomad workload management infrastructure. Multiple, different teams of users can have workloads specified to run on specific Workers according to security, performance and platform requirement profiles. Workloads can be both production scheduled processes as wells as interactive user processes. Nomad provides many facilities that are common in a Kubernetes containerized environment, but using conventional VMs instead of containers. A Containerised version may be offered in future, but the Nomad/VM version appears to be easier to manage for many organizations at this time.
5. **Opensource components**.
    a. Hashicorp's open-source Vault for back-end encryption for at-rest and in-flight passwords, database credentials, and secrets. This eliminates storing credentials in clear text when accessing resources.
    b. Minio. Worker nodes need a central location to store execution artifacts. For non-AWS deployments Minio provides an S3-like access method for shared resources created among Workers. This obviates deployment overhead for shared filesystems like IBM's SpectrumScale, FSx for Lustre (AWS), FileStore (GCP), or Azure Files.
    c. Nomad for orchestration. Nomad provides a flexible and scalable platform for managing applications across a wide range of infrastructure. Nomad is an open-source cluster manager and scheduler that automates the deployment, scaling, and management of containerized and non-containerized workload.
    d. PostgreSQL is used as a backing store for configuration and related Hub application data. PostgreSQL stores resources used to control the the Hub components. These include configuration files, namespace definitions, TLS certificates and so on.
    e. NGINX is used as a web-server to facilitate communication between the Hub components that are architected as micro services. No user-intervention is needed at install and configuration time. NGINX also serves HTML pages presented by the Hub's portal interface.

API PRODUCTION PROCESS

This section includes a step-by-step description of the workflow process described in the abstract. We start with a simple use case to illustrate an R script consuming a SAS language REST API:

1. Write a SLC program that outputs a range of dates. The start of the date range is a parameter (SAS macro variable &start_date) to be passed as part of the REST API call.
2. Test/validate this program by executing it using SLC Workbench to supply program attributes to determine how output is presented through the API call. For R, we chose to stream JSON output. Alternatively, we could have chosen to output an R DataFrame. Once the program is validated, we create an API package.
3. From SLC Workbench, upload the validated API package to SLC Hub storing it in a Hub-managed repository we named Pharmasug2023.
4. Using the Hub Console, we create a 'deployment package' to publish the program as a REST API.

5. Test the API call using Postman using Basic Auth and call the API with an arbitrary value for the &start_date parameter (SAS macro variable). This test ensure the API call is producing the expected output.
6. Write an R script using the httr library to call the API by passing basic authorization parameters and the value of the parameter representing the start_date. Load the jsonlite library to "flatten" the returned array and create the R DataFrame.

Create the SLC program called generate_dates

```
data dates;

do date = "&start_date" to "31Dec2023"d;
   output;
end;

format date YYMMDDd10.;
run;
```

In this form, the program would fail if executed in a "stand-alone" mode. For the API call we are using &start_date as a parameter whose value is determined at the time the rest API is called. SLC Workbench provisions configuration menus to determine the behavior of this parameter and related parameters to define the overall behavior of the REST API. See Figure 2. Hub Configuration Values for generate_dates.

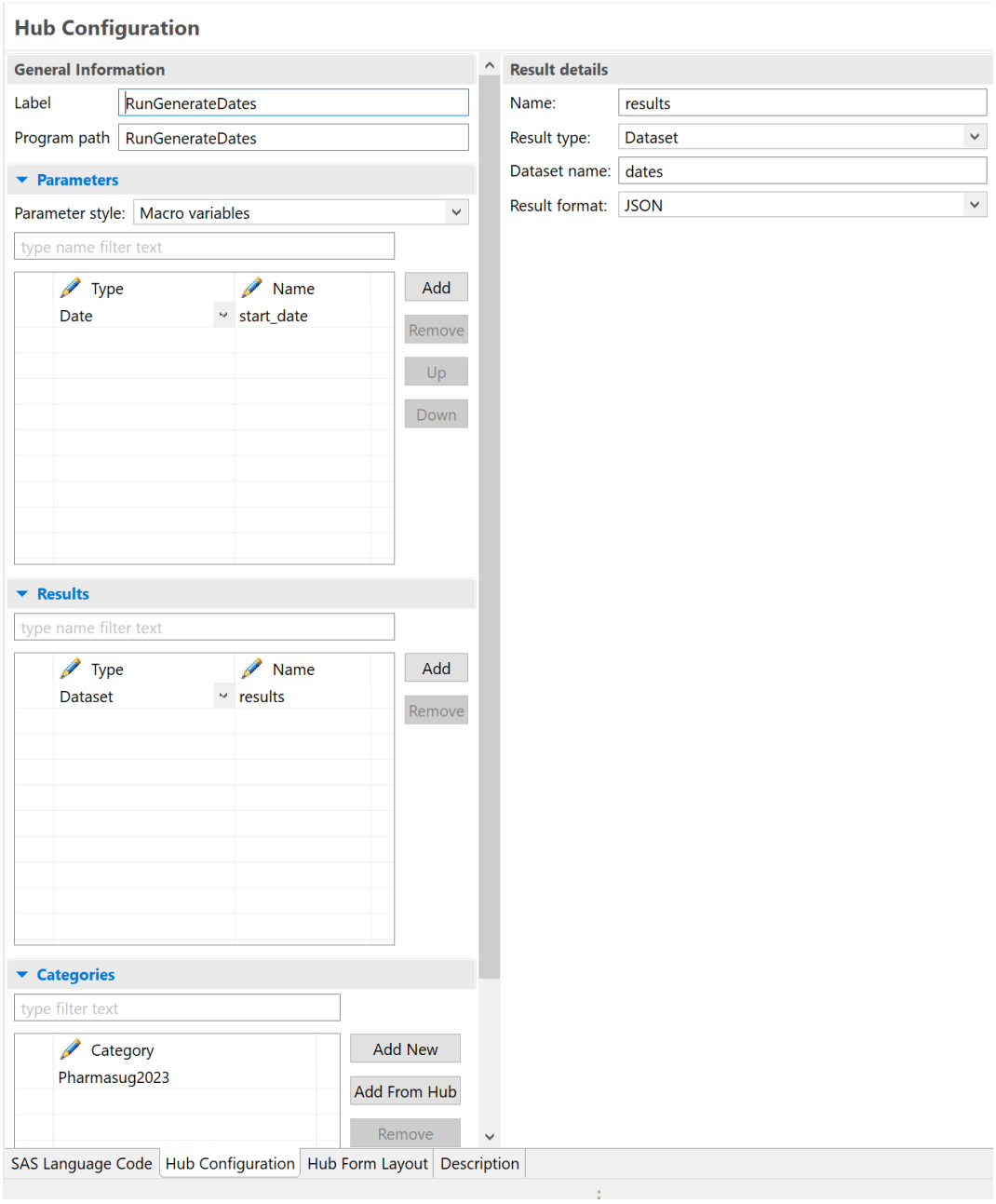

Figure 2. Hub Configuration Values for generate_dates

For Results Detail, we select Dataset for the Result type. For Results format we select JSON. These choices "extend" our original 'generate_dates.sas' program by calling PROC JSON to write output in JSON format.

