# Chuco Gives a FAQ 2024: Advanced GraphQL Examples

Version: 1.0

Last Update: Oct 22nd, 2024

# Table of Contents

# What is GraphQL

For our use case with the Tanium Gateway, GraphQL is a query language that allows us to ask Tanium for data or tell Tanium to change something. The functions we use to do that are ‘query’ and ‘mutation’ respectively.

If you’re just getting started with GraphQL and the Tanium Gateway, check out Tanium’s documentation along with their extensive list of examples.

- Overview: https://help.tanium.com/bundle/ug_gateway_cloud/page/gateway/overview.html
- Examples: https://help.tanium.com/bundle/ug_gateway_cloud/page/gateway/gateway_examples.html

If you already have a grasp on the basic concept and want to dive deeper, read on.

# Using GraphQL in Python

You may already be familiar with how to make GET and POST requests using your scripting language of choice. But GraphQL is a little different in that whether querying data or using a mutation to make a change, only the POST method will be used. The query will be sent in the payload of the request and the response will be JSON.

Let's see some python examples.

## Resources

Refer to the **Examples/GraphQL/** folder in the GitHub Repository for both Python and PowerShell example code: https://github.com/team-chuco/chuco-gives-a-faq-2024

## Query Example

### Import Libraries

First, we are going to take advantage of the following libraries:

- requests
- json

**NOTE**: The requests library will need to be installed, follow the documentation here.

python_graphql_query.py excerpt:

```python
import requests
import json
```

## Load Configuration Data From a JSON file

Next, we are going to load a config.json file. This file exists in the same directory as our python script and is used to store configuration data. It is best practice to avoid hardcoding sensitive data in your script, such as authorization tokens. This makes the script easier to configure and share.

python_graphql_query.py excerpt:

```python
# Import the config file with the URL and Token
with open('./config.json', 'r') as f:
    config = json.load(f)

instance_url = config['instance_url']
token = config['token']
```

config.json:

```
{
    "instance_url": "https://{instance_name}-api.cloud.tanium.com",
    "token": "token-...",
    "computer_name_filter_value": "win"
}
```

**NOTE**: When setting the instance URL, use the following formats (Notice there is no trailing forward slash):

- **Tanium Cloud**: https://{instance-name}-api.cloud.tanium.com
- **Tanium On-Prem**: https://{instance-name-and-domain}

We will cover the ‘computer_name_filter_value’ a little later on in this example.

## Create a Session

Now let's create a session using our url and token.

python_graphql_query.py excerpt:

```python
# Build the api url for the Tanium Gateway
api_path = "/plugin/products/gateway/graphql"
api_url = instance_url + api_path

# Create a requests session and add the token to headers
session = requests.Session()
session.headers.update({'session': token})
```

## Helper Function to Build the GraphQL Payload

Here we have a function to combine the query and variable if necessary. This allows you to define variables using a native python dictionary.

python_graphql_query.py excerpt:

```python
# A helper function to create the payload and include variables if needed
def graphql_query_body(query: str, variables: dict|None = None) -> dict:
    body = {
        "query": query
    }

    if variables is not None:
        body["variables"] = variables # type: ignore

    return body
```

## Build a Query

Now we can build the query using a python literal string as denoted by the three quotation marks. This allows you to enter the GraphQL query without having to escape special characters.

python_graphql_query.py excerpt:

```python
# Create a query string
query = """query getEndpoints($value: String) {
    endpoints(filter: {path: "name", op: CONTAINS, value: $value}) {
        edges {
            node {
                name
                os {
                    name
                }
                ipAddress
            }
        }
    }
}"""
```

## Query Breakdown

Let's pause here and look at the above query line by line:

1. `query getEndpoints($value: String)`
    - This declares a query named "getEndpoints" that accepts a variable called `$value` of type String. This variable will be used later to filter the endpoints. Note that the query name is whatever you want it to be and not a predefined value.
2. `endpoints(filter: {path: "name", op: CONTAINS, value: $value})`
    - This is the core of the query. It requests a list of "endpoints" and applies a filter to them.
        - `filter`: Specifies the filtering criteria.
        - `path: "name"`: Indicates that the filter should be applied to the "name" field of the endpoints.

