GraphQL API + Python Integration

Technical Brief

Last Updated: 10/8/2020
Verified using Python 2.7 and Juypter 0.33.12

Table of Content

Introduction

The Pluralsight GraphQL API can be utilized within Python. The method for pulling the data is easy to replicate for all graphql objects, though you must review the schema documentation if you want to know what entities and fields are available to pull. For a quick start, review the resources section below. For a more in depth explanation, read on.

Prerequisites

- Experience using Python
- You will need the following prior to following along:
  - Python Script Templates.zip
  - Pluralsight PaaS API Key

Basic Query Structure

Let's use a basic query that returns the name of the skills assessments in the catalog:

```
query {
  skillAssessmentCatalog {
    nodes {
      name
    }
  }
}
```

To begin, you will need to import the request library by writing the following line.

```
import requests
```

Next define the url, query and the request information. The script below leverage the following three variables:

1. url - stores the url to Pluralsight's endpoint
2. payload - stores the GraphQL query. Please note, that since the string is wrapped in double quotes, a backslash is needed inside the quotes (\")
3. headers - is an object that contains both the content-type and authorization.

```
url = "https://paas-api.pluralsight.com/graphql"
payload = "{\"query\":
             \"{ skillAssessmentCatalog
                  { nodes
                      { name }
                  }
               } \"}"

headers = {
    'content-type': "application/json",
    'authorization': "Bearer <API Key>"
    }

response = requests.request("POST", url, data=payload,
headers=headers)

print(response.text)
```

This will return a response similar to below:

```
{"data":{
  "skillAssessmentCatalog":{
    "nodes":[{"name":"Kotlin: App Data and Storage"}]}
  }
}
```

Pagination Structure

This section focuses on paging through the query. By default, GraphQL will return the first 100 rows unless specified. Pagination parameters allow you to specify how many pieces of data you want to read and where you want to start reading from. Each data entry has a cursor, which is an opaque string identifying the entry's place in the greater dataset. Queries that offer pagination will have the following two parameters:

- first: Specifies how many records to fetch per query (limited to 1000 per query).
- after: Used to request the next result set from the query. Your query will return the value PageInfo.endCursor, and you should put that value here for the next query. (has a rate-limit of 100 requests per minute)

Query-Size Pagination Method

The query-size pagination method leverages the first parameter to be inserted into the query string inside the body. As an example, let's use the basic query:

```
query {
  skillAssessmentCatalog {
    nodes {
      name
    }
  }
}
```

By default every query sent to GraphQL has additional properties that can be added with the query. For this method, it requires the following property **first**. **First** specifies the number of records to fetch per query.

With this logic, the query mentioned above will need to be modified to include the ***first*** keyword and the variable name (e.g. totalcount).

```
import requests

vTotalCount = 10;
vApiKey = "";
url = "https://paas-api.pluralsight.com/graphql"
payload = "{\"query\":
                \"{ skillAssessmentCatalog (first: %d )
                        { nodes {
                             name
                            }
                         }
                    }
            \"}" % (vTotalCount)

headers = {
    'content-type': "application/json",
    'authorization': "Bearer %s" % (vApiKey)
    }

response = requests.request("POST", url, data=payload,
headers=headers)
```

```
print(response.text)
```

Recursive Pagination Method

The recursive pagination method leverages the parameter **after** as well as it requires a new object called **pageInfo**. The **after** parameter allows the query to request the next result set. Your query will return the value PageInfo.endCursor, and you should put that value here for the next query.

As an example, let's use this basic query:

```
query { skillAssessmentCatalog { nodes { name } } }
```

Every Pluralsight query sent to GraphQL has additional properties that can be added with the query. For this method, it requires the following property **after** and a new object called **pageInfo** that contains two keys: **endCursor** and **hasNextPage** to be added. For example,:

```
query {
  skillAssessmentCatalog (after: "") {
    pageInfo {
      endCursor,
      hasNextPage
    }
    nodes {
      name
    }
  }
}
```