For Parameter Style, we select "Macro variables". Alternatively, if we had a larger number of parameters to manage as part of the API, we would select "Dataset". For Type we select Date from the following choices:

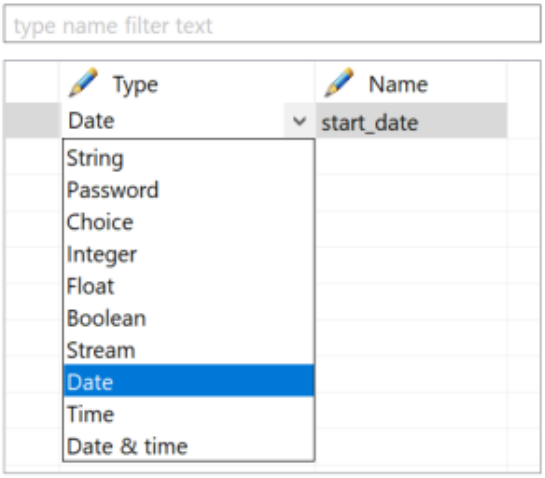

Figure 3. Hub Configuration for Typed-Parameters

For Categories, we chose the name Pharmasug2023. Within the Hub Console, we can search programs and deployment packages using this string as a search value. Once we save the Hub Configuration values, the program is marked as an API package shown in Figure 4. The API package contains the source code for the program as well as the selected Hub parameter values.

TEST AND VALIDATE THE 'GENERATE_DATES' PROGRAM

As we mentioned earlier, our 'generate_dates' program needs the appropriate execution context. Once the Hub parameters in Step 1 are defined, we test our program before uploading the API package to SLC Hub.

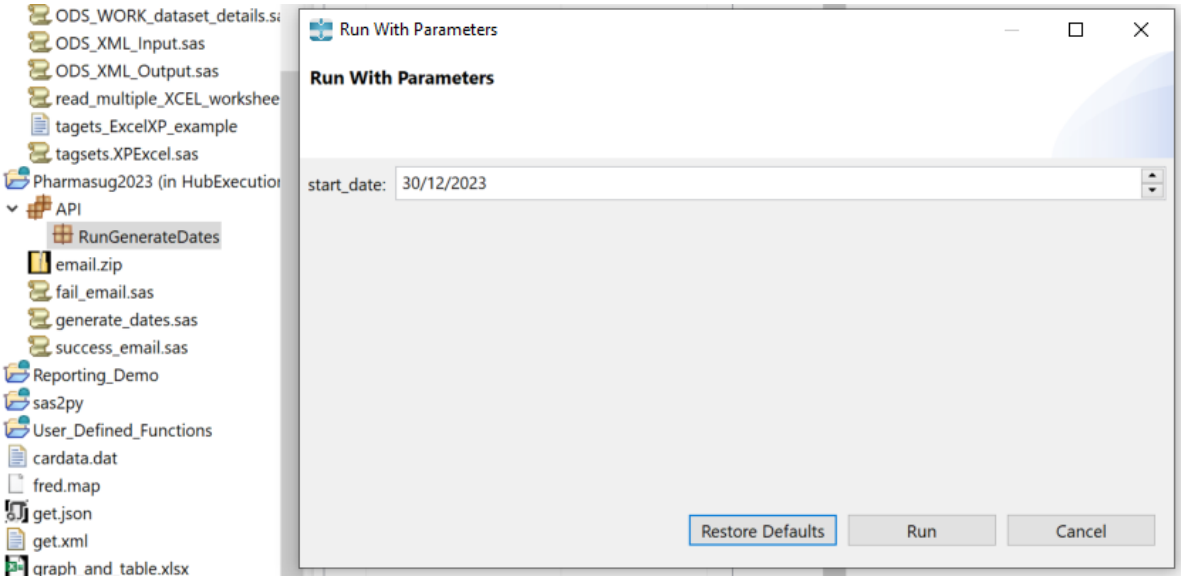

Figure 4. Validating the API Package created in Step 2

UPLOAD VALIDATE API PACKAGE TO SLC HUB

Once the API package is defined and validated it is uploaded to the SLC Hub instance we instantiated in AWS. SLC Hub deployments are cloud-agnostic.