        - `op: CONTAINS`: Defines the filter operation as "CONTAINS", meaning it will look for endpoints with names containing the provided `$value`.
        - `value: $value`: Uses the `$value` variable passed to the query as the value to filter by.
3. `edges { ... }`
    - In GraphQL, lists of items are often returned as connections with "edges" and "nodes". This line indicates that we want to access the "edges" of the endpoint connection.
4. `node { ... }`
    - Within each "edge", we access the actual "node", which represents the endpoint object itself.
5. `name`
    - This requests the "name" field of the endpoint object.
6. `os { name }`
    - This requests the "os" field of the endpoint object, which itself is an object with a "name" field.
7. `ipAddress`
    - This requests the "ipAddress" field of the endpoint object.

## Build Variables

The query above defines the $value variable used to filter the query results based on the computer name. Remember the ‘computer_name_filter_value’ from the config.json? You can set the query variables using a python dictionary and inject the computer_name_filter_value from the config. Now the query is configurable outside the script.

python_graphql_query.py excerpt:

```python
# Import the filter_value from config.json
filter_value = config['computer_name_filter_value']

# Create a dictionary with variables
variables = {
    "value": filter_value
}
```

## Create the Payload and Send the Request

Here we use the helper function to combine the query string and variables dictionary. Then we send the request to Tanium.

python_graphql_query.py excerpt:

```python
# Use the helper function to create the payload
body = graphql_query_body(query, variables)

# Post the request
response = session.post(api_url, json = body)
```

## Print the Response

We can print the raw data returned by the request.

python_graphql_query.py excerpt:

```python
# View the raw data
print(json.dumps(response.json(), indent=2))
```

Output:

```python
{
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
            "name": "Win10-LWSJS",
            "os": {
              "name": "Windows 10 Enterprise"
            },
            "ipAddress": "10.0.0.188"
          }
        },
        {
          "node": {
            "name": "Win11-WS02.chuco.lab",
            "os": {
              "name": "Windows 11 Pro"
            },
            "ipAddress": "10.10.1.137"
          }
        }
      ]
    }
  }
}
```

## Format the Response Using a List Comprehension

Or we can format the response to make the data easier to consume.

python_graphql_query.py excerpt:

```python
# Clean up the response using a list comprehension
data = [item['node'] for item in response.json()['data']['endpoints']['edges']]
print(json.dumps(data, indent=2))
```

Output:

```
[
  {
    "name": "Win10-LWSJS",
    "os": {
      "name": "Windows 10 Enterprise"
    },
    "ipAddress": "10.0.0.188"
  },
  {
    "name": "Win11-WS02.chuco.lab",
    "os": {
      "name": "Windows 11 Pro"
    },
    "ipAddress": "10.10.1.137"
  }
]
```

## Mutation Example

Let's dive deeper with a mutation example. Much of the code is the same as the query example above, we simply change the query, variables, and the way we output the results.

### Build a Mutation

In this example, the mutation is going to use some more “advanced” GraphQL features such as fragments and aliases to help make the query more configurable and readable. The goal of this example is to create a platform package for Windows and/or Non-Windows clients with dynamic parameters and filtering.

**NOTE**: There is a better way to deploy custom tags using the API Gateway, but the example below can be used with any set of platform packages.

python_graphql_mutation.py excerpt:

Python

```
# Create a query string
query = """mutation createTaniumAction($comment: String, $name: String,
$packageName_windows: String, $packageName_non_windows: String, $params:
[String!], $actionGroupName: String!, $filters: [ComputerGroupFilter!],
$include_windows: Boolean!, $include_non_windows: Boolean!) {
  ...windows_action @include(if: $include_windows)
  ...non_windows_action @include(if: $include_non_windows)
}

fragment windows_action on Mutation {
  windows: actionCreate(
    input: {comment: $comment, name: $name, package: {name:
$packageName_windows, params: $params}, targets: {platforms: [Windows],
actionGroup: {name: $actionGroupName}, targetGroup: {filter: {filters:
$filters}}}}
  ) {
    ...actionPayload
  }
}

fragment non_windows_action on Mutation {
  non_windows: actionCreate(
    input: {comment: $comment, name: $name, package: {name:
$packageName_non_windows, params: $params}, targets: {platforms: [Linux, Mac,
AIX, Solaris], actionGroup: {name: $actionGroupName}, targetGroup: {filter:
{filters: $filters}}}}
```

```
  ) {
    ...actionPayload
  }
}

fragment actionPayload on ActionCreatePayload {
  action {
    id
  }
  error {
    message
  }
}"""
```