In addition, since a new object is being introduced, the query string needs to reflect this new object. Therefore, the new python code will need to include a new set of variables to store the endCursor and hasNextPage values. Therefore the python will be updated with the following addition:

```
vResult = [] #Variable to store all results during a loop

while True:

  vNow = datetime.datetime.now()
  vStartTime = (vNow -
```

```
datetime.datetime(1970,1,1)).total_seconds()

  #Payload will query the skills assessment table to fetch all the
fields needed to be brought into python.
  #Below includes the pageInfo table as we will need it for
recursive.
  #To find the Schema definition per table, please go here:
https://dev-portal.pluralsight.com/schema-docs/query.doc.html
  payload = "{\"query\":\"{ skillAssessmentCatalog (first: %d
after: %s ) { pageInfo { hasNextPage, endCursor } nodes{ name } }
} \"}" % (vTotalCount, vEndCursor)

  #Response stores the result into a variable
  response = requests.request("POST", url, data=payload,
headers=headers)
  #JsonData converts the data into a json object
  jsonData = json.loads(response.content)

  #vEndCursor stores the endCursor value so that it can be pass
again in the next loop
  vEndCursor = "\\\"%s\\\"" %
(jsonData['data']['skillAssessmentCatalog']['pageInfo']['endCursor
'])

  #vResults is an array that stores all response into one object

vResult.append(jsonData['data']['skillAssessmentCatalog']['nodes']
)

  #vCounter stores the number of time Python has made a call to
GraphQL
  vCounter += 1

  #Checks to see if Python has made 100 calls in under a min.
  #If Yes, sleep for 60 seconds
  if (vCounter == 99 and vStartTime > vEndTime ):
    print('Going to sleep for 60')
    time.sleeip(60)
    print('Waking up from sleep and reseting')
    vCounter = 0
    vEndTime = ((vNow + datetime.timedelta(0,60)) -
datetime.datetime(1970,1,1)).total_seconds()
```

```
  elif vStartTime > vEndTime :
    vCounter = 0
    vEndTime = ((vNow + datetime.timedelta(0,60)) -
datetime.datetime(1970,1,1)).total_seconds()

  if
jsonData['data']['skillAssessmentCatalog']['pageInfo']['hasNextPag
e'] is False:
    break
```

As stated earlier, GraphQL has a rate-limit of 100 requests per minute and since we are not using cursor-based pagination (using property **first**); each query will return 100 rows. So after each query, a check must be made to ensure we are within the rate-limit or an error will return. In order to solve this problem, three variable needs to be introduced:

```
vCounter = 0
vNow = datetime.datetime.now()
vStartTime = (vNow - datetime.datetime(1970,1,1)).total_seconds()
vEndTime = ((vNow + datetime.timedelta(0,60)) -
datetime.datetime(1970,1,1)).total_seconds()
```

After a query returns 100 results, these four variables can be used to check if the following two conditions have occurred:

1. Is the vCounter at 99
2. Is vStartTime (time when a query started - needs to be converted to epoch) is greater than vEndTime (a minute added to when a query started - needs to be converted to epoch).

If the two conditions above are met, then the script should sleep for 1 minute so that the rate-limiter resets. Below is the conditional logic:

```
vCounter += 1

  if (vCounter == 99 and vStartTime > vEndTime ):
    print('Going to sleep for 60')
    time.sleeip(60)
    print('Waking up from sleep and reseting')
    vCounter = 0
    vEndTime = ((vNow + datetime.timedelta(0,60)) -
datetime.datetime(1970,1,1)).total_seconds()
  elif vStartTime > vEndTime :
```

