PharmaSUG 2020 - Paper AD-341

Opening Doors for Automation with Python and REST: A SharePoint Example

Mike Stackhouse, Atorus Research

ABSTRACT

A large part of increasing the efficiency of a process is finding new ways to automate. If manual components of a process can be replaced with automation, those components can then move to the background where they can be validated and trusted to be reliable. Large swaths of data sit within company intranets or other services that may seem 'disconnected' – but they may be more connected than one might think. This paper will explore extending automation capabilities with Python and REST APIs, using SharePoint as a primary example. Some topics covered will include using Python to make web requests, the basics of authentication, and how to interact with REST APIs. The paper will demonstrate how data repositories that may seem disconnected can be integrated into automated processes, opening doors for new data pipelines and data sources to pave the way for process improvements, efficiency increases, and better information. This paper additionally demonstrates my open source Python module for basic interaction with SharePoint's REST API, called spLite.

INTRODUCTION

More and more we in clinical research are being asked to automate. The demands of clinical trials have pushed us to find ways to accelerate and trim process fat as best we can, allowing us to focus on the nuance of a trial rather than repetition. As programmers, this is in our nature. We often drive to reuse code as much as possible, but sometimes we hit roadblocks that are difficult to overcome.

A common roadblock is connecting systems. At times, it can prove difficult to get different systems to talk to each other. Files are stored in many different places, where stakeholders are required to have varying levels of access. For example, consider the following scenario: aggregating information across studies on a SAS server is likely a simple process. But if project management does not have access to that server, making this information available to them may become a manual process – thus creating a roadblock in the chain of automation.

Another issue that hinders automation is when we step out of our standard programming environments. In a SAS driven world, it is simple for us to create scheduled scripts and run standard jobs on our servers, but there is a great deal of room to create efficiencies for those outside the programming group as well. This is one area in which utilizing languages like Python can prove quite useful.

This paper seeks to demonstrate how REST APIs and Python can prove to be practical solutions to the issues outlined above. SharePoint will be used as the primary example to demonstrate how systems thought to be disconnected may talk to each other after all. Python examples will be demonstrated using the open source module spLite, developed by myself and available for free on GitHub here[1].

WHAT IS A REST API

An API is an application programming interface. APIs allow programmers to abstract away complicated parts of a process so their program can 'talk', or interface, with other systems or languages. REST, or Representational State Transfer", is a set of rules that developers follow when creating an API – so REST governs how the API is created. For a deeper dive into REST APIs than what follows, reference this[2] article.

REST APIs work through URLs. When you link to the URL, a piece of data is returned (called a resource). Each URL is called a request, and the data sent back to you is called a response. Requests are made up of four different parts:

1. The endpoint
2. The method
3. The headers
4. The data

THE ENDPOINT

The endpoint itself is the URL that you request data from. This can be further broken down into parts. The root-endpoint is the starting point of the API from which you are requesting data. For example, the root-endpoint of the ClinicalTrials.gov API is simply https://www.clinicaltrials.gov/ct2/. For Reddit, the API root-endpoint is http://www.reddit.com/api/.

After the root-endpoint comes the path. The path is what determines the resource for which you are requesting. You can access these paths just like linking to different parts of a website. To understand how these paths work for an API, you need to consult the documentation. For example, the API documentation for ClinicalTrials.gov is here[3], or the Reddit API documentation here[4].

After the path comes the query parameters. Query parameters are not technically part of the REST architecture, but APIs commonly use them. The query parameters often take on the format:

```
www.somesite.com/api/path/?param1=value1&param2=value2
```

Where the parameter section begins with a question mark (?) and each parameter is separated with an ampersand (&). These parameters allow you to filter, restrict, or process the data being returned to you in whatever ways the API is designed to allow. Consider the ClinicalTrials.gov API. Following the documentation, consider the following query:

```
https://clinicaltrials.gov/ct2/results/download_fields?cond=cancer&down_count=10&down_fmt=csv
```

In this query, we are:

- Requesting clinical trial results from the API
- Downloading the data automatically from the request
- Filtering for only trials where the condition is ‘cancer’
- Downloading only 10 records
- Downloading in CSV format

Clicking the URL will automatically begin a download in your browser.