## Mutation Breakdown

It's kind of a lot, so let's break this down line by line:

1. Mutation Declaration:
    - `mutation createTaniumAction($comment: String, $name: String, $packageName_windows: String, $packageName_non_windows: String, $params: [String!], $actionGroupName: String!, $filters: [ComputerGroupFilter!], $include_windows: Boolean!, $include_non_windows: Boolean!)`
        - This line declares the mutation `createTaniumAction` and defines the variables it accepts. `createTaniumAction` is just a name I made up, you can call it anything you would like:
            - `$comment`: A string for the action's comment.
            - `$name`: A string for the action's name.
            - `$packageName_windows`: A string for the package name on Windows.
            - `$packageName_non_windows`: A string for the package name on non-Windows systems.
            - `$params`: An array of strings (non-nullable) for parameters.

        - `$actionGroupName`: A non-nullable string for the action group name.
        - `$filters`: A non-nullable array of `ComputerGroupFilter` objects for filtering target computers.
        - `$include_windows`: A non-nullable boolean to indicate whether to create the action for Windows.
        - `$include_non_windows`: A non-nullable boolean to indicate whether to create the action for non-Windows.
2. Conditional Inclusion:
    - `...windows_action @include(if: $include_windows)`
        - This line includes the `windows_action` fragment only if the `$include_windows` variable is true.
    - `...non_windows_action @include(if: $include_non_windows)`
        - This line includes the `non_windows_action` fragment only if the `$include_non_windows` variable is true.
3. Fragments:
    - `fragment windows_action on Mutation { ... }` and `fragment non_windows_action on Mutation { ... }`
        - These define reusable fragments for Windows and non-Windows actions. Each fragment calls the `actionCreate` mutation with specific input for the respective platforms.
        - The `windows:` and `non-windows:` before `actionCreate` are aliases. This not only helps us differentiate the results of our query, but also gives us the ability to have two `actionCreate` mutations in a single query.
        - Inside the `input`:
            - `comment`, `name`, `package`: Provide details about the action and the package to execute.
            - `targets`: Specifies the target computers for the action, including:
                - `platforms`: An array indicating the operating systems (Windows, Linux, Mac, etc.).

        - `actionGroup`: Specifies the action group.
        - `targetGroup`: Defines target computers with filters.
            - The `filters` variable lets us define our filter outside the query.

4. Action Payload:
    - `fragment actionPayload on ActionCreatePayload { ... }`
        - This fragment defines the structure of the response returned after creating the action. It retrieves the `id` of the created action and any potential `error` message.

## Build Variables

This time, we have quite a few more variables. This makes the above query reusable for multiple different use cases.

python_graphql_mutation.py excerpt:

```python
# Import the filter_value from config.json
computer_name_filter_value = config['computer_name_filter_value']
custom_tag_filter_value = config['custom_tag_filter_value']
custom_tags = config['custom_tags']

# Create a dictionary with variables
variables = {
  "name": "Add Tag via GraphQL",
  "comment": "GraphQL Tagging",
  "packageName_windows": "Custom Tagging - Add Tags",
  "packageName_non_windows": "Custom Tagging - Add Tags (Non-Windows)",
  "params": custom_tags,
  "actionGroupName": "Default - All Computers",
  "filters": [
    {
      "negated": False,
      "any": False,
      "sensor": {
        "name": "Computer Name"
      },
      "op": "CONTAINS",
      "value": computer_name_filter_value
    },
```

```
    {
      "negated": False,
      "any": False,
      "sensor": {
        "name": "Custom Tag Exists",
        "params": [
          {
            "name": "tag",
            "value": custom_tag_filter_value
          }
        ]
      },
      "op": "EQ",
      "value": "true"
    }
  ],
  "include_windows": True,
  "include_non_windows": True
}
```

## Format the Response Using a List Comprehension

We send the query just like before, but the response is formatted a little differently so we can revamp the list comprehension accordingly:

python_graphql_mutation.py excerpt:

```python
# Clean up the response using a list comprehension
data = [{"platform": platform, "action_id": data['action']['id']} for platform,
data in response.json()['data'].items()]
print(json.dumps(data, indent=2))
```

Output:

```
[
  {
    "platform": "windows",
    "action_id": "1859815"
```

```
  },
  {
    "platform": "non_windows",
    "action_id": "1859816"
  }
]
```