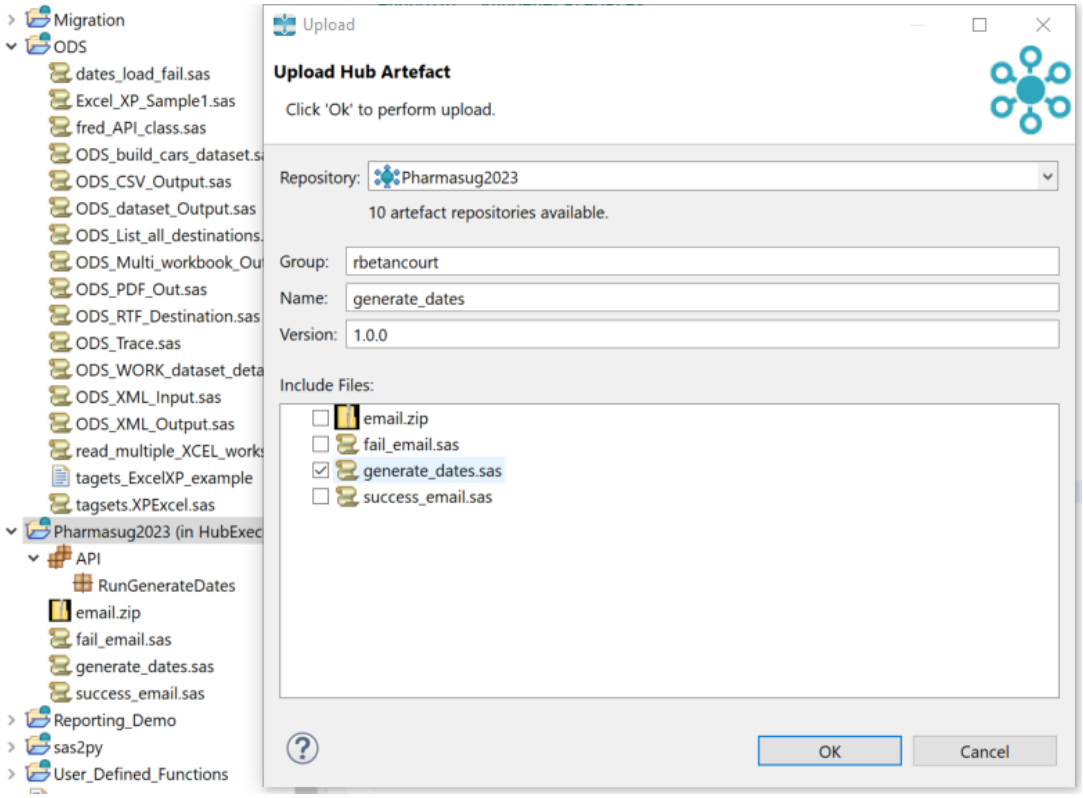

Figure 5. Upload API Package to SLC Hub in Step 3

The target deployment for this validated API package is the Pharmasug2023 repository that is a SLC Hub managed repo we created previously. We also support Git as a repository. After pressing the OK button, the API package is uploaded to SLC Hub as an artifact.

CREATE A 'DEPLOYMENT PACKAGE' TO PUBLISH THE PROGRAM AS A REST API

With the API package uploaded to SLC Hub, we use the Hub Console to create a Hub Deployment package which allows the program to be called through a REST API. The SLC Hub portal is designed to bring governance and control to all SLC assets.

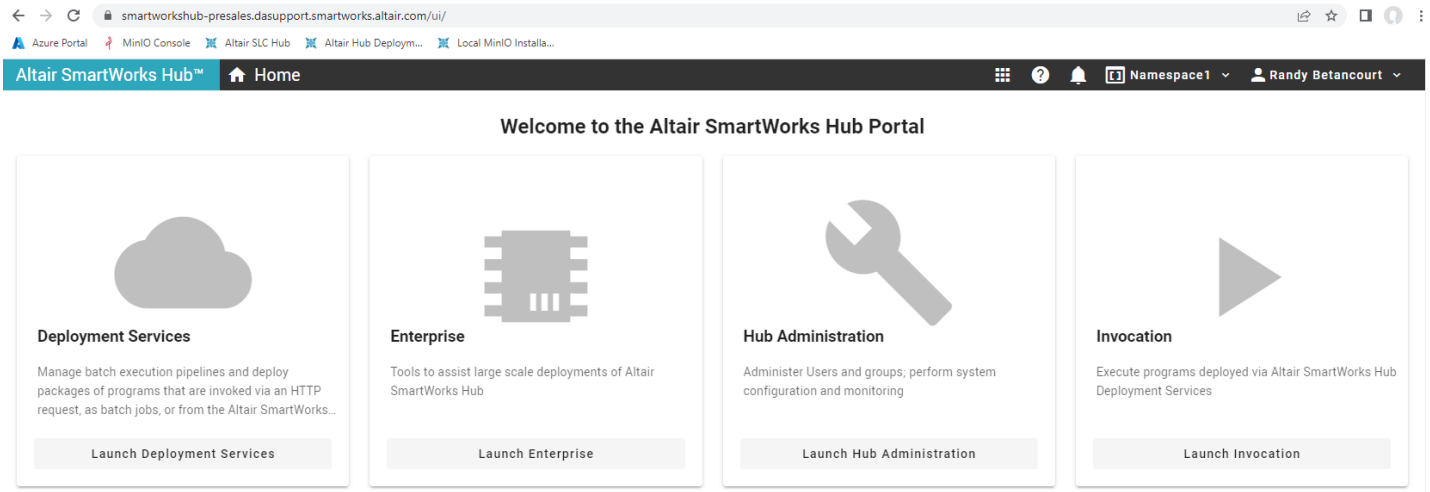

Figure 6. Main SLC Console Page

In this step, we concentrate on the Deployment Services. We confirm the artifact "generate_dates" is available in the Hub environment.

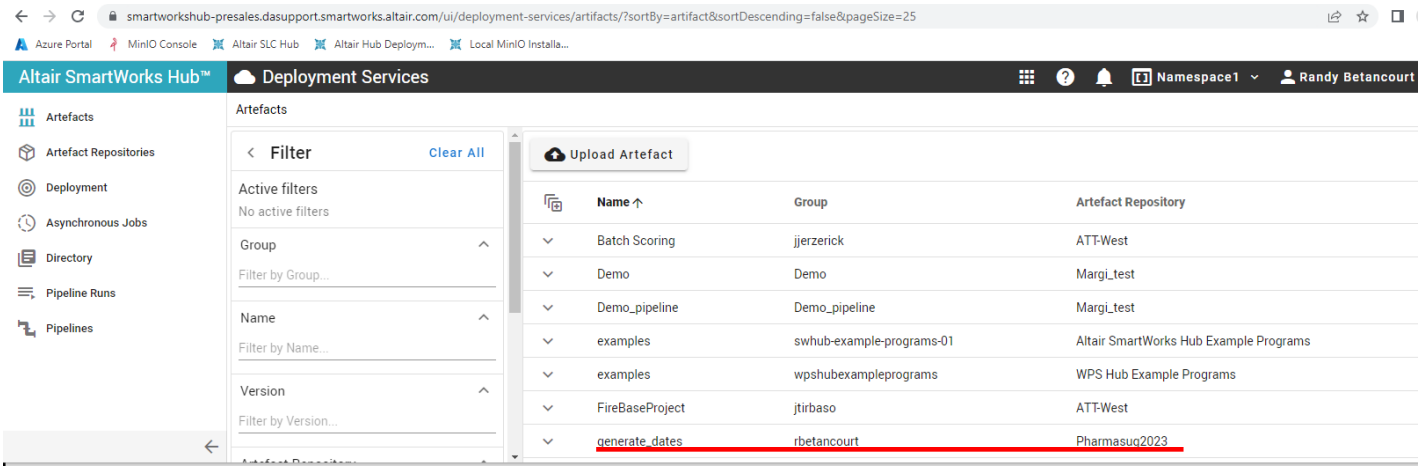

Figure 7. Artifact "generate_dates"

Next we create a Deployment package. Select the Deployment type, in our case a SLC Hub managed repository and the Execution profile. See Figure 8.