THE METHOD

The next part of the request is what the request is trying to do. This is called the Method. There are five different methods of requests you can make: GET, POST, PUT, PATCH, and DELETE. For the purpose of this paper, we will only focus on two – GET and POST.

GET	Get a resource from a server. The server looks for the data you requested and sends it back to you. The server performs a READ action.
POST	Create a new resource on a server. The server creates a new entry in the database and tells you whether the operation was successful. The server performs a WRITE action.

THE HEADERS

Headers provide information about the client and the server, but they are NOT in the URL itself, but rather header information sent along with the request. The HTTP headers are important for several reasons, but two pertinent points are telling the server how you want to receive information and authentication (which will be touched on later).

HTTP headers are made up of property-value pairs. For example, to tell the server that you accept data in JSON format, you would use the following:

```
Accept: application/json
```

By telling this to the server in the header, in some cases this may allow the server to return data to you in JSON format, rather than something like XML. An extensive list of valid headers can be found here[5].

THE DATA

The data, or the body, of the request contains the information you want to be sent to the server. This would not be necessary for a GET request, but for a POST request where you need to put a new entry in to the database. In that request, you need to specify which data will be given to the server. This is essentially like attaching a file to the request itself.

WEB REQUEUSTS WITH PYTHON

Any programming language that can submit web requests is capable of interfacing with a REST API, as you are simply consuming a URL following HTTP. Many languages support this. Python makes web requests quite easy by abstracting away some of the more complicated concepts. Let's consider the following use case: You would like to query the ClinicalTrials.gov API and look at the distribution of clinical trial statuses within their database. To simplify, we will only pull 1000 records.

The first thing we need to do is make the request.

```
>>> import requests
>>> r =  requests.get('https://clinicaltrials.gov/ct2/results/download_
    fields?cond=cancer&down_count=1000&down_fmt=csv')
```

Here we have submitted a GET request to the ClinicalTrials.gov API. We requested a download of 1000 records for trials with the condition of cancer in CSV format. The resulting request object is named r. The first thing of interest in this object is whether the request was successful. This can be found by checking the attribute `status_code`.

```
>>> print(r.status_code)
200
```

A successful request has a status code of 200. Successful responses in general are between 200-299, client errors (meaning your mistake) are between 400-499, and server errors (meaning the servers fault) are between 500-599. Common error codes you may see are as follows (see a full breakdown here[6]):

400 Bad Request	Something about the syntax of your request was incorrect – you likely have a mistake in your URL
401 Unauthorized	The server does not know who you are, but needs to for you to receive your desired request
403 Forbidden	The server knows who you are, but you do not have the proper permissions to receive the request

Moving on, the next thing you want from your request is the data you requested. The request object has two attributes for this: `text` and `content`. `content` is the binary form of the data you requested, while `text` is simply the same data provided as a string. To streamline, let's use the data requested to directly convert the request into a Pandas DataFrame and plot our result:

```
>>> import io
>>> import pandas as pd

>>> df = pd.read_csv(io.BytesIO(r.content))
>>> df['Status'].value_counts().plot(kind='bar')
```

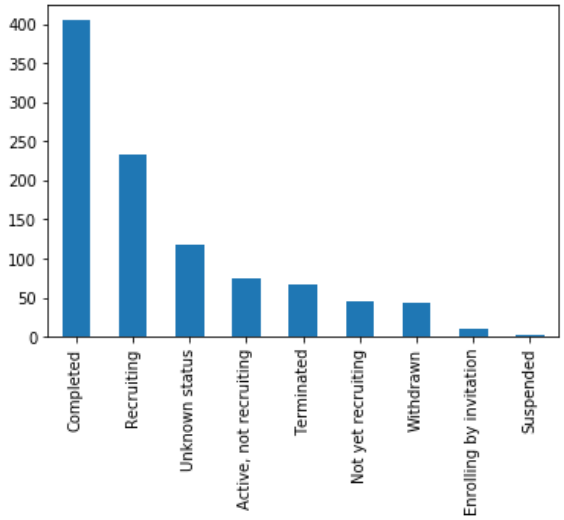

Figure 1. Plot of REST call results

Here we took the binary content of the request, use `io.BytesIO` to make it a file-like object, and then read that into Pandas like we would a CSV file to create a DataFrame. Once in the data frame, we could modify the data as we would any other dataset, so we took the Status column, got the frequency counts of each unique status, and plotted the result as a bar chart. All in less than ten lines of code.