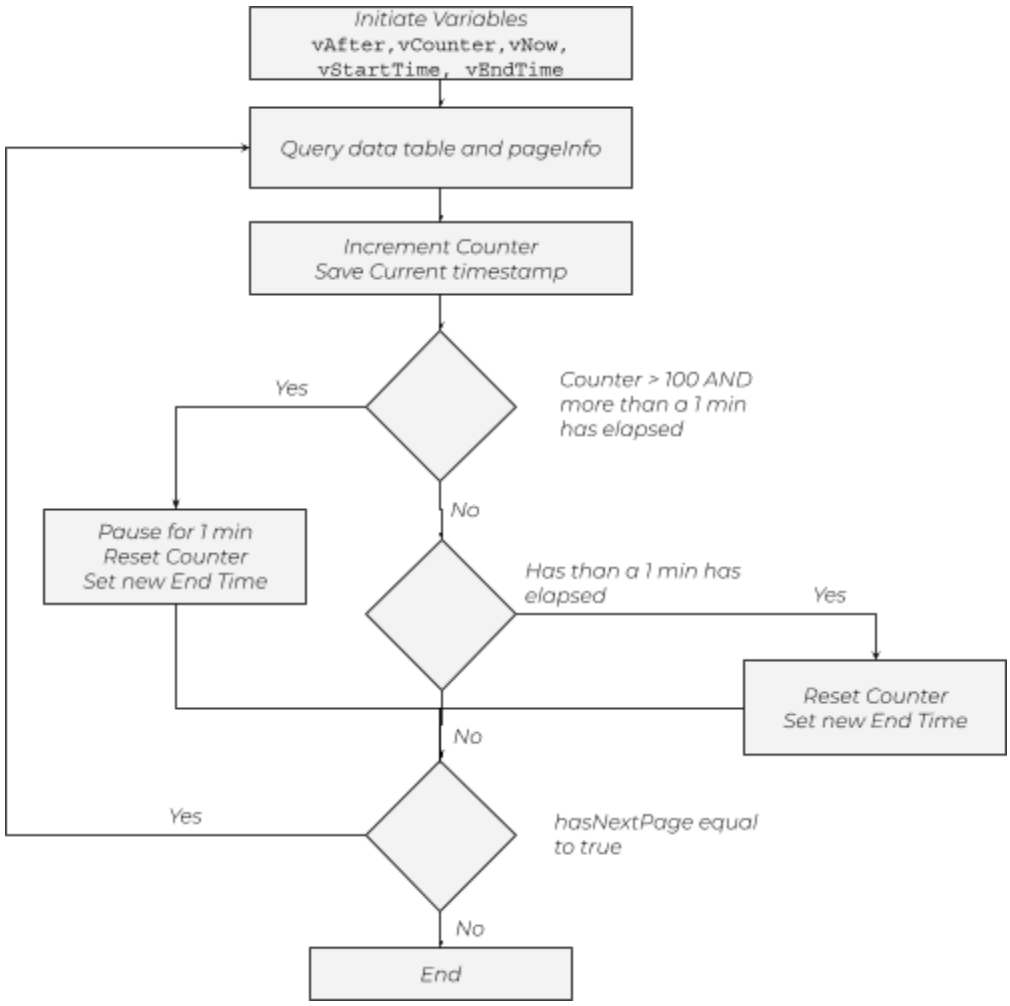
```
    vCounter = 0
    vEndTime = ((vNow + datetime.timedelta(0,60)) -
datetime.datetime(1970,1,1)).total_seconds()

  if
jsonData['data']['skillAssessmentCatalog']['pageInfo']['hasNextPag
e'] is False:
    break
```

As mentioned above, this approach comes with some pros and cons which are listed below:

Pros	Cons
• Unlimited number of rows to ingest • Takes into account the rate-limiter	• Complex coding, therefore more maintenance to up-keep • Requires multiple calls to GraphQL which will take longer for the data to load.

Finally the visual diagram below outlines how the script comes together.




```
#Sample Script provides
import requests
import json
import datetime
import time

vTotalCount = 10
vEndCursor = "\\\"\\\""

vCounter = 0
vNow = datetime.datetime.now()
vStartTime = (vNow - datetime.datetime(1970,1,1)).total_seconds()
vEndTime = ((vNow + datetime.timedelta(0,60)) -
datetime.datetime(1970,1,1)).total_seconds()

vApiKey = ""
url = "https://paas-api.pluralsight.com/graphql"
```

```
headers = {
    'content-type': "application/json",
    'authorization': "Bearer %s" % (vApiKey)
    }

vResult = []

while True:

  vNow = datetime.datetime.now()
  vStartTime = (vNow -
datetime.datetime(1970,1,1)).total_seconds()

  payload = "{\"query\":\"{ skillAssessmentCatalog (first: %d
after: %s ) { pageInfo { hasNextPage, endCursor } nodes{ name } }
} \"}" % (vTotalCount, vEndCursor)

  response = requests.request("POST", url, data=payload,
headers=headers)
  jsonData = json.loads(response.content)

  vEndCursor = "\\\"%s\\\"" %
(jsonData['data']['skillAssessmentCatalog']['pageInfo']['endCursor
'])


vResult.append(jsonData['data']['skillAssessmentCatalog']['nodes']
)

  vCounter += 1

  if (vCounter == 99 and vStartTime > vEndTime ):
    print('Going to sleep for 60')
    time.sleep(60)
    print('Waking up from sleep and reseting')
    vCounter = 0
    vEndTime = ((vNow + datetime.timedelta(0,60)) -
datetime.datetime(1970,1,1)).total_seconds()
  elif vStartTime > vEndTime :
    vCounter = 0
    vEndTime = ((vNow + datetime.timedelta(0,60)) -
datetime.datetime(1970,1,1)).total_seconds()
```

```
  if
jsonData['data']['skillAssessmentCatalog']['pageInfo']['hasNextPag
e'] is False:
    break

print(vCounter)
print(vResult)
```