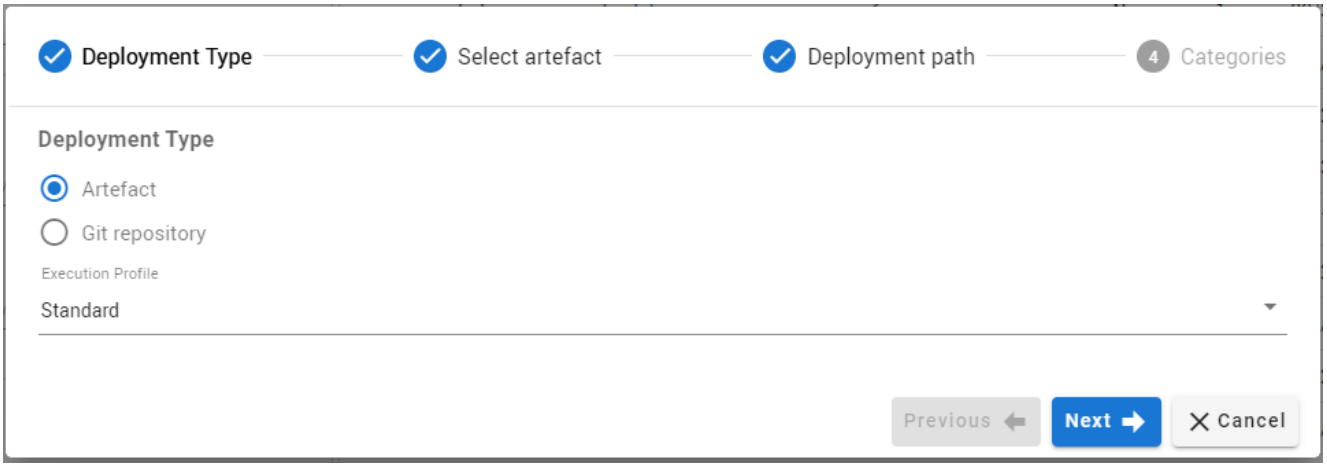

Figure 8. Select Deployment type and Execution Profile

A Hub administrator defines Execution profiles which map resources to specific worker nodes. For example, requests for access to read data from Oracle is a configuration defined for worker node instances 1 and 2 running Linux. Access to SQL/Server requirements are met by worker node 3 running Windows, and access to DB2 are met by worker node 4.

Here we select the Standard Execution profile. The repo where "generate_dates" defined in Step 1, and the Version number. See Figure 9.

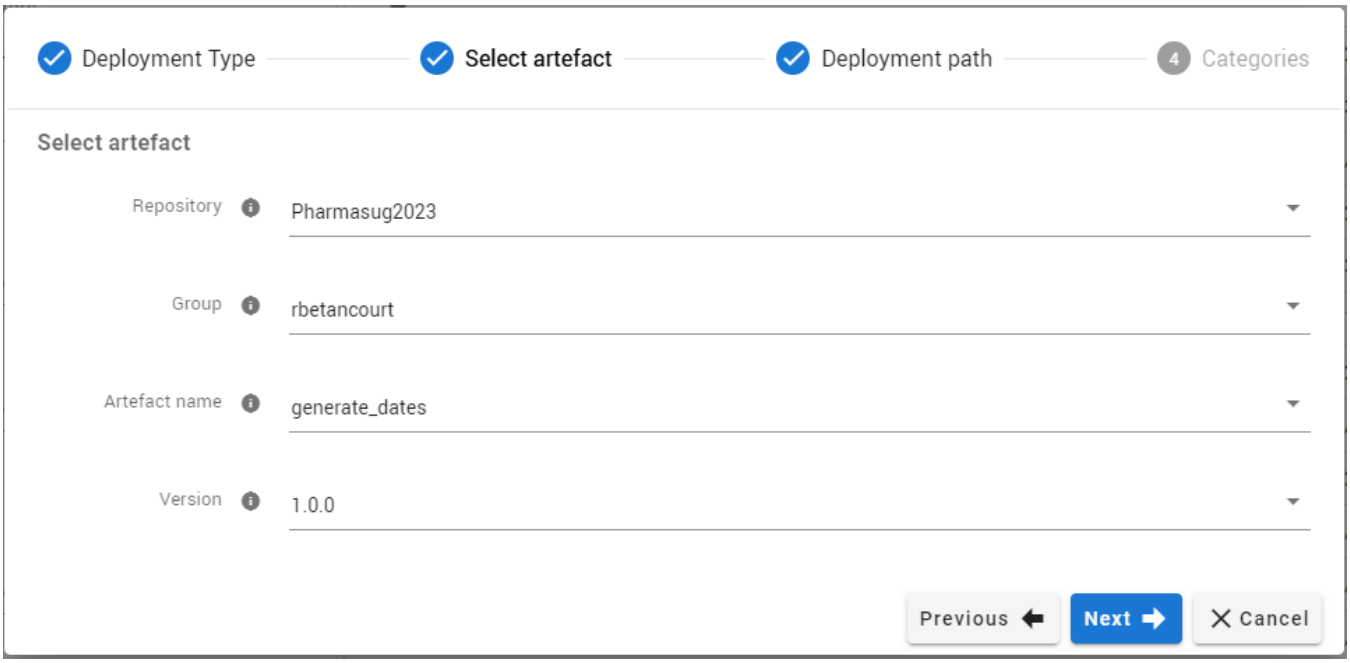

Figure 9. Select the "generate_dates" artifact

Next we select a unique deployment path (a portion of the URL string) clients use to call the API. We select the Create API endpoint check-box. See Figure 10.