While this is a very basic example, hopefully this begins to demonstrate how simple it can be to start getting systems to talk to each other. To extend the use case just presented, you could set up an automated process to monitor a customized query that runs every week or month and creates a report, instead of having someone go to the ClinicalTrials.gov site and build their query manually whenever you would like to view the results. This concept can extend to many other websites as well – and it is well worth the search to see if a site that you are interested has API documentation available.

AUTHENTICATION

Before we get into SharePoint, we need to first briefly discuss authentication. There are many different types of authentication available on the web, so an in-depth discussion of authentication is well out of scope of this paper, but it is worthwhile to cover a few points.

Authentication provides a barrier between your request and the server, ensuring that only those who have permission to get the requested resource can receive it. The server needs to have some means of understanding who you are, and some verification that the requestor is actually you. A very simple case is using HTTP Basic Authentication. HTTP Basic Authentication simply providers user credentials in the header of the request being sent to the server, encoded in base64. For example, if my user ID was 'USER' and my password was 'PASSWORD', the HTTP header of the request would be updated as:

```
Authorization: Basic VVNFUjpQQVNTV09SRA==
```

Though submitting a request with HTTP Basic Authentication does not require you to update the header yourself. Instead, you write the request call as:

```
>>> import requests
>>> from requests.auth import HTTPBasicAuth
>>> requests.get('www.someurl.com', auth=HTTPBasicAuth('USER','PASSWORD'))
```

The problem here is that decrypting the user credentials is extremely simple, as base64 is a very standard encoding:

```
>>> import base64
>>> base64.b64decode('VVNFUjpQQVNTV09SRA==')
b'USER:PASSWORD'
```

A more secure method, and one that may be used with your SharePoint installation, is NTLM authentication. The logistics of how it works are out of scope of this paper, but using this authentication in a Python request itself is no more difficult:

```
>>> import requests
>>> from requests_ntlm import HttpNtlmAuth # Not in base library
>>> requests.get('www.someurl.com', auth=HttpNtlmAuth('USER','PASSWORD'))
```

Another method of authentication necessary for interaction with the SharePoint REST API may be SAML Security Token Service for Office 365. Again, the details of this authentication method are out of scope of this paper, but a difference is that instead of re-authenticating every request, the header of your request is updated with cookies that contain authorization tokens that last for the life of their session. Once the session completes, the cookies are no longer valid, and the user would have to be re-authenticated.

SHAREPOINT AND REST

SharePoint's REST API has a great deal of capability, allowing you to programmatically interface with most of the application – but most of this goes out of scope of what we are looking to accomplish in this paper. When it comes to automation, we could greatly benefit from just 3 functions:

- Listing files in a SharePoint folder
- Downloading files
- Uploading files

These are the only functions that will be covered in this paper. Full documentation of the SharePoint REST API is available here[7].

Due to both authentication and API format, using the SharePoint REST API is more complicated than using something like the ClinicalTrials.gov API. There are some Python modules available that ease

interaction with the API, for example, Office365-REST-Python-Client[8] but as these are attempting to be full featured clients for the API, they themselves become complicated to use. To address this, I have developed a light-weight module called spLite. spLite aims to abstract away interaction with the API to provide simple methods to get what you want from the API, limited to the functions that I've outlined above. spLite is available on GitHub[1] with an open source MIT license, so you may use this library however you wish.

In the examples that follow, in my Python code I will use spLite to demonstrate, but the completed REST calls will also be provided.

DOWNLOAD A FILE

Going back to one of my original points in this paper, automation capabilities can be greatly enhanced simply by making a connection to a new system. Therefore, being able to download a file from SharePoint programmatically opens many doors.