Conclusion

The Pluralsight GraphQL API can be utilized from Python. By combining both Pluralsight's GraphQL data with internal customer data, customers can create unique analytical views that are tailored to their objectives.

Resources

- GraphQL Documentation hosted on our Developer portal
  - Pluralsight Developer Portal: Home
- Playground for exploring & converting query to basic python query
  - GraphQL Playground
- Python sample file
  - Python Script Templates.zip

FAQ

Q: TLDR. Got anything to quickly get started?
A: Python Script Templates.zip. Download. Open. Replace API token in Queries. Profit.

Q: Does the GraphQL API documentation cover data on SKILLS and FLOW plans?
A: Not at this time. It is only available for SKILLS data, though a plan that has both SKILLS and FLOW products can utilize the API to pull SKILLS data.

Q: Is there a limitation to the number of queries we create?
A: There is no limit to the number of queries. To minimize large data pulls, please use filters and pagination to create delta data pulls (and avoid all-time reports--especially for large datasets like CourseUsage, ChannelProgress, etc). This reduces load on our servers and provides a better experience for you and all customers who are using the API endpoint.

Q: How do I request an API key?
A: This can only be done by a Pluralsight Plan Admin on the Manage Keys tab of

the Developer Portal. If you do not know who your Plan Admin(s) is/are, please reach out to your dedicated CSM or support resource to identify them.

Q: My company has more than 1 Pluralsight plan. Is there a way to pull data for more than 1 plan?How do I request a multi-plan API key?
A: Yes, a multi-plan API key is what you need. To request one, please reach out to us at professionalservices@pluralsight.com. As long as each plan has an Integrations or ProServ SKU, we will be able to add it to a multi-plan API key for you.

Q: How long does it take to generate an API token?
A: After submitting a request through the Developer Portal link in the resource section, your API will be approved and ready to use immediately. If there is a delay for any reason or it does not function, please contact support@pluralsight.com.

Q: Will you create and maintain the Python script for our organization?
A: Not at this time. We may support a templated dashboard sample in the future.

Q: Does this cost extra?
A: If you have purchased the Integration's SKU, or have an Enterprise SKU that was purchased before July 1st 2020 and it has not yet expired and been renewed, then there is no additional charge. If you have not purchased our Integrations SKU and have a subscription starting after July 1st 2020, whether a new subscription or renewal, then you will have to speak to your CSM and Sales rep to get started.

Q: What data can I not see?
A: If it is not listed in the Developer Portal documentation schema, then it is not available. Keep an eye on the change log section of the webpage for updates.

Q: I do not like using GraphQL APIs or are unfamiliar with them. Can I continue to use your REST APIs?
A: You may continue to use our SKILLS REST APIs currently, however they will be decommissioned in Q2 2021.. The data availability and manipulation through the REST APIs is not as complete. For example you can't get SkillIQ, RoleIQ, etc from REST APIs. We will be using GraphQL APIs as the standard for the SKILLS product. FLOW (formerly GitPrime) will continue to use REST. If you are using REST APIs currently for SKILLS data and want to know how to make the switch to GraphQL, please see the help documentation here or email us at professionalservices@pluralsight.com.