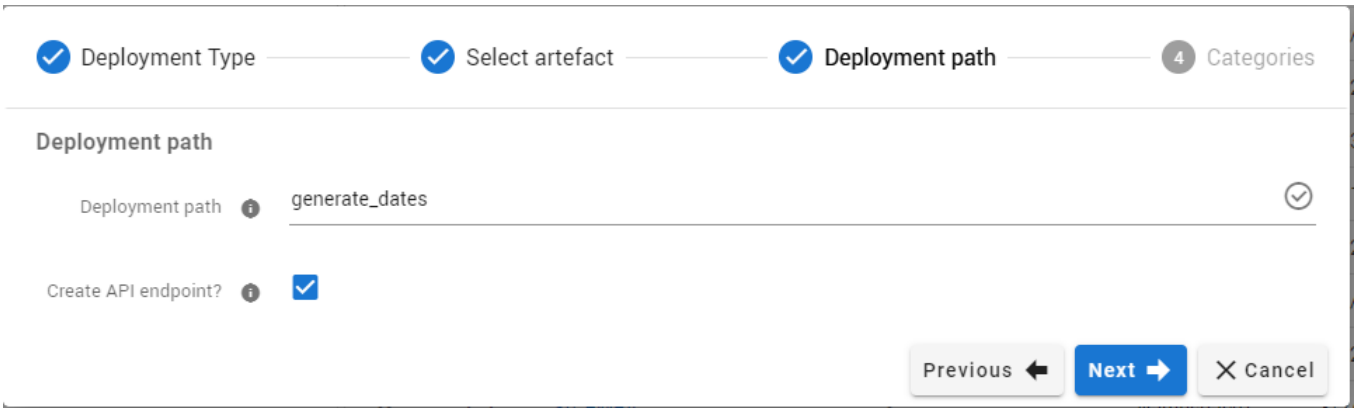

Figure 10. Select the Deployment path and enable creation of an API endpoint

The final action for this step is to select the category name. The category name was defined in Step 1, when defining the Hub attributes for the API package in SLC Workbench. This action provides a convenient filtering mechanism to easily locate artifacts beloging together.

An artifact can be a member of multiple, different categories. See Figure 11.

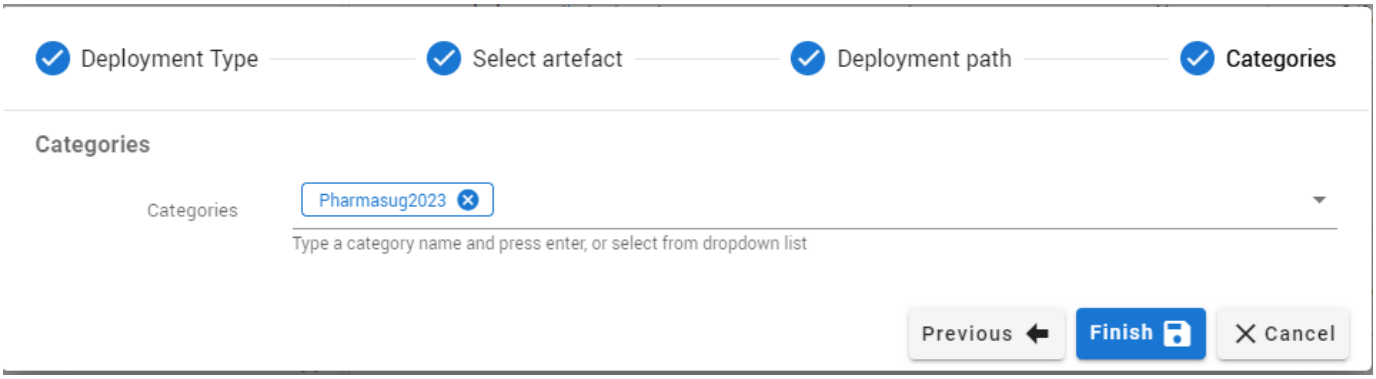

Figure 11. Select appropriate categories

Defining the “generate_dates” Deployment package is complete and available for any client to consume. The Hub Directory services returns the API end-point. See Figure 12.

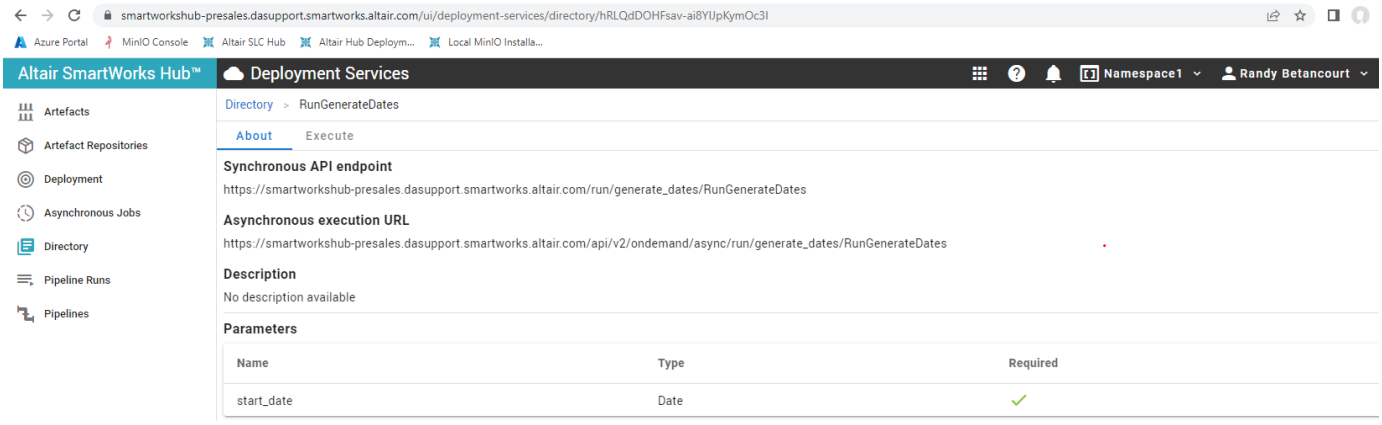

Figure 12. Deployment Service REST API Endpoint

In our case, the generated API endpoint is:

https://smartworkshub-presales.dasupport.smartworks.altair.com/run/generate_dates/RunGenerateDates

We also see “start_date” is a required parameter. Selecting the Execute tab on this page permits us to call the defined REST API. See Figure 13.