The REST API call to download a file is as follows:

```
GET https://{site_url}/_api/web/GetFolderByServerRelativeUrl
    ('/folder_name')/Files('{file_name}')/$value
```

There are a few components that you need to specific to your system:

`site_url`	The root-endpoint of your SharePoint site. This should be the home page of your SharePoint site (so it may not necessarily end in .com)
`folder_name`	The subfolder of the SharePoint site that your files resides in. This can occasionally be tricky – because the displayed folder name may be different than the link necessary. For example, 'Documents' may actually be 'Shared Documents'. Also – note that the text must be valid in a URL For example, spaces aren't valid, so 'Shared Documents' should be 'Shared%20Documents'. You can quote from the library urlib.parse in Python to ensure your string is formatted correctly.
`file_name`	The file name of the file that you want to download. Again, make sure that the string is valid in the URL as described in folder name.

We are retrieving data from the server, so note that this is a GET method. A complete URL will look something like this:

```
https://atorusresearchcom.sharepoint.com/sites/ATorusMain/_api/web/GetFolde
rByServerRelativeUrl('Shared%20Documents/General')/Files('test.txt')/$value
```

spLite abstracts away the process of forming the URL for you, and instead takes the fields specified in the table above as parameters. The first thing you must do using spLite to get started is to create a SharePoint session:

```
>>> from spLite import sharepoint
>>> s = sharepoint.SpSession(<site_url>,<username>,<password>)
```

The default authentication method is the SAML Security Token Service for Office 365, but HTTP Basic Authentication and NTLM Authentication are also supported. You can change this using the `context_type` parameter:

```
>>> s = sharepoint.SpSession(<site_url>,<username>,<password>,
    context_type='ntlm')
>>> s = sharepoint.SpSession(<site_url>,<username>,<password>,
    context_type='basic')
```

You can also provide a custom authentication context using the parameter `custom_context` if you require a different type of authentication, provided it returns a dictionary that unpacks into the `auth` or `headers` or a Python request object.

Once the session has been created, you can make your request:

```
>>> s.get_file(<folder>,<file>)
```

Here you are able to use spaces in file names, as the resulting string will be properly quoted when it is entered into the REST API URL.

If you do not provide an output location, then `get_file` will return a file-like object that you can use to process data in memory without writing to disk. This can be useful if you want to avoid having to clean up after your script once you finish processing but need to read information from SharePoint for use throughout the script. If you simply want to download the file locally, you provide an output location:

```
>>> s.get_file(<folder>,<file>, output_location=<folder path>)
```

DOWNLOAD A LIST OF FILES

Extending off downloading files, it's also useful to be able to grab multiple files from a folder, either through a specified list, by extensions, or all the files in the folder. The REST call to list files is very similar to the call to download an individual file:

```
GET https://{site_url}/_api/web/GetFolderByServerRelativeUrl
    ('/{folder_name}')/Files
```

Note that again, we are requesting data so this requires a GET method. spLite supports this as well through the method `list_and_get_files`. This method actually performs multiples REST calls – one call to list the files, then a `get_file` call for each file in the list that will be downloaded. If you want all the files in a folder:

```
>>> s.list_and_get_files(<folder>, output_location=<folder path>)
```

Just a subset of files by name:

```
>>> s.list_and_get_files(<folder>, output_location=<folder path>,
    files=['a.txt','b.txt'])
```

If a specified file that you request does not exist in the folder, you will be warned, but other files will be downloaded. If you want a subset of files by one extension or multiple extensions, the use the code as follows:

```
>>> s.list_and_get_files(<folder>, output_location=<folder path>,
    extensions='.xlsx')
>>> s.list_and_get_files(<folder>, output_location=<folder path>,
    extensions=['.xlsx', .xls])
```

UPLOAD A FILE

Uploading a file is a slightly more complicated process. Instead of retrieving data from the server, we are now creating data in the server – and therefore this is a POST method instead of a GET. The REST call is as follows:

```
POST https://{site_url}/_api/web/GetFolderByServerRelativeUrl
     ('/{folder_name}')/Files/add( url='a.txt',overwrite=true)
```

The structure of the API call is very similar, where the path points to the folder of interest and focuses on files, but now we are using add and setting some parameters on the method: `url`, which creates the file name in SharePoint, and overwrite, which forces the overwrite of the file in SharePoint if it already exists.

A notable change about posting files is that it requires an extra layer of security called a *request digest*[9]. Uploading data into the server has different implications than reading it, so a request digest is a token that proves the validity of your request. The is only valid for a limited period of time, so you must ensure it's valid before making your post request to upload data. This process is handled for you in spLite.

To upload a file to SharePoint using spLite, first make sure you have the correct path to your folder. With this information correct, the method call is very similar to downloading a file:

```
>>> s.upload_file(<folder>,<file>)
```

spLite is set by default to overwrite an existing file, but you can also control this behavior by setting overwrite to false:

```
>>> s.upload_file(<folder>, <file>, overwrite=False)
```

If the parameter data is not provided, file must refer to the file path of an existing file on your local system. Alternatively, you can use the data parameter to provide:

- A `bytes` object
- A file-like object that can be read

For example, if you are to provide a bytes object:

```
>>> import io
>>> my_string = 'Here is some text'
>>> data = bytes(my_string, encoding='utf8')
>>> s.upload_file('some/folder', 'some_file.txt', data=data,
    overwrite=False)
```

Providing a file-like object warrants some more explanation as to a valid use case. Consider the following: you create an automated process the aggregates data on your server. This results in a `pandas.DataFrame` called `df` that is held in memory within your program. One option is to write this file to disk, then read it back in to upload to SharePoint. Alternatively, you could do the following:

```
>>> import pandas as pd

>>> data = df.to_csv()
>>> file = io.BytesIO(bytes(data, encoding='utf8'))
>>> s.upload_file('some/folder', 'my_data.csv', data=file, overwrite=False)
```

Here, the aggregated data is never written to disk, so there is no clean-up to be done after the program is executed, and the data could still be uploaded to SharePoint with the file name `my_data.csv`.

CONCLUSION

Any chance you have to remove a manual process is an opportunity to generate value. When you open doors to connect two systems, many of those opportunities are created. REST APIs open those doors for systems to talk, and SharePoint's API is one that is very advantageous to us in this industry. We can automate the generation of certain reports and post them for stakeholders without access to our servers. We can integrate files stored on SharePoint into our programming processes themselves. We can use languages like Python to put the power of these automation capabilities into the hands of those without access to our typical development environments, like project managers. It does not take a web developer or a software engineer to interface with this technology anymore. If the doors look closed, they may not be locked – so it is well worth seeing which ones you can open.

REFERENCES

1. Stackhouse, M. (2020, January 21). mstackhouse/spLite. Retrieved from https://github.com/mstackhouse/spLite
2. Zell, & Liew. (2018, January 17). Understanding And Using REST APIs. Retrieved from https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/
3. Downloading Content for Analysis. (n.d.). Retrieved from https://clinicaltrials.gov/ct2/resources/download
4. (n.d.). Retrieved from https://www.reddit.com/dev/api/
5. HTTP headers. (n.d.). Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
6. HTTP response status codes. (n.d.). Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
7. Spdevdocs. (n.d.). Get to know the SharePoint REST service. Retrieved from https://docs.microsoft.com/en-us/sharepoint/dev/sp-add-ins/get-to-know-the-sharepoint-rest-service?tabs=csom
8. Gremyachev, V. (2020, January 13). vgrem/Office365-REST-Python-Client. Retrieved from https://github.com/vgrem/Office365-REST-Python-Client
9. Spdevdocs. (n.d.). Work with __REQUESTDIGEST. Retrieved from https://docs.microsoft.com/en-us/sharepoint/dev/spfx/web-parts/basics/working-with-requestdigest

CONTACT INFORMATION

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

Mike Stackhouse
Atorus Research
Mike.Stackhouse@atorusresearch.com
https://www.linkedin.com/in/michael-s-stackhouse/