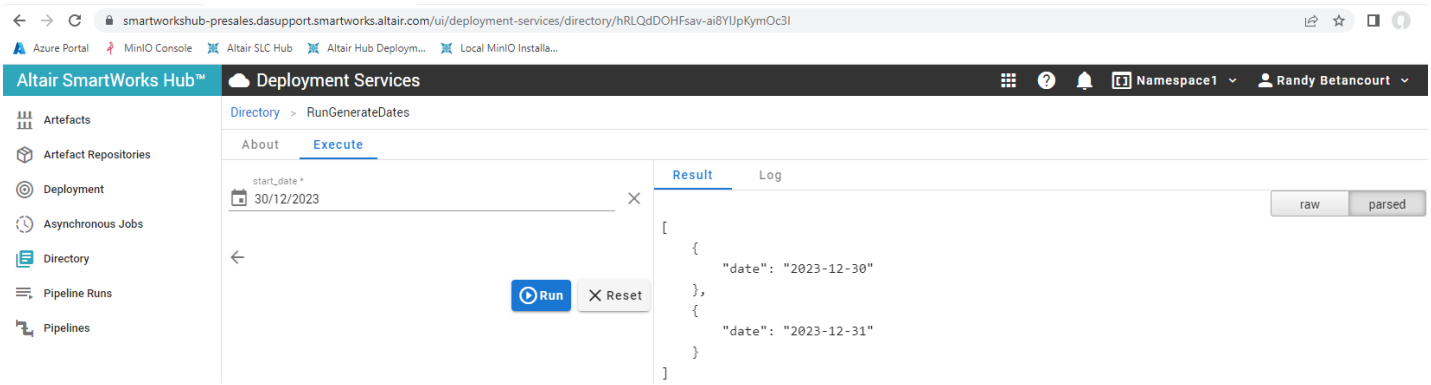

Figure 13. Calling the REST API Endpoint

In the Results pane we see the JSON array returned when the start_date parameter value is set to 30/12/2023. Recall the date values are formatted in our initial program with the SLC-supplied YYMMDDd10. date format.

TEST THE REST API CALL USING POSTMAN USING BASIC AUTH

Once an API endpoint is defined, we describe a validation process to certify calls into the API are working and produce intended results. This is accomplished by using different clients; Microsoft Excel, a browser, and a language of SAS program calling the API endpoint and so on.

An indispensable tool for testing REST API endpoint is Postman. Postman is an API platform for building and using APIs. Postman simplifies each step of the API lifecycle and streamlines collaboration.

Access to Hub's APIS require client-side authentication. SLC Hub API endpoints support Bearer Tokens as illustrated in the Postman screen-shots. See Figure 14.

The API endpoint passed is:

https://smartworkshub-presales.dasupport.smartworks.altair.com/run/generate_dates/RunGenerateDates?start_date=2023-12-28

Notice how the start_date parameter value is 2023-12-28 returning 4 dates as a JSON array in the response body. In addition, we confirm with Postman that the SLC Hub endpoint will accept Basic Auth which is simpler to implement in subsequent endpoint API calls than using Bearer tokens.

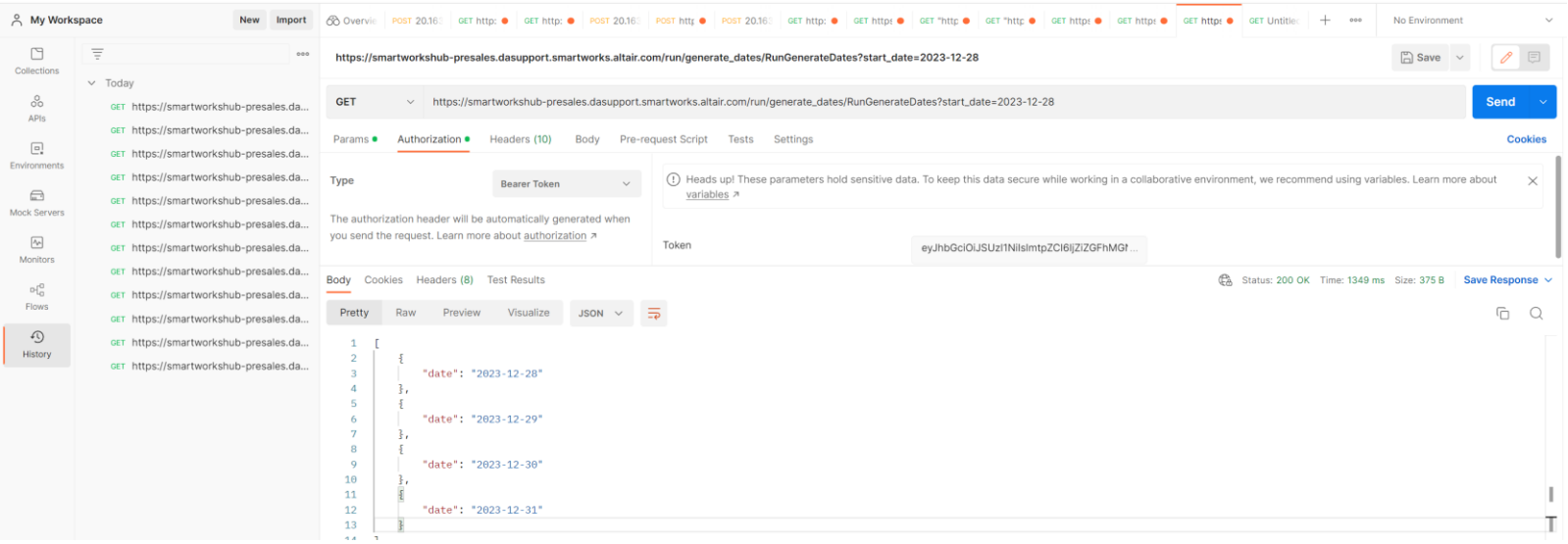

Figure 14. Postman calling SLC Hub REST API endpoint using Bearer Auth token

As an example of client-side Basic Auth for authentication to access the API endpoint, consider the use of a browser such as Chrome.

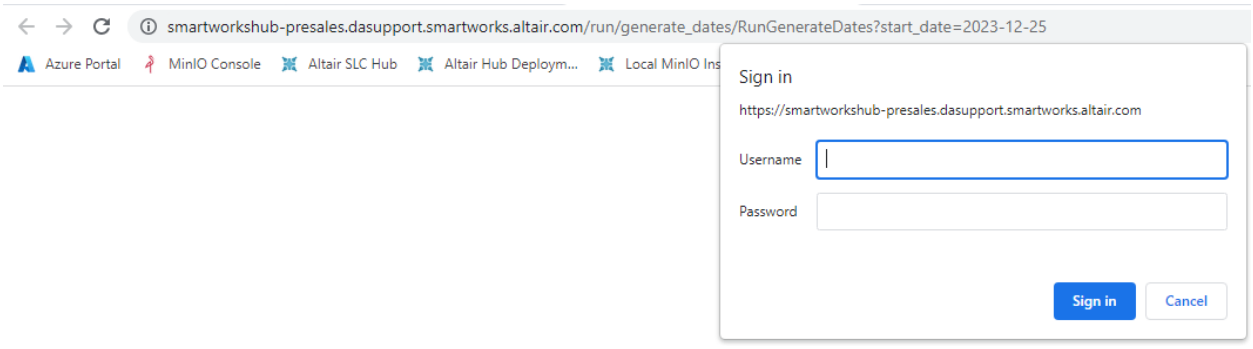

Figure 15. Chrome browser authentication to access SLC REST API endpoint

The user must authenticate with their Hub credentials which can be managed internally by SLC Hub, or though Vault, or by establishing an auth domain to a credentials management services like Active

Directory. Creating an auth domain within SLC Hub offers the ability for organizations to support Single Sign-On (SSO).

Once user credentials are validated, then the request is completed. In our case the five date are returned as a JSON array.

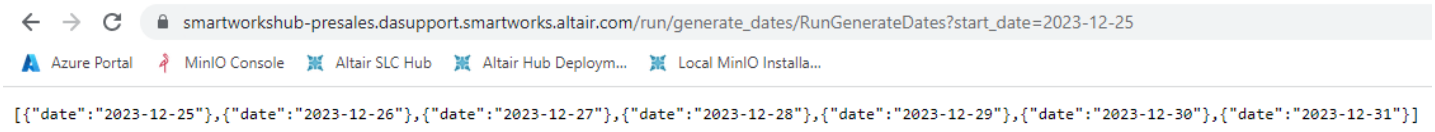

Figure 16. Chrome browser calling REST API after authentication to SLC Hub

WRITE AN R SCRIPT USING THE HTTR LIBRARY TO CALL THE REST API

By understanding how authentication is handled by SLC Hub, we can complete our use-case. Write an R script that calls the defined API.

```
# ChatGPT3 Prompt:
# write an R program to illustrate basic auth for httr library

# Set up the authentication details
auth_user <- "rbetancourt"
auth_pass <- "<HUB Password String>"
auth <- paste(auth_user, auth_pass, sep = ":")

# Encode the authentication details in base64 format
auth_enc <- enc2utf8(base64_enc(auth))

# Create a request with the authentication header
req <- GET("https://smartworkshub-
presales.dasupport.smartworks.altair.com/run/generate_dates/RunGenerateDates?
start_date=2023-12-01",
  add_headers("Authorization" = paste("Basic", auth_enc))
)

# Extract the JSON content from the response
json_content <- content(req, "text")

# Convert the JSON content to a DataFrame
df <- fromJSON(json_content)
```

```
# Print the DataFrame
print(df)
```

The output from the R script executed in VS Code Studio is shown in Figure 17.

Figure 17. R DataFrame from calling SLC Hub's REST API Endpoint

OPEN API STANDARD

In designing SLC Hub's ability to publish API endpoints, Altair engineers relied on the Open API Initiative. The OpenAPI Initiative (OAI) is a collaborative open-source project that aims to create, maintain and evolve a standard specification for building and documenting RESTful APIs. The initiative was created in 2015 by a group of industry experts and companies including Google, IBM, Microsoft, and Apigee, and is now governed by the Linux Foundation.

The OpenAPI Specification (OAS), formerly known as Swagger Specification, is the foundation of the OAI. It is a machine-readable format for describing APIs that can be used to generate documentation, code, and other resources automatically. The OAS provides a common language for describing API operations, parameters, responses, authentication mechanisms, and other aspects of a RESTful API.

The OAI aims to promote interoperability and standardization among API providers and consumers. By using a common specification, developers can create and consume APIs in a more consistent, efficient, and scalable way, regardless of the programming language, framework, or platform they are using. The OAI also provides tools, resources, and best practices to help developers design, test, and deploy APIs that conform to the OAS.

To illustrate the standardization offered by OAS, we wrote a SAS language program using PROC HTTP as the method for returning metadata about the SLC Hub example programs deployed at Hub configuration time. This program illustrates the handling of Bearer Tokens as the authentication method.

This program calls the auth API endpoint using PROC HTTP by passing a request body containing username and password key/value pairs. The returned auth string is included as part of the response header in a subsequent API call (all calls for resources must be authenticated) to return metadata about the composition of the Hub's repositories. See Appendix A: PROC HTTP Calls to Hub's Open APIs.

CONCLUSION

With the large scale use of the SAS language in the pharma and life sciences industry, program reusability is essential. The utilization of SAS Autocall macros as a common method for code re-usability

among SAS users and programmers has not changed over the past 30 years. Further, SAS Autocall libraries are not accessible to non-SAS language applications.

In today's world of multiple languages for reporting and analysis, REST APIs for critical SAS language macros and code templates extend code reusability beyond SAS programmers to include R, Python, Lua and other programming languages. The innovation behind exposing SAS language programs as REST APIs translates directly to increased staff productivity, less redundancy, and less re-work.

REFERENCES

1. Marshall, James (1997). "HTTP Made Really Simple"
2. Introduction to SLC Hub Administration: https://hubdoc.worldprogramming.com/5EA-2.2.0.108/use/introduction/#hub-administration
3. Introduction to SLC Deployment Services. https://hubdoc.worldprogramming.com/5EA-2.2.0.108/use/introduction/#deployment-services
4. Learning Postman. https://learning.postman.com/docs/introduction/overview/
5. R-project.org. "Getting Started with httr", httr QuickStart Guide – CRAN
6. HTTP Authentication. https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication
7. An Introduction to Accessing RESTFUL APIs Using R. https://rpubs.com/plantagenet/481658
8. Introducing ChatGPT. https://openai.com/blog/chatgpt
9. Open API Initiative, OAS. https://www.openapis.org/

CONTACT INFORMATION

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

Randy Betancourt
Altair
rbetancourt@altair.com
www.pythonforsasusers.com

Oliver Robinson
Altair
orobinson@altair.com

APPENDIX A: PROC HTTP CALLS TO HUB'S OPEN APIS

```
4          /* Program variables */
5          %let url = http://20.163.76.XXX:9090/api/v2/auth/login;
6          /* https://hubdoc.worldprogramming.com/5EA-
2.2.0.108/api/auth/restapi.html#operation/passwordLogin */
7
8          filename in TEMP;
9          filename out TEMP lrecl=32767 recfm=v;
10         filename headerin TEMP lrecl=32767 recfm=v;;
11
12         /* Macro to print FILE output to log */
13         %macro prntfile(file);
14         option nonotes;
15         data _null_;
16            infile &file end=eof;
17            input;
18            put _infile_;
19            if eof then put /;
20         run;
21         option notes;
22         %mend;
23
24         /* Create parameters to form Request Body in JSON format */
25         data _null_;
26         file in;
27         put  '{' /
28              '"username": "rbetancourt",' /
29              '"password": "XXXXXXXXXXXX"' /
30              '}';
31         run;

NOTE: The file in is:
      Filename='C:\Users\RBETAN~1\AppData\Local\Temp\WPS Temporary
Data\_TD19812\#LN00002',
      Owner Name=PROG\rbetancourt,
      File size (bytes)=0,
      Create Time=13:27:19 Jan 02 2023,
      Last Accessed=13:27:19 Jan 02 2023,
      Last Modified=13:27:19 Jan 02 2023,
      Lrecl=256, Recfm=V

NOTE: 4 records were written to file in
      The minimum record length was 1
      The maximum record length was 28

32
33         /* Create Request Header */
34         data _null_;
35         file headerin;
36
37         put 'Content-Type: application/json, text/plain' /
38             'Accept: application/json, text/plain' /
39             'Accept-Language: en-US';
40
41         run;
```

```
NOTE: The file headerin is:
      Filename='C:\Users\RBETAN~1\AppData\Local\Temp\WPS Temporary
Data\_TD19812\#LN00004',
      Owner Name=PROG\rbetancourt,
      File size (bytes)=0,
      Create Time=13:27:19 Jan 02 2023,
      Last Accessed=13:27:19 Jan 02 2023,
      Last Modified=13:27:19 Jan 02 2023,
      Lrecl=32767, Recfm=V

NOTE: 3 records were written to file headerin
      The minimum record length was 22
      The maximum record length was 42

42
43        /* Request (POST) for Auth Bearer Token string */
44        proc http
45           url = "&url"
46           method = "post"
47           ct="application/json, text/plain"
48           in = in
49           out = out
50           headerin = headerin;
51        run;
NOTE: Call to [http://20.163.76.XXX:9090/api/v2/auth/login] returned [200:OK]

52
53        /* Reformat Auth Bearer Token for passing to next HTTP request */
54        data _null_;
55           infile out;
56           length line $ 1024;
57           input line :$1024.;
58
59              token_x = substr(line,11);
60              token = substr(token_x,1,index(token_x,'"')-1);
61
62           call symput('bearer_token', token);
63        run;

NOTE: The infile out is:
      Filename='C:\Users\RBETAN~1\AppData\Local\Temp\WPS Temporary
Data\_TD19812\#LN00003',
      Owner Name=PROG\rbetancourt,
      File size (bytes)=995,
      Create Time=13:27:20 Jan 02 2023,
      Last Accessed=13:27:20 Jan 02 2023,
      Last Modified=13:27:20 Jan 02 2023,
      Lrecl=32767, Recfm=V

NOTE: 2 records were read from file out
      The minimum record length was 0
      The maximum record length was 992

64
65        /* Update Request Header file to include properly formatted Auth
Bearer Token */
66        data _null_;
```

```
67        file headerin mod;
68        length hdr_rqst $ 1024;
69
70        %let prefix = %nrbquote(Authorization: Bearer);
71        %let auth_str = &prefix %trim(&bearer_token);
72
73        hdr_rqst = symget('auth_str');
74
75        put hdr_rqst;
76        run;

NOTE: The file headerin is:
      Filename='C:\Users\RBETAN~1\AppData\Local\Temp\WPS Temporary
Data\_TD19812\#LN00004',
      Owner Name=PROG\rbetancourt,
      File size (bytes)=106,
      Create Time=13:27:19 Jan 02 2023,
      Last Accessed=13:27:20 Jan 02 2023,
      Last Modified=13:27:19 Jan 02 2023,
      Lrecl=32767, Recfm=V

NOTE: 1 record was written to file headerin
      The minimum record length was 985
      The maximum record length was 985

77
78         %let url =
http://20.163.76.XXX:9090/api/v2/ondemand/artifactrepos;

79         /* https://hubdoc.worldprogramming.com/5EA-
2.2.0.108/api/ondemand/restapi.html#operation/queryArtifactRepos */
80
81         proc http
82           url = "&url"
83           method = "get"
84           ct="application/json, text/plain"
85           out = out
86           headerin = headerin;
87        run;
NOTE: Call to [http://20.163.76.XXX:9090/api/v2/ondemand/artifactrepos]
returned [200:OK]

88
89
90
91        libname get_repo json fileref=out;
NOTE: Library get_repo assigned as follows:
      Engine:        JSON
      Physical Name: C:\Users\RBETAN~1\AppData\Local\Temp\WPS Temporary
Data\_TD19812\#LN00003

92        data _null_;
93        file log;
94        set get_repo.alldata(where=(P2 ne ""));
95
96        put P2 " = " @32 Value;
97        run;
```

```
_created  =                    2022-12-14T17:58:33.086846Z
_id  =                         c0c57ded-a17b-4a19-80e0-6059a8720e64
_modified  =                   2022-12-28T21:43:15.472878Z
allowPrereleaseVersions  =     true
allowReleaseVersions  =        true
description  =                 Altair SmartWorks Hub Example Programs
name  =                        Altair SmartWorks Hub Example Programs
_created  =                    2022-12-29T21:31:15.108314Z
_id  =                         ebc9bce6-5ad1-4e92-a583-9599a62c53bb
_modified  =                   2022-12-29T21:35:13.085284Z
allowPrereleaseVersions  =     true
allowReleaseVersions  =        true
description  =                 Added on 30Dec2022
name  =                        SLCRepo

NOTE: 14 observations were read from "GET_REPO.alldata"
```